Predicting Energy Consumption for Potential Effective Use in Hybrid Vehicle Powertrain Management Using Driver Prediction

Brian Magnuson

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington
2017

Committee:
Robert Darling
Brian Fabien
Jim Peckol

Program Authorized to Offer Degree:
Electrical Engineering

University of Washington

Abstract

Predicting Energy Consumption for Potential Effective Use in Hybrid Vehicle Powertrain Management Using Driver Prediction

Brian Magnuson

Chair of the Supervisory Committee:
Professor Robert Darling
Electrical Engineering

A proof-of-concept software-in-the-loop study is performed to assess the accuracy of predicted net and charge-gaining energy consumption for potential effective use in optimizing powertrain management of hybrid vehicles. With promising results of improving fuel efficiency of a thermostatic control strategy for a series, plug-ing, hybrid-electric vehicle by 8.24%, the route and speed prediction machine learning algorithms are redesigned and implemented for real-world testing in a stand-alone C++ code-base to ingest map data, learn and predict driver habits, and store driver data for fast startup and shutdown of the controller or computer used to execute the compiled algorithm. Speed prediction is performed using a multi-layer, multi-input, multi-output neural network using feed-forward prediction and gradient descent through back-propagation training. Route prediction utilizes a Hidden Markov Model with a recurrent forward algorithm for prediction and multi-dimensional hash maps to store state and state distribution constraining associations between atomic road segments and end destinations. Predicted energy is calculated using the predicted time-series speed and elevation profile over the predicted route and the road-load equation. Testing of the code-base is performed over a known road network

spanning 24x35 blocks on the south hill of Spokane, Washington. A large set of training routes are traversed once to add randomness to the route prediction algorithm, and a subset of the training routes, testing routes, are traversed to assess the accuracy of the net and charge-gaining predicted energy consumption. Each test route is traveled a random number of times with varying speed conditions from traffic and pedestrians to add randomness to speed prediction. Prediction data is stored and analyzed in a post process Matlab script.

The aggregated results and analysis of all traversals of all test routes reflect the performance of the Driver Prediction algorithm. The error of average energy gained through charge-gaining events is 31.3% and the error of average net energy consumed is 27.3%. The average delta and average standard deviation of the delta of predicted energy gained through charge-gaining events is 0.639 and 0.601 Wh respectively for individual time-series calculations. Similarly, the average delta and average standard deviation of the delta of the predicted net energy consumed is 0.567 and 0.580 Wh respectively for individual time-series calculations. The average delta and standard deviation of the delta of the predicted speed is 1.60 and 1.15 respectively also for the individual time-series measurements. The percentage of accuracy of route prediction is91%. Overall, test routes are traversed 151 times for a total test distance of 276.4 km.

CONTENTS

1. INTRODUCTION

Over a driver's route, changes in kinetic and potential energy through accelerations and changes in elevation along the way can be used to supplement the usage of petroleum energy in hybrid vehicle architectures to maintain the state of charge of traction batteries. To an extent typical thermostatic, load-following, and hybrid powertrain control strategies exploit these changes in kinetic and potential energy in real-time to command torque at the wheels of the hybrid vehicle. For instance, less torque and thus energy is needed to accelerate a vehicle downhill as acceleration due to gravity assists in propulsion. Conversely, negative wheel torque is needed to brake a vehicle requiring zero energy consumption. Rather, energy is typically gained via regenerative braking in deceleration events. To improve upon conventional powertrain management of hybrid vehicles, understanding the time-series energy consumption along the drivers' intended route can be greatly beneficial to optimize petroleum usage.

For example, a thermostatic control strategy used in a plug-in, series hybrid charges and discharges the traction battery whenever the state-of-charge reaches a lower and upper limit of the target state-of-charge. In the case where the time-series energy consumption is known over the intended route, thermostatic control could be improved by skipping charge cycles if energy could be added to the traction battery through regenerative braking at approximately the same time charging is needed. Load-following control strategies, on the other hand, may draw energy from the gas portion of the hybrid when driver torque requests draw more current from the traction batteries than the battery management systems may allow. Battery management systems regulate the discharge of current as a function of state-of-charge and typically allow a discharge current for a limited amount of time before an emergency power-off is required. Similarly, understanding the time-series energy consumption may improve upon load-following control strategies by optimizing the time in which the gas portion of the hybrid is utilized.

To understand the time-series energy consumption over the intended route ahead of time, however, the driver must manually enter the intended route to assess the changes in potential energy due to varying elevation along the way. Additionally, the speed at which the driver travels the intended route must also be known to assess the changes in kinetic energy along the way. In an attempt to manage the randomness of driver route selection and travel speed, the Driver Prediction algorithm attempts to predict an end-destination and predicted route as well as the time-series speed over the route using a Hidden Markov Model and neural network respectively. The intention is to provide a production-ready algorithm rooted in machine learning to predict accurately energy consumption of the vehicle over the intended route that minimizes driver input to maintain the conventional turn-key-and-go driving experience. The accuracy and precision of energy consumption is studied to assess the potential effective use of the Driver Prediction algorithm in improving conventional hybrid vehicle powertrain management.

