VISUALIZATION, STATISTICAL ANALYSIS, AND MINING OF HISTORICAL VESSEL DATA

SABARISH SENTHILNATHAN MUTHU

February 2015

GEODESY AND GEOMATICS ENGINEERING
UNB

TECHNICAL REPORT
NO. 296

VISUALIZATION, STATISTICAL ANALYSIS, AND MINING OF HISTORICAL VESSEL DATA

Sabarish Senthilnathan Muthu

Department of Geodesy and Geomatics Engineering
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada
E3B 5A3

February 2015

PREFACE

This technical report is a reproduction of a thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Geodesy and Geomatics Engineering, February 2015. The research was supervised by Dr. Emmanuel Stefanakis, and funding was provided by the Natural Sciences and Engineering Research Council – Canada Research Chair Program.

As with any copyrighted material, permission to reprint or quote extensively from this report must be received from the author. The citation to this work should appear as follows:

Senthilnathan Muthu, Sabarish (2015). *Visualization, Statistical Analysis, and Mining of Historical Vessel Data.* M.Sc.E. thesis, Department of Geodesy and Geomatics Engineering, Technical Report No. 296, University of New Brunswick, Fredericton, New Brunswick, Canada, 150 pp.

ABSTRACT

An important area of research in marine information systems is the management and analysis of the large and increasing amount maritime spatio-temporal datasets. There are a lack of systems that may provide visualization and clustering techniques for large spatiotemporal datasets (Oliveira, 2012). This thesis describes the design and implementation of a prototype web-based system for visualizing, computing statistics, and detecting outliers of moving vessels over a massive set of historic AIS data from the Aegean Sea in the Mediterranean. This historic AIS data was acquired from the Marine Traffic project (MarineTraffic, 2014) which collects the raw location points of the vessels. The web-based system provides the following functionalities: (i) user interface to upload the location points of vessels into a database, (ii) detailed and simplified trajectory construction of the uploaded location points of vessels, (iii) distance, speed, direction, and turn angle computation of the constructed trajectories, (iv) identify vessels that intersect the European Union's Natura 2000 protected areas, (v) identify spatio-temporal outliers in the location points of vessels using DBSCAN algorithm, and (vi) heat map visualization to show the traffic load and highlight sea zones of high risk.

The architecture of the web-based system employed is based on open standards, and allows for interoperable data access. The system was implemented using PHP as the server-side scripting language, and Google Maps API as the client-side scripting language. Furthermore, improved system responsiveness, and server performance was

achieved by asynchronous interaction between client and server by utilizing AJAX to send and receive requests. In addition, data transfer between client and server was achieved using the platform-independent and light weight JSON format.

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude to my supervisor, Dr. Emmanuel Stefanakis for all the help, guidance, and support he has provided me during my Master’s at the University of New Brunswick. I am very grateful for the time he took to give me valuable suggestions, and feedback. I would like to thank the rest of my thesis committee members, Dr. David Coleman, and Dr. Paul Peters for kindly accepting at the last moment to be the members of my thesis committee, and for their suggestions. I would also like to thank Sylvia Whitaker for all the administrative help.

Also, I would like to thank Dr. Dimitrios Lekkas for providing the historical vessel dataset. I would also like to thank my friends at Fredericton for their support, and friendship. Last but not least, I would like to thank my mother, and father for their unconditional love, and support all throughout my life.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

AIS	Automatic Identification System
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interfaces
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
DBMS	Database Management System
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
ECQL	Extended Common Query Language
EPSG	European Petroleum Survey Group
GeoRSS	Geographically Encoded Objects for RSS feeds
GIS	Geographic Information System
GiST	Generalized Search Tree
GPS	Global Positioning System
GML	Geography Markup Language
GRASS	Geographic Resources Analysis Support System
GUI	Graphical User Interface
HTML	HyperText Markup Language

HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
KDD	Knowledge Discovery from Databases
KML	Keyhole Markup Language
MMSI	Maritime Mobile Service Identity
OGC	Open Geospatial Consortium
PHP	HyperText Preprocessor
PyWPS	Python Web Processing Service)
RFID	Radio-Frequency Identification
SQL	Structured Query Language
TIGER	Topologically Integrated Geographic Encoding and Referencing
UML	Unified Modeling Language
URL	Uniform Resource Locator
WCS	Web Coverage Service
WFS	Web Feature Service
WKT	Well-Known Text
WMS	Web Map Service
WPS	Web Processing Service

XML	Extensible Markup Language

CHAPTER 1 INTRODUCTION

The recent, rapid advancements in capturing the location of moving objects through, for example, GPS sensors and RFID tags have increased the possibility of locating moving objects with improved accuracy. These mobile positioning technologies have combined to produce vast amounts of spatiotemporal data. This has also developed a need to provide an interface through which these data could be stored, viewed, and analyzed. Knowledge Discovery from Databases (KDD) is a response to the enormous volumes of data being collected and stored in operational and scientific databases (Miller and Han, 2008). Mobility data typically includes large amount of trajectory data of concrete objects. Analysis of trajectory data is the key to a growing number of applications aiming at global understanding and management of complex phenomena that involve moving objects (e.g. worldwide courier distribution, city traffic management, bird migration monitoring) (Spaccapietra et al., 2008). A *trajectory* is described as a series of individual time-stamped positions, representing geographical coordinates at a certain time.

The effects caused by major sea collisions and oil spills have direct impacts on the environment and human lives. Such damages greatly affect the maritime ecosystem and causes serious problem to the marine protected species and protected areas. This greatly affects the economic sector. This has developed a need to develop functionality to determine the potential environmental impact of ship-based sea collisions and oil spills.

For purposes of this research, the location information of the ships has been acquired from the real historic Automatic Identification System (AIS) data posted between the time period ’01-Aug-2012’ and ‘01-Aug-2014’ across a region in the Aegean Sea.

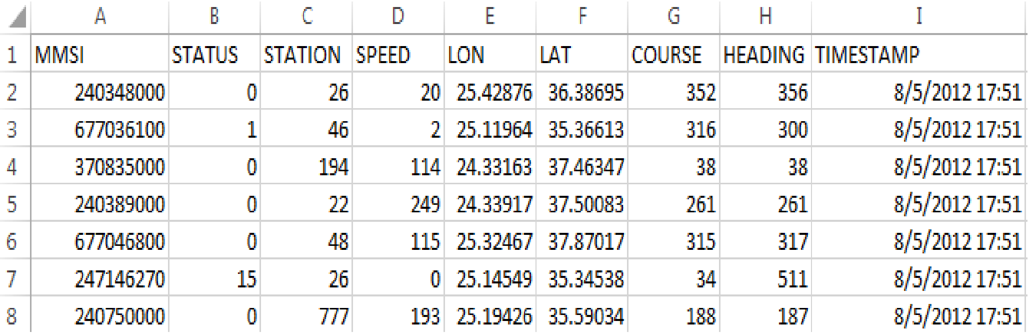
An AIS is a low cost system that can range in price between $500 and $4,000 (USCG Navigation Center, 2014) and provides information - including the ship's identity, type, position, course, speed, navigational status and other safety-related information - automatically to appropriately equipped shore stations, other ships and aircraft (IMO, 2015). The primary purpose of an AIS system is to allow ships to identify the locations of other ships that are in the vicinity. The base stations are equipped with an AIS receiver, a PC, and an Internet connection (MarineTraffic, 2014). The AIS unit in each ship processes the received data and sends these information into a central database from which ships will be able identify the identity, type, position, course, speed, navigational status and other safety-related information of ships that are in the vicinity.

The real, historic, and raw AIS dataset in Comma-Separated Values (CSV) format comes from the Marine Traffic project (MarineTraffic, 2014), an academic, open, community-based research project which provides the position of ship movements. The dataset is acquired for an area in the Aegean Sea extending between 35° 0' 0.0396" - 37° 59' 59.9598" East longitude and 24° 0' 0.0684" - 26° 59' 59.964" North longitude. The data representing ship movements is an example of trajectory data. The following are the details that were acquired from the Marine Traffic project dataset:

Table 1.1: Attributes of AIS data

Attribute	**Description**
LON and LAT	The position of the ships in WGS 84 coordinate reference system
TIMESTAMP	The time at which position of the ship was recorded in Coordinated Universal Time (UTC)
MMSI	The Maritime Mobile Service Identity (MMSI) is the unique identification number of ships
STATUS	The status of the ships, whether it is anchored or moving
STATION	The station providing the signal
HEADING	The azimuth of the ship bow
COURSE	The ship course direction (azimuth) 0-360 deg

Figure 1.1 illustrates a snapshot of the properties such as the MMSI identifier, spatio-temporal location, type, heading, etc. of moving vessels in the Aegean Sea acquired from the MarineTraffic project. Figure 1.2 shows MarineTraffic user interface that displays the various ships' current position in the Aegean Sea.



	A	B	C	D	E	F	G	H	I
1	MMSI	STATUS	STATION	SPEED	LON	LAT	COURSE	HEADING	TIMESTAMP
2	240348000	0	26	20	25.42876	36.38695	352	356	8/5/2012 17:51
3	677036100	1	46	2	25.11964	35.36613	316	300	8/5/2012 17:51
4	370835000	0	194	114	24.33163	37.46347	38	38	8/5/2012 17:51
5	240389000	0	22	249	24.33917	37.50083	261	261	8/5/2012 17:51
6	677046800	0	48	115	25.32467	37.87017	315	317	8/5/2012 17:51
7	247146270	15	26	0	25.14549	35.34538	34	511	8/5/2012 17:51
8	240750000	0	777	193	25.19426	35.59034	188	187	8/5/2012 17:51

Figure 0.1: Historical AIS data

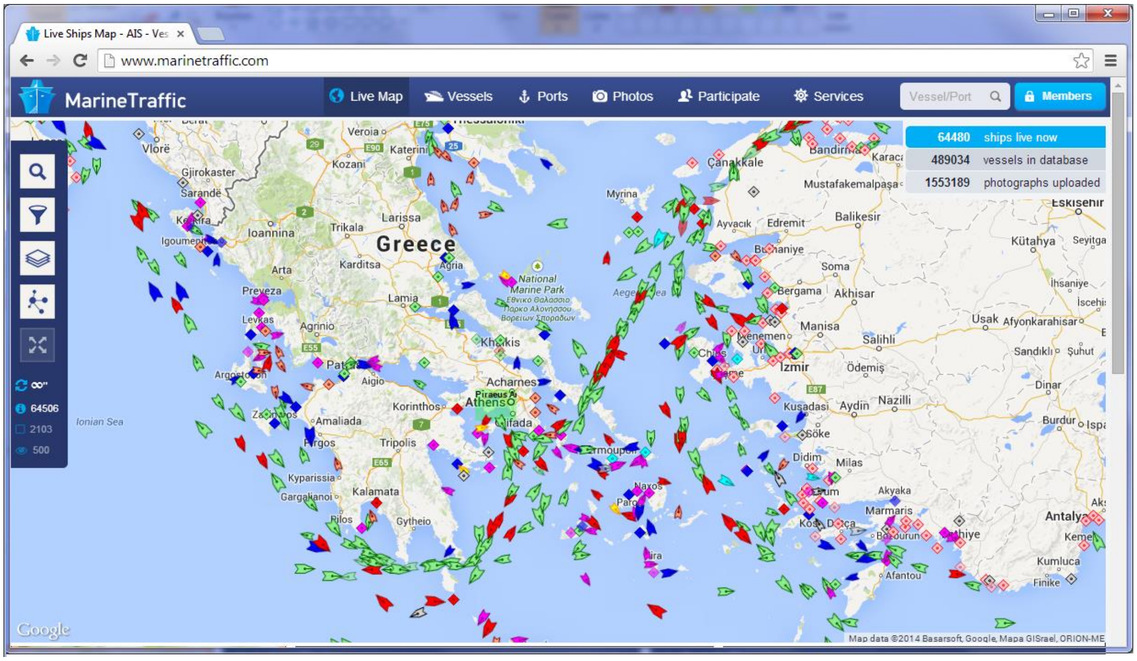


Figure 0.2: Marine Traffic AIS interface

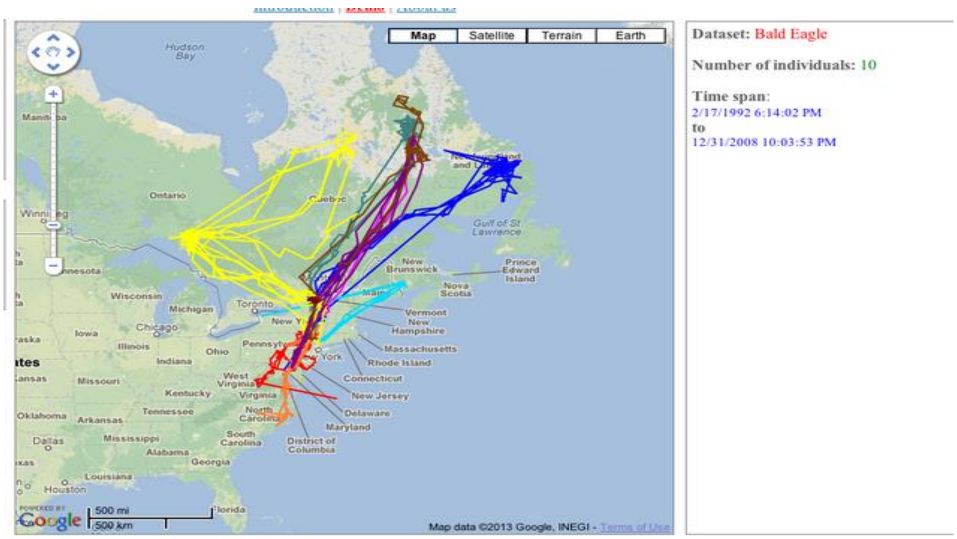
1.1 Related Work and Motivation

Other web-based data visualization tools using complex spatio-temporal data have been developed for other purposes. First, Lu et al. (2006) developed a web-based data visualization tool for traffic information in Washington D.C. The data visualization is only by means of 2D plots, and scatter plots. However, the web application lacked integration with any mapping and data mining component as will be developed for this thesis.

Second, Yawen et al. (2010) implemented a web-based spatio-temporal data visualization tool for visualizing Argo float data, sea surface temperatures, sea current fields, salinity, and in-situ investigation data. The floats' paths are represented as trajectories, and the sea flow field data are represented by arrows with size and direction. The tool does not enable users to perform temporal queries and the time visualization is static.

Third, Zheng (2013) implemented a desktop tool that display the characteristics of spatio-temporal trajectories (turn angles, length, duration) of marine mammals and boats, and the identification of high risk zones as density maps. The desktop application lacks effective presentation and visualization of the spatio-temporal datasets as the integration with web mapping APIs is not provided. Further, the tool does not implement sophisticated data mining and KDD algorithms.

The work done in this thesis is closely related to MoveMine (Li et al., 2010). MoveMine is a web-based system developed for processing, mining, and visualizing animal movement based on the raw location data of animals captured using GPS tracking system which was acquired from MoveBank.org. The web application provides an interface for visualizing movement data of various animals such as bald eagles, white pelicans, and artic terns in Google Maps and Google Earth. Along with the visualization of the movement data, Euclidean distance between selected objects is presented to users. Further, various data mining functions -- such as finding clustered movements using clustering algorithms, finding periodic movements, identifying density distribution of data points, and identifying relationships among animal movements by identifying groups of objects that move together -- are integrated into the web application. Figure 1.3 shows the web interface of MoveMine where a user has selected the path of ten Bald Eagles. Unlike MoveMine, the work done as part of this thesis focuses on mining historical AIS data based on PostGIS (Refractions Research Inc., 2012), extended with several custom functions to manipulate trajectories. Furthermore, the web application enables users to upload massive AIS data that can be mined and visualized.



Dataset: Bald Eagle

Number of individuals: 10

Time span:
2/17/1992 6:14:02 PM
to
12/31/2008 10:03:53 PM

Figure 0.3: Web interface of MoveMine

In the context of marine traffic management, several web-based visualization applications such as MarineTraffic (MarineTraffic, 2014), ShipFinder (Pinkfroot, 2014), and MyShipTracking (MyShipTracking, 2014) that track vessels based on AIS data and provide an interface for visualizing the real-time position of vessels exist. However, these applications are not capable of management, processing, analysis and visualizing of large maritime spatio-temporal datasets.

There is a lack of systems that may provide visualization and clustering techniques for large spatiotemporal datasets (Oliveira, 2012). Furthermore, one of the interesting research area opened for development of marine information systems relies in the management and analysis of the large and increasing amount maritime spatio-temporal datasets. The web application was developed as part of this thesis in order to address the problem of the lack of systems that provide visualization and clustering techniques for large maritime spatiotemporal datasets.

MarineTraffic (Lekkas et al. 2008) is an interactive and open web application that provides real-time geospatial information about vessel movement and port traffic based on Google Maps. This application exploits AIS data to map the location of vessels in real-time. In addition to the map visualization, it presents a list of ships that have recently arrived and of those that are expected to arrive for any given port. However, this application does not provide an interface to process past ship routes stored in the database and acquire critical information. However, a cost-effective processing and storage database, in order to preserve the history of vessels traffic for long periods is provided. This thesis exploits the massive collection of vessel position data acquired from MarineTraffic to develop a framework and data model for building an information system that will process and reconstruct trajectories, extract geometric properties from the reconstructed trajectories, compute turn angles, develop and disseminate heat maps, and visualize the trajectories, geometric properties, turn angles, and heat map on a web interface.

An additional tools is ShipFinder (Pinkfroot, 2014), which is a web mapping application similar to MarineTraffic that enables visualization of real-time ship locations, port arrivals and departures. Unlike MarineTraffic, in ShipFinder the coverage area is less, does not differentiate the vessels based on its type, and the history of vessel location data is not preserved. Similarly, MyShipTracking (MyShipTracking, 2014) is a web interface that provides real-time ship locations, port arrivals and departures. However, unlike MarineTraffic the history of vessel location data is not preserved.

1.2 Thesis Objectives

The purpose of this Master's thesis is to propose and develop an innovative method to store, analyze, mine, and disseminate useful information from trajectories, focusing on moving ships. Visualization is a powerful strategy for integrating high-level human intelligence and knowledge into the KDD process (Miller and Han, 2008). The objective of this project is to develop a visualization tool for the interactive presentation and interpretation of vessel movements from large spatio-temporal datasets generated by AIS. The other main objective research is to incorporate visualization techniques that support extraction of a portion of trajectories that intersect with the EU Natura 2000 protected areas. Natura 2000 is the centrepiece of EU nature and biodiversity policy (Natura 2000 network, 2014). It is an EU wide network of nature protection areas established under the 1992 Habitats Directive. The aim of the network is to assure the long-term survival of Europe's most valuable and threatened species and habitats. The detailed objectives are given below.

The following are objectives of the research project:

1. To develop a framework and data model for storing, analysing, mining, and visualizing massive collection of historical AIS data.
2. To implement data mining functions that exploits the historical AIS data.
3. To develop a web application that integrates the data mining functions and enables the results to be visualized in Google Maps.
4. Determine the potential negative environmental impact through spatio-temporal thematic maps.

1.2.1 Tasks

The following are the tasks that need to carried out to achieve the objectives:

1. Develop a web user interface to upload the raw AIS dataset in CSV format into a object-relational database with spatio-temporal capabilities.
2. Provide a methodology to process and reconstruct the raw records into vessel trajectories. Two types of trajectory reconstruction was incorporated, namely, detailed, and simplified using the Douglas-Puecker algorithm.
3. Extract geometric properties from trajectories, namely, travelled time, travelled distance, speed, and azimuth.
4. Develop a methodology to determine the locations where a ship has made sharp changes in direction by computing the turn angles at each time point. The

travelled time, travelled distance, speed, azimuth, and turn angle are presented for visualization as they represent the sailing habit/characteristics of vessels (Zheng, 2013).

5. Extract the information of ships that are in the vicinity of a ship. Identifying ships are in the vicinity of a ship at a given time point is important for analyzing risk and near-collision situations.
6. Identify ships that go through protected areas in the Aegean Sea, by comparing the trajectories and their patterns against EU Natura2000 protected areas (Figure 1.4).
7. Determine the outliers in the ships' position using an outlier detection algorithm.
8. Develop a web based user interface to facilitate disseminating the results of (b) to (h) through standard OGC web services (e.g, WMS, WFS, etc.) (Open Geospatial Consortium, 2014) and other web mapping libraries.

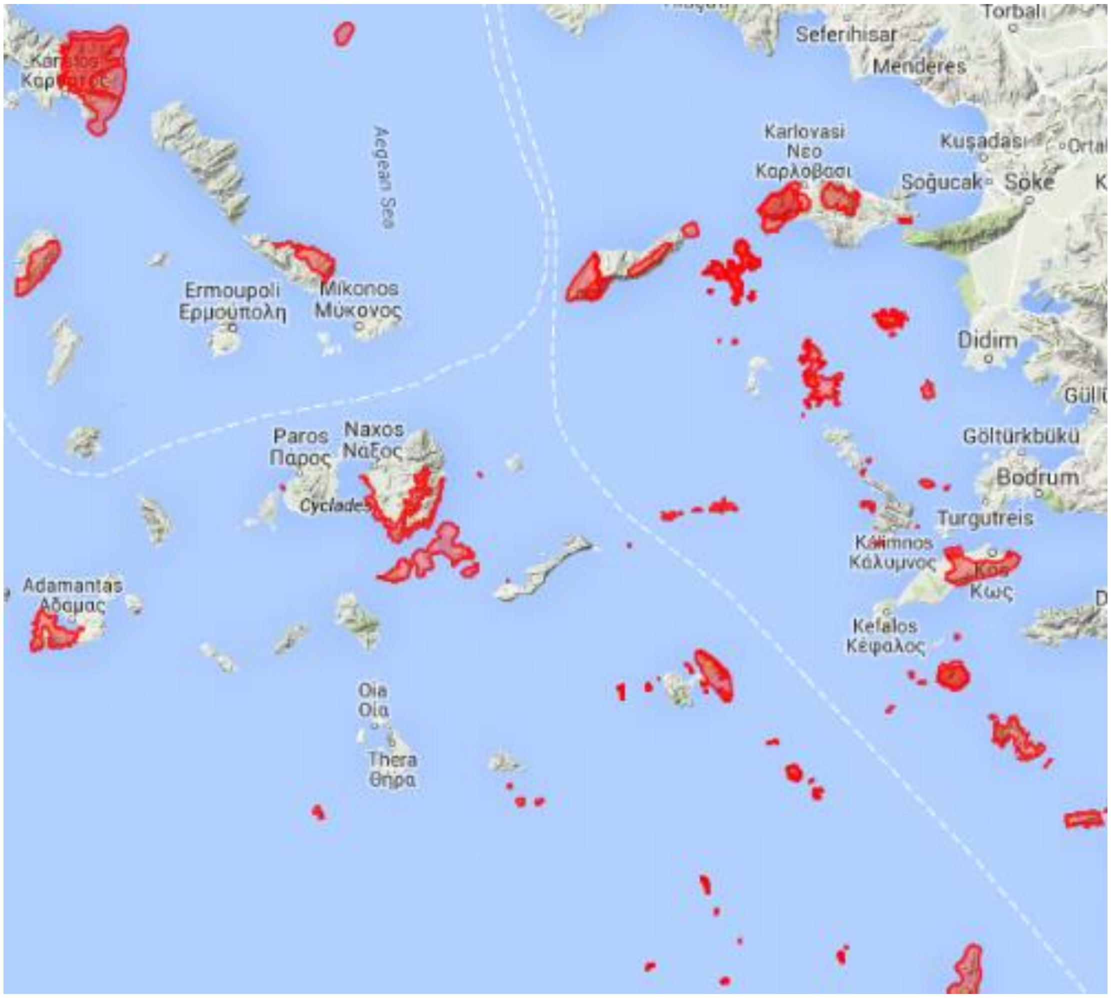


Figure 0.4: Sites of Community Importance (SCI) as defined by the European Union (Natura 2000) for Cyclades

1.3 Thesis Outline

This thesis is organized into six chapters. This chapter provides an introduction to the thesis and discusses topics such as the objectives, and the methodology used for the thesis. Further, this chapter discusses the web-based tools that exist in marine traffic management and how this thesis improves upon them. Finally, this section provides an outline of the thesis. Chapter 2 discusses the system architecture. More specifically, the tools and the underlying technologies employed in the server-side, and the client-side to achieve the objectives will be discussed. Chapter 3 discusses the process involved in designing, implementing, and populating the database. Further, the chapter discusses framework in terms of the data model used for the development of the web application. Chapter 4 discusses the methodology employed in order to accomplish the objectives of the thesis. The development process for the statistical analysis, and mining from the uploaded raw location data will also be discussed. More specifically, Chapter 4 covers topics such as the trajectory reconstruction, extraction of geometric properties from reconstructed trajectories, turn angle computation, and outlier detection. The results acquired as a result of the development process serves as the back-end data for the web application. Chapter 5 discusses in detail the design, and implementation of the web interface to visualize the results of the methodology used for the statistical analysis, and mining. This chapter will discuss the components of the Google Maps APIs and how it forms the basis for visualization. Further, the chapter discusses the advantages of integrating AJAX into the web application. Chapter 6 discusses the thesis outcomes and limitations. Recommendations for future research on this topic are also discussed.

CHAPTER 2 SYSTEM ARCHITECTURE

2.1 Introduction

This chapter discusses the system architecture employed to achieve the objectives. Furthermore, the underlying technologies are discussed in detail to illustrate the capabilities of the technologies for organizing and managing various types of spatial data. The system architecture used is a typical three-tier architecture proposed by Eckerson (Eckerson, 1995). The three-tier architecture is a client-server architecture that consists of the top most "Presentation Tier", the middle "Application Tier", and the bottom "Data Tier". The presentation tier consists of the user interface where the user services such as session management, text input, and display management reside and work together to disseminate the output from a client request. The user interface is typically accessed through a web browser using the HTTP protocol. In the system prototype proposed, the presentation tier is composed of HTML, CSS, JavaScript, AJAX, OpenLayers and Google Maps API. The application tier performs the business logic, executes queries, processes commands, and performs calculations. It also transfers requests from the browser to the data tier to read or write data. In the system prototype proposed, the components of the application tier are a web server, a web scripting language and a geoprocessing tool. The web server used is the Apache HTTP server and the scripting language used is PHP. The geoprocessing tool used is PyWPS, that facilitates access to GRASS modules via the web interface. The data tier is used for data management. The data management should support data storage, data retrieval, and concurrent access by multiple application tier processes, data backup, security and integrity of data. These functions are supported by a typical Database Management System (DBMS). In the system prototype proposed, the components comprising the data tier include a spatial DBMS, comprising of PostgreSQL, PostGIS and a Geographic Information System (GIS) server, GeoServer. Figure 2.1 illustrates the system architecture employed for this thesis.

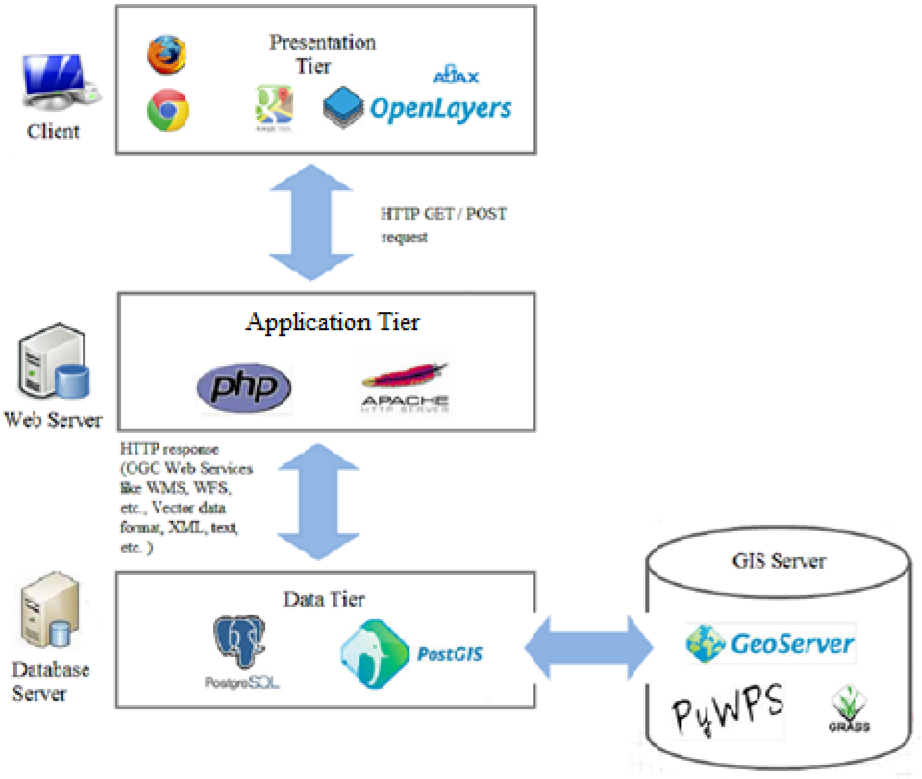


Figure 0.1: System architecture

2.2 Presentation Tier

The presentation tier is the top level of the application and contains the Graphical-User Interface (GUI) through which the dynamic nature of web applications is made possible. The presentation tier is used to disseminate the results of the knowledge discovery process of the trajectory data to the web browser running on the client system.

2.2.1 OpenLayers

OpenLayers (OpenLayers Dev Team, 2014) is a fast, high-performance, open-source JavaScript API used in web mapping applications. OpenLayers was initially developed by MetaCarta to provide an open-source alternative to Google Maps. Now OpenLayers is developed by the Open Source Geospatial Foundation under the 2-clause BSD license.