2. DESIGN SPECIFICATION

2.1 Development

Development of the Driver Prediction algorithm was a two-phase process over two years. Initially, a proof-of-concept software in the loop (SIL) model was developed in Matlab. A virtual world and plug-in hybrid vehicle were used to assess the performance of Driver Prediction to improve simple rule-based thermostatic control of the hybrid powertrain. With promising results,

the code base was altered, added to and implemented in C++ for real-world testing as part of the second phase. In the second phase, however, more emphasis is placed on the energy prediction performance of the algorithm rather than the algorithms energy savings once implemented in a hybrid powertrain management controller to better assess feasibility of potential effective use in production. The intent of the second phase of development is also to create a standalone code base capable of integration into prototyping controller with simple plug-and-play functionality.

2.2 System Use Case

This system is designed to have minimal user interface to prevent interference with the drivers' natural driving routine. Thus, the only decision the driver/user must make is whether to place Driver Prediction in effect while driving to improve powertrain efficiency.

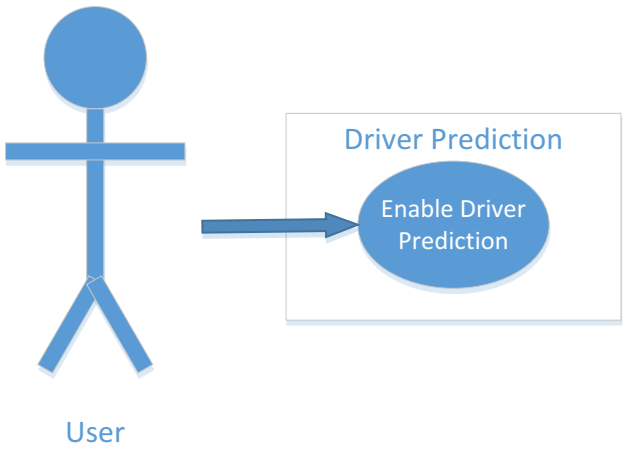

Figure 1: Use Case Diagram

There is no other form of user interaction with the algorithm. Unlike "suggestive" energy saving strategies, energy-savings takes place with the driver unaware to the powertrain optimizations made by the Driver Prediction system. The number of use-cases of Driver Prediction is intentionally limited to one to broaden the variety of powertrain applications from efficient to high-performance hybrids by reducing the need for sophisticated user interfaces. It is believed the algorithm offers the potential to save comparable amounts of energy as seen with "suggestive" strategies.

2.3 Top-Level Design Specifications

The following table is a brief description of the design requirements to implement successfully the Driver Prediction algorithm.

Table 1: Top-Level Design Specification

General Requirements	Specific Requirements	Acceptable Performance
Route Prediction	Hidden Markov model implementation	Employ recursive forward algorithm with link jitter prevention to predict end destination and intended route

	Road and intersection abstraction classes	Abstract road and intersection information to only identifying numbers and road direction
	State hashing	Hash current road/next road/end-destination states to improve run-time complexity
	Hashing of constraints on state distribution	Hash road/end-destination state distribution constraints to improve run-time complexity
	Low prediction error	prediction error is < 10% nominally
Speed Prediction	Multi-input multi-output neural network implementation	Employ standard feed-forward and gradient descent through back propagation algorithms.
	Input scaling	Scale input to prevent saturation of the objective function
	Low prediction error	Prediction error is < 15% nominally
Route and Speed Prediction Interface (Driver Prediction)	Generate time-series prediction data	Gather predicted route and employ speed prediction over it.
	Train route and speed prediction	Train machine learning algorithms on the fly and at the end of a trip
	Buffer time-series speed measurements	Buffer all speed measurements for input to speed prediction
Data Serialization	Serializes and deserializes road network data	Includes road, intersection and boundary identifying numbers, GPS locations to all road network features and graph-oriented associations of road network features. Execution time < 0.33 seconds.
	Serializes and deserializes route prediction	Includes identifying features of hash maps and records of observed road and intersections. Execution time < 0.33 seconds
	Serializes and deserializes speed prediction	Includes neural network weight, input and output matrices for all layers. Execution time < 0.33 seconds.
Data Management	Template data structure for ordered and unordered data sets	Data structure must be capable of being iterated over and provide a tool box of manipulator functions.
Data Collection	Generate connected graph representing road-network	Identifying connections must comprise greater than 90% of the actual road network represented.
	Gather road curvature and elevation profiles	Gather raw elevation and GPS waypoints of map data

Road Network Management	Offer graph-theory tool box over collection of intersections and atomic road segments	Include functions for shortest path, random path, connecting roads, and adjacent intersections.
	Store road network	Store roads, intersections, and known road network perimeter
	Generate speed and elevation time-series data along predicted route	perform interpolation when needed of elevation measurements over the predicted speed time-series.
GPS	Offer toolbox of localization algorithms over road network	Includes calculation of heading, distance along the current road, degree-to-meter conversions, and identifying the nearest road to the current vehicle location
	Gather latitude and longitude from GPS receiver in real-time	Gather latitude and longitude over RS-232 and log the data.
Vehicle Diagnostics	Gather vehicle data in real-time	Gather speed, fuel mass flow, and engine load
Kinematics	Forecast energy consumption from predicted speed and elevation time-series data	Use road-load equation. Error < 20% nominally

3. DESIGN PROCEDURE

Development of Driver Prediction spanned two years in which a MatLab version of the code-base was first generated as a proof-of-concept and then a C++ version was created for real-world testing.

3.1 Proof of Concept

For the scope of this section, design test-benching and result analysis is limited to software-in-the-loop testing where the Driver Prediction algorithms and virtual city are executed in conjunction with a vehicle plant model. The vehicle plant model uses a series, plug-in, hybrid-electric architecture to maximize control of energy flow to the powertrain while minimizing torque interruptions at the wheels. The goal of this study is to determine the quantity of petroleum energy saved by optimizing generator and regenerative braking control within the thermostatic propulsion controller. This is accomplished by predicting driver habits while still providing enough electric energy to maintain driver torque requests.

3.1.1 Driver Prediction

Predicting driver intent is broken up into two distinct steps, each with a distinct prediction algorithm. The first step is predicting where the driver will go and the route to be taken to get there; this is Route Prediction. The next step is predicting the speed of the driver over the predicted route; this is Speed Prediction. Breaking up Driver Prediction into different steps has two advantages. First, it simplifies the machine learning algorithms necessary for accurate prediction. This reduces the memory size required for prediction and increases the speed of predictions. Second, each algorithm can be run individually and only when necessary, thus

improving computation efficiency. The predicted speed can update constantly, but the Route Prediction only needs to run whenever the vehicle encounters road intersections.

3.1.1.1 Route Prediction

The primary assumption made when predicting the driver destination and route is that most drivers use the same route to a given location. This means that, in general, there is a finite set of destinations and routes a driver will take. However, without asking the driver directly, this information is unknown to the vehicle. All that is known is where the vehicle has been and where it currently is [1]. This makes hidden Markov models [2] a logical choice for route prediction. The Markov algorithm has been used significantly to solve the problem of driver route prediction. Driver route prediction is a well-researched topic, and because the focus of this paper is not the prediction itself but the application, the algorithm described in 'Learning to Predict Driver Route and Destination Intent' by Simmons, Browning, Zhang, and Sadekar is extensively relied upon.

Too simplify the hidden Markov model used for the Route Prediction algorithm, a basic assumption made is that the model has complete accuracy in its observations. In simulation, this is easier to achieve. By then adding sensory error coupled with modern GPS accuracy, approximating near-exact vehicle location becomes feasible. Roads are split into different segments by intersection or destinations. Each segment is then treated as an atomic unit, for if the vehicle enters a segment from one direction, the assumption made is that the vehicle must exit at the other end. At the end of each road segment, the vehicle is allowed to travel along any segment that is connected to the intersection. Each time the driver chooses a new road section, the model updates prediction probabilities and the predicted end-destination. This is done by implementing a model that holds a set of all possible destinations the driver might use, where each is weighted by the probability the driver will choose the predicted path.

3.1.1.1.1 Hidden Markov Model

Hidden Markov models contain a set of states X, observations or emissions e, and a transition function T, which maps one state to another for a given observation. For the purposes of this paper, states represent road intersections, emissions are the set of all possible routes for a given intersection, and the transition function assigns a probability of moving to the next intersection for each observation. See Figure 1.

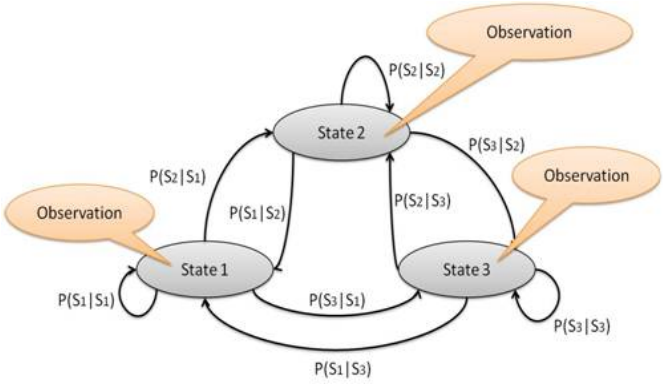

Figure 2: Hidden Markov Model Diagram [3]