OpenLayers has no server-side dependencies and can render maps in web browsers. Without server-side dependencies, OpenLayers is capable of connecting to a wide variety of OGC web services such as WMS, WFS and WCS. OpenLayers is well integrated with GeoServer, and is capable of requesting and rendering the services implemented in GeoServer. The main advantage of using OpenLayers is that it can embed other publicly available web mapping applications such as Google and Bing Maps.

2.2.2 Google Maps API

Google Maps JavaScript Application Programming Interfaces, or APIs (Google, 2014), are a suite of tools that enable developers to use the functions available in Google Maps components to create custom mapping application using Google Maps as the base map. They are one of the most sophisticated AJAX-based web applications. These APIs contains classes, and objects that allow developers to create maps, provide map controls such as zoom control, pan control, scale control, street view control, rotate control, and overview control. Further, the APIs provides support to display several types of layers on top of the base map. Layers are one or more separate objects on the map, and are manipulated as a single unit. The types of layers include, KML layer, GeoJSON layer, heatmap layer (renders geographic data as a density map), and fusion tables layer. Each Map object contains a number of named events such as mouseout, mouseover, drag, mouseup, and mousedown. Based on the user events triggered, functions listening to these events are called and an action is executed. Google Map API provides support to display overlays to the map to demarcate points using markers, lines, polygon, and collections of objects. In addition to these features, Google Maps API provides web Services using the HTTP GET requests to Google services. The HTTP GET requests are made to specific URLs by passing parameters as arguments to these services. Table 2.1 shows the different types of web services that Google Maps API provide.

Table 2.1: Types of web services available in Google Map API

Service	**Description**
Directions API	Service that calculates routing directions between two different locations
Distance Matrix API	Service that computes the travel distance and time for a matrix of origins and destinations
Elevation API	Service that provides the elevation for a location (Latitude/Longitude pair) on the earth surface
Geocoding API	Service that provides the geographic coordinates from an address
Time Zone API	Service that provides the time offset data for a location (Latitude/Longitude pair) on the earth surface

2.2.3 AJAX

AJAX (Garrett, 2005) is Asynchronous JavaScript and XML. It encompasses several technologies that facilitate enhanced user experience. AJAX comprises of the following five technologies:

(a) front-end UI using XHTML and CSS

(b) dynamic web pages using the Document Object Model (DOM)

(c) data transport using XML or JSON

(d) asynchronous data retrieval using XMLHTTPRequest

(e) scripts using JavaScript that binds all components together

In traditional web applications, based on the HTTP request sent by the user, data is retrieved from the server, and processed on the server. The result is sent back to the user as an HTML page. However, in an AJAX model, there is an intermediate AJAX engine between the client and the server. A HTTP request that does not require data retrieval from the database, such as data validation, editing data in memory, and page navigation, is handled by the AJAX engine. If the AJAX engine requires data from the database such as retrieving new data, and data processing, the engine sends the request to the server asynchronously, using XML or JSON. This is done without halting the user interaction with the web application. This prevents having to reload an entire web page when a user request is made, and only the portion of the page when data is required from the server is updated. This enhances the dynamic nature and user experience of the web pages. Some of the applications that use AJAX include, Google Maps, Gmail, Facebook, and Flickr.

2.3 Application Tier

The application tier contains the business logic to construct trajectories from the raw AIS dataset, extract geometric properties from the trajectories, and for other analytical methods that were applied to these trajectories. It acts as an interface between the presentation tier and the data tier, and contains the data access logic that queries the data stored in the data tier. The requested results are returned to the client.

2.3.1 Web Server

A web server is a software application that provides the capability for storing, processing and delivering web pages to clients. The basic function of a web server is to receive requests from the web browsers and respond with a document from the server to the web browser (Peng & Tsuo, 2003). A web server, therefore, is the communication between client and server and this communication takes places through the Hypertext Transfer Protocol (HTTP) protocol. A typical web server needs to facilitate processing of various operations such as allowing users to submit data to the server through web forms, providing server-side scripting capabilities, uploading files to the server, data processing to retrieve information from a query, and building documents to the client.

Apache HTTP Server (Apache Software Foundation, 2014) is the web server used in the prototype. Apache HTTP Server is the product of the Apache Software Foundation, a community of developers developing open-source software applications. The Apache HTTP Server Project is a collaborative effort aimed at developing a free-available implementation of the HTTP web server. It is a flexible, highly configurable, extensible web server that can run on a wide variety of operating systems such as UNIX, Linux, Solaris, OS X, and Microsoft Windows. It provides modules for various operations such as authenticating users, supporting a wide variety of server-side scripting, including, Perl, Python, and PHP, compression methods, password authentication, and digital certificate authentication.

Compression methods help reduce the size of web pages when server over HTTP. Another important feature in Apache HTTP server is virtual hosting. Virtual hosting enables a single Apache HTTP Server installation to host multiple web sites.

2.3.2 PHP

PHP (PHP Group, 2013) is an open-source, server-side scripting language that allows rapid application development of dynamic web pages. It was originally developed by Rasmus Lerdorf in 1994. In typical three-tier architecture, PHP acts as the application tier. PHP can either be embedded in HTML pages or it can be used with web frameworks. Using web frameworks for development allows the developer to produce fast, reliable, secure, and customizable web pages at a reduced development time and cost. When a client makes a request, a PHP-based website can quickly connect to the backend database, run a function, process the result, and return the result back to the client. One important feature in PHP is the built-in flexibility that enables developers to execute an external program from within PHP and inject the result back into the web application. For example, a Python Web Processing Service (PyWPS) script could be executed using PHP and the resulting output could be sent back to the client.

The ease of development, availability of object-oriented features and integration with Apache web server made PHP a natural choice as the server-side scripting language. Functions responsible for communication between server and client side components, providing file upload feature for end-users, retrieving information from the database as a result of spatial operations were written using PHP and deployed on the web server.

2.4 Data Tier

The data tier is used to store the raw AIS dataset in an object-relational data structure. The data tier receives the retrieval and data processing requests using a query language from the middle or business tier. The query is executed, and data pertaining to the query is returned back to the middle tier. The data tier in web-GIS has capabilities to store, and process spatio-temporal data.

2.4.1 PostgreSQL

To effectively store, analyze and manipulate spatial data such as the location and heading of the vessels, a spatial database is needed. Traditional relational database management systems in the form currently present are not very well suited for handling large spatio-temporal data. This is because of the inherent nature of spatio-temporal data. Traditional relational databases support basic data types such as strings and integers, whereas spatial databases must handle complex data types such as points, lines and polygons. Although several spatial database implementations exist, some are open-source such as MySQL, others are proprietary such as Oracle Spatial and Microsoft SQL Server, only Free and Open Source Software (FOSS) was considered to develop the web application. PostgreSQL was chosen to serve as the back-end database for the web application.

PostgreSQL (PostgreSQL Global Development Group, 2013) is an object-relational database management system that was developed from the Ingres project at the University of California, Berkley during the early 1980's. PostgreSQL is the most advanced open source database system available and its speed and functionality competes with any proprietary enterprise database system. It uses the Spaghetti Model (Rigaux et al., 2001) for modeling spatial data objects. It includes several geometric data types to represent two-dimensional spatial objects. Apart from the standard SQL functions, PostgreSQL provides functions for producing XML (Extensible Markup Language) and JSON (JavaScript Object Notation) from SQL data. XML and JSON are the most used format for transporting data between different systems. Table 2.2 shows the geometry data types supported in PostgreSQL.

Table 2.2: Geometric types in PostgreSQL

Geometric Type	**Representation**	**Description**
Point	Point on a plane	(x,y)
Line	Infinite line	((x1,y1),(x2,y2))
Box	Rectangular box	((x1,y1),(x2,y2))
Path	Closed path	((x1,y1),...)
Path	Open path	[(x1,y1),...]
Polygon	Similar to closed path	((x1,y1),...)
Circle	Circle	<(x,y),r> (center point)

One advantage of PostgresSQL is that it uses a balanced, tree structure for indexing called GiST (Generalized Search Tree) that allows users to develop custom data types in addition to the data types provided. One example of this is PostGIS, the PostgreSQL spatial extension that consists of extensive spatial data types. GiST is used to speed up the search queries on different types of data. A table containing all the entries for a typical data type would consist of several thousand rows, and building an index on the table would speed up the spatial queries of the data. Although PostgreSQL has geometric data types, these types are very limited for storing and analyzing complex GIS data.

2.4.2 PostGIS

PostGIS (Refractions Research Inc, 2012) is a spatial extension of PostgreSQL object-relational database that facilitates spatial objects to be stored, queried and analyzed in the database. It is an open-source project developed as an extension for PostgreSQL by Refractions Research under the GNU General public license. PostGIS follows the specifications from the Open Geospatial Consortium (OGC) called Simple Features for SQL (SFSQL). SFSQL provide the specifications for the SQL routines that could be performed on Simple Features such as point, line, polygon, and multi-point.

2.4.2.1 Data Types

There are two main data types by which spatial objects are stored in PostGIS – Geometry and Geography. The "geometry" type is the most widely used, as this can store spatial data in any projection and Coordinate Reference Systems (CRS). The "geography" data type was recently introduced in version 1.5. In the geography data type, all coordinates are stored in WGS84 CRS (longitude/latitude coordinates) and only a few functions are available for it. For data covering a small region, choosing the right projection and storing the data in geometry type is the ideal solution as it offers better performance and a greater number of available spatial functions. For global or data covering large areas, storing the data as a geography type without specifying the projection system would be an ideal solution. Here, all data are stored in latitude/longitude coordinates.

2.4.3 GeoServer

GeoServer (OpenPlans, 2014) is an open-source Java based GIS server used to publish geospatial content. GeoServer is analogous to the Apache HTTP Server Project. With Apache HTTP Server, one can publish HTML web pages. With GeoServer, one can publish geospatial information. The Open Planning Project (TOPP) conceived GeoServer in 2001 to facilitate citizen participation in government planning and decision making, thereby helping make the government more transparent. GeoServer is built on top of GeoTools (GeoTools, 2014), a set of Java APIs for analyzing, querying and manipulating geospatial content. GeoServer facilitates users to insert, update and delete geographic data. Using desktop and web-based client side tools like Quantum GIS, uDig and OpenLayers, GeoServer allows users to serve maps and data to these client tools.

GeoServer implements the main OGC's web services – Web Map Service (WMS) version 1.1.1 and 1.3.0, Web Feature Service (WFS) version 1.0.0, 1.1.0 and 2.0, Web Coverage Service (WCS) version 1.0.0 and 1.1.1 and Web Processing Service (WPS) version 1.0.0. It supports various output formats such as PNG, JPEG, TIFF, SVG, GeoTIFF, PDF, KML GML, GeoJSON, GeoRSS and shapefiles. Mainly designed for interoperability, GeoServer is capable of publishing data from various sources such as raster (e.g., GeoTIFF, WorldImage, GTOPO30, ArcGRID, Oracle Georaster, PostGIS raster, and GDAL image formats), vector (e.g., ESRI Shapefile, and GML) and database (e.g., PostGIS, MySQL, Oracle, and Microsoft SQL Server). GeoServer provides a REST (REpresentational State Transfer) API by which users can read, insert, update and delete through HTTP methods – GET (to read data), POST (to insert data), PUT (to update data) and DELETE (to delete data).

2.4.4 PyWPS

Python Web Processing Service (HS-RS, 2014) is an open-source Python based implementation of the OGC's Web Processing Service (WPS) developed in 2006. PyWPS was originally developed to be used on Linux based environments. However, recent versions can be used in the Windows environment as well. PyWPS is a translator-proxy application between client (web browser, desktop GIS, command line tool, etc.) and working tool installed on the server (Cepicky, 2009). PyWPS provides support for executing GRASS GIS operations. So, PyWPS enables access to all GRASS modules via the Web. By executing PyWPS scripts via PHP, users can have access to WPS geoprocessing operations through a web browser. In addition to offering support for running GRASS GIS scripts, PyWPS also supports other libraries such as R, GDAL, and Proj4.

2.5 Summary

This chapter overviewed the system architecture to perform statistical analysis and mining of raw location points of vessels. The architecture consists of three major parts; namely, the presentation, application, and data tiers. Statistical analysis and mining functions utilize the spatial data stored in PostgreSQL / PostGIS database. GeoServer was used to publish WMS documents as services on the Web. The Google Maps JavaScript API was used to develop the web application for visualization of the knowledge discovery process elaborated in this thesis. The architecture is completely open-source, and thus facilitates flexibility and reusability. In addition, this architecture provides a framework for further research and development.

CHAPTER 3 DATABASE DESIGN

3.1 Introduction

Database design consists of the following: (i) defining the database structure that is used to store end-user data by determining the entities, the relationships among the entities, and the constraints on the entities; (ii) transforming the database structure into a database schema by creating tables, columns, primary keys, foreign keys, and constraints; and (iii) populating the database schema with real-world data. Navethe (Navethe, 1992) describes the structure of a database as the data types, relationships, and constraints that define the "template" of that database. A data model should provide operations on the database that allow retrievals and updates including insertions, deletions, and modifications (Navethe, 1992). A data model represents an abstraction of complex real-world data. The data models are partitioned into three levels of abstraction, namely: conceptual, logical, and physical (Elmashri, 1989). This chapter will describe the conceptual, logical, and physical database design to build a GIS database for representing, storing, and querying historical AIS data. Each phase taken together creates the corresponding data model.

3.2 Conceptual Design

The conceptual design of the database is the first phase in the database design process and it involves creating the conceptual data model. The first step in conceptual database design is data analysis and requirements gathering. Based on the application needs, a conceptual data model, which represents the entities, the attributes for each entity, the relationships among these entities, and the integrity constraints (rules) that are required for the application is constructed. A conceptual data model can be described using a natural language, such as the English language, or an advance data modeling language such as an entity-relationship model (Chen, 1976) and Unified Modeling Language (Booch et al., 1996). The goal of the conceptual database design is to develop a model which is independent of any physical system such as software, programming language, application program, and system hardware. A well designed conceptual data model is important for the design of the subsequent models – logical and physical. An abstract data model that represents the real world details is created in this stage. The database implementation details are not governed in this stage.

The first aspect in the conceptual data model deals with entities. The geospatial data entity types could be any of the geometric data types such as point, line, polygon, multipoint, multiline, etc. These entities could have spatial relationships (topology) and these relationships must be computed. The second aspect of the conceptual data model deals with the relationships between the entities. The relationships are a set of rules that govern how spatial data objects are stored in the database, and these rules help create data with greater integrity. The third aspect of the conceptual data model deals with the integrity constraints (rules). The integrity constraints provide a mechanism to prevent loss in data consistency when changes are made to the database. The most common types of constraints include UNIQUE constraints (ensures that a given column is unique), NOT NULL constraints (ensures that no null values are allowed), FOREIGN KEY constraints (ensures that two keys share a primary key to foreign key relationship). These constraints prevent introduction of redundant data and are useful in query optimization.

3.2.1 Unified Modeling Language (UML)

The conceptual data model is built using UML. UML was designed as a standard, unifying modeling language that uses object-oriented techniques for documenting the components in software applications (Marcos et al., 2003). UML was originally developed for modeling classes in object-oriented applications. Being an extensible language, UML is the best choice in GIS applications which deal with complex spatial objects. The Data Model Profile is an UML extension that supports modeling databases in UML. The purpose of the Data Model Profile is to show the entities being modeled in the system. Modeling the database helps to understand how the tables are structured and how the tables are related in a particular schema. Using the Data Model Profile, the tables, columns, data base schema, table keys, triggers, constraints, indexes, stored procedures, stored functions and relationships are modeled effectively.

Table 3.1 shows how the database elements are translated into an UML Data Model Profile. A table in an UML data model is modeled as a class with a table icon on the top right hand corner. The database columns are modeled as attributes of the table class. The data type associated with the column is also indicated. A column can be a key or a non-key column. A key column can be a primary, foreign, or a combination of both primary and foreign. The key associated with a column should be indicated. A primary key uniquely identifies each record in a table and a foreign key is a column wherein it is a primary key in a parent table and indicates the relationship between the parent and child table. The relationship among the entities should be indicated. The relationship can be identifying or non-identifying. The relationship is termed identifying if the child foreign key includes all the values of the parent primary key and non-identifying if only some values of the primary key are included.

Table 3.1: Database elements and their appropriate icons in an UML Data Model Profile

Database Element	**UML icon**
Table	⊞
Primary Key	PK
Foreign Key	FK
Primary/Foreign Key	$^{P}_{f}$K
Identifying Relationship	0..* 1 ———
Non-identifying Relationship	———◆ 1..* 1

The behavior associated with columns such as indexes, keys, triggers and procedures should also be described. Behavior is represented as stereotyped operations. Figure 3.1 shows an example of a UML diagram where a Customer entity having the attributes, namely, "OID, "Name", and "Address" is modeled. Here, the PK flag on the OID column indicates OID is a primary key, while the stereotyped operation «PK» idx_customer00 defines the behavior of the primary key, i.e., the idx_customer00 is the name of the primary key definition in the database. Similarly, additional behavior such as triggers, unique constraints and stored procedures are indicated below the attributes.

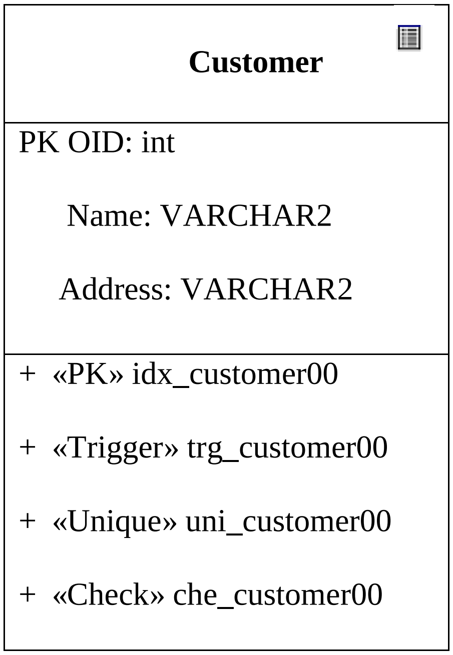


Figure 3.1: An example of a UML diagram

Figure 3.2 shows the UML Data Model Profile developed for storing the following: (i) the raw AIS dataset acquired from the Marine Traffic project, (ii) the EU Natura2000 protected areas, (iii) the ship types (e.g, tanker, passenger, etc.), and (iv) other extracted data from the raw AIS dataset. The `LOCATION_SHIPS` class represents a record in the AIS dataset. It is characterized by having the following attributes, namely, MMSI identifier, status, station, the geographical coordinates (latitude-longitude), course, heading, ship type (foreign key that references the `ship_type_id` in `ship_type` class), and the time at which the position was recorded. Further, `LOCATION_SHIPS` class encapsulates all the methods needed for the knowledge discovery process such as `DistanceAndSpeedCalc`, `DirectionCalc`, `TurnAngleCalc`, and

ProtectedAreasIntersection. The ship_type class contains the attributes needed to store the id of the type of ship and its description. The PROTECTED_AREAS class represents stores the names and the polygon geometries of the EU Natura 2000 protected areas. The route_ships class diagram to represents the extracted detailed ship trajectories. It is characterized by having the MMSI identifier, the start time of the trajectory, the end time of the trajectory, and the geometry of the trajectory typically represented as a PostGIS LineString. Similarly, the route_simplified class diagram represents the extracted simplified ship trajectories.

The extracted geometric properties from trajectories, namely travelled time, travelled distance, speed, and azimuth are represented through distanceandspeed and direction classes respectively. The distanceandspeed class is characterized by having information about the MMSI identifier, the latitude-longitude coordinates, the time when the position was recorded, the distance between two time points, the total distance, the speed calculated between two time points, the average speed along the entire trajectory. The direction class is characterized by having information about the MMSI identifier, the latitude- longitude coordinates, the time when the position was recorded, the direction of travel, and the major direction along the entire trajectory. The turnangle class contains information about the turn angles made the ship at each turn. The protected_area_intersection class contains information about the MMSI identifier, and its trajectory that intersect the protected areas in the Aegean Sea. The outlier class encapsulates the attributes that are needed to detect outliers in the trajectory dataset. Applying an outlier detection algorithm, the result is stored in the `cluster` attribute. If a recorded position is an outlier, the value of the `cluster` attribute is `noise`, else the value of the cluster attribute is the cluster group that to which the recorded position belongs.

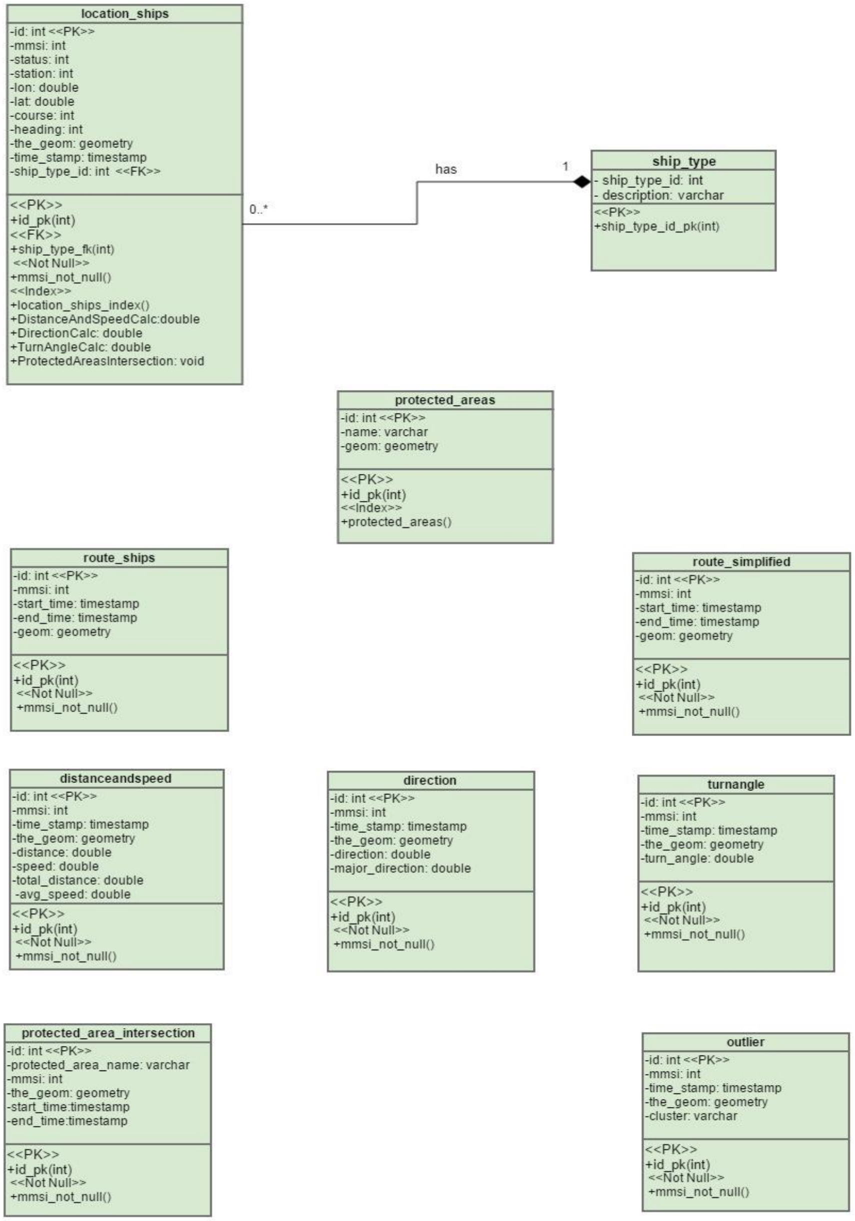


Figure 0.2: UML Data Model Profile that represents the database schema

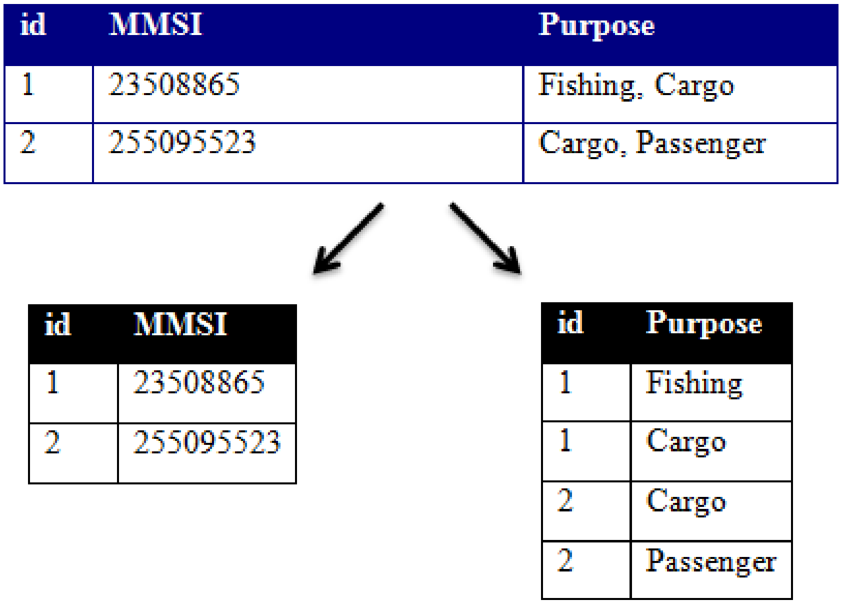
3.3 Logical Database Design

The logical database design is the second step in the database design process. Once the structure of the conceptual data model is identified, the logical data model is built from this structure. The conceptual data model identifies the data contents of the database, whereas the logical data model helps define the logical structure of the data that is stored in the database. The goal of the logical design is to develop an enterprise level mode independent of the physical details. It closely models the structure of the data that a specific database displays to the user. The logical data model represents the tables, columns, relationships, primary keys, foreign keys, constraints, etc. as available of the DBMS used in the application. These form a relational data model, where the basic unit of this data model is the table.

3.3.1 Transformation from Conceptual Model to Logical Model

The transformation from conceptual model to logical done is done by using as input the UML diagram, which is the result of the conceptual database design phase. This transformation produces another UML diagram which represents the relational model of the database used. The following are the steps used to transform a conceptual model to a logical model:

1. Each class in the UML diagram is transformed into a table.
2. For each table generated as result of Step 1, a column is created and assigned as the primary key. Each element in this primary key column will be assigned a unique number in order to uniquely identify each record in the table.
3. Each attribute in a class is transformed into a column. For simple valued attributes, the attribute is transformed into one column, and the data type for the column is assigned to be that of the data type of the class attribute. However, if the exact data type is not available in the implementation of the DBMS used in the application, an appropriate data type available in the implementation is chosen. For multivalued attributes, a new table is created with a foreign key, the primary key of the table containing this attribute. Figure 3.3 shows how to represent multivalued attributes. Here, to represent the multivalued attribute "purpose", a new table is created and joined via the same "id" primary key.
4. The relationships among classes are transformed into tables using a primary-foreign key pair. For classes having a one-to-one relationship, the attributes corresponding to both classes are placed in a single table, hence creating a single table out of two classes. For classes having a one-to-many relationship, a relationship is created with a foreign key set on the table in the "many" side of the relationship and the primary key set on the table in the "one" side of the relationship. For classes having a many-to-many relationship, a separate table having a composite primary key composed of the primary key of the two tables is created. The newly created table is linked to the other two tables by making the primary keys as foreign keys to the other two tables.



id	MMSI	Purpose
1	23508865	Fishing, Cargo
2	255095523	Cargo, Passenger

id	MMSI
1	23508865
2	255095523

id	Purpose
1	Fishing
1	Cargo
2	Cargo
2	Passenger

Figure 0.3: Mapping a conceptual multivalued attribute to a logical model

In addition to the tables, columns and relationships, the logical model needs to contain details about the indexes, stored procedures, triggers, unique constraints and check constraints that will be created in the database.

3.4 Physical Database Design

In the physical database design, the logical data model created during the logical database design phase is implemented in the database using Structured Query Language (SQL) statements such as Data Definition Language (DDL) statements, which are used to define the database schema, and Data Manipulation Language (DML) statements, which are used to read and write data into the database. Further, to ensure better performance and enhance storage space, fine tuning the logical data model is performed during this stage. For example, by creating indexes on a database the speed of the search queries are improved. Physical database design largely depends on the hardware and software such as the DBMS used to build the system. The decisions made during the physical database design stage affect the speed of the database, the accessibility of the database, the security implemented on the database and the user-friendliness of the database (McKearney, 2003).

Physical database design consists of the following steps:

i) Database schema implementation

ii) Populating the database

3.4.1 Database Schema Implementation

Database schema implementation consists of loading the schema into the database management system. This includes creating a spatial database, the tables and columns necessary for the application. The database management system (DBMS) used for this visualization application is PostgreSQL 9.1 with the PostGIS 2.1 extension.

PostgreSQL stores both spatial and non-spatial data in the same schema. This allows interaction between spatial and non-spatial data. PostgreSQL allows spatial data to be stored in columns of type GEOMETRY. To be able to use PostGIS in PostgreSQL, a spatial database has to be created using "template_postgis" template. This template allows the database created to load the PostGIS spatial datatypes and functions. pgAdmin (pgAdmin Development Team, 2013) was used for the schema implementation. pgAdmin is a open-source Graphical User Interface (GUI) used for administration, management and development of the PostgreSQL database for Unix and Windows systems. It is freely available under the terms of the PostgreSQL license and is managed by the pgAdmin Development Team. pgAdmin is used to connect to the PostgreSQL hosted on gaia.gge.unb.ca at port 5432. Appendix A.1 shows the SQL command for creating a spatial database in PostgreSQL. Here, a database schema called MovingObjectDB is created. The template used here is called template_postgis_20, which is the template available in 2.1 the extension. The tablespace used is the default called pg_default. Tablespace is the physical location in the disk where the database schema and values are stored.

Once the MovingObjectDB database is created, a new schema called "Topology" is created. The Topology schema was introduced in version 2.0 and it contains tables and functions necessary to manage and process topological features such as faces, edges and nodes. Topology is needed to support topological integrity, reduce the storage space, define explicit spatial relationships and have a normalized data structure. In addition to the creation of topology schema, two tables called spatial_ref_sys and geometry_columns are created in the public schema. The spatial_ref_sys table consists of details of over 3000 spatial reference systems that can be used in the database. Custom projections can also be added into this table using PROJ.4 constructs (Evenden & Warmerdam, 1999). PROJ.4 is a cartographic projections library which is used for transformation of geographic coordinates from one projection to another. The geometry_columns consists of details of the tables that hold geometries, and columns that hold the type of geometries such as point, line, polygon, etc.

After creating the database, the tables in the UML class diagram that were developed during the logical database design phase must be transformed to the database schema in PostgreSQL. Three such database schema called LOCATION_SHIPS, SHIP_TYPE and PROTECTED_AREAS were created to store the details of the vessel locations, vessel type, and the protected areas respectively. Each table's primary key is of SERIAL data type. SERIAL data type provides an automatically increasing or decreasing unique identifier by creating a sequence. So, when each record is created in the database, the primary key value need not be set explicitly and PostgreSQL will assign a new unique sequence number.

After the schema creation, the relationship between LOCATION_SHIPS and SHIP_TYPE is physically enforced by creating a FOREIGN KEY constraint. Again, a foreign key is a column that creates a link between data in two tables (one-to-many relationship). Appendix A.2 shows the creation of a foreign key constraint between

LOCATION_SHIPS and SHIP_TYPE table. Here, the SHIP_TYPE column in the LOCATION_SHIPS table references the SHIP_TYPE_ID column in the SHIP_TYPE table.

After the relationships are established, the final step in the creation of the database schema is to create indexes on the columns. By default, the primary key column is indexed. Indexes are mainly created to increase database performance by increasing the speed of database operations. The main advantage of creating indexes is that the speed of the search operations on the database is vastly increased. Without using indexes, search queries would require a sequential scan of every record in the table. Using indexes, the data in a table gets organized into a search tree: traversing in a tree data structure is faster, and hence indexes improve the performance of search queries. However, the major drawback in using indexes is that write operations such as insert, update and delete are slowed down. However, the advantages for this application far outweigh the drawbacks and indexes are used extensively for performance improvement in a database.

Indexes are typically created on columns that will the used for most selection queries. Most often, a number of columns will often be used together in search queries. In that scenario, creating indexes on multiple columns is the best approach. Appendix A.3 shows the SQL command to create an index on a column. Here, an index on mmsi and geom columns are created on the table LOCATION_SHIPS. The reason for choosing to index on multiple columns is that the majority of the searches and spatial queries will be specific to the unique identification number of a vessel (mmsi) and its location (geom). Data types that are sorted along one axis -- such as integers, strings and dates -- are indexed using B-trees and GiST indexes break up data into "things to one side", "things which overlap", "things which are inside", and is used to index geometric data types. Apart from creating an index on columns most frequently used in search queries, creating an index on foreign keys improves the speed of joins between relations.

Once the schema is created, the spatial component as a geometric column has to be added to the `LOCATION_SHIPS` and `PROTECTED_AREAS` table as these are normal non-spatial tables. The position of the vessels has to be added as a point data in `LOCATION_SHIPS` schema as no geometric column has been included. For this, PostGIS provides a function called `AddGeometryColumn`. This function adds a spatial column of geometry datatype to a table. This function takes as parameters the name of the table into which the geometric column has to be added, the European Petroleum Survey Group (EPSG) ID of the spatial reference system, the type of geometry, and the number of dimensions in the geometry data type.

3.4.2 Populating the Database

The last step in the physical design is to load the data into the database schema. The location of the vessels is loaded into the `LOCATION_SHIPS` database schema. The web application facilitates the end-user to populate the vessels' location by uploading a Comma-Separated Values (CSV) file containing details about the vessels' MMSI, location, heading, etc. Figure 3.4 shows the screenshot of the tool which allows uploading CSV files and populating the LOCATION_SHIPS schema.

Facilitating the uploading of files from the browser to the server requires that three attributes in the HTML <form> tag have to be set, namely: (1) enctype attribute, which is set to multipart/form-data; (2) method attribute, which is set to POST, as the data has to be summited to the server to be processes; and (3) action attribute, which is set to the PHP script that will process the form. The enctype attribute determines the way in which the form data is encoded by the browser. For files, the enctype is set to multipart/form-data. In addition to the three attributes, the form's input type has to be set to file. Clicking this input type would let the browser to enter into a file selection mode and user's local directory is displayed, letting the user to select the file from the different directories.

Once the file has been selected and submitted to the server, the information about the uploaded file information is stored in $_FILES array in PHP. $_FILES is a two-dimensional array that contains the form's input name as the first index and the file's attribute such as name type, size, temporary filename and error code as the second index. Once the file is sent to the server, only a temporary copy of the uploaded file would exist on the server and the file would be deleted once the script ends. So it is moved and stored in a permanent location for further processing and for populating the database using the move_uploaded_file() function, which takes two parameters, namely, the temporary filename of the file uploaded and the original name of the file on the client machine. An important feature of this function is that it will ensure that the file passed in the input parameter is the actual file uploaded via PHP's HTTP POST method. To prevent the server from being filled and for security purposes, the application is designed to accept only CSV files. The PHP script restricts only files of CSV format to be uploaded. Appendix A.4 shows the PHP script for a file upload that checks whether the uploaded file is in CSV format and moves it to a permanent location.

Once the file has been uploaded, the file is opened for reading and a filehandle is returned that will allow to access the file using PHP's `fopen()` function. Once the CSV file is read its data is accessed using the `fgetcsv()` function in PHP by passing in two parameters, namely, the filehandle that is returned from the `fopen()` function, total number of lines to be read, and a delimiter. This function splits up the lines of the file by a delimiting character. The CSV file has a comma (,) as a delimiting character and hence this is used as the delimiter in the `fgetcsv()` function. Each line will return an array as the result. The elements of the array are passed into the SQL INSERT DDL statement and this INSERT statement is executed using the PHP's `pg_query()` function. The `pg_query` function is used to execute SQL statements on the database connected. The algorithm for populating the `LOCATION_SHIPS` schema from the CSV file having the location information of the vessels is given below.

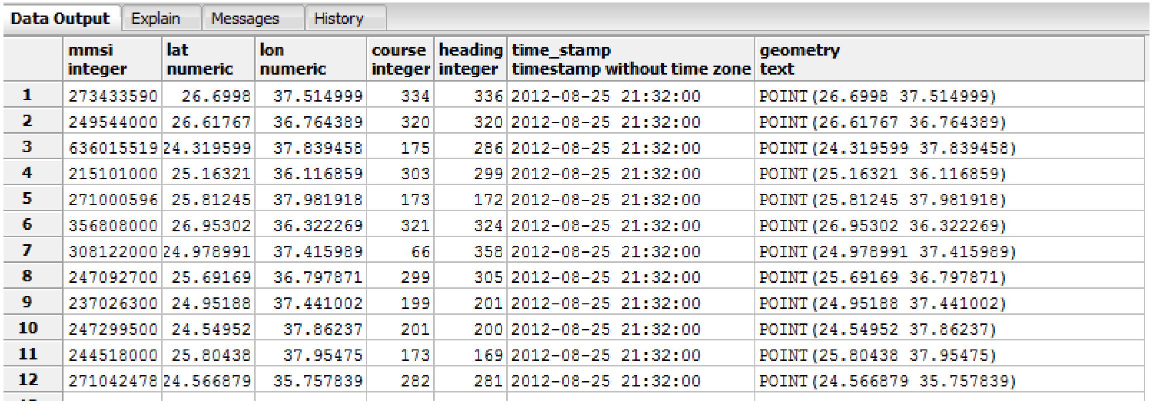
- Read the CSV file

- **While** the End of File (EOF) is not found, parse the CSV file by calling the function **fgetcsv** using the delimiter ',' and set it to a **line** array variable

```
        If  line is empty
            Break
        Else
            - Call the Insert DML statement by passing the elements of the line
            array and set it to query string variable.
            - Call the pg_query function by passing the query string variable
        End
    End
```

The `ST_GeomFromText()` PostGIS function is used to convert a Well Known Text (WKT) version of a geometry into a PostGIS geometry. This function is used to populate the geometry column while inserting the data from each line of the CSV into the database. Appendix A.5 shows the PHP script for reading and inserting data into `LOCATION_SHIPS` schema. Here, the file is opened, read and inserted into the `LOCATION_SHIPS` schema. A WKT representation of a point in WGS 84 format is inserted into a PostGIS geometry field using `ST_GeomFromText()` function.

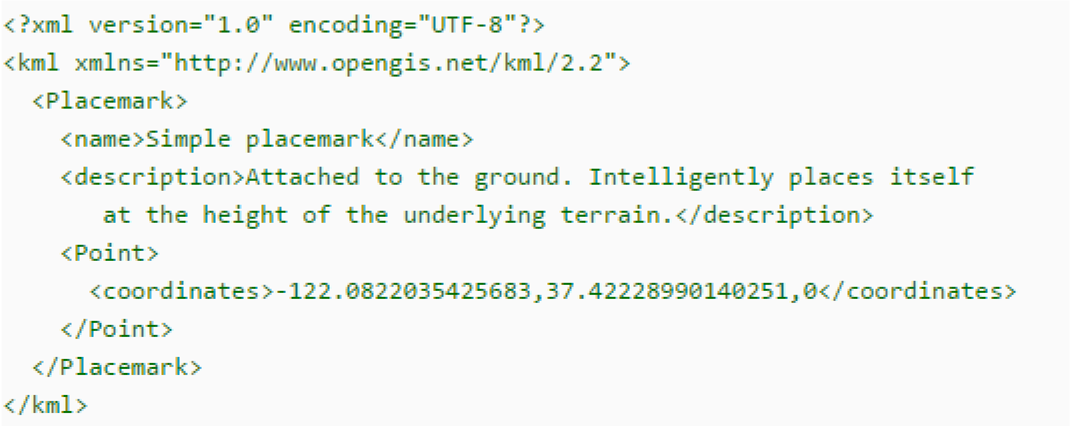
Figure 3.4 shows the snapshot of the populated `LOCATION_SHIPS` schema from the CSV file as a result of uploading the CSV file into the PostgreSQL database.



	mmsi	lat	lon	course	heading	time_stamp	geometry
	integer	numeric	numeric	integer	integer	timestamp without time zone	text
1	273433590	26.6998	37.514999	334	336	2012-08-25 21:32:00	POINT(26.6998 37.514999)
2	249544000	26.61767	36.764389	320	320	2012-08-25 21:32:00	POINT(26.61767 36.764389)
3	636015519	24.319599	37.839458	175	286	2012-08-25 21:32:00	POINT(24.319599 37.839458)
4	215101000	25.16321	36.116859	303	299	2012-08-25 21:32:00	POINT(25.16321 36.116859)
5	271000596	25.81245	37.981918	173	172	2012-08-25 21:32:00	POINT(25.81245 37.981918)
6	356808000	26.95302	36.322269	321	324	2012-08-25 21:32:00	POINT(26.95302 36.322269)
7	308122000	24.978991	37.415989	66	358	2012-08-25 21:32:00	POINT(24.978991 37.415989)
8	247092700	25.69169	36.797871	299	305	2012-08-25 21:32:00	POINT(25.69169 36.797871)
9	237026300	24.95188	37.441002	199	201	2012-08-25 21:32:00	POINT(24.95188 37.441002)
10	247299500	24.54952	37.86237	201	200	2012-08-25 21:32:00	POINT(24.54952 37.86237)
11	244518000	25.80438	37.95475	173	169	2012-08-25 21:32:00	POINT(25.80438 37.95475)
12	271042478	24.566879	35.757839	282	281	2012-08-25 21:32:00	POINT(24.566879 35.757839)

Figure 0.4: Populated `LOCATION_SHIPS` schema

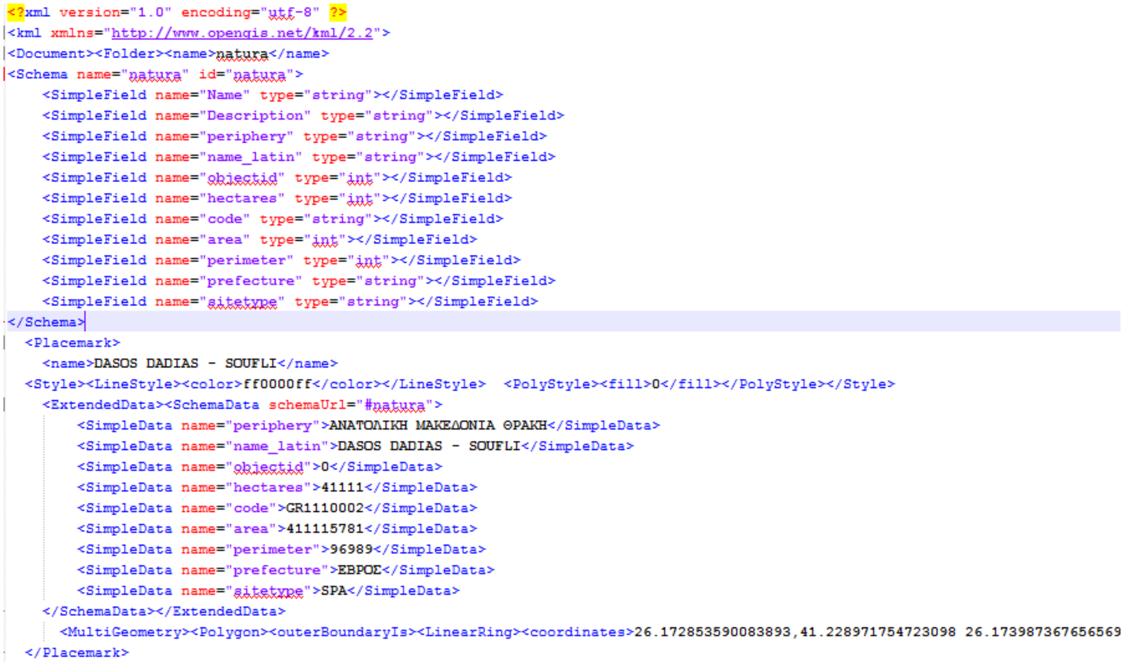
After populating the `LOCATION_SHIPS` schema, the `PROTECTED_AREAS` schema is populated. The input for populating `PROTECTED_AREAS` is expressed in Keyhole Markup Language (KML) format acquired from the European Union's Natura 2000 network. KML (Google, 2013) was originally developed by Keyhole Incorporated and acquired by Google in 2004. KML is an XML variant that describes elements for storing geodata. It is primarily used to display geographic data in applications such as Google Earth and Google Maps. KML is based on an object structure and each object may contain several elements. Figure 3.5 shows a simple KML file. Here, placemark is an object and the elements for this placemark object are name, description and point.



```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Placemark>
    <name>Simple placemark</name>
    <description>Attached to the ground. Intelligently places itself
       at the height of the underlying terrain.</description>
    <Point>
      <coordinates>-122.0822035425683,37.42228990140251,0</coordinates>
    </Point>
  </Placemark>
</kml>
```

Figure 0.5: A KML file

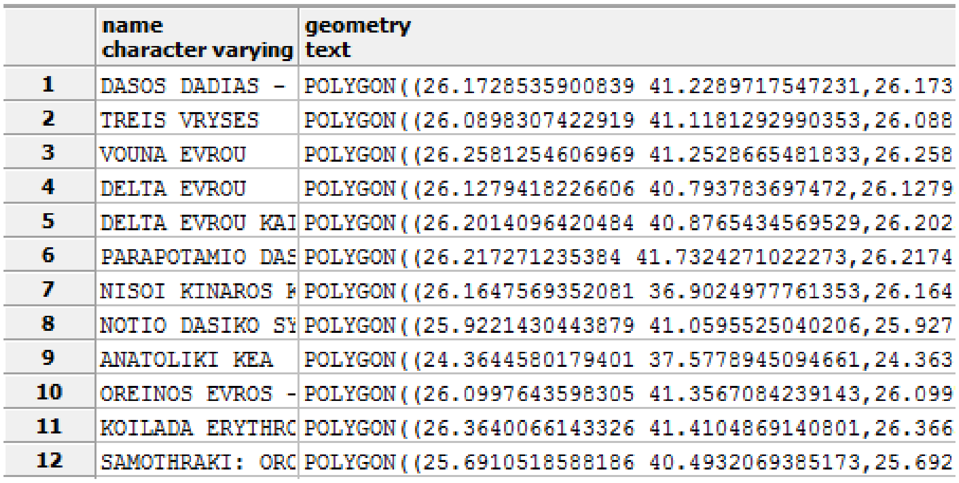
Figure 3.6 shows a snapshot of the KML file that is uploaded and processed to populate the `PROTECTED_AREAS` schema. First, the KML file is read and processed by creating a SimpleXML object using the `simplexml_load_file()` function, which takes the filename as the parameter. The SimpleXML object is populated with the XML elements from the file and this object is used to populate the `PROTECTED_AREAS` table. We extracted the name and the coordinates from the SimpleXML object and populated the table. To get the values of the name and coordinates attribute from the Placemark object, the `children()` method is used on the object. The name attribute is the first child of Placemark, whereas the coordinate attribute is nested deep within the MultiGeometry attribute. Hence, iteration through several child nodes is necessary. Appendix A.6 shows the PHP script used to read data from the KML file and populate the `PROTECTED_AREAS` table. The name attribute is the first child of Placemark, whereas the coordinate attribute is nested deep within the MultiGeometry attribute. Hence, iteration through several child nodes using foreach method is performed. The coordinates is got as string and to be stored in PostgreSQL, the commas are replaced with spaces and the spaces are replaced with commas using the `strtr()` function. This is because the format of the of the WKT representation that is given as a parameter to the `ST_GeomFromText()` function.



```
<?xml version="1.0" encoding="utf-8" ?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document><Folder><name>natura</name>
<Schema name="natura" id="natura">
    <SimpleField name="Name" type="string"></SimpleField>
    <SimpleField name="Description" type="string"></SimpleField>
    <SimpleField name="periphery" type="string"></SimpleField>
    <SimpleField name="name_latin" type="string"></SimpleField>
    <SimpleField name="objectid" type="int"></SimpleField>
    <SimpleField name="hectares" type="int"></SimpleField>
    <SimpleField name="code" type="string"></SimpleField>
    <SimpleField name="area" type="int"></SimpleField>
    <SimpleField name="perimeter" type="int"></SimpleField>
    <SimpleField name="prefecture" type="string"></SimpleField>
    <SimpleField name="sitetype" type="string"></SimpleField>
</Schema>
  <Placemark>
    <name>DASOS DADIAS - SOUFLI</name>
  <Style><LineStyle><color>ff0000ff</color></LineStyle>  <PolyStyle><fill>0</fill></PolyStyle></Style>
    <ExtendedData><SchemaData schemaUrl="#natura">
        <SimpleData name="periphery">ΑΝΑΤΟΛΙΚΗ ΜΑΚΕΔΟΝΙΑ ΘΡΑΚΗ</SimpleData>
        <SimpleData name="name_latin">DASOS DADIAS - SOUFLI</SimpleData>
        <SimpleData name="objectid">0</SimpleData>
        <SimpleData name="hectares">41111</SimpleData>
        <SimpleData name="code">GR1110002</SimpleData>
        <SimpleData name="area">411115781</SimpleData>
        <SimpleData name="perimeter">96989</SimpleData>
        <SimpleData name="prefecture">ΕΒΡΟΣ</SimpleData>
        <SimpleData name="sitetype">SPA</SimpleData>
    </SchemaData></ExtendedData>
      <MultiGeometry><Polygon><outerBoundaryIs><LinearRing><coordinates>26.172853590083893,41.228971754723098 26.173987367656569
  </Placemark>
```

Figure 0.6: KML file used to populate `PROTECTED_AREAS` table

Figure 3.7 shows the snapshot of the populated `PROTECTED_AREAS` schema from the CSV file as a result of uploading the CSV file into the PostgreSQL database.



	name character varying	geometry text
1	DASOS DADIAS -	POLYGON((26.1728535900839 41.2289717547231,26.173
2	TREIS VRYSES	POLYGON((26.0898307422919 41.1181292990353,26.088
3	VOUNA EVROU	POLYGON((26.2581254606969 41.2528665481833,26.258
4	DELTA EVROU	POLYGON((26.1279418226606 40.793783697472,26.1279
5	DELTA EVROU KAI	POLYGON((26.2014096420484 40.8765434569529,26.202
6	PARAPOTAMIO DAS	POLYGON((26.217271235384 41.7324271022273,26.2174
7	NISOI KINAROS K	POLYGON((26.1647569352081 36.9024977761353,26.164
8	NOTIO DASIKO SY	POLYGON((25.9221430443879 41.0595525040206,25.927
9	ANATOLIKI KEA	POLYGON((24.3644580179401 37.5778945094661,24.363
10	OREINOS EVROS -	POLYGON((26.0997643598305 41.3567084239143,26.099
11	KOILADA ERYTHRC	POLYGON((26.3640066143326 41.4104869140801,26.366
12	SAMOTHRAKI: ORC	POLYGON((25.6910518588186 40.4932069385173,25.692

Figure 0.7: Populated `PROTECTED_AREAS` schema

3.5 Summary

This chapter has explored in detail the design and implementation of the spatial database using PostgreSQL / PostGIS. The three main stages in database design, namely, conceptual, logical, and physical design have been explained. Data were acquired from a number of different sources. The historical AIS data were acquired the marine traffic project, and the European network of protected areas from the European Environment agency. The quality of the statistical analysis and mining ultimately depends on the quality of the database design. A well designed database improves the performance of the SQL queries. Further, the spatial relationship between entities has been defined. These kind of relationships help perform analysis on different data objects.

CHAPTER 4 STATISTICAL ANALYSIS AND MINING

4.1 Introduction

In this chapter, the methodology employed for the trajectory reconstruction will be discussed. These trajectories are of two types, namely "Detailed" and "Simplified". The simplified trajectory reconstruction implemented using the Douglas-Peucker algorithm will be explained in detail. Further, the methodology employed for computing distance, speed, direction, and turn angle at each intermediate location point of trajectories will be discussed and implemented. This chapter will also discuss the implementation of processes for identifying vessels that pass through EU Natura 2000 protected areas, determining vessels around a vessel at a particular instant in time, and detecting outliers using the DBSCAN algorithm.

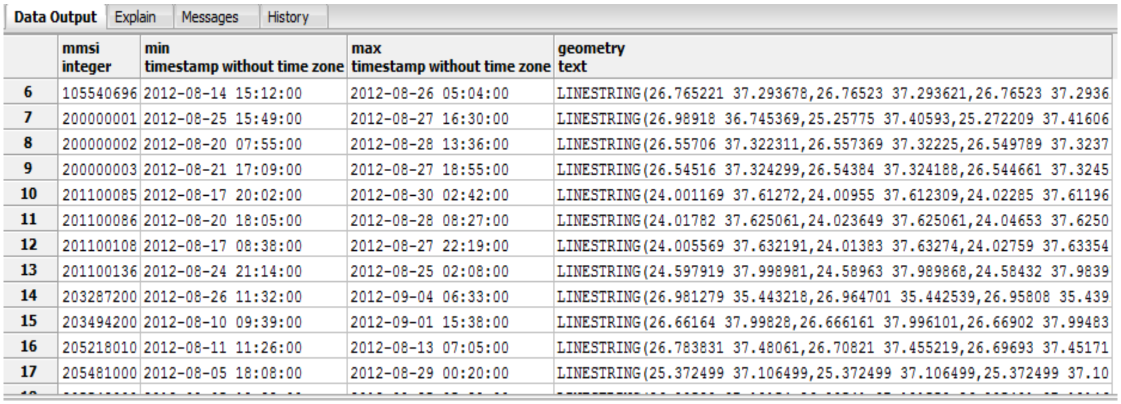
4.2 Detailed Trajectory Construction

Once the `LOCATION_SHIPS` database has been populated, as discussed in Chapter 3, the next step is to construct the trajectory for each ship based on its unique MMSI. The series of location points for each MMSI are ordered by the time at which the location was captured, as the location points might not be in order. To construct the detailed trajectories of each individual ship, a new database table called `ROUTE_SHIPS` is created automatically when the file upload process is completed. This table contains the trajectory traversed by each ship between its start and end time. This is implemented using the ST_Makeline(geometry) function in PostGIS. This function is an aggregate function that takes a sequence of point geometries of the same MMSI identifier and is transformed to make a linestring and returns the geometry of the linestring created. The algorithm for creating a single trajectory for a given MMSI identifier (ship_id) is shown below.

```
-   Select the all point geometries from the LOCATION_SHIPS table where
    MMSI = ship_id by ordering by time
-   Loop
        Store the point geometries into an array
-   End loop
-   If array length is greater than 1
        trajectory  ⟵ ST_MakeLine(array)
-   End If
-   Return trajectory
```

The algorithm described above was implemented as a function in PHP. The function that creates a trajectory for a given MMSI identifier is given in Appendix B.1. Here, the function createTrajectory takes in a MMSI identifier as parameter and returns the geometry of the linestring that is created from the point geometries.

For constructing the trajectories for all vessels, the `ROUTE_SHIPS` table is created automatically when the file upload process is completed as described earlier in this section. The PHP script that transforms a sequence of point geometries into trajectories for each MMSI in the `LOCATION_SHIPS` table is shown in Appendix B.2. First, SELECT query is used to fetch the MMSI, the starting time `(min(time_stamp))`, the ending time `(max(time_stamp))`, and the line geometry which is the detailed trajectory constructed through the `ST_MakeLine()` function, after ordering the timestamp values ascending. For each record fetched from this SELECT statement, the INSERT statement is executed to store the MMSI identifier, the starting time, the ending time and the line geometry into `ROUTE_SHIPS` table (Figure 4.1).



	mmsi integer	min timestamp without time zone	max timestamp without time zone	geometry text
6	105540696	2012-08-14 15:12:00	2012-08-26 05:04:00	LINESTRING(26.765221 37.293678,26.76523 37.293621,26.76523 37.2936
7	200000001	2012-08-25 15:49:00	2012-08-27 16:30:00	LINESTRING(26.98918 36.745369,25.25775 37.40593,25.272209 37.41606
8	200000002	2012-08-20 07:55:00	2012-08-28 13:36:00	LINESTRING(26.55706 37.322311,26.557369 37.32225,26.549789 37.3237
9	200000003	2012-08-21 17:09:00	2012-08-27 18:55:00	LINESTRING(26.54516 37.324299,26.54384 37.324188,26.544661 37.3245
10	201100085	2012-08-17 20:02:00	2012-08-30 02:42:00	LINESTRING(24.001169 37.61272,24.00955 37.612309,24.02285 37.61196
11	201100086	2012-08-20 18:05:00	2012-08-28 08:27:00	LINESTRING(24.01782 37.625061,24.023649 37.625061,24.04653 37.6250
12	201100108	2012-08-17 08:38:00	2012-08-27 22:19:00	LINESTRING(24.005569 37.632191,24.01383 37.63274,24.02759 37.63354
13	201100136	2012-08-24 21:14:00	2012-08-25 02:08:00	LINESTRING(24.597919 37.998981,24.58963 37.989868,24.58432 37.9839
14	203287200	2012-08-26 11:32:00	2012-09-04 06:33:00	LINESTRING(26.981279 35.443218,26.964701 35.442539,26.95808 35.439
15	203494200	2012-08-10 09:39:00	2012-09-01 15:38:00	LINESTRING(26.66164 37.99828,26.666161 37.996101,26.66902 37.99483
16	205218010	2012-08-11 11:26:00	2012-08-13 07:05:00	LINESTRING(26.783831 37.48061,26.70821 37.455219,26.69693 37.45171
17	205481000	2012-08-05 18:08:00	2012-08-29 00:20:00	LINESTRING(25.372499 37.106499,25.372499 37.106499,25.372499 37.10

Figure 0.1: A snapshot of the constructed trajectories in the `ROUTE_SHIPS` table

4.3 Simplified Trajectory Construction

The AIS records a far denser collection of points than necessary. The raw trajectory data is therefore usually very large, becomes expensive to store, and decreases the performance of the web application significantly. This may cause the web browser to crash when the trajectories are analyzed and mined. This creates a need to employ trajectory simplification algorithms to reduce the number of data points defining each trajectory while still maintaining the shape information that trajectory.

The simplified trajectories are constructed using the Douglas–Peucker algorithm (Douglas & Peucker, 1973). The reason for choosing this algorithm is because it is one of the most commonly available algorithms in GIS to simplify lines (Joao, 1988). The Douglas–Peucker algorithm is a generalization algorithm that is used to reduce the number of points in a curve that is approximated by a series of points. Given a set of points and a threshold, this algorithm generates a simplified line connecting these set of points. The start and end points will remain in the simplified line. First, a straight line segment connecting the start and end points is constructed. Second, the perpendicular distances between each point on the constructed line segment and the line connecting the set of points is calculated. The maximum distance among them is compared with the threshold value and, if it is greater than the threshold value, the corresponding point remains in the simplified line and the line is split into two parts. These steps are recursively run to each part of the line segment. The stopping criteria is when the there is no maximum perpendicular distance greater than the threshold value. The result consists of only a subset of the original set of points. Figure 4.2 shows how a line segment has been simplified using the Douglas–Peucker algorithm.