Usually, hidden Markov models also include an observation function, which is used when observations are unreliable. Since model observations are assumed to be completely accurate, the observation function is left out. The initial probability distribution for each state is set *a priori*, which sets the uncertainty of each state by using knowledge of routes previously taken by the driver. The transition function, *T*, returns the probability that starting at state i and given emission e, the next state is state *j*

$$T(s_i, e, s_j) = p(s_i \,|s_j, e). \tag{1}$$

To predict the probability of state i at time t using notation $p^t(s_i)$, the products of probability of each state s_j and the transition function from state s_j to state s_i are summed as

$$p^t(s_i) = \sum_{s_j} p^{t-1}\left(s_j\right) * T\left(s_i, e^t, s_j\right). \tag{2}$$

With this, the probability of each state can be updated each time a new piece of evidence is observed.

3.1.1.1.2 Route Prediction Algorithm

The list of possible states contained in the hidden Markov model at time t are all tuples of $< d, l_t, c >$ where c represents the starting conditions such as time-of-day or location, l_t is the route link the vehicle is on at time t, and d is a destination previously seen. A route link, or simply link, represents a road segment traveled in a single direction. There are two links per road segment, one for each direction. This allows the algorithm to distinguish between traveling along a road segment in one direction from traveling down the same segment in the opposite direction.

By encoding the evidence, l_t, into the state itself, the transition function is reduced to $T(s_i \,|s_j)$, which returns the probability of state s_i, given state s_j and is redefined as

$$T(s_i \,|\, s_j) = p(< l_i, d_i, c > \;|\; s_j). \tag{3}$$

However, this means that the set of states at each time t is different, which is acceptable as long as it is taken into consideration when developing the predictive algorithm. To simplify the transition function further, s_i is broken up, so that

$$p(< l_i, d_i, c > | \, s_j) = \; p(l_i, < d_i, c > \,|\, s_j). \tag{4}$$

Further reduction is possible by splitting the single probability into two probabilities redefined as

$$p(l_i, < d_i, c > | \, s_j) = \; p\left(< d_i, c > \left|l_i, s_j\right.\right) * p\left(l_i\right|s_j). \tag{5}$$

Combining (3), (4), and (5)

$$T(s_i|s_j) \;=\; p(< d_i, c > | \, l_i) * p(l_i|s_j). \tag{6}$$

Multiple layers of hash tables [4, p. 171] allows for quick lookup time for each of these probabilities at the expense of memory requirements. Links are hashed by both the name of the route segment the link represents and the direction the vehicle is traveling along the link. Destinations and conditions are hashed together. Because the conditions are often continuous values with a large discrete range, sorting values into bins reduces the range of the conditions. For example, time-of-day could be sorted into three bins, one for the morning, afternoon, and night. Three possible categories improve accuracy without significantly increasing the complexity of the model by comparing destinations visited at similar times to one another. To find $p(l_i \,|s_j)$, the probabilities are hashed by destination, d_j, conditions, c, and link, l_j. For each link, l_i, that can be taken from the intersection at the end of link l_j, a counter, m, is stored and incremented each time a transition from link l_j to l_i occurs, when d_j is the destination and conditions c are true in the training set or at the end of test route. Letting

$$m \;=\; m(< d_j, c >, l_i, l_j) \tag{7}$$

then

$$p(l_i|s_j) = \frac{m(< d_j, c >, l_i, l_j\,)}{\sum_{l_x} m(< d_j, c >, l_i, l_x)}. \tag{8}$$

The data structure that is used for the second probability $p(< d_j, c > |l_i)$ is a 2-level hash table, where the elements of the first hash table are also hash tables, instead of the 3-level hash table, $p(l_i \,|s_j)$. The first hash table is indexed by the hash of $< d_i, c >$, while the second hash table, which is the element of the first, is hashed by l_i. The entries to this second hash table are the number of times l_i is visited when $< d_i, c >$ is the destination and condition, which is the value m. For this second probability, m is defined by

$$m = \; m(< d_i, c >, l_i\,). \tag{9}$$

To find $p(< d_i, c > |l_i)$, the equation is very similar to equation (8) with

$$p(< d_i, c > |l_i\,) = \frac{m(< d_i, c >, l_i\,)}{\sum_{d_x} m(< d_x, c >, l_i\,)}. \tag{10}$$

Reversing the order of the hash table starting with l_i then hashing to $< d_x, c >$ would improve the speed of the probability calculation. However, there are significantly more links than destination and condition pairs, so reversing the order would also require reserving more memory space to store the data structure.