The threshold value determines the level of simplification. The smaller the threshold value, the greater will be the number of points retained in the simplified line and the better will be the approximation. Selecting an appropriate threshold value is an important criterion in line generalization using the Douglas–Peucker algorithm. The proposed system will have an option for selecting different threshold values, and users can experiment to get the output desired. Figure 4.3 illustrates the steps involved for the simplified trajectory construction for each vessel in the `LOCATION_SHIPS` table.

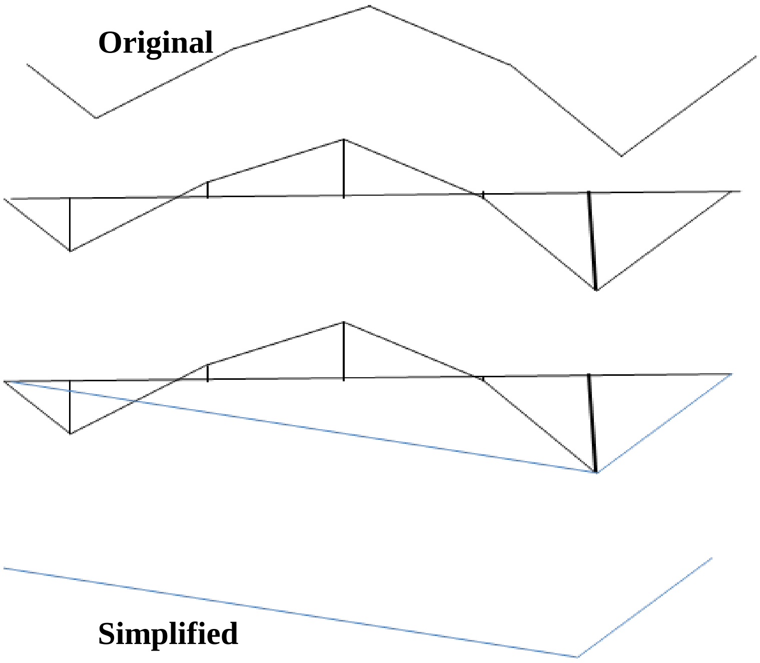


Figure 0.2: Line simplification using Douglas–Peucker algorithm

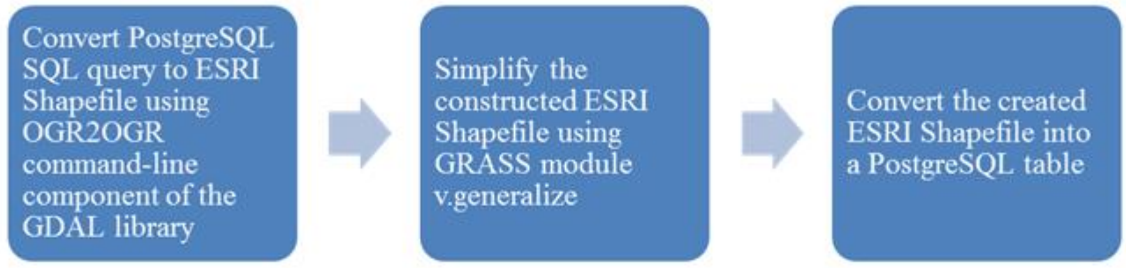

Convert PostgreSQL SQL query to ESRI Shapefile using OGR2OGR command-line component of the GDAL library
Simplify the constructed ESRI Shapefile using GRASS module v.generalize
Convert the created ESRI Shapefile into a PostgreSQL table

Figure 0.3: Flow diagram to construct a simplified trajectory using Douglas–Peucker algorithm

4.3.1 PostGIS Data to ESRI Shapefile

The first step to construct the simplified trajectories is to connect directly to the PostgreSQL database and convert a SQL query into an ESRI shapefile using the OGR2OGR command-line component of the GDAL library. OGR2OGR (Open Source Geospatial Foundation, 2010) is a component which is part of the GDAL used for reading, writing and, processing of vector data formats, including ESRI shapefile, spatial databases like PostGIS and Oracle Spatial, MapInfo file, GML, KML, and TIGER. It is powerful and also free, licensed under the MIT-style open source license. It is widely used in the commercial GIS community due to its widespread use and comprehensive set of functionalities (Neteler & Raghavan, 2006).

Appendix B.3 illustrates the creation of a shapefile called `route.shp` based on a SQL query that retrieves the route traversed by MMSI identifier 23700000 between

2012-08-25 08:17:00 and 2013-01-15 12:25:00. To import data from PostgreSQL to an ESRI shapefile, the output file format is set as "ESRI Shapefile". The location to store the created shapefile is set as a full path setting. The contents of the SQL query set in `-sql` flag is imported into a shapefile `route.shp at location C:\Users\Desktop\route.`

PHP executes external commands which are normally executed in the command line from the script using the `proc_open()`function (PHP Group, 2013) function. It basically creates a process and has pipes for the following: (i) reading the command from the standard input, (ii) returning the output of the command through the standard output, and (iii) handling any errors that the command may cause. The parameters for `proc_open()` function include: (1) the string representing the command to execute; (2) an array of descriptors; (3) an empty array variable called `$pipes`; and (4) the absolute directory path of the initial working directory for the command. The array of descriptors should consist of three elements and it includes: (1) an array describing the pipe that represents the standard input from which the process writes; (2) an array describing the pipe that represents the standard output to which the process writes; and (3) name and location of a file to which the process logs error statements if any. The result of the `proc_open()` function is a resource that populates the `$pipes` array. The command that is to be piped is written to the first element of the `$pipes` array, `$pipes[0]`. The result of the command written is read from the second element of the $pipes array, `$pipes[1]`.

Appendix B.4 illustrates the PHP script that executes the creation of the shapefile based on a PostGIS spatial query using the org2ogr command given in Appendix B.3. Based on the MMSI identifier, and the time range selected by the user, the PHP function `convertPostGISToShapefile()` generates a shapefile of the route traversed by the selected MMSI between the time range called `route.shp` stored in location `C:\Users\Desktop\route\`. PHP's `proc_open()` function is called by passing the org2ogr command and the client supplied arguments in a single string. In addition, the $descriptors property, the empty `$pipes` array, and the location of the directory that has the org2ogr application is also passed. No other command needs to be piped to the org2ogr command and so, an empty string is passed to `$pipes[0]`. The process is closed using the `proc_close()` function. If the command is successfully executed, proc_close() function returns 0; else, it returns -1.

4.3.2 Simplification of the Constructed ESRI Shapefile using GRASS

The second step in the trajectory generalization is to create a PyWPS process that is capable of executing GRASS GIS modules. The generalization module `v.generalize` offers a generalization service that facilitates simplification and smoothing of linear geometries using the Douglas-Peucker and several other algorithms. A PyWPS process that is capable of executing the `v.generalize` GRASS GIS module was created. In turn, this PyWPS process is executed by PHP. Table 4.1 shows the parameters used in the `v.generalize` module.

Table 4.1: Parameters available in v.generalize module

Parameter	Description
`input`	Name of input shapefile in the GRASS GIS location
`output`	Name of output shapefile that is to be created
`type`	Type of the shapefile geometry. E.g., line, boundary, and area.
`method`	The algorithm that is to be used for generalization. Several line simplification algorithms, such as, Douglas-Peucker Algorithm, Douglas-Peucker Reduction Algorithm, Lang Algorithm, Vertex Reduction, Reumann-Witkam Algorithm, Remove Small Lines/Areas are available.
`threshold`	The maximum tolerance value
`where`	A SQL query using the WHERE without 'where' keyword

The route shapefile was copied within the GRASS location at `C:\Users\ \Desktop\route_simplified` to enable the `v.generalize` to be executed. A PHP function using exec() method was created to execute the `xcopy` command that copies the route folder created from the previous step to the GRASS location. Appendix B.5 shows the PHP script that was used copy the folder to the GRASS location using the `xcopy` command.

Appendix B.6 illustrates the way to apply the Douglas-Puecker algorithm using the `v.generalize` module to the route shapefile that was created based on parameters set by the user during the previous step and copied to the GRASS GIS location. Here, the threshold is given in map unit degrees. The result of this command is the creation of a generalized shapefile called `route_simplified` at `C:\Users\Sabarish\Desktop\route_simplified\newLocation\Sabarish\vector`.

To facilitate access to the `v.generalize` GRASS module via web interface, a PyWPS process was created. This process would execute the command in Appendix B.4 and subsequently this process is invoked as a service through PHP. In PyWPS, a process is defined in a single Python file. The processes can be executed within a temporary GRASS Location or within an existing GRASS Location, within temporary created Mapset. The PyWPS process was executed in the location `C:\Users\\Desktop\route_simplified` that was created during GRASS GIS installation. This location and mapset were created during GRASS GIS installation. A process is defined in a Python file by creating a class or instance that implements two methods, namely, the `__init__()`, and `execute()` method. The `__init__()` method initializes the process and describes the main attributes of the process such as the title, abstract, version, identifier (name of the Python file), and grassLocation. The grassLocation is the name of the GRASS Location within the configured grassdbase. This property was set to `C:\Users\Sabarish\Desktop\route_simplified`. The `execute()` method is called once PyWPS accepts execute request type. This method retrieves the information in the input data and executes the process algorithm. Appendix B.7 shows a PyWPS process called `generalize.py` that creates a generalized geometry using the `v.generalize` GRASS module. Here, `self.cmd()` inside the execute method is a WPSProcess method used to execute GRASS GIS commands. Each parameter-value pair is enclosed in " " and separated by ‘,’.

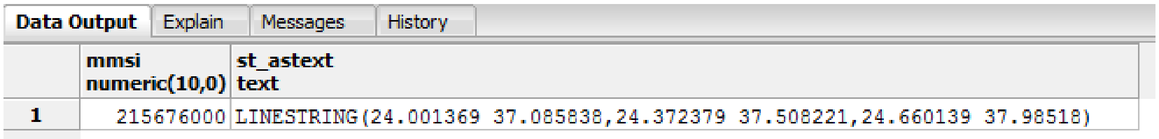
The PyWPS process consisting of a single Python file is saved in the server and executed by passing the string `"http://localhost/cgi-bin/generalize.py?Service=WPS&request=execute&version=1.0.0&identifier=generalize"` into PHP’s `file_get_contents()` method. The result of this execution is the creation of an ESRI shapefile called `route_simplified` that contains the generalized geometry of the trajectory.

4.3.3 Loading the Simplified Shapefile into a PostgreSQL Table

Once the line simplification process is completed using PyWPS and the resulting geometry stored as an ESRI shapefile spatial data format in the server, the third step is to load the shapefile into a PostgreSQL table for visualization through Google Maps API. The shapefile that was created in the previous was imported into PostgreSQL. Again, OGR2OGR, the command-line component of the GDAL is used to import the shapefile into a PostGIS table. Appendix B.8 illustrates the command to import data from a shapefile into PostgreSQL. Here, the output format is stated as `"PostgreSQL"` and the destination is specified through the connection string of the target PostgreSQL database.

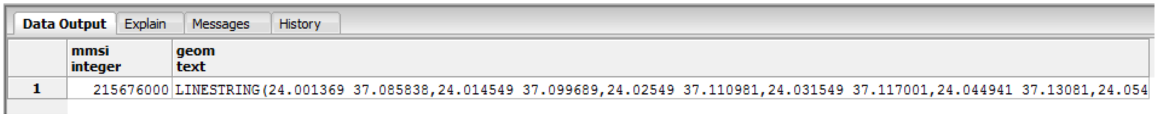
By default, a new table in the destination database having the same file name as that of the source data is created. If data needs to be inserted into a specific table, the `-nln` flag is used. This flag will insert data into a named table in the destination database and default behavior is ignored. Data can either be appended as new records into an existing table using the `-append` flag or data can be inserted into a table by deleting and re-creating it using the `-overwrite` flag. In Appendix B.8, the contents of the simplified geometry `route_simplified.shp` are loaded into the `route_simplified` table at host `gaia.gge.unb.ca` PostgreSQL instance (Figure 4.4).

The external ogr2ogr command in Appendix B.8 imports the simplified trajectory shapefile into a PostgreSQL table with a PHP script via `proc_open()` function. The PHP function `convertShapefileToPostgreSQL()` is called after the execution of the Python script that is used to generalize the trajectories. Figure 4.4 shows the `route_simplified` table that contains the simplified trajectory for the MMSI identifier 215672000. Here, the number of points has been reduced to 3, whereas the original detailed trajectory consisted of several points (figure 4.5).



	mmsi numeric(10,0)	st_astext text
1	215676000	LINESTRING(24.001369 37.085838,24.372379 37.508221,24.660139 37.98518)

Figure 0.4: Snapshot of the simplified trajectory for MMSI 21567600



	mmsi integer	geom text
1	215676000	LINESTRING(24.001369 37.085838,24.014549 37.099689,24.02549 37.110981,24.031549 37.117001,24.044941 37.13081,24.054

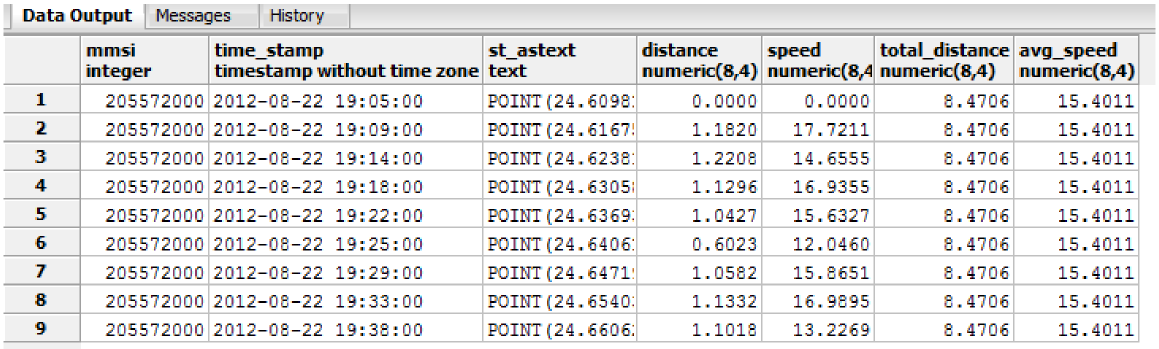
Figure 0.5: Snapshot of the detailed trajectory for MMSI 21567600

4.4 Calculating Distance, Speed, and Direction of Trajectories

The distance travelled by a vessel is calculated as Euclidean distance between all points traversed by it. The function to compute this measure in implemented by using PostGIS's `ST_Length()` function, which returns the Cartesian 2D length of the geometry. Since this function returns the result in units of spatial reference with which the geometry is stored, the trajectory polylines were transformed into a meter-based projection. Hence, WGS84 curvilinear coordinates were transformed into plane coordinates using the spherical Mercator projection (EPSG: 3857), which is a common projection in web mapping and visualization applications using the `ST_Transform(geometry, SRID)` PostGIS function. Appendix B.9 illustrates a query used to determine the distance travelled by a vessel (here the MMSI identifier is equal to a value X). Here, `route_ships1` is a temporary view created to store the results of the user query that retrieves the route traversed by a single vessel between a start and end provided by the user. Appendix B.10 illustrates the way in which a temporary view was created to store the results of the route traversed by a single vessel. Here, X, Y, and Z are the MMSI identifier, start, and end time respectively provided by the user through the web interface.

Based on the travelled distance and travelled time, the speed is calculated. The speed can be calculated as the speed between two time periods, and the average speed along the entire trajectory. For calculating the average speed, the total distance travelled between these time periods is divided by the total travelled time along the trajectory. A stored function that calculates the travelled distance along the trajectory, and between two consecutive time points was created and stored in the table `distanceandspeed`. In addition, the average speed along the trajectory and the speed between two consecutive timepoints are calculated. This stored procedure takes in the MMSI identifier and the time intervals for which the distance and speed are to be calculated as input parameters, which is given by the user. Appendix B.11 illustrates the code snippet of the function `DistanceAndSpeedCalc` that shows how the distance and speed are calculated given a MMSI identifier and a time interval.

In the code given in Appendix B.11, a cursor variable `location_ships_cur` is created, which is associated with a SQL SELECT statement and can hold different values at run time. The records that are associated with `location_ships_cur` contain the position and the time at which the position was recorded for the vessels. The inputs are the MMSI identifier and the time period. These records are iterated over, and the distance between two consecutive time points, the speed between two consecutive time points, the total distance of the entire trajectory, and average speed of the entire trajectory are computed using the `ST_Distance()` function. The values are stored in the `distanceandspeed` table. Figure 4.6 shows the snapshot of the computed total distance travelled along a trajectory, average speed along the trajectory, distance between two consecutive time points, and speed between two consecutive time points of the MMSI identifier 205572000 between the time interval ‘2012-08-22 19:05:00’ and ‘2016-08-22 19:05:00’. Here the distance is in nautical miles and speed is in nautical miles per hour.

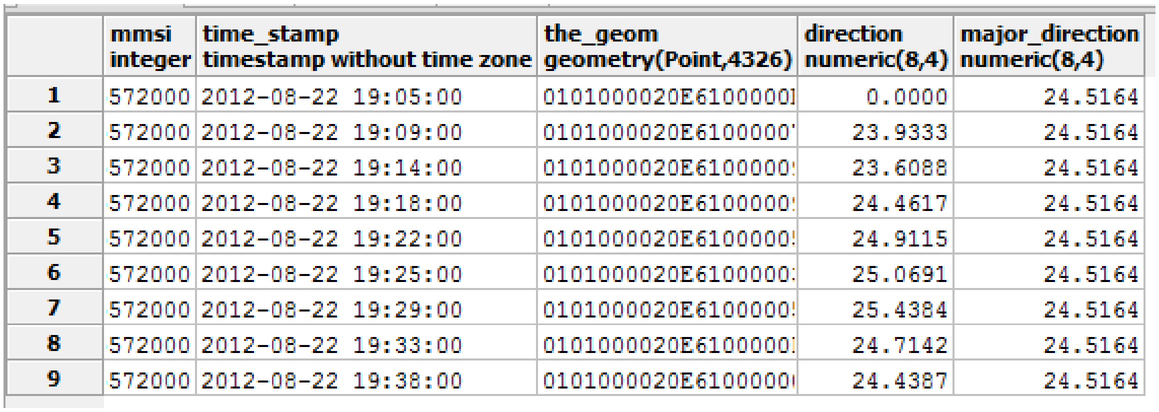


	mmsi integer	time_stamp timestamp without time zone	st_astext text	distance numeric(8,4)	speed numeric(8,4	total_distance numeric(8,4)	avg_speed numeric(8,4)
1	205572000	2012-08-22 19:05:00	POINT(24.6098:	0.0000	0.0000	8.4706	15.4011
2	205572000	2012-08-22 19:09:00	POINT(24.6167!	1.1820	17.7211	8.4706	15.4011
3	205572000	2012-08-22 19:14:00	POINT(24.6238:	1.2208	14.6555	8.4706	15.4011
4	205572000	2012-08-22 19:18:00	POINT(24.6305	1.1296	16.9355	8.4706	15.4011
5	205572000	2012-08-22 19:22:00	POINT(24.6369:	1.0427	15.6327	8.4706	15.4011
6	205572000	2012-08-22 19:25:00	POINT(24.6406:	0.6023	12.0460	8.4706	15.4011
7	205572000	2012-08-22 19:29:00	POINT(24.6471'	1.0582	15.8651	8.4706	15.4011
8	205572000	2012-08-22 19:33:00	POINT(24.6540:	1.1332	16.9895	8.4706	15.4011
9	205572000	2012-08-22 19:38:00	POINT(24.6606:	1.1018	13.2269	8.4706	15.4011

Figure 0.6: Snapshot of the calculated distance and speed given a MMSI and time interval

Similarly, the direction of the trajectory can either be calculated as a major direction of the trajectory between the starting time and ending time or between two consecutive intermediate time points. PostGIS’s built-in `ST_Azimuth()` function was used to calculate the direction in radians. The `ST_Azimuth()` function takes two geometries as input parameters and Returns the north-based azimuth. The radian measure is converted to degree units by multiplying the radian measure by 180/π. A stored function that calculates the direction of the trajectory between the starting time and ending time, and between two consecutive time points was created and the result stored in the table `azimuthcalc`. This stored function takes in the MMSI identifier and the time intervals for which the direction are to be calculated as input parameters, which is given by the user. Appendix B.12 illustrates the code snippet of the function `DirectionCalc` that shows how the major direction of a trajectory between the initial and final time point and between two consecutive intermediate time points are calculated given a MMSI identifier and a time interval.

In the code given in Appendix B.12, a cursor variable `location_ships_cur` is created, which is associated with a SQL SELECT statement and can hold different values at run time. The records that are associated with `location_ships_cur` contain the position and the time at which the position was recorded for the vessels. The inputs are the MMSI identifier and the time period. These records are iterated over, and the direction between two consecutive time points, and the major direction of the trajectory are computed using the `ST_Azimuth()` function. The values are stored in the `direction` table. Figure 4.7 shows the snapshot of the computed direction between intermediate time points, and the major direction of the trajectory between the first and the last time point of the MMSI identifier 205572000 between the time interval ‘2012-08-22 19:05:00’ and ‘2016-08-22 19:05:00’.



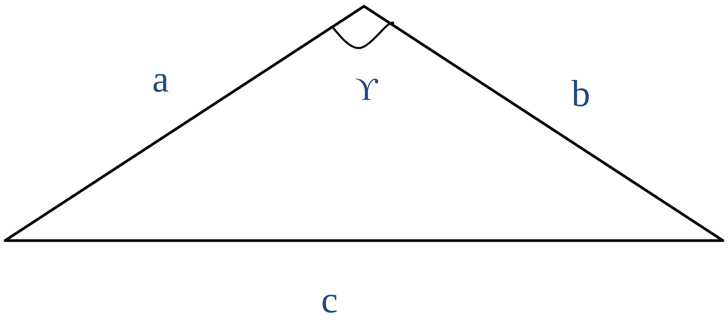
	mmsi integer	time_stamp timestamp without time zone	the_geom geometry(Point,4326)	direction numeric(8,4)	major_direction numeric(8,4)
1	572000	2012-08-22 19:05:00	0101000020E6100000	0.0000	24.5164
2	572000	2012-08-22 19:09:00	0101000020E6100000	23.9333	24.5164
3	572000	2012-08-22 19:14:00	0101000020E6100000	23.6088	24.5164
4	572000	2012-08-22 19:18:00	0101000020E6100000	24.4617	24.5164
5	572000	2012-08-22 19:22:00	0101000020E6100000	24.9115	24.5164
6	572000	2012-08-22 19:25:00	0101000020E6100000	25.0691	24.5164
7	572000	2012-08-22 19:29:00	0101000020E6100000	25.4384	24.5164
8	572000	2012-08-22 19:33:00	0101000020E6100000	24.7142	24.5164
9	572000	2012-08-22 19:38:00	0101000020E6100000	24.4387	24.5164

Figure 0.7: Snapshot of the calculated direction given a MMSI and time interval

4.5 Identifying Sharp Turns in the Trajectory

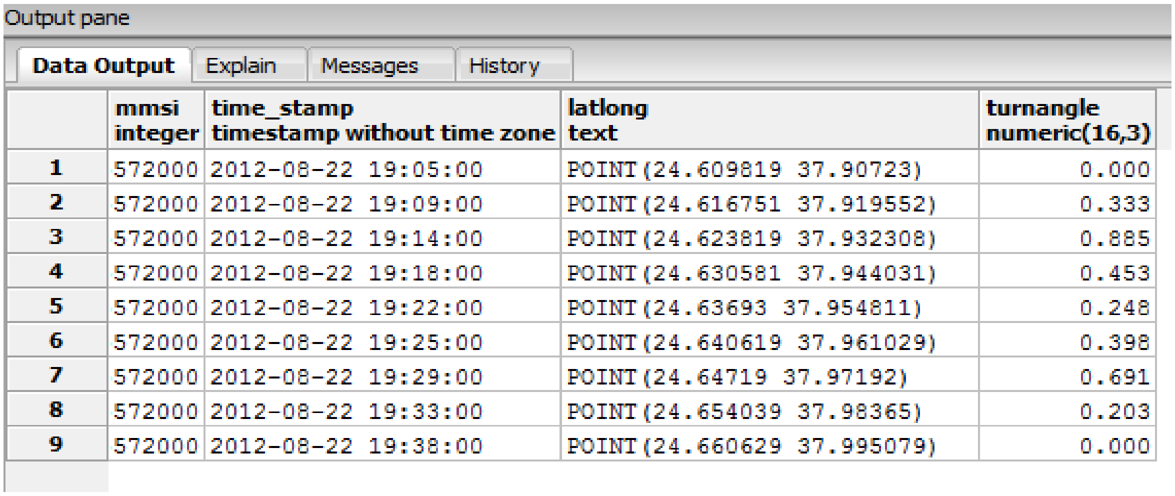
To determine the locations where a vessel has made sharp turns, the turn angle between successive locations along the trajectory of the vessel is computed and the values which exceed a user specified threshold are determined. The turn angles are calculated and presented to analysts as it a very good indicator of maritime risk and it is one of the important factors which contribute to a shipping accident. The coordinates at these angles are where the vessel has made a sharp turn. The turn angle is computed using the cosine formula, which relates the length of the sides a triangle and the angles formed by this triangle. Knowing the length of the three sides of a triangle a, b, and c, the angle Υ of the triangle can be computed using the equation given below. Due to the result of a periodic lost in transmission, some turn angles computed would have very large values. To prevent such a scenario, the time difference between consecutive points were considered and the turn angles are computed for location points only when the difference is less than that of the sampling interval.

$$\Upsilon = \arccos\left(\frac{a^2+b^2-c^2}{2ab}\right)$$



Appendix B.13 given below illustrates the code snippet of the function `turnanglecalc` that shows how turn angles at each location a vessel has traversed are calculated given a MMSI identifier and a time interval. At each intermediate location, the previous and the next position of the vessel are determined from the `LOCATION_SHIPS` table. These three points form a triangle and distances between the points are determined using the `ST_Distance()` function**.** The turn angle at the intermediate point is computed only if the time difference between the successive points is less than the sampling interval using the law of cosines, and the values are inserted into the `turn_angle` table. The angle computed is in radian measure and is converted into degrees. From the turn angles computed, the locations where a vessel has made sharp turns can be determined if the turn angles at the location exceeds a user-specified threshold.

Figure 4.8 shows the snapshot of the computed turn angles along the trajectory between the first and the last time point of the MMSI identifier 205572000 between the time interval ‘2012-08-22 19:05:00’ and ‘2012-08-23 19:05:00’. Since the trajectory is nearly a straight line, the turn angles are nearly 0.



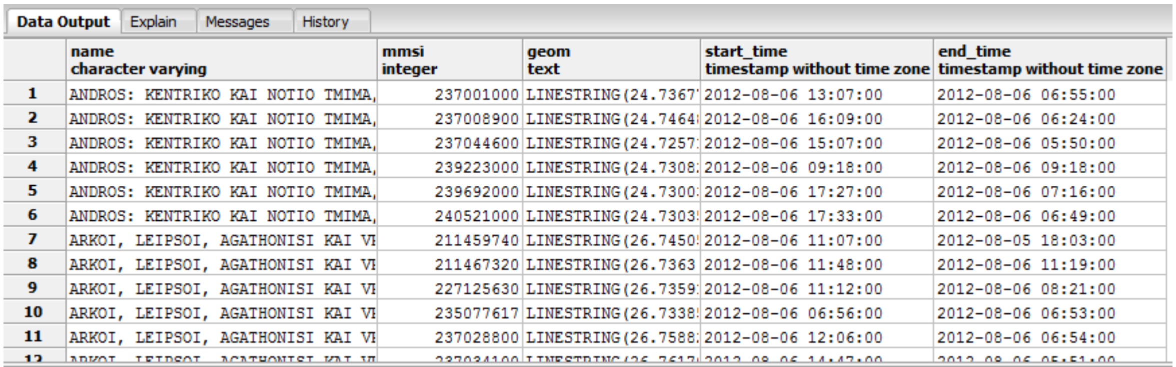
	mmsi integer	time_stamp timestamp without time zone	latlong text	turnangle numeric(16,3)
1	572000	2012-08-22 19:05:00	POINT(24.609819 37.90723)	0.000
2	572000	2012-08-22 19:09:00	POINT(24.616751 37.919552)	0.333
3	572000	2012-08-22 19:14:00	POINT(24.623819 37.932308)	0.885
4	572000	2012-08-22 19:18:00	POINT(24.630581 37.944031)	0.453
5	572000	2012-08-22 19:22:00	POINT(24.63693 37.954811)	0.248
6	572000	2012-08-22 19:25:00	POINT(24.640619 37.961029)	0.398
7	572000	2012-08-22 19:29:00	POINT(24.64719 37.97192)	0.691
8	572000	2012-08-22 19:33:00	POINT(24.654039 37.98365)	0.203
9	572000	2012-08-22 19:38:00	POINT(24.660629 37.995079)	0.000

Figure 0.8: Snapshot of the calculated turn angles along a trajectory given a MMSI and a time interval

4.6 Identifying Vessels That Intersect the Protected Areas

The vessels that pass through the protected areas are determined using PostGIS’s `ST_Intersects()` function. The `ST_Intersects()` function takes as input two geometry objects, and returns true if one geometry spatially intersect the another geometry and false it the geometries do not intersect. Appendix B.14 illustrates the SQL query used to determine the vessels that intersect each of the protected areas using the `ST_Intersects()` function between a given time period, and the trajectory of each vessel in each protected area is created using the `ST_MakeLine()` function. The query statement illustrates an SQL inner join operation that combines the records of the `LOCATION_SHIPS` and `PROTECTED_AREAS` table and is filtered using the `ST_Intersects(geometry A, geometry B)` function that returns true if the geometry A is intersecting geometry B completely. Here, geometry A is the collection of `LOCATION_SHIPS` points that are intersecting the `PROTECTED_AREAS` as defined by geometry B.

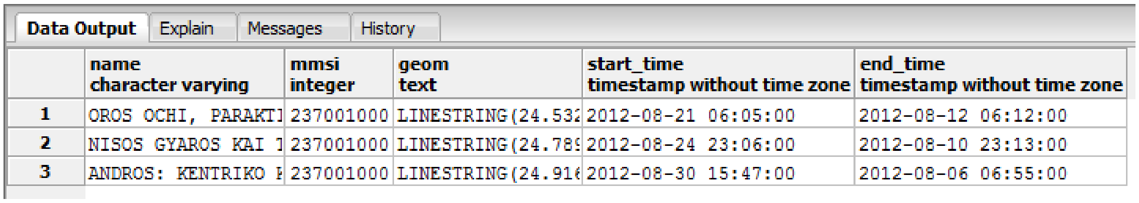
Figure 4.9 illustrates the result from the query in Appendix B.14. Here, the route traversed by the vessels in each of the protected area is determined. In addition, the time period in which the vessel traverses a protected area is also determined.



	name character varying	mmsi integer	geom text	start_time timestamp without time zone	end_time timestamp without time zone
1	ANDROS: KENTRIKO KAI NOTIO TMIMA,	237001000	LINESTRING(24.7367	2012-08-06 13:07:00	2012-08-06 06:55:00
2	ANDROS: KENTRIKO KAI NOTIO TMIMA,	237008900	LINESTRING(24.7464	2012-08-06 16:09:00	2012-08-06 06:24:00
3	ANDROS: KENTRIKO KAI NOTIO TMIMA,	237044600	LINESTRING(24.7257	2012-08-06 15:07:00	2012-08-06 05:50:00
4	ANDROS: KENTRIKO KAI NOTIO TMIMA,	239223000	LINESTRING(24.7308	2012-08-06 09:18:00	2012-08-06 09:18:00
5	ANDROS: KENTRIKO KAI NOTIO TMIMA,	239692000	LINESTRING(24.7300	2012-08-06 17:27:00	2012-08-06 07:16:00
6	ANDROS: KENTRIKO KAI NOTIO TMIMA,	240521000	LINESTRING(24.7303	2012-08-06 17:33:00	2012-08-06 06:49:00
7	ARKOI, LEIPSOI, AGATHONISI KAI VI	211459740	LINESTRING(26.7450	2012-08-06 11:07:00	2012-08-05 18:03:00
8	ARKOI, LEIPSOI, AGATHONISI KAI VI	211467320	LINESTRING(26.7363	2012-08-06 11:48:00	2012-08-06 11:19:00
9	ARKOI, LEIPSOI, AGATHONISI KAI VI	227125630	LINESTRING(26.7359	2012-08-06 11:12:00	2012-08-06 08:21:00
10	ARKOI, LEIPSOI, AGATHONISI KAI VI	235077617	LINESTRING(26.7338	2012-08-06 06:56:00	2012-08-06 06:53:00
11	ARKOI, LEIPSOI, AGATHONISI KAI VI	237028800	LINESTRING(26.7588	2012-08-06 12:06:00	2012-08-06 06:54:00
12	ARKOI, LEIPSOI, AGATHONISI KAI VI	237034100	LINESTRING(26.7617	2012-08-06 14:47:00	2012-08-06 05:51:00

Figure 0.9: Vessels intersecting protected areas generated using `ST_Intersects()` function

The query in Appendix B.14 could be modified to determine the protected areas a vessel goes through. Appendix B.15 illustrates the query which determines the protected areas that the vessel with MMSI identifier 237001000 passes through during the month of August in 2012. Figure 4.10 illustrates the result of this query. Here, it is identified that there are three protected areas that the vessel with MMSI identifier 237001000 passes through.



	name character varying	mmsi integer	geom text	start_time timestamp without time zone	end_time timestamp without time zone
1	OROS OCHI, PARAKTI	237001000	LINESTRING(24.53	2012-08-21 06:05:00	2012-08-12 06:12:00
2	NISOS GYAROS KAI	237001000	LINESTRING(24.78	2012-08-24 23:06:00	2012-08-10 23:13:00
3	ANDROS: KENTRIKO	237001000	LINESTRING(24.91	2012-08-30 15:47:00	2012-08-06 06:55:00

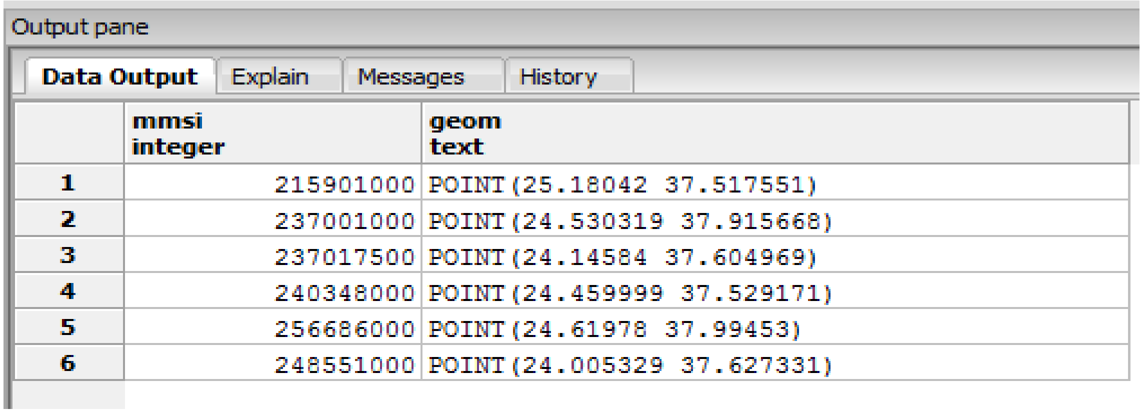
Figure 0.10: Areas that the vessel with MMSI identifier 237001000 passes through

4.7 Identifying Vessels That Are in the Vicinity of a Vessel

Identifying vessels that are in the vicinity of a vessel is essential as it indicates a shipping risk and a near-collision condition. Based on the chosen MMSI identifier and a time stamp, the vessels that are in the vicinity of a vessel defined by the chosen MMSI identifier and a time stamp can be determined. First, a simple search is performed to find the location of the vessel based on the user inputted MMSI identifier and a time stamp. Second, a buffer is defined based on a user-specified range around the location obtained in the first step within which vessels that are within this buffer are to be searched. A buffer is defined using the ST_Buffer(geometry A, float radius_of_buffer) function which returns a geometry that represents the geometry of the polygon whose boundary is defined by all points whose distance from this geometry A fall within the distance defined by the radius_of_buffer parameter. Third, points from LOCATION_SHIPS are filtered to identify the vessels that are within the buffer defined using the ST_Within(geometry A, geometry B) function which returns true if the geometry defined in A is completely inside that of geometry B. In this case, A is the geometry of the collection of points in the LOCATION_SHIPS table and B is the geometry of the buffer defined around the vessel within which other vessels are to be determined.

The spatial SQL query that is used to determine the vessels that are in the vicinity of a vessel is given in Appendix B.16. Here, the vessels around 50 nautical miles of the MMSI identifier 237001000 at '2012-08-21 06:05:00' is determined. The query uses an inner join operation that joins two SQL SELECT statements. First, the geometry of the vessel with MMSI identifier 237001000 at '2012-08-21 06:05:00' is determined using ST_Buffer() function. Second, the geometries of the vessels at '2012-08-21 06:05:00' from LOCATION_SHIPS table is extracted. These two SELECT statements are joined, and the MMSI identifier along with its geometry are filtered using the ST_Within(geometry A, geometry B) function that returns true if the geometry A is completely inside of geometry B.

The result of the SQL query in Appendix B.16 is shown in Figure 4.11. The result returned five vessels that are around 50 nautical miles of the MMSI identifier 237001000 at '2012-08-21 06:05:00'.



	mmsi integer	geom text
1	215901000	POINT(25.18042 37.517551)
2	237001000	POINT(24.530319 37.915668)
3	237017500	POINT(24.14584 37.604969)
4	240348000	POINT(24.459999 37.529171)
5	256686000	POINT(24.61978 37.99453)
6	248551000	POINT(24.005329 37.627331)

Figure 0.11: Vessels that are located within 50 nautical miles of the MMSI identifier 237001000 at '2012-08-21 06:05:00'

4. 8 Outlier Detection

Outliers are erroneous location points of a trajectory that exist because of data entry errors, and errors in data recorded by the AIS. These data errors are typically identified if the distance travelled by a vessel is large within a short interval of time. The vessels are associated with spatio-temporal points, where the location of the vessel is monitored over time. In this scenario, spatio-temporal "outliers" are defined to be anomalous points which are at a large distance from other points in a given trajectory. This creates a need to perform outlier detection on the spatial data to determine whether there were any outliers which do not conform to general behavior. Density based outlier detection using DBSCAN algorithm was considered in this thesis.

Density Based Spatial Clustering of Applications with Noise (DBSCAN) (Ester et al., 1996) is a density based clustering algorithm that is used to cluster points based on the similarities with respect to distance. DBSCAN is well suited for large spatial databases. One of the main advantages of DBSCAN is that it is robust to outliers, and hence can detect outliers in large spatial data sets. DBSCAN is employed to identify geographical outliers in the vessels data.

DBSCAN classifies each point as (a) core point, (b) border point, and (c) noise. A core point is point which is inside a cluster. A border point is a point which is on the border of a cluster. A point which is neither a core nor a border is noise. Two input parameters are: (i) ε-neighborhood, which is the radius of a cluster, and (ii) MinPts, which is minimum number of points required to form a cluster.

Key concepts

The following are the key concepts that define the DBSCAN algorithm:

1) The ε-neighborhood of a point P, denoted by $N\varepsilon_{(p)}$, is defined by $N_{\varepsilon(p)} = \{q \in D \mid dist(p,q) \leq \varepsilon \}$, i.e, the ε-neighborhood of a point p consists of all points around it where the distance between the point p and all other points is less

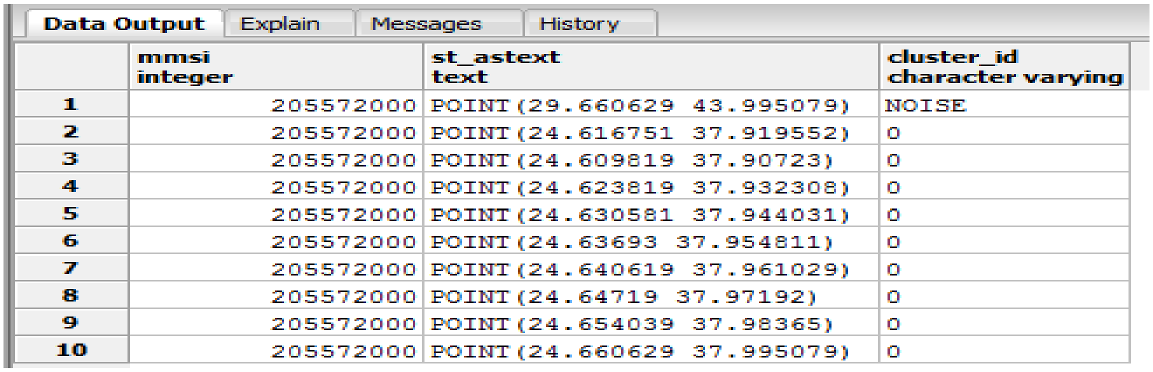
than or equal to ε. A point is a core point if the number of points around its ε-neighborhood is more than MinPts. A border point has less than MinPts around its ε-neighborhood, but it is within the ε-neighborhood of the core point. A noise point is a point which is neither a core point nor a border point.

2) A point p is directly-density reachable from a point q if p is within the ε-neighborhood of q and the number of points within the ε-neighborhood of q is greater than or equal to MinPts.
3) A point p is density reachable from a point q if there is a series of points p_1, p_2,.. p_n, p_1 = p, p_n = q such that p_{i+1} is directly density-reachable from $p_{i.}$
4)) A point p is density-connected to a point q if there is a point r such that both p and q are density-reachable from r with respect to ε and MinPts.

DBSCAN starts from an arbitrary point p and determines all points density-reachable from p with respect to ε and MinPts. If the number of points around point p is greater than or equal to MinPts, then point p is a core point, and a cluster is started. All density-reachable points along with the core-point are assigned to this cluster. Else, the point p is labeled as noise, and the next point which is unvisited is determined and the process of determining the density-reachable points, and assigning the points to a cluster, or labeling the point as noise is repeated. A point can belong to a cluster, and still be labeled as noise, as the point may have been initially labeled as noise and in the subsequent iteration be classified into a cluster. Thus, the points which are not in any cluster are considered as outliers.

The DBSCAN algorithm was implemented using PHP. This implementation is capable of reading the data from the `LOCATION_SHIPS` table and determines the outliers. This implementation, like the original algorithm, takes in two input parameters, namely MinPts, the minimum number of points within a cluster, and ε, neighborhood radius of a cluster. The PHP implementation consists of two functions, namely `DBSCAN()` and `ExpandCluster()`. The `DBSCAN()` function takes as input, the geometry (latitude / longitude) of the vessels for which the outliers has to be detected as an array, MinPts as an integer, and ε as an integer. This function iterates over each geometry and determines all points within its ε-neighborhood using PostGIS's `ST_Buffer()` and `ST_Within()` functions. If the number of these points including the geometry point is less than MinPts, then the geometry point is labelled as noise, else `ExpandCluster()`function is called, and a cluster is started. The `ExpandCluster()`function takes as input four parameters, namely, a geometry point, the ε-neighborhood points of the geometry point as an array, the number of the cluster as an integer, the ε-neighborhood, and MinPts. First, the inputted geometry point is added to the cluster. Next, all points within the ε-neighborhood of each of the inputted neighborhood point are determined and if this number is greater than or equal to MinPts, then these points are added to the inputted neighborhood points, and if the point is not in any cluster it is added to the cluster. This process is repeated until every unvisited point processed, and classified into a cluster or labelled as noise. As described earlier, a point can be in a cluster, and it can be a noise. So, all points which are stored as noise points and not belonging to any cluster are the outliers. The points along with information whether it belongs to cluster or noise is stored in a new table. Appendix B.17 illustrates the DBSCAN algorithm implementation using

PHP. The results are stored in the `outliers` table. Figure 4.12 shows the snapshot of the results of executing the DBSCAN algorithm for identifying outliers for MMSI identifier 205572000.



	mmsi integer	st_astext text	cluster_id character varying
1	205572000	POINT(29.660629 43.995079)	NOISE
2	205572000	POINT(24.616751 37.919552)	0
3	205572000	POINT(24.609819 37.90723)	0
4	205572000	POINT(24.623819 37.932308)	0
5	205572000	POINT(24.630581 37.944031)	0
6	205572000	POINT(24.63693 37.954811)	0
7	205572000	POINT(24.640619 37.961029)	0
8	205572000	POINT(24.64719 37.97192)	0
9	205572000	POINT(24.654039 37.98365)	0
10	205572000	POINT(24.660629 37.995079)	0

Figure 0.12: The outlier points from DBSCAN algorithm for the MMSI identifier 205572000

4. 9 Experimental Results

The proposed methods for detailed trajectory construction, simplified trajectory construction, distance calculation, speed calculation, direction calculation, sharp turn determination, and outlier detection were evaluated with two datasets, the raw location points from August 2012, and the raw location points from August 2013. First, the trajectory reconstruction algorithm was applied to these two datasets. Figure 4.13 illustrates the number of trajectories reconstructed in August 2012, and August 2013 as a bar graph. A total of 2,952 trajectories were reconstructed in August 2012 for a set of

1,709,308 points, and 7,910 trajectories were reconstructed in August 2013 for a set of 5,322,687 points.

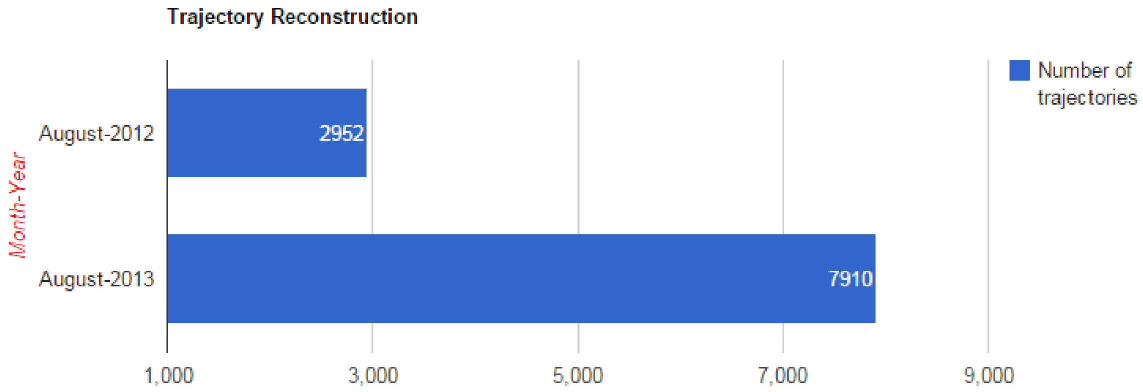


Figure 0.13: A bar graph showing the number of trajectories reconstructed in August 2012, and August 2013

The trajectory simplification algorithm was applied to two subsets of trajectories, namely, trajectories reconstructed for the months of August 2012, and August 2013. Figure 4.14 illustrates the result of the trajectory simplification process as a bar graph. In August 2012, 1,709,308 location points were simplified to 297,930, and in August 2013, 5,322,687 location points were simplified to 2,124,060.

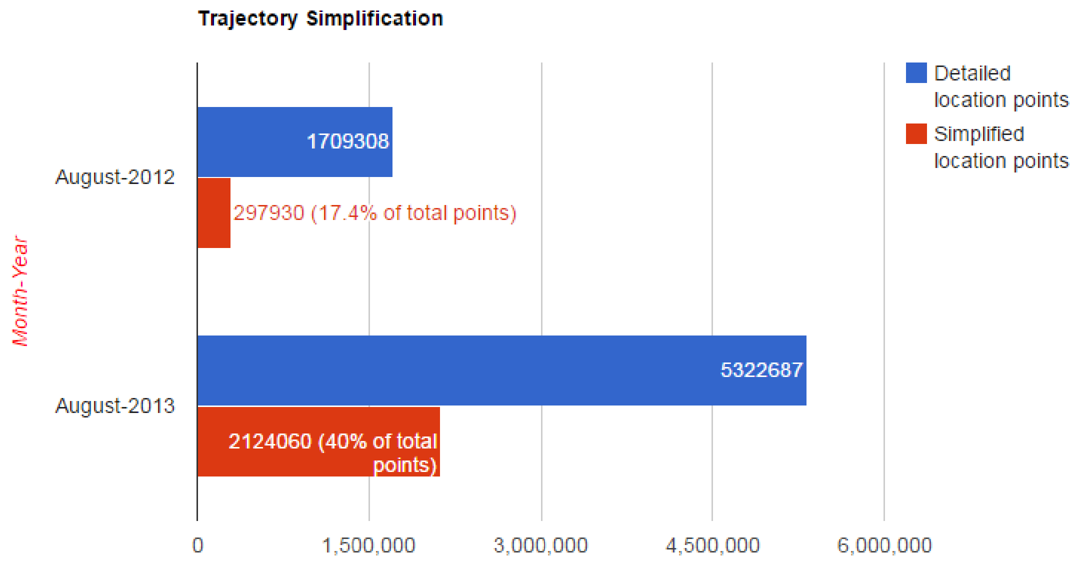


Figure 0.14: A bar graph showing the respective number of detailed and simplified location points for the months of August 2012, and August 2013

The total distance and the average speed of the trajectories were computed at different time periods, during August 2012, and August 2013. The total distance covered in an area indicates how busy the area is, and is a measure of shipping risk. Figure 4.15 illustrates the total distance, and the average speed of the trajectories computed during August 2012, and August 2013 as a bar graph. Here, 1,051,358.17 nautical miles of distance at an average speed of 8.2 nautical miles per hour was covered during August 2012, and 6,249,364.06 nautical miles of distance at an average speed of 4.9 nautical miles per hour was covered during August 2013.

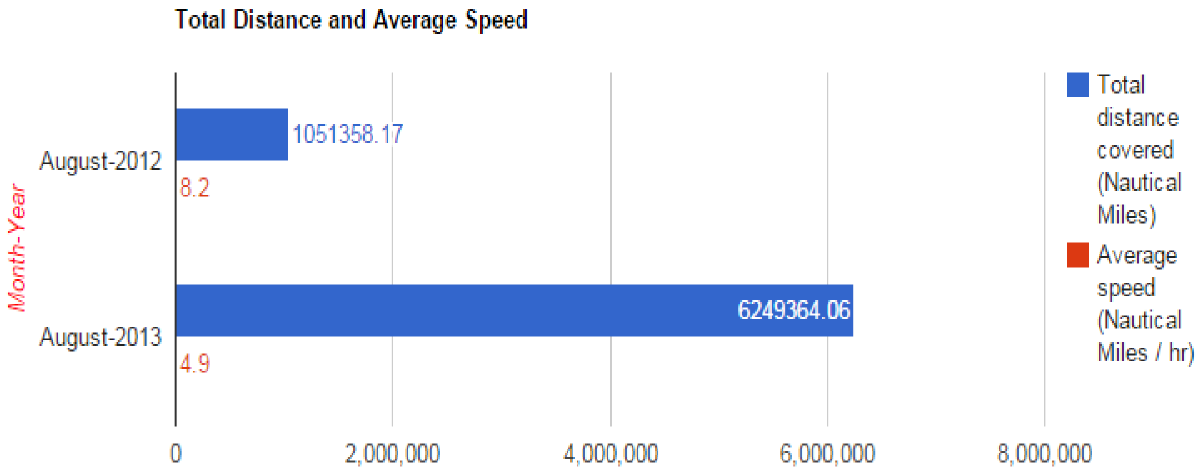


Figure 0.15: A bar graph showing the total distance and average speed of trajectories during August 2012, and August 2013

The trajectories that intersect the protected areas with two datasets with different time periods, August 2012, and August 2013 were computed. Among the 2,952 trajectories that were reconstructed in August 2012, 689 passed through the protected areas during August 2012, and among the 7,910 trajectories that were reconstructed during August 2013, 1,798 passed through the protected areas. Figure 4.16 illustrates the number of reconstructed trajectories that passed through the protected areas during August 2012, and August 2013 as a bar graph.

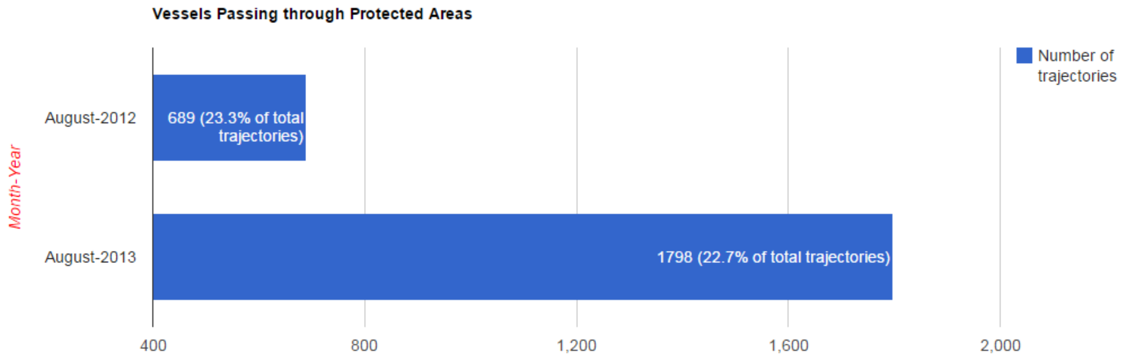


Figure 0.16: A bar graph showing the number of trajectories passing through the protected areas during August 2012, and August 2013

The number of sharp turns (turn angles greater than 10 and 20 degrees) was determined using the turn angle computation algorithm explained in section 4.5 for two datasets with different time periods, August 2012, and August 2013. In August 2012, 52,784 location points had turn angles greater than 10 degrees, and 27,913 location points had turn angles greater than 20 degrees. In August 2017, 102,586 location points had turn angles greater than 10 degrees, and 61,294 location points had turn angles greater than 20 degrees. These results are presented as a bar graph in Figure 4.17.

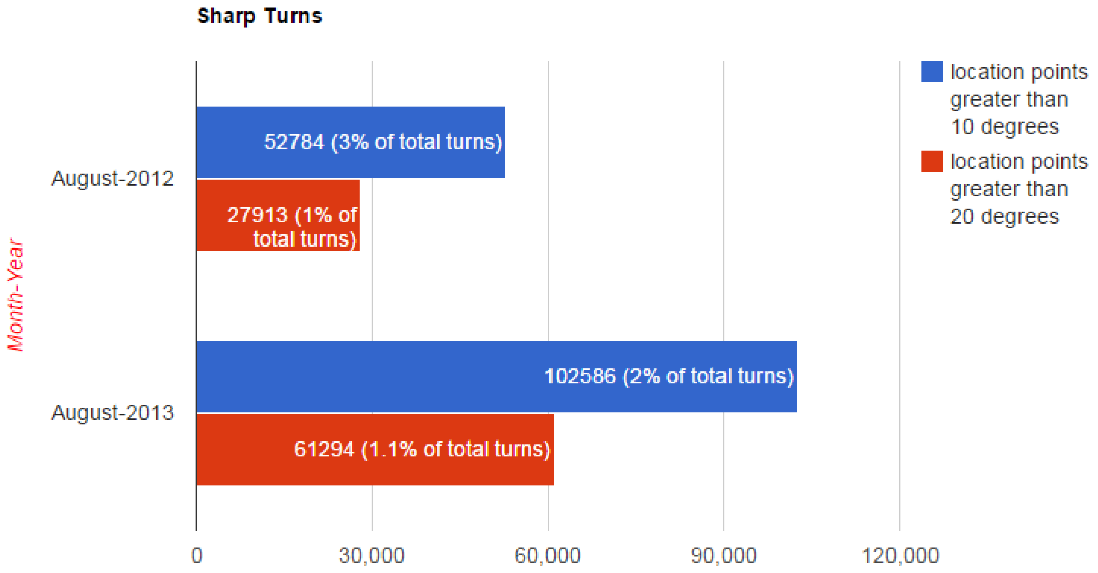


Figure 0.17: A bar graph showing the number of sharp turns during August 2012, and August 2013

4.10 Summary

This chapter discussed the procedure and implementation methodology employed for detailed and simplified trajectory construction, extracting statistics from the constructed trajectories, and detecting spatio-temporal outliers. In order to get the quantitative measures for evaluation, the algorithm for the statistical analysis and mining of the raw location points was applied on two different subsets of data at different time periods. Further, the results derived from applying the algorithms on the different data sets have been presented. The functionality for spatially enabled SQL queries were provided by PostGIS that utilized data that was stored in a PostgreSQL database. The resulting web application that was developed for the visualization of the results of the methodology is presented in the next chapter.

CHAPTER 5 VISUALIZATION

5.1 Introduction

This chapter describes the web application developed for visualization of the results of the statistical analysis methodology. The first section discusses the design and implementation of the web user interface. The second section discusses how AJAX was incorporated into the application, and its advantage in how it helps to improve usability, and add dynamic nature to the web application. The third section describes the various components of Google Maps API that was used for visualization. The fourth section discusses the implementation of the client-side scripts that use Google Maps API to visualize the following: (i) detailed, and simplified trajectories, (ii) distance, speed, direction, and turn angle of each intermediate point of the trajectories, (iii) vessels around the vicinity of a vessel at a particular instant in time, (iv) outlier, (v) vessels passing through protected areas, and (v) heat maps.

5.2 User Interface

The standard web technologies such as HTML, CSS, and JavaScript were used to develop the client side user interface. In addition, jQuery JavaScript API (The jQuery Foundation, 2013) was used to develop certain user interface elements and add AJAX capabilities to the application. jQuery JavaScript API is a cross-platform JavaScript API that has functions and methods that simplifies client-side scripting of HTML documents. jQuery provides the following: (i) a simplified syntax for Document Object Model (DOM) element selection and manipulation, (ii) capabilities for event handling, (iii) animations and effects, (iv) AJAX, and (v) multi-browser support. Some of the advantages of using jQuery over JavaScript include simplified syntax structure, small size of the jQuery JavaScript files, extensive extensions such as jQuery UI, excellent documentation, and great online support. The jQuery library is a single JavaScript (`.js`) file that is embedded in HTML pages. Appendix C.1 illustrates the way to include jQuery directly from the source.