3.1.1.1.3 Training Route Prediction

To train the route prediction algorithm, routes with known end-destinations and starting conditions are used, such as routes in the training set and the end of routes in the test set. The set of training routes is given directly to the prediction algorithms without running the vehicle simulation. Vehicle efficiency is not measured for these routes as the machine learning algorithms must learn the behavior of the driver before accurate predictions can be made. The test routes are the set of routes over which the vehicle is simulated. This is where route and speed

predictions are made and energy usage is recorded and compared against results using a non-predictive thermostatic powertrain control strategy.

3.1.1.1.4 Making Predictions

At the start of the drive, the route prediction algorithm weights each stored destination based on starting parameters, such as time of day and day of the week, as well as the percentage of times the predicted destination is the actual destination relative to all of the other destinations. This is the posterior distribution, *a priori*, for the set of initial drive conditions. A route is created from the current location to the most likely end-destination based on the stored normal routes the driver takes. When the simulated vehicle reaches an intersection and chooses a new road link to travel on, the route prediction compares the chosen link to the predicted route link. If the link is on the predicted route, the probably of each possible destination is updated with the new evidence. However, if the driver deviates from the predicted path, the probabilities are updated, and a new predicted route is created based on the most likely end-destination given the vehicle location and trajectory. When the vehicle reaches the end of the actual route, the route prediction will update the probability hash maps [4, p. 171] with the route just driven, the destination the route ended at, and the initial conditions.

3.1.1.2 Speed Prediction

Predicting a driver's future speed is done separately from Route Prediction. Once the Route Prediction algorithm creates a route from the current location, the Speed Prediction algorithm extrapolates the current speed along the predicted route from the current vehicle speed and speed limits along the route. At each time step, independent of whether the predicted route has changed, the Speed Prediction algorithm updates its prediction using the current speed limit of the new road segment.

3.1.1.2.1 Regression Tree Model

The machine-learning model used for Speed Prediction is a regression tree [5]. Regression trees separate training data by thresholds in different features with the predicted value as the average of all the data sets at a particular node. See Figure 4 as to how prediction features, 'x3' and 'x2', are separated at each level of the regression tree.

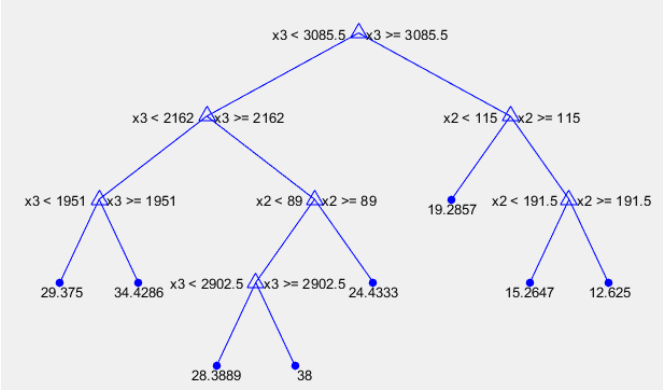

Figure 3: Matlab Regression Tree Visualizer: Feature Separation [6]

At each none-leaf node of the tree, the data points are split by one feature at a single threshold. Data points above the threshold go to one child node, while the remaining data points go to the other separating out the data at each level of the tree. Prediction features include speed limits at the current vehicle-location, 50 meters ahead, and 100 meters ahead as well as the current vehicle speed and acceleration. Predicted value is the speed of the vehicle in 10 meters. Distance is chosen instead of time because the simulated city is based on the same 10-meter, measurement interval. Because of this distance constant, it is easier to compute data points for distance than time.

Instead of creating a custom algorithm for calculating speed prediction like route prediction, a 'fitrtree' is used. A 'fitrtree' is a binary regression tree that is part of the Matlab Statistics and Machine Learning Toolbox. To predict the speed over the entire route produced from Route Prediction, the current speed and location of the vehicle are first taken into account, and the regression tree is used to predict the speed at the next 10-meter interval to find the predicted acceleration from the difference in the predicted speed and the current speed. To predict speed across an entire route, the algorithm iteratively predicts the speed at every interval using the predicted speed and acceleration at the previous interval as well as the speed limit values along the route.

3.1.1.2.2 Training Speed Prediction

To create data points from multiple drive routes and reduce the overall number of data points stored in memory, a 2-level hash map is created whose elements are also hash maps. The first map is the hash of the three speed limit values, the speed limit at the current position as well as the speed limit at 50 meters and 100 meters ahead of the vehicle. The second is the hash of the current speed and acceleration, both rounded to nearest whole number to reduce the map size. The final element stored is the number of times this combination of speed limits, current speed, and acceleration has been seen. When converting from the hash map to data points to give to the fitrtree, all of the predicted values are averaged and given a weight of the length of the array. This way, data points that are seen frequently are weighted heavily but only have to occur once in the data set.