The client side user interface allows users to select the various parameters, such as the MMSI identifier, and time period. Figure 5.1 shows the HTML form which allows users to select the MMSI identifier through a drop down menu that is populated from the data stored in the PostgreSQL database, the time period that uses the jQuery datepicker widget to display the calendar as a drop down for the visualization of the detailed, and simplified trajectories using the Douglas-Peucker algorithm. The form elements are constructed using HTML and Cascading Style Sheets (CSS3). Cascading Style Sheets (CSS) are a simple mechanism for adding style (e.g., fonts, colors, spacing) to web documents (W3C, 2014). CSS3 provides several new design elements which were used to design the form. These elements are (i) `border-radius` property, which was used to add rounded borders to the `input` HTML elements, (ii) `opacity` property, which is used to control the transparency of the background. The values submitted in this form are captured and sent as a GET request using jQuery's AJAX capability to the PHP file that processes the request to generate the trajectories, and sends back the data as a JSON encoded array.

jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library (The jQuery Foundation, 2014). jQuery UI simplifies client-side scripting and facilitates client-side user interface to include complex widgets such as date picker, slider bar, etc. These widgets enhance the interactivity of the web application. The date picker widget used in the application facilitates users to select the required data from the calendar, thus eliminating the need for the user to enter the date manually.

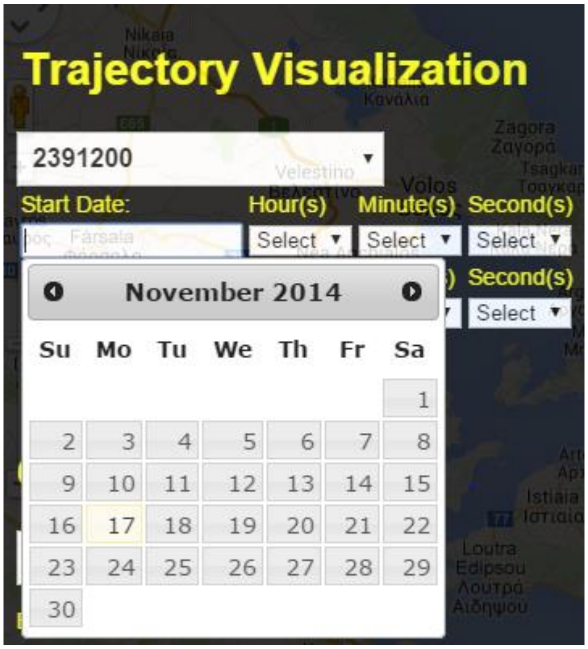


Figure 0.1: Trajectory visualization user interface

5.3 AJAX Calls through jQuery Library

The jQuery library provides methods and functions for AJAX capabilities using the `jquery.ajax()` method, thereby facilitating the client to make calls to the server asynchronously. Thus, the request is handled in the background and user interaction is not hampered. This is the backbone for visualizing geometry objects using Google Maps. Table 5.1 illustrates the parameters required for making AJAX calls through the jQuery library. Appendix C.2 shows an example of the `$.ajax()` function to communicate with the server using the POST HTTP method by sending the following: (i) data to the server as a JSON object, (ii) the specific URL to hit, (iii) a success callback wherein the data would be processed and manipulated.

Table 5.1: Parameters required for making AJAX calls through jQuery

Parameter	**Description**
`type`	The string of the type of the HTTP request (GET / POST).
`url`	The string of the URL to which the request has to be sent, usually a PHP script.
`data`	The data that is to be sent to the server. The data can be sent contain either a query string of the form `key1=value1&key2=value2` if the request is a GET, or an object of the form `{key1: 'value1', key2: 'value2'}` if the request is a POST.
`datatype`	The string of the type of data that is received from the server (e.g., xml, json, html, text, etc.).
`cache`	A Boolean to indicate whether to cache the results by the browser.
`success`	The function to be called if the request is successful.

5.4 Dynamic Web Visualization Using Google Maps API

The visualization and exploration platform for the historic vessel data is created using JavaScript, AJAX and the Google Maps API. AJAX is leveraged in this application to handle the transaction between client and the server to be asynchronous, and thereby client interaction is not limited during processing by the server. This allows updating of a portion of the web page without reloading the entire page. Figure 5.2 illustrates the flow diagram of a typical AJAX call made by passing data from the client to the server and back. On the client-side, an `XMLHTTPRequest` object is created using JavaScript and data sent to the server through GET or POST requests. The server-side gets the request with the user-entered parameters as a JSON object and generates a JSON encoded array which contains the data from a SQL query in response to the request using PHP.

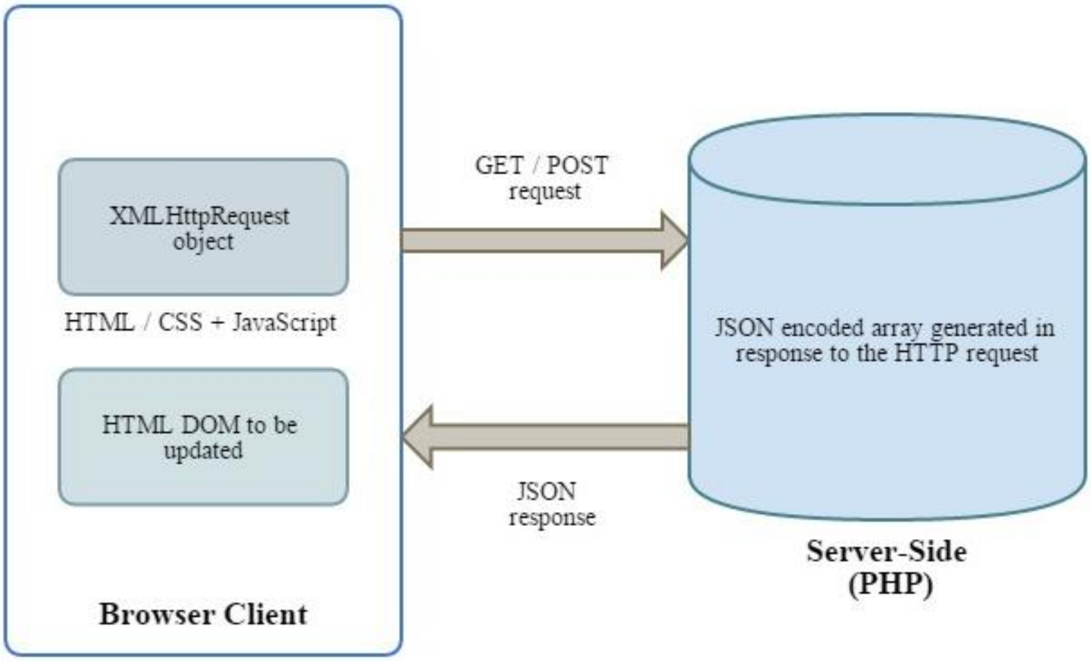
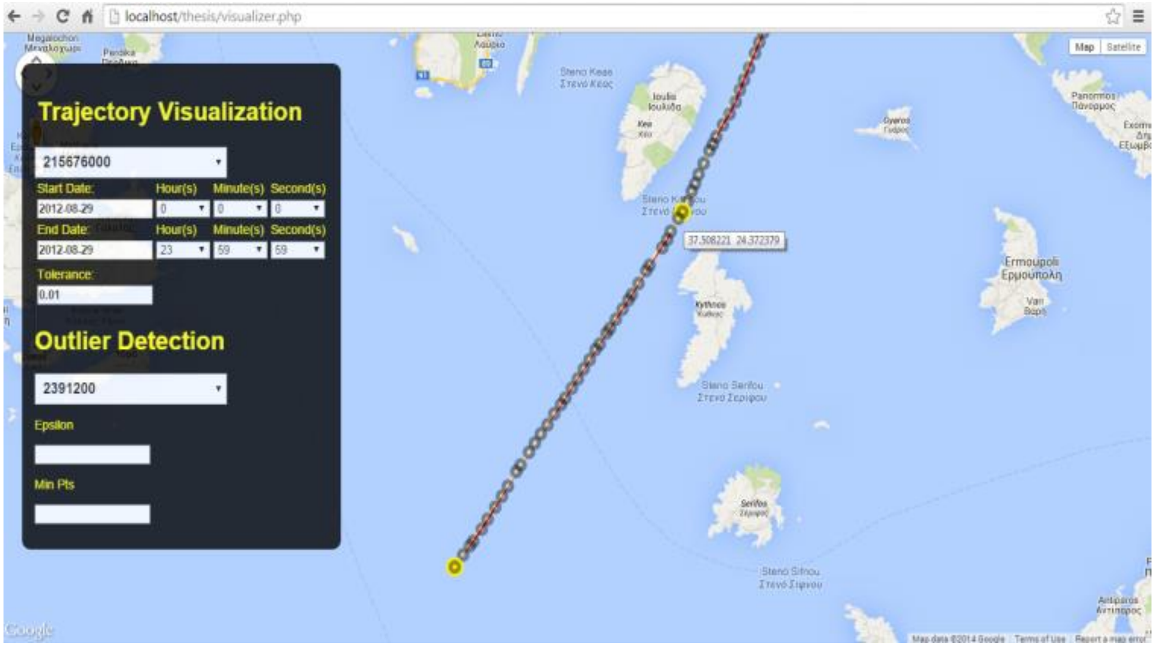


Figure 0.2: Flow diagram of a typical AJAX call

5.4.1 Visualizing Detailed and Simplified Trajectory

The user defines the parameters such as the MMSI identifier, start date, end date, and the tolerance value by which the Douglas-Peucker algorithm computes the simplified trajectories through the trajectory visualization user interface. The client jQuery function concatenates the parameter's names and values into a data string and passes it as a POST request through an AJAX call into the PHP script illustrated in Appendix C.3. The PHP script gets the query parameters, and connects to the PostgreSQL database using the `pg_connect()`function by passing in the names of the PostgreSQL host, database, user and password. The points data from the LOCATION_SHIPS table are extracted which is used to construct the detailed trajectory. The simplified trajectory constructed using the Douglas-Peucker algorithm is stored in the route_simplified table is extracted. The extracted values containing details about latitude / longitude coordinates of the simplified and the detailed trajectory is sent to client as a JSON encoded array. Appendix C.3 shows the jQuery function that performs the AJAX call. The JSON encoded array containing the latitude / longitude coordinates of the simplified and the detailed trajectory received from the PHP script is iterated over and the latitude / longitude coordinates is converted into an array of Google Maps points using the google.maps.LatLng() function. The points array for the detailed and simplified trajectory extracted is passed into the path property of the google.maps.Polyline()function to construct the trajectories and the DOM element that defines the Google Maps container is updated, and thereby page reloading is prevented. In addition to setting the path property, other properties for the PolylineObject such as the strokeColor, strokeOpacity and strokeWeight is set. To differentiate between the simplified and detailed trajectory, different colors were used. In addition to creating the trajectory, markers for the individual points comprising the detailed and simplified trajectory are created to differentiate between the detailed and simplified trajectory using the google.maps.Marker()function.

Figure 5.3 illustrates the simplified and the detailed trajectory for MMSI identifier 215676000 on 29-August-2012 with the tolerance set to 0.01. The red line with the black markers indicates the detailed trajectory, and the yellow line with yellow markers indicates the simplified trajectory. With tolerance set to 0.01 the Douglas-Peucker algorithm has simplified the number of points to three.



localhost/thesis/visualizer.php

Trajectory Visualization

215676000

Start Date: 2012-08-29 Hour(s) 0 Minute(s) 0 Second(s) 0

End Date: 2012-08-29 Hour(s) 23 Minute(s) 59 Second(s) 59

Tolerance: 0.01

Outlier Detection

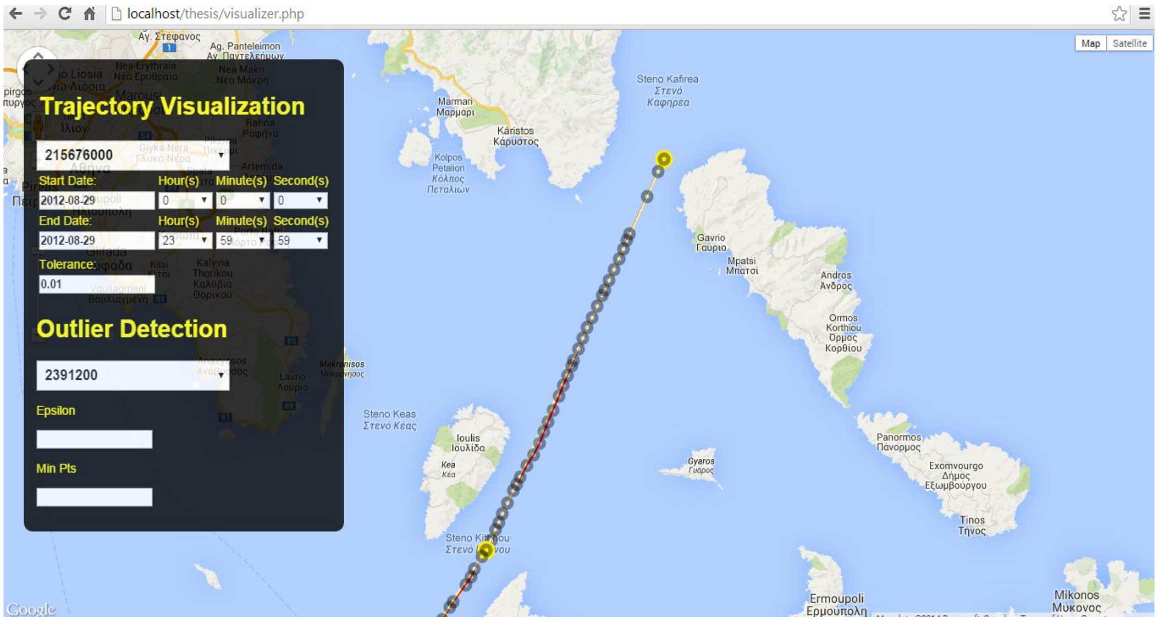
2391200

Epsilon

Min Pts

37.508221 24.372379

(a)



localhost/thesis/visualizer.php

Trajectory Visualization

215676000

Start Date: 2012-08-29 Hour(s) 0 Minute(s) 0 Second(s) 0

End Date: 2012-08-29 Hour(s) 23 Minute(s) 59 Second(s) 59

Tolerance: 0.01

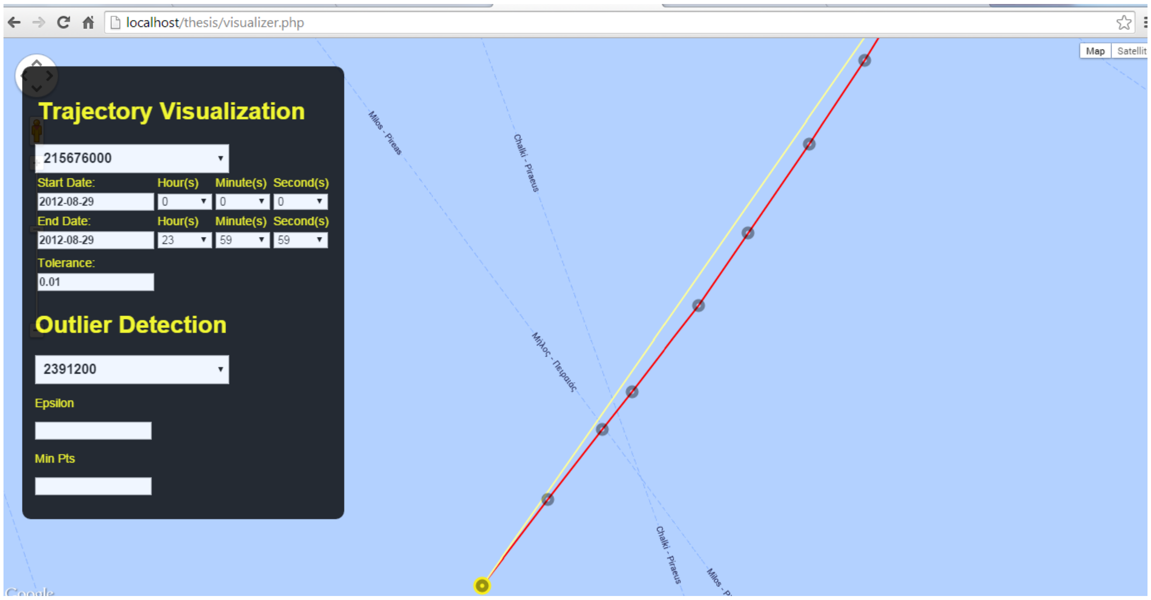
Outlier Detection

2391200

Epsilon

Min Pts

(b)



Trajectory Visualization

215676000

Start Date:	Hour(s)	Minute(s)	Second(s)
2012-08-29	0	0	0
End Date:	Hour(s)	Minute(s)	Second(s)
2012-08-29	23	59	59

Tolerance: 0.01

Outlier Detection

2391200

Epsilon

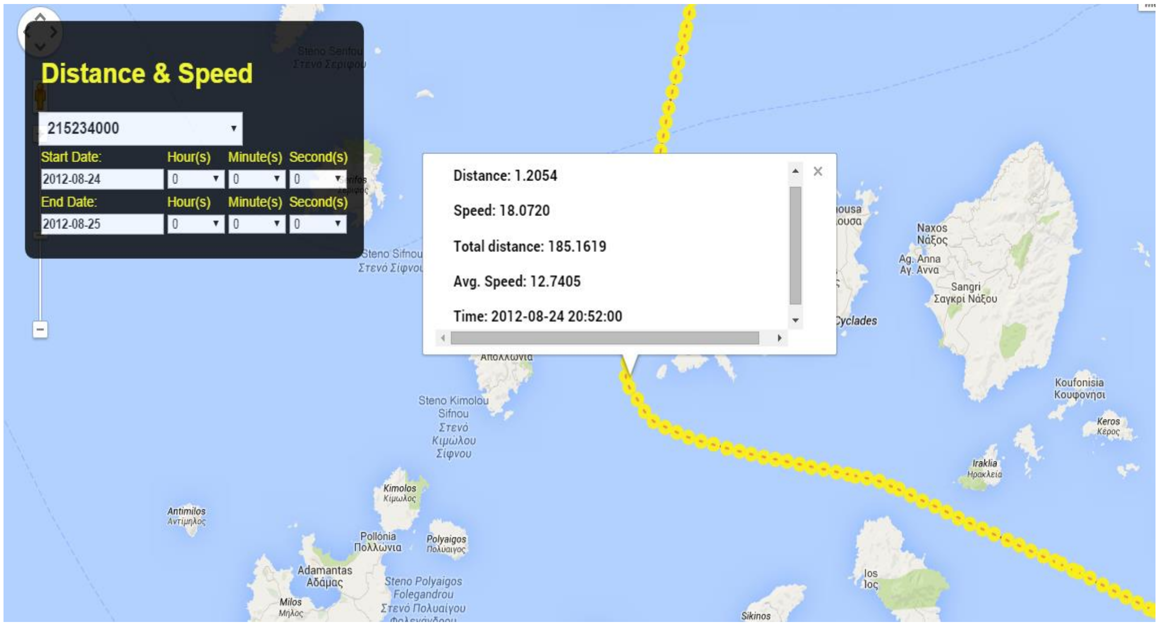
Min Pts

(c)

Figure 0.3: Detailed and simplified trajectory visualization using Google Maps API. The red line indicates the detailed trajectory, and the yellow line indicates the simplified trajectory

5.4.2 Visualizing Distance, Direction, Speed, and Turn Angle

Figure 5.4 illustrates an example of visualizing distance, direction, speed, and turn angle at each intermediate point of a trajectory through Google Maps InfoWindow object. The functions that determine the distance and speed, direction, and turn angle are `distanceandspeed`, `direction`, and `turn_angle` respectively. These functions are executed through SQL queries by PHP and the result of these queries are returned as a JSON-encoded array. The client, through JavaScript, processes the result, and displays the distance, direction, speed, and turn angle as within Google Maps infowindows. Appendix C.4 illustrates the JavaScript code fragment that creates the markers and info windows displaying the direction, major direction, and time for each intermediate point in a trajectory. Here, `darray` is a multi-dimensional array that contains details about the latitude / longitude coordinates, time, direction, and major direction that is parsed from the JSON-encoded array. This array is iterated over and markers created at each latitude / longitude coordinate using the `google.maps.LatLng()` function. The icon used for these markers are defined in the icon property (`google.maps.SymbolPath.CIRCLE`). The `title` is the property that sets the text to be displayed when the mouse is hovered over the marker. This `title` property is set to the latitude / longitude coordinate. A click event is created using the `google.maps.event.addListener()` function which displays a info window displaying the direction, major direction, and time at which the coordinate was recorded. Figure 5.4 (a) shows the distance, and speed, 5.4 (b) shows the direction, and major direction, and 5.4 (c) shows the turn angle of each intermediate point of MMSI identifier 215234000 between 24-Aug-2014 and 25-Aug-2014.



Distance & Speed

215234000

Start Date: 2012-08-24 Hour(s) 0 Minute(s) 0 Second(s) 0

End Date: 2012-08-25 Hour(s) 0 Minute(s) 0 Second(s) 0

Distance: 1.2054

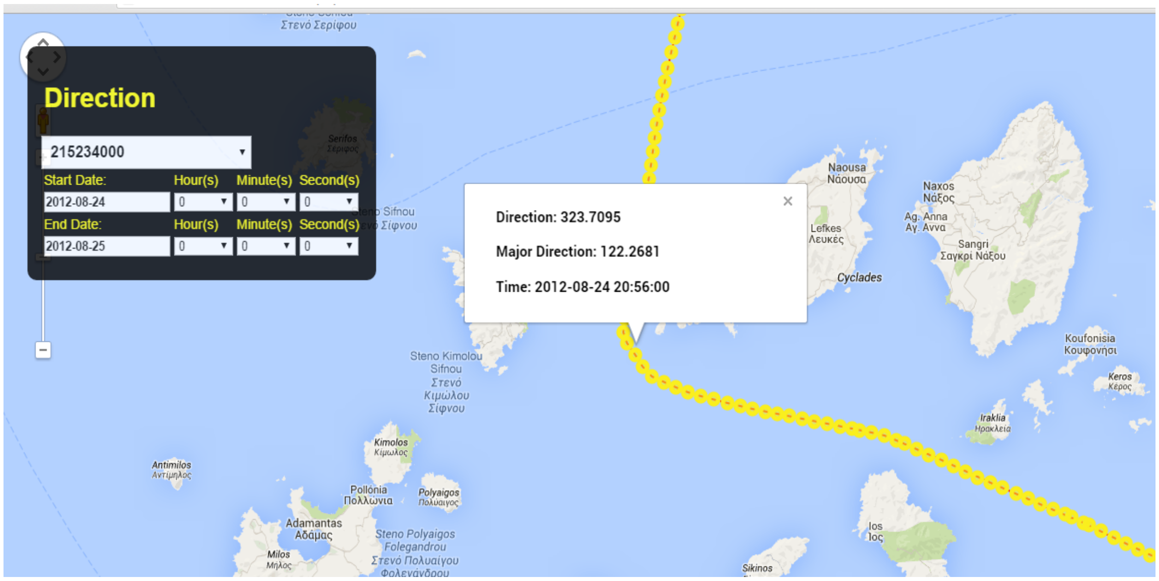
Speed: 18.0720

Total distance: 185.1619

Avg. Speed: 12.7405

Time: 2012-08-24 20:52:00

(a)



Direction

215234000

Start Date: 2012-08-24 Hour(s) 0 Minute(s) 0 Second(s) 0

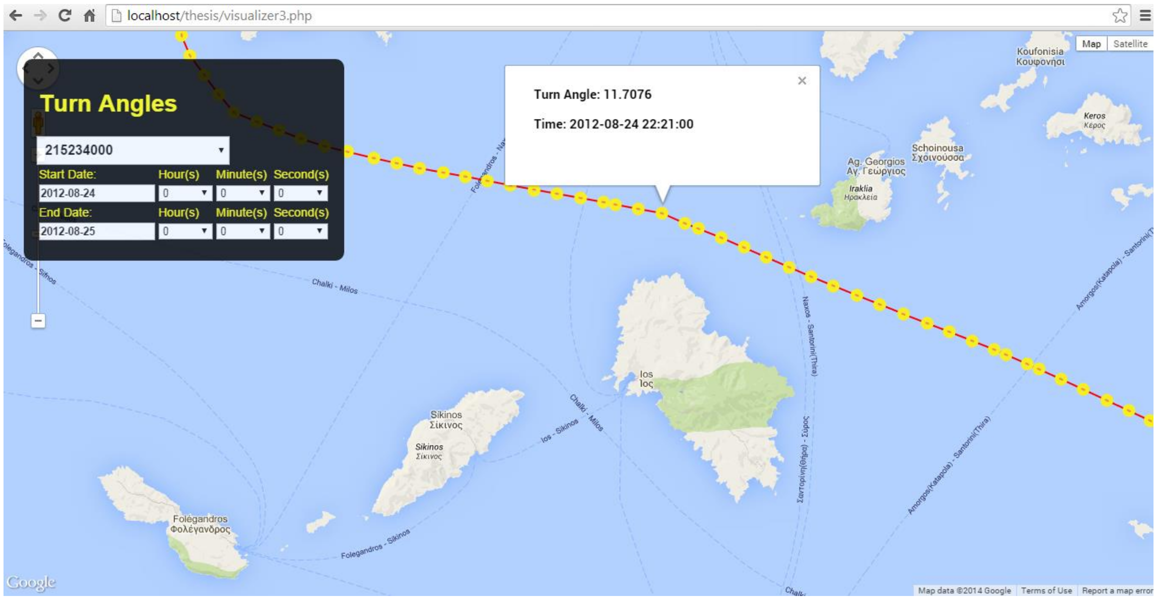
End Date: 2012-08-25 Hour(s) 0 Minute(s) 0 Second(s) 0

Direction: 323.7095

Major Direction: 122.2681

Time: 2012-08-24 20:56:00

(b)


localhost/thesis/visualizer3.php

Turn Angles

215234000

Start Date: 2012-08-24 Hour(s) 0 Minute(s) 0 Second(s) 0

End Date: 2012-08-25 Hour(s) 0 Minute(s) 0 Second(s) 0

Turn Angle: 11.7076

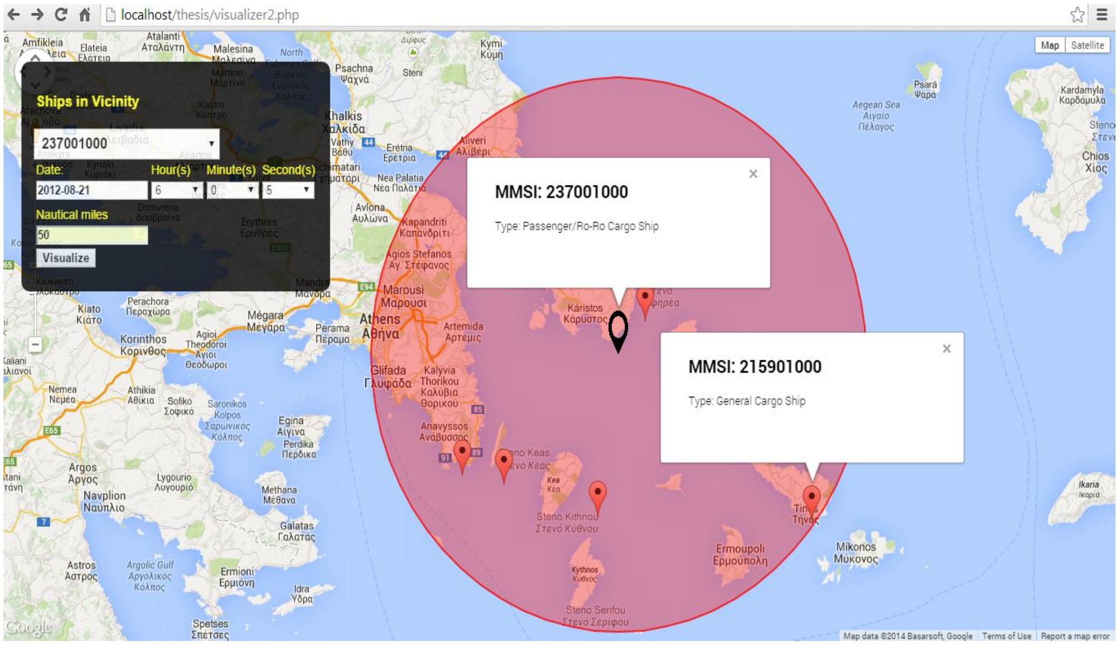
Time: 2012-08-24 22:21:00

(c)

Figure 0.4: Markers displaying (a) the distance and speed, (b) the direction and major direction, and (c) turn angle of each of each intermediate point of MMSI identifier 215234000 between 24-Aug-2014 and 25-Aug-2014

5.4.3 Visualizing Ships around the Vicinity of a Ship

Figure 5.5 gives an example for visualizing ships around the vicinity of a ship. Here, ships around the vicinity of 50 nautical miles of MMSI identifier 237001000 are identified and displayed as markers on Google Maps. Two steps were required to achieve this as described in section 4.7. First, a buffer is defined according to the user-inputted radius using the `ST_Buffer(geometry A, float radius_of_buffer)` function. The resulting buffer is of type polygon geometry and is retrieved from a PHP script after an AJAX call (with all the necessary user-defined parameters set) is made. This geometry is rendered on Google Maps using the `google.maps.Polygon` function. Second, the specific vessels that were identified to be located around the user-inputted MMSI identifier is computed through PostGIS query as described in section 4.6 and the resulting latitude / longitude coordinates of these vessels are sent as a JSON-encoded array to the client. In addition, the ship type is also sent to the client. A marker will be shown on the map based on the coordinates of each vessel using the `google.maps.Marker` function along with its type through the `InfoWindow` object.