3.1.2 Improving Thermostatic Control with Driver Prediction

For all vehicle optimization, speed and incline slope is inputted to a vehicle plant model and virtual thermostatic powertrain controller via a vector of values as a function of time referred to as trace-data.

3.1.2.1 Predictive Vehicle Powertrain Control

A medium-fidelity thermostatic powertrain controller and plant model are developed in MathWorks Simulink to simulate energy losses of the hybrid vehicle under test. From the algorithms described in the Driver Prediction step, predicted speed and incline slope trace-data are made available to the thermostatic powertrain controller to improve efficiency of the generator and regenerative braking by leveraging hill descents and deceleration events.

A simple plugin, hybrid-electric propulsion control strategy is used to dissipate all allowable energy in the traction batteries, known as charge-depleting mode, providing an all-electric range of approximately 80 km with a full charge. Only after the vehicle has entered a charge-sustaining mode, where the generator is used to maintain a 30-percent target state-of-charge, are improvements made to torque-blending of the generator and traction motors via Driver

Prediction. By default, thermostatic generator control is used to maintain the target state-of-charge while the vehicle operates in charge-sustaining mode.

Events in which energy can be saved are detected via a force breakdown on the vehicle, represented as a point mass defined as the sum of inertial, aerodynamic, and frictional forces respectively. $F_{Road\ Load}$ is positive when propelling the vehicle forward using the predicted θ via Route Prediction and V via Speed Prediction. a_v is calculated iteratively by dividing the difference of adjacent speed values (v_a and v_b) in the predicted speed trace by the average time required to accelerate to the next adjacent speed using the specified interval distance, 10 m.

$$F_{Road\ Load} = ma_v + mg * sin(\theta) + {}^{1}/_{2}\,\rho A_v C_d V^2 + C_{rr} mg. \tag{11}$$

This method assumes linear acceleration between adjacent speed values defined as

$$a_v = {}^{{v_b}^2 - {v_a}^2}/_{2d}. \tag{12}$$

Energy gained in an event, $E_{Road-Load}$, with length, $n * d_{int}$, is

$$E_{Road-Load} = \sum_{0}^{n} F_{Road-Load}(\theta_n, a_{v,n}, V_n) * d_{int}. \tag{13}$$

n is the number of time-series measurements recorded and is kept small to minimize discrepancies between predicted and actual energy spent in an optimization event. If the net energy spent over the predicted event is negative, the generator is prevented from charging the traction batteries over the length of the event, where a margin of safety is used to ensure state-of-charge does not drop too far below target if the prediction is incorrect. During a charge-gaining event, fuel flow to the engine is prevented and all torque commands to the generator motor are overridden to zero.

3.1.3 Virtual Testing Environment

To test the Driver Prediction algorithms and improvements made to thermostatic powertrain control, a set of models are created to simulate the vehicle, driver, and the environment. The environment is simulated with a city model, which is interfaced via a GPS model. A vehicle plant model represents the vehicle equipped with a vehicle specific predictive thermostatic powertrain control algorithm. Finally, the driver model allows for different types of drivers to be simulated and compared.

3.1.3.1 Simulated City

To simulate the environment the vehicle drives in, a city is simulated as an undirected graph [4, p. 400]. The nodes of the graph are the intersections of the city and the edges represent the roads segments between the intersections. Each road segment contains the speed limit and elevation at specified distances along the length of the segment. Speed limits and elevation are chosen, because both are accessible to a vehicle from services like Google Maps or a GPS navigation system. To simplify the creation of routes and increase the speed of the simulation, the simulated roads are made significantly longer than normal roads. This significantly reduces the number of intersections and roads while still having the same total drivable distance. The only downside is that the predictive thermostatic controller has fewer roads to choose from when creating the

predicted path, affecting the accuracy of predictions. However, this also makes Route Predictions faster and reduces the number of predictions that need to be made, which increases simulation speed. Considering the high accuracy of Route Prediction, over 98% [1], this limitation is acceptable.

Because a vehicle does not know if it must stop at a given intersection, each simulated intersection is split into different types. The first type resembles stop signs, where the simulated vehicle must stop. The next type is a stop-light intersection, where there is a random probability of stopping. Finally, there are destination intersections where the vehicle will only stop if the final destination is reached. In this case, the model will only expect the driver to stop if it has predicted that this intersection is the destination the driver intends.

3.1.3.2 GPS Model

Just like a GPS in a real vehicle, the GPS model provides the Driver Prediction model with the location and speed of the vehicle. Instead of acquiring the location by coordinates, the model integrates the speed of the vehicle acquired from the vehicle plant model plotting the distance against the drivers' route comprised of road links. Location is based on the current road link the vehicle is on and how far it is along that link. The entire route the driver will take, along with the current vehicle location, is passed to the driver model from the GPS model. While a normal driver would usually know the end-destination, storing this data first in the GPS model is a necessity required by the implementation details of the prediction and driver simulation models. This way, the driver model has information a real driver would have including intersection, speed, and elevation data. The Driver Prediction model shares the same data to base predictions upon, but is prevented from knowing the end-destination.