localhost/thesis/visualizer2.php

Ships in Vicinity

237001000

Date: 2012-08-21

Hour(s): 6

Minute(s): 0

Second(s): 5

Nautical miles: 50

Visualize

MMSI: 237001000

Type: Passenger/Ro-Ro Cargo Ship

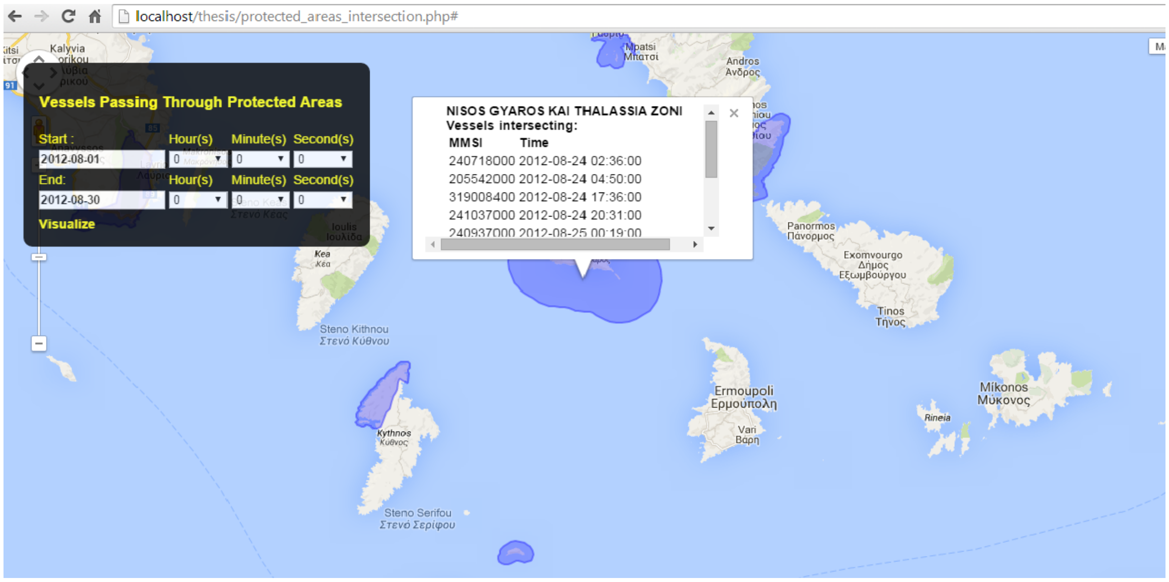
MMSI: 215901000

Type: General Cargo Ship

Figure 0.5: Ships around the vicinity of 50 nautical miles of MMSI identifier 237001000

5.4.4 Visualizing Vessels Intersecting Protected Areas

Section 4.6 explained how to determine the vessels that pass through EU Natura 2000 protected areas. Each resulting record consists of the name of the protected area, the geometry of the protected area, the vessels, and the time each vessel passes through the protected area as a comma-seperated string. Figure 5.6 shows the vessels that pass through a particular protected area between the time period 01-Aug-2012 and 30-Aug-2012. Here, on clicking a polygon the MMSI identifiers that pass through that polygon are listed as a table. Two steps were employed to achieve this visualization. First, the geometry of the protected areas were fetched as a JSON-encoded array from a PHP script that queries the geometry from the `PROTECTED_AREAS` schema. This JSON-encoded array containing the geometry of each proctected area is parsed and the OGC latitude / longitude coordinate is converted to Google Maps coordinate using the `google.maps.LatLng()` function using JavaScript. Each polygon representing the protected area thus would contain an array of `google.maps.LatLng()`points. An array of polygons is created to store the geometries of the polygons. This array is iterated over, and using the `google.maps.Polygon()` function, the polygons representing the protected areas were constructed as overlays on the map instance. Second, the comma-seperated vessel MMSI identifier and the time is parsed using the `split()` function and displayed as a table through info windows on the map instance.



localhost/thesis/protected_areas_intersection.php#

Vessels Passing Through Protected Areas

Start : 2012-08-01 Hour(s) 0 Minute(s) 0 Second(s) 0

End: 2012-08-30 Hour(s) 0 Minute(s) 0 Second(s) 0

Visualize

NISOS GYAROS KAI THALASSIA ZONI
Vessels intersecting:

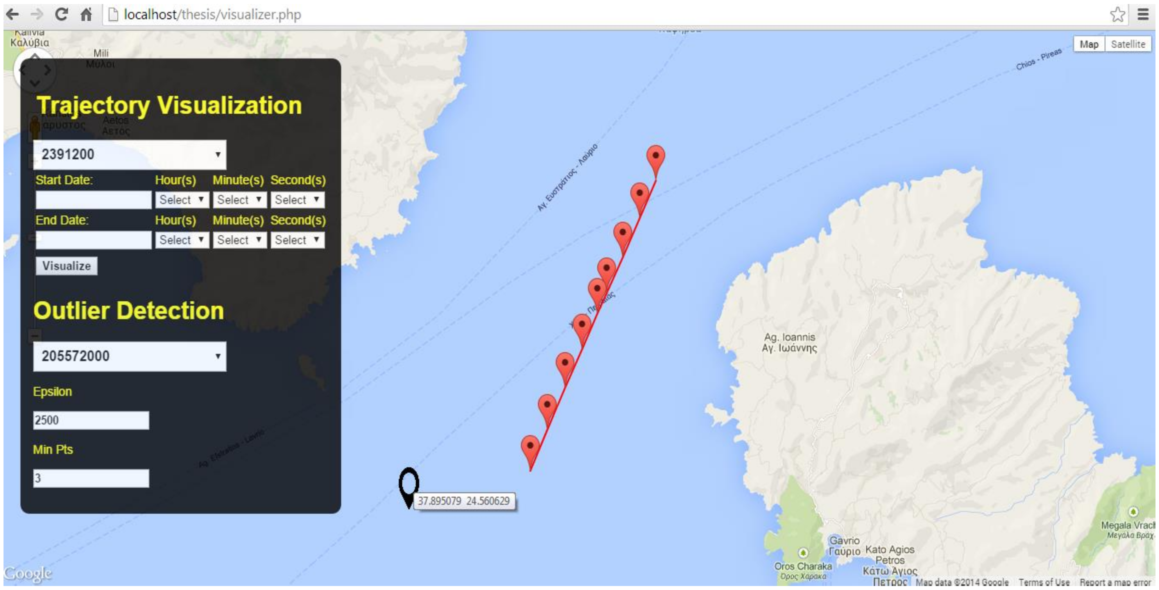
MMSI	Time
240718000	2012-08-24 02:36:00
205542000	2012-08-24 04:50:00
319008400	2012-08-24 17:36:00
241037000	2012-08-24 20:31:00
240937000	2012-08-25 00:19:00

Figure 0.6: Vessels passing through EU Natura 2000 protected areas using Polygon, and InfoWindow objects

5.4.5 Visualizing Outliers

Section 4.8 explained the DBSCAN algorithm for outlier detection. Based on the user-inputted MMSI identifier, the epsilon, and minimum points, which are the input parameters for DBCAN algorithm, the cluster id / outlier to which the points comprising the trajectory of the MMSI identifier are computed and stored in the `outliers` database. The on change event which is called when the user has inputted all the required inputs -- namely, the MMSI identifier, the epsilon, and minimum points -- triggers an AJAX call that retrieves a JSON-encoded array from a PHP script that queries the geometry, and the cluster / outlier from the `outliers` schema .

The client JavaScript iterates over the JSON-encoded array, and converts the OGC geometry to a Google Maps geometry using the `google.maps.LatLng()` function. The `cluster_id` column in `outlier` table indicates whether a geometry is a noise or whether it belongs to a cluster. Based on this value, a multidimensional array was created to store the geometries belonging to different cluster and the geometries that are noise. The geometries identified as noise are indicated using a different marker using the `icon` property in the `MarkerOptions` object. Figure 5.7 illustrates an example to visualize the outlier points for the MMSI identifier 20557200 with the epsilon set to 2.5 nautical miles and minimum points set to 3.



localhost/thesis/visualizer.php

Trajectory Visualization

2391200

Start Date: Hour(s) Select Minute(s) Select Second(s) Select

End Date: Hour(s) Select Minute(s) Select Second(s) Select

Visualize

Outlier Detection

205572000

Epsilon

2500

Min Pts

3

Figure 0.7: Outlier visualization using Google Maps API

5.4.6 Visualizing heat maps

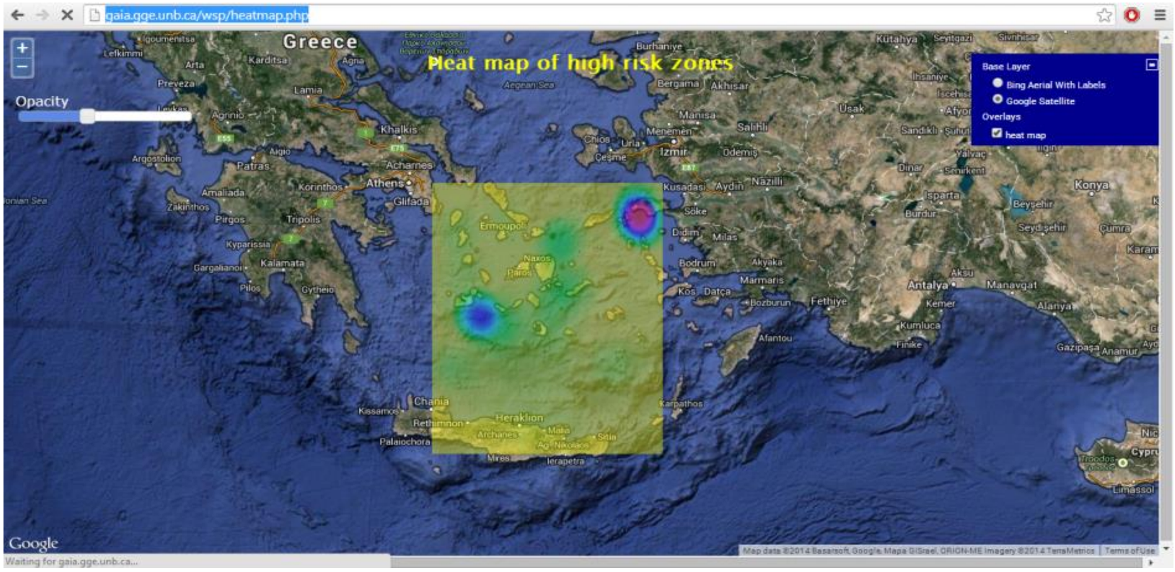
A heat map is a visualization technique that allows decision makers to identify key areas of environmental risk zones and where most interaction has taken place. The users can produce heat maps for different time periods and compare the results. The heat map is rendered as an overlay on top of the base map, the Google Maps. The areas of higher intensity will be colored red, and areas of lower intensity will appear green. The users can zoom in and out to visualize the relative changes at different map scales. Various technologies provide capabilities to process the vessel position data to generate the heatmap either through the client-side, or through the server-side. Inverse distance weighting is the simplest interpolation technique that defines a neighborhood as weighted average of the observation values within this neighborhood. This interpolation technique is used to create the heat maps.

OpenLayers was used to render the heat map through the server-side. OpenLayers supports OGC WMS specification and can display and manipulate geospatial data without server-side dependencies. In conjunction with GRASS GIS, a PyWPS script that generates the heat map is disseminated as a WMS through GeoServer. OpenLayers provide the capability to serve the WMS as an overlay on top of Google Maps. Using the `v.surf.idw` command, the environmental risk zones were determined. The point snapshot data containing the location of the vessels were passed to the `v.surf.idw` GRASS command. The resulting raster in GeoTIFF is served as a WMS layer in

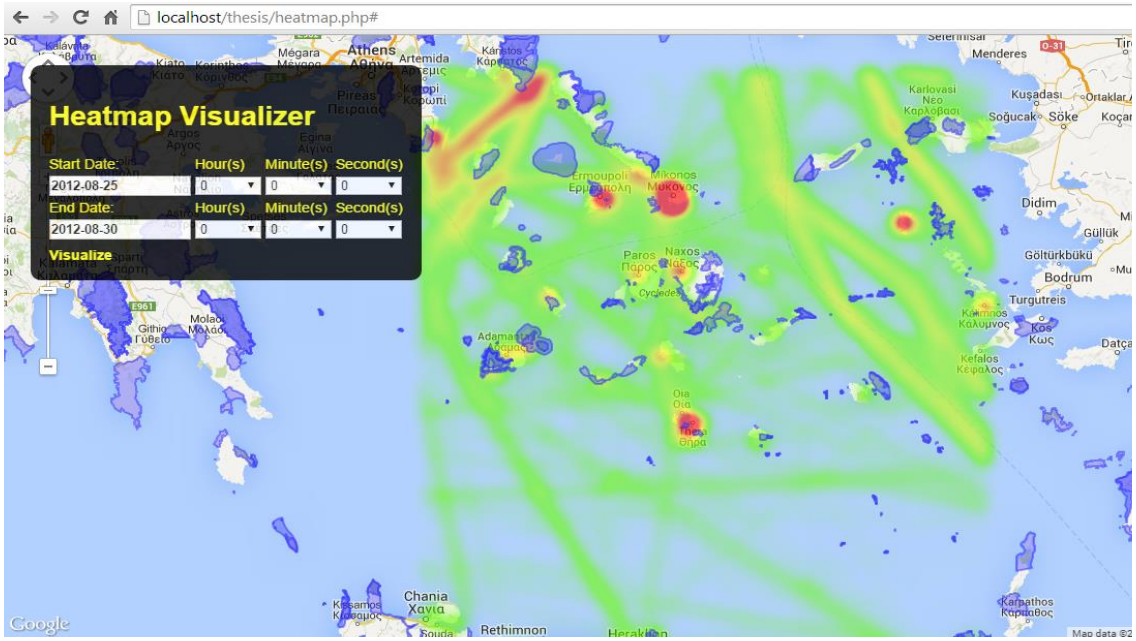
GeoServer and visualized using OpenLayers JavaScript API (Figure 5.8 (a)). Here, the users would be able to adjust the transparency of the layer, thereby facilitating comparison with different base maps. In addition, the opacity of the base layer could also adjusted using the opacity tool developed through jquery UI.

Google Maps API provides the capability to process geographical points and render the heat map client-side by on-the-fly calculation through the Heatmap Layer. The API provides several customization options to change the color gradient, radius of the points, and intensity of the points. The Heatmap Layer is part of the `google.maps.visualization` library that contains various methods for data visualization. This library is loaded through JavaScript. To initiate the Heatmap Layer, the `HeatMapLayer` object has to be initiated by passing in to this object an array of `google.maps.LatLng` objects, which contains the geographical points for which the heat map has to rendered. The rendered is georeferenced on-the-fly and rendered as an overlay on top on Google Maps. Figure 5.8 (b) illustrates the client-side rendering of a heat map between ’25-Aug-2012’ and ’30-Aug-2012’.

Appendix C.5 illustrates how to add a OGC WMS layer using OpenLayers JavaScript API. Any number of layers could be overlayed. This example shows a single WMS layer overlayed on top of Google Maps satellite image. Here, the variable map object defines the layers that need to be added to the map `div` element. The variable wms creates a new WMS layer object of name `'heat map'` using the `OpenLayers.Layer.WMS()` function. The location of the WMS layer on GeoServer is defined by the URL of the host. In addition, the width, height, the spatial reference system, the layer name on GeoServer, format, the transparency, and the background color of the layer is defined. It should be noted that the spatial reference system that facilitates map overlaying through OpenLayers and other web mapping APIs such as Google and Bing Maps is the Spherical Mercator and this is defined by setting the `srs` property using the EPSG code 900913. Setting the `transparent` property to be "True" allows jQuery to adjust the transparency of the WMS layer. The `maxExtent` property defines the extent of the WMS layer that is extracted and sent to the client. The unit in which the `maxExtent` property is defined corresponds to the units of the coordinate system. The format property specifies the WMS layers overlayed on top of Google Maps is of PNG format. The `isBaseLayer` property specifies whether the layer is a baselayer or an overlay layer on top of baselayer. The `addLayers` method takes an array of map layers that are to be superimposed.


gaia.gge.unb.ca/wsp/heatmap.php
Heat map of high risk zones
Opacity
Base Layer
○ Bing Aerial With Labels
● Google Satellite
Overlays
☑ heat map
Greece
Aegean Sea
Athens
Izmir
Konya
Heraklion
Google
Waiting for gaia.gge.unb.ca...

(a)



localhost/thesis/heatmap.php#

Heatmap Visualizer

Start Date:	Hour(s)	Minute(s)	Second(s)
2012-08-25	0	0	0

End Date:	Hour(s)	Minute(s)	Second(s)
2012-08-30	0	0	0

(b)

Figure 0.8: Heat map of high risk zones using (a) OpenLayers and GRASS, and (b) JavaScript and Google Maps API

5.5 Summary

This chapter presented the design and implementation of the web application for visualization. The web application contains a wide array of features presented with a simple user interface. The architecture was created using open standards to achieve interoperability. The web application uses Google Maps API for visualizing the results of the statistical analysis discussed in Chapter 4, and OpenLayers API for disseminating WMS layers. However, other web mapping APIs such as ArcGIS JavaScript API could be used for visualization. Further, by incorporating AJAX into the web application for client-server communication, the application runs faster and has responsiveness closer to that of desktop applications (Puder, 2006).

CHAPTER 6 CONCLUSIONS

6.1 Summary

This thesis presents an efficient framework and data model for representing, storing, and querying historical AIS data. The data model developed for this thesis is flexible and can represent complex spatial data types. The data model was designed by taking into consideration the application usage of the data (i.e., queries, updates, and processing of the data).

A web-based GIS application was developed for the visualization of the vessel trajectories and the results of the statistical information processing. This is achieved by developing a methodology for (i) the storage of historical AIS data by developing a parsing software for converting raw coordinates from a CSV file to database records and (ii) querying the historical AIS data to extract useful statistical information. Administrators of the system can upload CSV files containing the coordinate details of vessels. The vessel trajectories are created automatically after the file upload and by means of stored procedures written in PostgreSQL and Google Maps API. The following were made available for visualization: (i) distance, speed, direction, and turn angle; (ii) trajectories passing through EU Natura 2000 protected areas; (iii) vessels in the vicinity of the vessel; and (iv) heat maps for the extraction and analysis of interesting patterns.

6.2 Thesis Contributions

Web applications such as MarineTraffic (MarineTraffic, 2014), ShipFinder (Pinkfroot, 2014), and MyShipTracking (MyShipTracking, 2014) that track vessels based on AIS data and displays real time ship positions has no capability for visualizing and mining of historical AIS data. However, historical AIS data since 2009 is available in MarineTraffic (MarineTraffic, 2014) and can be ordered for a price. This thesis contributes by describing a framework and data model for the organization, storage, visualization, and data mining of past historical AIS datasets. Further, a web-based GIS application to analyse and visualize historical AIS datasets has been implemented that facilitates the exploration of the spatial and temporal dimensions. Further, data mining algorithms have been integrated into the visualization.

The components and protocols of a client-server based architecture used in the implementation of the web application have been presented. The architecture makes the application secure and scalable since it reduces the overhead caused by placing the components on the client-side alone. By creating a standard client-server architecture which consists of nine main components with unique roles, the time to develop new features is decreased as integrators can focus on the specific requirements of the new features.

The web-based GIS application developed follows the standards of Web 2.0. In Web 2.0, acquiring data from users, following open standards, and enhancing user interaction are the main characteristics. The web-based GIS application allows spatial data to be uploaded by users and compatibility in terms of data format and data type has been addressed. In addition, open standards and data protocols were used to develop the application to ensure interoperability between the components of the architecture. Further, by incorporating AJAX into the web application, an efficient mechanism to disseminate large digital spatial data with fast response time and enhanced user interaction has been developed. The network traffic cost is reduced and displays the trajectories and results of the statistical analysis faster by applying AJAX techniques.

6.3 Limitations

Several limitations of this research are noted.

(i) Due to difficulties in collecting ground truth data, validating the results with ground truth data has not been performed in this research.

(ii) The data acquisition and conversion of the CSV format consisting of the raw coordinates of the vessels into a spatial database is a very time consuming process. This could be prevented by feeding the real time data acquired in web based real time AIS services such as MarineTraffic (MarineTraffic, 2014), ShipFinder (Pinkfroot, 2014), and MyShipTracking (MyShipTracking, 2014) directly into a spatial database.

(iii) Due to the fact that none of the major web browsers, including Google Chrome, Mozilla Firefox, and Microsoft Internet Explorer, support all features of CSS, the user interface of the web application developed in this thesis is not rendered consistently across the major browsers. Testing the web application in different web browsers was not performed.

(iv) A major problem of large spatial data sets is the uncertainty in the geographic data. As a result, the credibility of the data analysis and processing remains questionable. This thesis does not provide a framework to quantify and visualize uncertainty in the geographic data.

(v) The web application developed focuses on historical AIS data, and real time visualization and statistical analysis of trajectories has not been implemented in this thesis.

6.4 Recommendations for Future Research

The thesis describes a framework and a web application prototype for storing, querying, and mining historical AIS data. The following areas are identified for future research by which efficient spatial data mining and processing of the large spatial data could be addressed:

(i) Integrate spatial databases such as PostgreSQL / PostGIS with Hermes Moving Object Database (MOD) engine (Pelekis et al., 2011). The

Hermes MOD engine is designed to efficiently process and mine moving object trajectories. Hermes MOD engine support several data types that allow spatial, temporal, and spatio-temporal queries. One of the major advantages of MODs over spatial databases is the efficient processing of topological queries. Topological queries answer questions such as: "Which are the vessels that entered a particular port?", "Which vessels crossed a particular canal?", and "Which vessels started at a particular port and ended at a particular port?".

(ii) Integrate Hadoop (Apache Software Foundation, 2014) with PostgreSQL. Hadoop File System (HDFS) stores large data sets by partitioning the file systems into data blocks and then replicates these blocks into several data nodes. Hadoop allows SQL queries to be executed on the HDFS data. By setting up multiple nodes within Hadoop clusters the processing time of SQL queries is drastically reduced. Thus, by leveraging Hadoop with PostgreSQL, processing of large spatial data sets within minutes could be achieved by distributing to several clusters.

(iii) Implement within the web application hierarchical clustering algorithms for mining periodic behaviors, and identifying periodic patterns in the vessel trajectories.

(iv) Incorporate within the web application clustering algorithms to identify congested routes in the seas.

(v) Incorporate within the web application mechanisms for updating of the trajectories of moving objects in real-time.

REFERENCES

Apache Software Foundation (2014). *Apache HTTP Server Version 2.2 Documentation*. Retrieved 02/10, 2014, from http://httpd.apache.org/docs/2.2.

Apache Software Foundation (2014). *Apache Hadoop 2.6.0.*. Retrieved 06/01, 2015, from http://hadoop.apache.org/docs/current/.

Arctur, D., & Zeiler, M. (2004). *Designing Geodatabases: case studies in GIS data modeling* (Vol. 380). Redlands, CA: Esri Press.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide*. Pearson Education India.

Chen, P. P. S. (1976). The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, *1*(1), 9-36.

Douglas, D. H., & Peucker, T. K. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, *10*(2), 112-122.

Eckerson, W. W. (1995). Three tier client/server architectures: achieving scalability, performance, and efficiency in client/server applications. *Open Information Systems*, *3*(20), 46-50.

Evenden, G., & Warmerdam, F (1990). *Proj. 4–Cartographic Projections Library*. Retrieved 10/05, 2014, from http://www. trac.osgeo.org/proj.

European Environment Agency (2013). *Natura 2000 European protected areas*. Retrieved 11/05, 2013, from http://ec.europa.eu/environment/nature/natura2000/index_ htm#.

Garrett, J.J (2005). *Ajax: A New Approach to Web Applications*. University of Washington, Seattle. Retrieved 09/12, 2014, from https://courses.cs.washington.edu/courses/cse490h/07sp/readings/ajax_adaptive_path.pdf.

Google (2013). *KML Reference – Keyhole Markup Language – Google Developers*. Retrieved 10/05, 2014, from https://developers.google.com/kml/documentation/kmlreference.

Google (2014), *Getting Started – Google Map JavaScript API v3 - Google Developers*. Retrieved 02/10, 2014, from https://developers.google.com/maps/documentation/javascript/tutorial.

HS-RS (2014). *PyWPS Documentation*. Retrieved 10/02, 2014, from http://pywps.wald.intevation.org/documentation/index.html.

IMO (2015). *Automatic Identification Systems (AIS)*. Retrieved 05/01, 2015, from http://www.imo.org/OurWork/Safety/Navigation/Pages/AIS.aspx.

Joao, E. (1998). *Causes and consequences of map generalization*. CRC Press.

jQuery Foundation (2013). jQuery: The write less, do more, JavaScript library. Retrieved 04/01, 2013, from https://jquery.com.

jQuery Foundation (2014). jQuery UI API Documentation. Retrieved 10/05, 2014, from http://api.jqueryui.com.

Lekkas, D., Vosinakis, S., Alifieris, C., & Darzentas, J. (2008). MarineTraffic: Designing a Collaborative Interactive Vessel Traffic Information System. In *Proceedings of the 2008 International Workshop on Harbour, Maritime & Multimodal Logistics Modelling and Simulation*. HMS.

Li, Z., Ji, M., Lee, J. G., Tang, L. A., Yu, Y., Han, J., & Kays, R. (2010). MoveMine: mining moving object databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (pp. 1203-1206). ACM.

Lu, C. T., Boedihardjo, A. P., Zheng, J., & Transportation Research Board. (2006). Towards an Advanced Spatio-Temporal Visualization System for the Metropolitan Washington DC. In *5th International Visualization in Transportation Symposium and Workshop*.

MarineTraffic (2014). *Live Ship Map - AIS - Vessel Traffic and Positions*. Retrieved 09/05, 2014, from https://www.marinetraffic.com.

MarineTraffic (2014). *Frequently Asked Question about AIS and Marine Traffic Features*. Retrieved 09/05, 2014, from http://www.marinetraffic.com/en/p/faq.

Mckearney, S (2000). *Physical Database Design - Overview. Bournemouth University, Bournemouth, UK*. Retrieved 10/05, 2014, from http://www.smckearney.com/hn cdb/notes/lec.physicaldesign.2up.pdf.

Miller, H. J., & Han, J. (Eds.). (2009). *Geographic data mining and knowledge discovery.* CRC Press.

Muthu, S.S., Stefanakis, E., & Lekkas, D. (2014). Discovery of Environmental Risk from Historical Vessel Trajectories. In the *Proceedings of the Joint International Conference on Geospatial Theory, Processing, Modelling and Applications.* Toronto, Canada.

MyShipTracking (2014). *Shiptracking*. Retrieved 09/05, 2014, from http://www.myshiptracking.com/.

Neteler, M., & Raghavan, V. (2006). Advances in free software geographic information systems. *Journal of Informatics*, *3*(2).

Oliveira, M., Baptista, C., & Falcão, A. (2012). A Web-based Environment for Analysis and Visualization of Spatio-temporal Data provided by OGC Services. In *GEOProcessing 2012, The Fourth International Conference on Advanced Geographic Information Systems, Applications, and Services* (pp. 183-189).

Open Geospatial Consortium (2014). *OGC Standards and Supporting Documents*. Retrieved 09/05, 2014, from http://www.opengeospatial.org/standards.

OpenLayers Dev Team (2013). OpenLayers Documentation. Retrieved 09/18, 2013, from http://docs.openlayers.org/#openlayers-documentation.

OpenPlans (2013). *GeoServer User Manual*. Retrieved 04/17, 2014, from http://docs.geoserver.org/stable/en/user.

Open Source Geospatial Foundation (2010). *GDAL: ogr2ogr*. Retrieved 10/05, 2014, from http://www.gdal.ogr2ogr.html.

Pelekis, N., Frentzos, E., Giatrakos, N., & Theodoridis, Y. (2011). HERMES: A trajectory DB engine for mobility-centric applications. *International Journal of Knowledge-based Organizations*.

Peng, Z. R., & Tsou, M. H. (2003). *Internet GIS: distributed geographic information services for the internet and wireless networks*. John Wiley & Sons.

Pinkfroot (2014). *Ship Finder - The Live Marine Traffic Tracking App*. Retrieved 09/05, 2014, from http://shipfinder.co/.

PostgreSQL Global Development Group (2013). Retrieved 02/10, 2013, from http://www.postgresql.org/files/documentation/pdf/9.1/postgresql-9.1-A4.pdf.

Puder, A. (2006). A code migration framework for ajax applications. In *Distributed Applications and Interoperable Systems* (pp. 138-151). Springer Berlin Heidelberg.

Rigaux, P., Scholl, M., & Voisard, A. (2001). *Spatial databases: with application to GIS*. Morgan Kaufmann.

Refractions Research Inc (2012*). PostGIS 2.1.5dev Manual*. Retrieved 02/11, 2013, from http://postgis.net/stuff/postgis-2.1.pdf.

Spaccapietra, S., Parent, C., Damiani, M. L., de Macedo, J. A., Porto, F., & Vangenot, C. (2008). A conceptual view on trajectories. *Data & knowledge engineering*, *65*(1), 126-146.

USCG Navigation Center. (2015). AIS Frequently Asked Questions. Retrieved 05/01, 2015, from http://www.navcen.uscg.gov/?pageName=AISFAQ#1.

W3C (2013). World Wide Web consortium. Retrieved 02/10, 2013, from http://www.w3.org/.

Yawen, H., Fenzhen, S., Yunyan, D., & Rulin, X. (2010, June). Web-based visualization of marine environment data. In *Geoinformatics, 2010 18th International Conference on* (pp. 1-6). IEEE.

Zheng, B. (2013). *Interactive Visualization to Reveal Activity Patterns of Marine Mammals and Boat Traffic in the St. Lawrence Estuary in Quebec, Canada* (Master's thesis, University of Calgary, Calgary, Canada). Retrieved December 18, 2013, from http://theses.ucalgary.ca//handle/11023/718

Appendix A   Database Design Scripts

A.1 Creation of a Spatial Database in PostgreSQL

```
CREATE DATABASE MovingObjectDB

WITH ENCODING = 'UTF8'

TABLESPACE = pg_default

TEMPLATE = template_postgis_20
```

A.2 Creation of a Foreign Key Constraint in PostgreSQL

```
ALTER TABLE LOCATION_SHIPS  ADD CONSTRAINT ship_type_fk

FOREIGN KEY(ship_type) REFERENCES SHIP_TYPE (ship_type_id)

MATCH FULL;
```

A.3 Creation of an Index on a Column in PostgreSQL

```
CREATE INDEX location_ships_index on

LOCATION_SHIPS(mmsi,geom);
```

A.4 PHP Script for File Uploading

```
<?php
 $filename = basename($_FILES['file']['name']);
 if (($_FILES['file'][type] != "application/vnd.ms-excel" )
 {
      echo  "<div class='content-center'> <p>File should be
      in CSV      format.</p> <br></div> ";
 }
 else {
      move_uploaded_file($_FILES["file"]["tmp_name"],
      $_FILES["file"]["name"]);
 }
 ?>
```

A.5 PHP Script for Reading and Inserting Data into `LOCATION_SHIPS` Schema

```
$filein=fopen($filename,"r");
while (($line = fgetcsv($filein, 1000, ",")) !== FALSE) {
        if(empty($line[0])){
           break;
       }
```

```
     $query = "INSERT INTO location_ships VALUES ('". $
     line[0] ."','". $line[1] ."','". $ line[2] ."','". $
     line[3] ."','". $ line[4] ."','". $ line[5] ."','". $
     line[6] ."','". $ line[7] ."','". $ line[8] ."','".
     $data[9] ."' ,ST_GeomFromText('POINT(". $line[5] ."
     ". $line[6] .")',4326) ,'". $line[10] ."')";
     $result = pg_query($query);
}
```

A.6 PHP Script to Read Data from a KML File and Populate PROTECTED_AREAS

```
$xml = simplexml_load_file($filename) or die("Error: Cannot
 create object");
  foreach($xml->children() as $nameAttribute){
   foreach($nameAttribute->children() as $style){
     foreach($style ->children() as $extendeddata){
       foreach($extendeddata->children() as $multigeometry){
          foreach($multigeometry->children() as $polygon){
            foreach($polygon->children() as $outerb){
               foreach($outerb ->children() as $ring){
                  foreach($ring ->children() as
                  $coordinates){
                       $name = $ nameAttribute->name;
```

```
                     $trans = array("," => " ", " " =>
                     ",");
                     $polygon = strtr($coordinates,
                     $trans);
                     $query = "INSERT INTO protected_areas
                     VALUES ('". $name   ."' ,
                     ST_GeomFromText
                     ('POLYGON((". $polygon .."))',4326))";
                     $result = pg_query($query);
                }
             }
          }
       }
     }
   }
}
```

Appendix B   Data Mining Scripts

B.1 PHP Function for Creating a Trajectory as a Linestring

```
<?php
function createTrajectory($ship_id) {
$query = "SELECT loc.the_geom  FROM location_ships As loc
WHERE 4 loc.mmsi = $ship_id  ORDER BY loc.time_stamp";
$result = pg_query($query);
$pointArray = array();
while ($row = pg_fetch_row($result)) {
      $pointArray[ ]  = $row[0];
}
if(count($pointArray)  > 1) {
      $trajectory  = SELECT ST_Makeline(rtrim(implode(',',
$arr),
      ','));
}
return  $trajectory;
}
?>
```

B.2 PHP Script for Creating Trajectories for All Ships and Storing It in ROUTE_SHIPS Table

```
<?php
$query = "SELECT loc.mmsi, min(loc.time_stamp),
max(loc.time_stamp), ST_MakeLine(loc.the_geom ORDER BY
loc.time_stamp) FROM location_ships As loc GROUP BY
loc.mmsi;";
     $result = pg_query($query);
     while ($row = pg_fetch_row($result)) {
     $query_insert = "INSERT INTO route_ships VALUES
     ('". $row[0}  ."','". $row[1] ."','". $row[2]
   ."','".$row[3] ."')";
          $result1 = pg_query($query1);
      }
?>
```

B.3 OGR2OGR Command to Import a PostGIS Query into a Shapefile

```
ogr2ogr -f "ESRI Shapefile"
"C:\Users\sabarish\Desktop\route\rout  e.shp"
PG:"host=gaia.gge.unb.ca user=sabarish dbname=sabarish
sabarish password=xxxxx" -sql "SELECT loc.mmsi , min(loc.
time_stamp) as start_time,max(loc.time_stamp) as end_time,
ST_MakeLine(loc.the_geom ORDER BY loc.time_stamp) as geom
FROM location_ships As loc WHERE loc.mmsi = 23700000 and
loc.time_stamp >='2012-08-25 08:17:00' and loc.time_stamp
<= '2013-01-15 12:25:00' GROUP BY loc.mmsi"
```

B.4 PHP Script for the Creation of a Shapefile Based on PostGIS Query

```
<?php
function convertPostGISToShapefile($mmsi, $startdate,
$enddate){
$path="C:\Program Files\PostgreSQL\9.3\bin";
chdir($path);
exec('rd "C:\Users\sabarish\Desktop\route" /S /Q');
exec('mkdir "C:\Users\sabarish\Desktop\route" ');
$descriptorspec = array(
   0 => array("pipe", "r"),
   1 => array("pipe", "w"),
   2 => array("file", "error-output.txt", "a")
);
$env = array('some_option' => 'aeiou');
$cwd  = "C:\OSGeo4W64\bin";
$process = proc_open('ogr2ogr -f "ESRI Shapefile"
"C:\Users\sabarish\Desktop\route\route.shp"
PG:"host=gaia.gge.unb.ca user=sabarish dbname=sabarish
password= xxxxx" -sql "SELECT loc.mmsi ,
min(loc.time_stamp) as start_time,max(loc.time_stamp) as
end_time, ST_MakeLine(loc.the_geom ORDER BY loc.time_stamp)
as geom FROM location_ships As loc WHERE  loc.mmsi = ' .
$mmsi .' and loc.time_stamp >=\'' . $startdate. ' \' and
loc.time_stamp <= \'' . $enddate. '\' GROUP BY loc.mmsi"',
$descriptorspec, $pipes, $cwd);
if (is_resource($process)) {
    fwrite($pipes[0], '');
    fclose($pipes[0]);
    echo stream_get_contents($pipes[1]);
    fclose($pipes[1]);
    $return_value = proc_close($process);
    echo "command returned $return_value\n";
}
}
?>
```

B.5 PHP Script to Copy a Folder to the GRASS GIS Location

```
<?php
function copyToGrassLocation(){
      echo exec('xcopy "C:\Users\Sabarish\Desktop\route"
      "C:\Users\Sabarish\Desktop\route_simplified\newLocatio
      n\Sabarish\vector\route' /i /r /y);  } ?>
```

B.6 Trajectory Simplification Using v.generalize Module

```
v.generalize input=route output=route_simplified
method=douglas threshold=0.01
```

B.7 A PyWPS Process to Generalize a Line Geometry

```
from pywps.Process.Process import WPSProcess
from types import *
class Process(WPSProcess):
  def __init__(self):
      WPSProcess.__init__(self,
      identifier = " generalize ",
      title="TRAJECTORY SIMPLIFICATION",
      version = "0.1",
      storeSupported = "true",
      statusSupported = "true",
      abstract="PyWPS process for geometry simplification",
      grassLocation = "")
  def execute(self):
        self.cmd(["v.generalize", "input=route", "output=
        route_simplified", "method=douglas", ,"threshold=
        0.01"])
```

B.8 OGR2OGR Command to Import a Shapefile into a PostgreSQL Table

```
ogr2ogr -overwrite -f  "PostgreSQL"
PG:"host=gaia.gge.unb.ca user=sabarish dbname=sabarish
password=xxxxx" "C:\Users\Sabarish\
Desktop\route_simplified\route_simplified.shp" -nln
route_simplified
```

B.9 SELECT Query to Determine a Trajectory Length

```
SELECT mmsi, ST_AsText(geom), ST_ Length(ST_Transform(geom,
3857)), start_time, end_time FROM temp_route
WHERE mmsi = ‘X’;
```

B.10 CREATE VIEW Query to Store the Results of the Route Traversed by a Vessel

```
CREATE VIEW  temp_route as
SELECT loc.mmsi, min(loc.time_stamp), max(loc.time_stamp),
ST_MakeLine(loc.the_geom ORDER BY loc.time_stamp)
FROM location_ships As loc
WHERE  loc.mmsi='X'   AND loc.time_stamp  BETWEEN 'Y' AND
'Z' GROUP BY loc.mmsi
```

B.11 A Stored Function `DistanceAndSpeedCalc` for Distance and Speed Calculation given a MMSI Identifier and a Time Interval

```
CREATE OR REPLACE FUNCTION DistanceAndSpeedCalc(mmsiint
integer, starttime timestamp with time zone , endtime
timestamp with time zone) RETURNS DECIMAL AS  $BODY$
DECLARE
      location_ships_cur CURSOR FOR
      SELECT loc.mmsi, loc.time_stamp, loc.the_geom
      FROM location_ships As loc WHERE  loc.mmsi= mmsiint
      AND loc.time_stamp >= starttime and loc.time_stamp <=
      endtime GROUP BY loc.mmsi,loc.time_stamp,loc.the_geom;
      location_ships_rec location_ships%ROWTYPE;
      temp_count integer := 0;
      temp_dist decimal (8,4):=0;
      temp_timediff decimal(8,4);
      temp_time timestamp;
      start_time timestamp;
      end_time timestamp;
      total_time_diff decimal(8,4);
      total_distance_covered decimal(8,4):= 0.0;
      avg_speed_trajectory decimal(8,4) := 0.0;
      temp_geom geometry(POINT,4326);
BEGIN
      DELETE FROM distanceandspeed;
      FOR location_ships_rec in location_ships_cur
      LOOP
            IF (temp_count = 0) THEN
                  INSERT INTO distanceandspeed ( mmsi,
                  time_stamp,the_geom, distance, speed)
                  values(location_ships_rec.mmsi,location_shi
                  ps_rec.time_stamp,ST_GeomFromText(ST_AsText
                  (location_ships_rec.the_geom)), 0.0,0.0);
                                 start_time:=
                  location_ships_rec.time_stamp;
            ELSE
                  temp_dist:= (SELECT ST_Distance(
                  ST_Transform(location_ships_rec.the_geom,38
                  57),ST_Transform(temp_geom, 3857))) /1000 *
                  0.6214;temp_timediff:= (SELECT ((extract
                  (epoch from (location_ships_rec.time_stamp
                  ::timestamp - temp_time::timestamp )))
                  )::decimal)/(60 * 60 );
                                 INSERT INTO
                  distanceandspeed (
                  mmsi, time_stamp,the_geom, distance, speed)
                  values (location_ships_rec.mmsi,
                  location_ships_rec.time_stamp,
                  ST_GeomFromText(ST_AsText(location_ships_re
                  c.the_geom)),temp_dist,temp_dist/temp_timed
                  iff); end_time:=
                  location_ships_rec.time_stamp;
            END IF;
            temp_count:=temp_count + 1;
            temp_geom:= location_ships_rec.the_geom;
            temp_time:= location_ships_rec.time_stamp;
            total_distance_covered:= total_distance_covered +
            temp_dist;
      END LOOP;
      total_time_diff:= (SELECT ((extract (epoch from
      (end_time::timestamp - start_time::timestamp )))
      )::decimal)/(60 * 60 );
      avg_speed_trajectory:=total_distance_covered
```

```
     /total_time_diff;
     UPDATE distanceandspeed set total_distance =
     total_distance_covered;
     UPDATE distanceandspeed set avg_speed =
     avg_speed_trajectory;
     return avg_speed_trajectory;
     END;$BODY$
     LANGUAGE plpgsql
```

B.12 A Stored Function `DirectionCalc` for Direction Calculation given a MMSI Identifier and a Time Interval

```
CREATE OR REPLACE FUNCTION DirectionCalc(mmsiint integer,
starttime timestamp with time zone , endtime timestamp with
time zone) RETURNS DECIMAL AS  $BODY$
DECLARE
     location_ships_cur CURSOR FOR
     SELECT loc.mmsi, loc.time_stamp, loc.the_geom
     FROM location_ships As loc WHERE  loc.mmsi= mmsiint
     AND loc.time_stamp >= starttime and loc.time_stamp <=
     endtime GROUP BY loc.mmsi,loc.time_stamp,loc.the_geom;
     location_ships_rec location_ships%ROWTYPE;
     temp_count integer := 0;
     temp_dir decimal (8,4):=0;
```

```
      temp_timediff decimal(8,4);
      temp_time timestamp;
      first_loc geometry(POINT,4326);
      last_loc geometry(POINT,4326);
      total_time_diff decimal(8,4);
      avg_speed_trajectory decimal(8,4) := 0.0;
      temp_geom geometry(POINT,4326);
BEGIN
      DELETE FROM direction;
      FOR location_ships_rec in location_ships_cur
      LOOP
            IF (temp_count = 0) THEN
                  INSERT INTO direction ( mmsi, time_stamp,
                  the_geom , direction) values
                  (location_ships_rec.mmsi,
                  location_ships_rec.
                  time_stamp,ST_GeomFromText(ST_AsText(locati
                  on_ships_rec.the_geom),4326), 0.0);
                  first_loc:= location_ships_rec.the_geom;
            ELSE
                  temp_dir:= (SELECT ST_Azimuth(

                  ST_Transform(location_ships_rec.the_geom,38
                  57),ST_Transform(temp_geom, 3857)) ) *
                  (180/pi());
                  temp_timediff:= (SELECT ((extract (epoch
                  from
                  (location_ships_rec.time_stamp::timestamp -
                  temp_time::timestamp )))::decimal)/(60 *
                  60 );
                  INSERT INTO direction (
                  mmsi, time_stamp,the_geom, direction)
                  values (location_ships_rec.mmsi,
                  location_ships_rec.time_stamp,
                  ST_GeomFromText(ST_AsText(location_ships_re
                  c.the_geom),4326), temp_dir);
                  last_loc:= location_ships_rec.the_geom;
            END IF;
            temp_count:=temp_count + 1;
            temp_geom:= location_ships_rec.the_geom;
            temp_time:= location_ships_rec.time_stamp;
      END LOOP;
      temp_dir:= SELECT ST_Azimuth(
      ST_Transform(first_loc,3857),
      ST_Transform(last_loc, 3857)) ) * (180/pi());
UPDATE direction set major_direction = temp_dir;
return temp_dir;
END;$BODY$
LANGUAGE plpgsql
```

B.13 A Stored Function `Turnanglecalc` for Turn Angle Calculation given a MMSI Identifier and a Time Interval

```
CREATE OR REPLACE FUNCTION turnanglecalc (mmsiint integer,
starttime timestamp, endtime timestamp)
RETURNS integer AS
$BODY$
DECLARE
     location_ships_cur SCROLL CURSOR FOR
     SELECT  loc.time_stamp, loc.the_geom
     FROM location_ships As loc
     WHERE  loc.mmsi= mmsiint
     AND loc.time_stamp >= starttime and loc.time_stamp <=
     endtime
     GROUP BY loc.mmsi,loc.time_stamp,loc.the_geom;
     location_ships_rec location_ships%ROWTYPE;
     location_ships_rec2 location_ships%ROWTYPE;
     temp_count integer := 0;
     row_count integer := 0;
     temp_dist decimal (16,9):=0;
     temp_distA decimal (16,9):=0;
     temp_distB decimal (16,9):=0;
     temp_distC decimal (16,5):=0;
     cosi double precision:=0;
```

```
    deg decimal (8,4):=0;
    temp_timediff decimal(8,4);
    temp_time timestamp;
    temp_time1 timestamp;
    temp_time2 timestamp;
    start_time timestamp;
    end_time timestamp;
    total_time_diff decimal(8,4);
    total_distance_covered decimal(8,4):= 0.0;
    avg_speed_trajectory decimal(8,4) := 0.0;
    temp_geom geometry(POINT,4326);
    temp_geom1 geometry(POINT,4326);
    temp_geom2 geometry(POINT,4326);
BEGIN
    DELETE FROM angle;
    row_count:= (SELECT  count(*)
    FROM location_ships As loc
    WHERE  loc.mmsi= mmsiint
    AND loc.time_stamp >= starttime and loc.time_stamp <=
    endtime
        );
    FOR location_ships_rec IN location_ships_cur
    LOOP
        IF (temp_count = 0) THEN
```

```
      INSERT INTO turn_angle ( mmsi, time_stamp,
      geom, angle) values
      (mmsiint,location_ships_rec.time_stamp,ST_G
      eomFromText(ST_AsText(location_ships_rec.th
      e_geom),4326), 0.0);
END IF;
IF (temp_count <= (row_count-3)) THEN
      FETCH NEXT FROM location_ships_cur into
      temp_time,temp_geom;
      FETCH NEXT FROM location_ships_cur into
      temp_time1,temp_geom1;
      temp_distA:= (SELECT
      ST_Distance(ST_Transform(location_ships_rec
      .the_geom,3857),
      ST_Transform(temp_geom, 3857))) /1000 *
      0.6214 ;
      temp_distB:= (SELECT ST_Distance(
      ST_Transform(temp_geom,3857),
      ST_Transform(temp_geom1, 3857))) /1000  *
      0.6214;
      temp_distC:= (SELECT ST_Distance(
      ST_Transform(location_ships_rec.the_geom,38
      57),ST_Transform(temp_geom1, 3857))) /1000
      *0.6214;
```

```
cosi  = ((temp_distA * temp_distA) +
(temp_distB * temp_distB) - (temp_distC *
temp_distC)) / (2 * temp_distA * temp_distB
) ;
deg = 180 - (acos(cosi) * (180 / pi()));
temp_timediff:= (SELECT ((extract (epoch
from (location_ships_rec.time_stamp
::timestamp -
temp_time1::timestamp ))))::decimal);
temp_timediff2:= (SELECT ((extract (epoch
from (temp_time::timestamp-
location_ships_rec.time_stamp::timestamp
))))::integer);

temp_timediff3:= (SELECT ((extract (epoch
from (temp_time1::timestamp -
temp_time::timestamp ))))::integer);

IF( (temp_timediff2/60) < 15 and
(temp_timediff3/60) < 15) THEN
     INSERT INTO turn_angle ( mmsi,
     time_stamp,
     geom,angle)values(mmsiint,temp_time,ST
     _GeomFromText(ST_AsText(temp_geom),432
     6),deg );
```

```
            IF (temp_count = (row_count-3)) THEN
                  INSERT INTO turn_angle ( mmsi,
                  time_stamp, geom, angle) values
                  (mmsiint,temp_time1,ST_GeomFromText(ST
                  _AsText(temp_geom1),4326), 0.0);
            END IF;
            ELSE
                  INSERT INTO turn_angle ( mmsi,
                  time_stamp,
                  geom,angle)values(mmsiint,temp_time,ST
                  _GeomFromText(ST_AsText(temp_geom),432
                  6),0.0000 );
                  IF (temp_count = (row_count-3)) THEN
                        INSERT INTO turn_angle ( mmsi,
                        time_stamp, geom, angle) values
                        (mmsiint,temp_time1,ST_GeomFromT
                        ext(ST_AsText(temp_geom1),4326),
                        0.0);
                  END IF;
            END IF;
      FETCH PRIOR FROM location_ships_cur into
      temp_time,temp_geom;
      FETCH PRIOR FROM location_ships_cur into
      temp_time1,temp_geom1;
      END IF;
      temp_count:= temp_count +1;
END LOOP;
temp_time:=  temp_time1;
return row_count  ;
END;$BODY$
LANGUAGE plpgsql
```

B.14 SQL Query to Determine the Vessels That Intersect the Protected Areas

```
SELECT  p.name as name, p.mmsi as mmsi,
ST_ASTEXT(ST_MAKELINE(p.point)) as geom , max(p.time) as
start_time, min(p.time) as end_time
FROM
      (SELECT pro.name as name, loc.mmsi as mmsi,
      loc.time_stamp as time, loc.the_geom  as point
      FROM location_ships loc, protected_areas pro
      WHERE ST_INTERSECTS (loc.the_geom, pro.geom)
      and loc.time_stamp >= '2012-08-05 17:51:00' and
      loc.time_stamp <= '2012-08-06 17:51:00' ) p
GROUP BY p.name, p.mmsi
```

B.15 SQL Query to Determine the Protected Areas That the Vessel with MMSI Identifier 237001000 Passes Through

```
SELECT  p.name as name, p.mmsi as mmsi,
ST_ASTEXT(ST_MAKELINE(p.point)) as geom , max(p.time) as
start_time, min(p.time) as end_time
FROM
     (SELECT pro.name as name, loc.mmsi as mmsi,
     loc.time_stamp as time, loc.the_geom  as point
     FROM location_ships loc, protected_areas pro
     WHERE ST_INTERSECTS (loc.the_geom, pro.geom)
     and loc.mmsi = 237001000 and loc.time_stamp >= '2012-
     08-01 00:00:00' and loc.time_stamp <= '2012-08-30
     12:59:59' ) p
GROUP BY p.name, p.mmsi
```

B.16 SQL Query to Identify Vessels That Are in the Vicinity of a Vessel

```
SELECT b.mmsi, ST_AsText(b.geom) from
     (SELECT ST_transform(ST_Buffer(
     ST_Transform(loc.the_geom,
     3857), 1852*50),4326) as geom  FROM location_ships As
     loc WHERE loc.mmsi=237001000 AND loc.time_stamp =
     '2012-08-21 06:05:00') a,
     (SELECT loc.mmsi as mmsi, loc.the_geom as geom,
     loc.time_stamp  FROM location_ships As loc
     WHERE loc.time_stamp  = '2012-08-21 06:05:00') b
WHERE ST_Within(b.geom , a.geom)
```

B.17 DBSCAN Algorithm Implementation (a) DBSCAN() Function, and (B) ExpandCluster() Function Using PHP

```
function DBSCAN($a, $eps, $minpts){
      $result = count($a);
      $c  = 0;
      $GLOBALS["clusters"][$c] = array();
      $GLOBALS["unvisited"] = $a;
      foreach($a as  $i => $geom){
                  if (in_asso_array2($geom,
                  $GLOBALS["unvisited"])) {
                        unset($GLOBALS["unvisited"][$i]);
                        $GLOBALS["visited"] =  $i;
                        $mmsi = $GLOBALS["id"];
                        $st = $GLOBALS["start"];
                        $end = $GLOBALS["end"];
                        $query = "SELECT b.geom from (SELECT
                        ST_transform(ST_Buffer(ST_Transform(ST
                        _GeomFromText(st_astext('$geom')
                        ,4326),3857), 1500),4326) as geom ) a,
                        (SELECT loc.mmsi as mmsi, loc.the_geom
                        as geom  FROM location_ships As loc
                        where loc.mmsi='$mmsi' AND
                        loc.time_stamp  between  '$st'  and
                        '$end' )b
                        where ST_Within(b.geom , a.geom)";
                        $result = pg_query($query);
                  $count = 0;
                  $neighborpts = array();
                  while ($row = pg_fetch_row($result)) {
                        $neighborpts[$count] = $row[0];
                        $count++;
                  }
                  if (count($neighborpts) < $minpts){
                        $GLOBALS["noise_points"][] = $i;
                  }
                  else{
                        ExpandCluster($geom, $neighborpts, $c,
                        $minpts);
                        $c = $c + 1;
                        $GLOBALS["clusters"][$c] = array();
```

```
                  }
            }
      }
}
```

(a)

```
function ExpandCluster($point, $ne_pts, $ct, $min){
      $GLOBALS["clusters"][$ct][] = $point;
      $GLOBALS["in_cluster"][]  = $point;
      $neighbor_point = reset($ne_pts);
      $mmsi = $GLOBALS["id"];
      $st = $GLOBALS["start"];
      $end = $GLOBALS["end"];
      while ($neighbor_point){
            if (in_asso_array2($neighbor_point,
            $GLOBALS["unvisited"])) {
            $k = retkey($neighbor_point,
            $GLOBALS["unvisited"]);
            unset($GLOBALS["unvisited"][$k]);
            $query = "SELECT b.geom from (SELECT
            ST_transform(ST_Buffer(ST_Transform(ST_GeomFromTe
            xt(st_astext('$neighbor_point') ,4326),3857),
            1500),4326) as geom ) a,(SELECT loc.mmsi as mmsi,
            loc.the_geom as geom  FROM location_ships As loc
            where loc.mmsi='$mmsi' AND loc.time_stamp
```

```
        between  '$st'  and '$end' )b where
        ST_Within(b.geom , a.geom)";
        $result = pg_query($query);
        $count = 0;
        $neighborpts1 = array();
        while ($row = pg_fetch_row($result)) {
            $neighborpts1[$count] = $row[0];
            $count++;
        }
        if (count($neighborpts1) >= $min ){
            $ne_pts = array_merge($ne_pts,
            $neighborpts1);
        }
    }
    if(!in_array($neighbor_point, GLOBALS["in_cluster"])){
    $GLOBALS["clusters"][$ct][] = $neighbor_point;
    $GLOBALS["in_cluster"][]  = $neighbor_point;
    }
    $neighbor_point = next($ne_pts);
    }
}
```

(b)

Appendix C   Visualization Scripts

C.1 Including jQuery Library in HTML

```
<script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jque
ry.min.js">
</script>
```

C.2 AJAX Function in jQuery That Loads a Remote Page Using HTTP Request

```
$.ajax
({
type: "POST",
url: "XXXX.php",
data: {id: 1},
dataType: 'json',
cache: false,success: function(html)
{
. . .
}});
```

C.3 PHP Script to Extract the Coordinates for the Detailed and Simplified Trajectory as a JSON Encoded Array to Be Sent to the Client

```
$conn_string = "host=localhost port=5432 dbname=mod
user=postgres password=etentis";
$db_conn = pg_connect($conn_string);
$query =  "SELECT loc.mmsi ,  loc.time_stamp ,
st_astext(loc.the_geom)  FROM location_ships As loc WHERE
loc.mmsi =  $mmsi  and loc.time_stamp >= '$startdate' and
loc.time_stamp <=  '$enddate' ";
$result = pg_query($query);
$query_simplified = " select st_astext(wkb_geometry) from
route_simplified ";
$simplifiedtrac = pg_query($query_simplified);
$straj = pg_fetch_row($simplifiedtrac, 0);
$points = array();
while ($row = pg_fetch_row($result)) {
           $points[] = array(
                'mmsi' =>   $row[0] ,
                'time' => $row[1],
                'geom' => $row[2],
                'simp' => $straj,
           );
     }
```

```
echo json_encode($points);
```

C.4 JavaScript Code Fragment to Displaying the Direction through Google Maps Info Window

```
for (tys = 0; tys < darray.length; tys++) {
      var marker1 = new google.maps.Marker({
      position: new
      google.maps.LatLng(darray[tys][0],darray[tys][1]),
     map: map,
      icon: {
            path: google.maps.SymbolPath.CIRCLE,
            strokeColor : '#FAEF1E',
           fillColor : '#FAEF1E',
            fillOpacity : 0.6,
            scale: 5
      },
       title: darray[tys][0]  +"  "+ darray[tys][1]
});

(function(marker1, tys) {
// add click event
google.maps.event.addListener(marker1, 'click', function()
{
```

```
           infowindow = new google.maps.InfoWindow({
           content: ‘<div style=”width:300px;margin:0 0 20px
           20px;height:90px;”><h3>Direction: ‘+
           darray[tys][2]  +’</h3> <h3>Major Direction: ‘+
           darray[tys][4] +  ‘</h3><h3>Time: ‘ +
           darray[tys][3]  + ‘</h3></div>’
                            });
           infowindow.open(map, marker1);
      });
     })(marker1, tys);
}
```

C.5 Adding a OGC WMS Layer Using OpenLayers JavaScript API

```
var map = new OpenLayers.Map('map',options);
var wms = new OpenLayers.Layer.WMS( 'heat map',
'http://gaia.gge.unb.ca:8080/geoserver/wms',
{
     width: '600', height: '400',
     srs: 'EPSG:900913',
     layers: 'gn:rs6565',format: 'image/png',
     bgcolor: '0x80BDE3',transparent:true
},{
singleTile: true, ratio: 1,
maxExtent: [2674366,4166515,3002844,4576045],

isBaseLayer: false}

);

map.addLayers([gsat,wms]);
```

Curriculum Vitae

Candidate's full name: Sabarish Senthilnathan Muthu

Universities attended:

2012 - present, MScE. University of New Brunswick, NB, Canada

2010, B.E. Geoinformatics. College of Engineering, Guindy, Anna University, India

Publications:

Muthu, S.S., Gkadolou, E., and Stefanakis, E. (2013). Historical Map Collections on Geospatial Web. *Geomatica Journal*, Vol. 67, No. 3, pp. 279-290.

Muthu, S.S., Stefanakis, E., & Lekkas, D. (2014). Discovery of Environmental Risk from Historical Vessel Trajectories. In the *Proceedings of the Joint International Conference on Geospatial Theory, Processing, Modelling and Applications*. Toronto, Canada.