3.1.3.3 Simulated Drivers

The prediction algorithm is tested with four different types of drivers 1) an aggressive driver that drives at the speed limit but accelerates quickly, 2) a meek driver that drives at the speed limit but accelerates slowly 3) an aggressive driver that drives above the speed and accelerates quickly 4) a meek driver that drives above the speed limit but accelerates slowly. In effect, drivers are varied by nominal speed and rate of acceleration.

A single PID controller-based driver model with different proportional, integral and differential gains is used to simulate the different drivers. Low-frequency Gaussian noise is added to the driver model output to make the driver more realistic. The model has a vehicle velocity input where velocity is provided by the vehicle plant model and a predetermined route input, which includes the distance between intersections and speed limits at different sections of the route. The driver stop time at intersections depends on whether the intersection is a stop sign or a traffic signal.

3.1.3.4 Simulated Vehicle and Controllers

The simulated vehicle utilizes a series plugin, hybrid-electric architecture. Components included in the model are the traction batteries, traction motors, gearboxes, generator engine, generator motor, generator coupler, chassis, and wheels, each with specified terminal power-flow characteristics. One 200 kW motor is used per rear wheel, and an 85 kW generator is used to increase tractive power and extend range. Energy storage consists of an 18.9 kWh battery pack and a 30.3 liter on-board fuel tank providing a total range of 232 km. A diagram of the energy flow breakdown is provided in Figure 5.

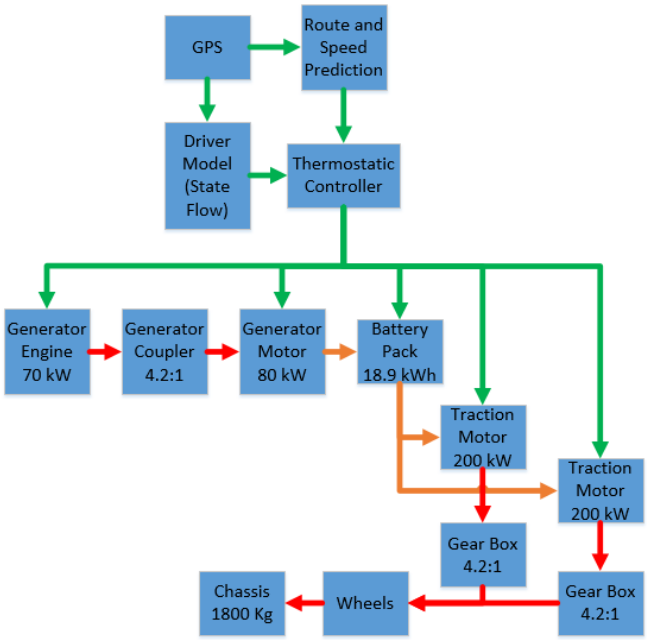

Figure 4: Full Model Breakdown

Figure 5 demonstrates the flow of control (green) from the Driver Prediction algorithms, GPS, and Driver Model to the Thermostatic Controller, which manages powertrain components where flow of mechanical (red) and electrical (orange) energy propel the chassis model. Speed feedback from the chassis model is provided to the GPS model to update current vehicle location in the virtual city.

3.1.3.5 Driving Schedules

For each driver, two sets of routes are created, where each route has a start intersection, an end intersection, and a set of links that go from the first to the second. Schedules also contain information like time-of-day and day-of-the-week that help improve prediction algorithm results. Each set shares similar routes and many have repeated routes. The goal is to create the set of routes a driver might take over the course of a month. The first set is the training set and the second is the test set. When the first set is run, the predictive algorithm is not used, because at this point the hidden Markov model is not trained for the driver leading to inaccuracy. Instead, the model simply learns about the driver. For the second set, the predictive algorithm is enabled and the predictions are used by the predictive portion of the thermostatic controller to improve fuel economy. During the simulated drive, the driver model is given the entire route including whether or not it will stop at each intersection. However, the predictive model is only given the current link and where along that link the driver is located at each time step.

3.1.4 Results

For each of the four driver types, a simulation is performed using a parallelized testing script to first train the hidden Markov model and regression tree through supervised learning. Energy usage of the predictive and conventional thermostatic propulsion controllers is then compared. Route and speed prediction are inputted to the Simulink vehicle model by interrupting simulations at every road intersection to update the GPS model, record necessary model output

such as cumulative petroleum and electric energy consumption, and provide new predictions if needed.

All four modeled drivers are tested on the same set of training and testing routes, and all tests are completed in charge-sustaining mode where target and initial state-of-charge are set to 70 % to ensure the vehicle powertrain has enough power to accelerate to requested speeds. The pre-generated city is comprised of 192 intersections and 452 roads ranging in length from 0.01 km to 3.0 km. Speed limits range from 30 to 120 kph and elevation ranges from 0 m to 700 m above sea-level. Road and climate conditions are kept constant as well as the vehicle architecture and powertrain components. Table 25 demonstrates the energy used per driver.

Table 2: Driver Type Energy Consumption

Drivers	Predictive Controller Energy Used (kWh)		Thermostatic Controller Energy Used (kWh)	
	Electric	Petrol	Electric	Petrol
Aggressive / Fast	13.03	7.659	12.39	8.124
Aggressive / Slow	12.31	6.969	11.83	7.682
Meek / Fast	12.68	6.307	11.93	6.992
Meek / Slow	12.45	5.527	11.66	6.019

An average 8.24 % of petroleum energy is saved from the thermostatic controller using predicted route and speed to skip charging cycles of the generator, yet an average of 5.5 % more electric energy is consumed from the predictive controller. Table 26 details the percent-increase in electric energy consumed and percent-decrease in petroleum energy consumed using a predictive thermostatic controller over a conventional implementation.

Table 3: Predictive Thermostatic Controller Energy Consumption Baseline Comparison Percentages

Drivers	Electric Energy Percent-Increase (%)	Petrol Energy Percent-Decrease (%)
Aggressive / Fast	4.84	5.72
Aggressive / Slow	3.90	9.28
Meek / Fast	5.91	9.80
Meek / Slow	6.35	8.17

Confidence in the predictions used are tabulated in Table 27 where the absolute error is calculated using the absolute discrepancy of the predicted and actual trace values over all testing routes within all simulations.

Table 4: Result Error Metrics

Measured Simulation Output	Absolute Error Average	Absolute Error Standard Deviation
Speed [kph]	5.833	11.33
Slope [y/x]	0.014	0.041
State-of-Charge [%]	0.1967	1.533

A small absolute error average and standard deviation in predicted speed and slope aids in justifying the small discrepancy in state-of-charge between the predictive and thermostatic controllers. The predictive controller accurately makes use of hills and decelerations to supplement petroleum powered charging cycles with regenerative braking exemplified in Figure 6 demonstrating trace data from a single testing route in a simulation. Note the activation of the 'Torque Cut-Off' and 'Regen Scaler' control signals during times of decelerations or negative slope.

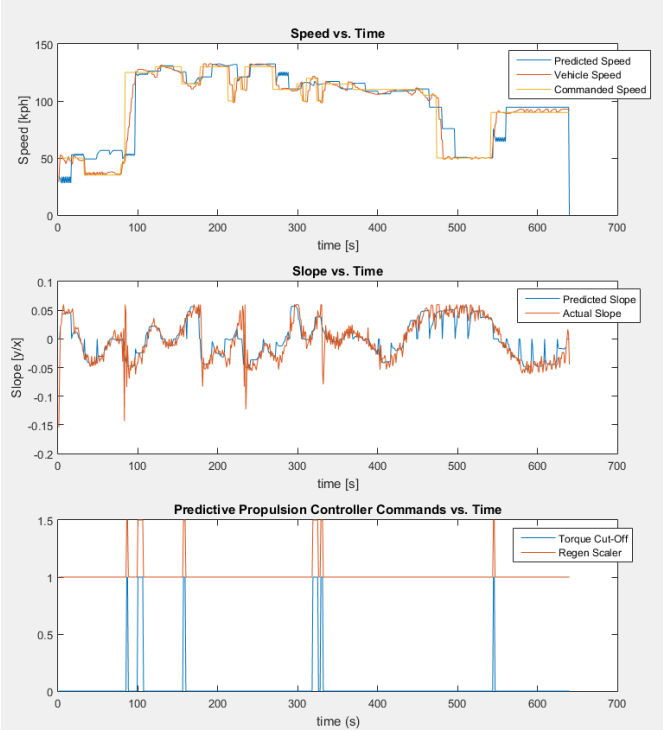

Figure 5: Speed, Slope, and Predictive Controller Data

Predictive controller command, 'Torque Cut-Off', is used to zero torque requests to the generator engine and motor and cut fuel to the engine, where 'Regen Scaler' is used to amplify negative torque requests. Effects from the same test route on energy consumption and state-of-charge are provided in Figure 6.

Vehicle simulations using the predictive powertrain controller, however, do not account for transient energy spent when the predictive controller eliminates energy consumption of the generator during a predicted charge-gaining event evidenced in Figure 7, 'Petroleum Energy Used vs. Time'. Petroleum energy saving results collected, in effect, may be higher than what would be expected if energy transients were included, because the energy required to regain generator engine speed when reloaded by the generator motor is not modeled in the simulated vehicle.

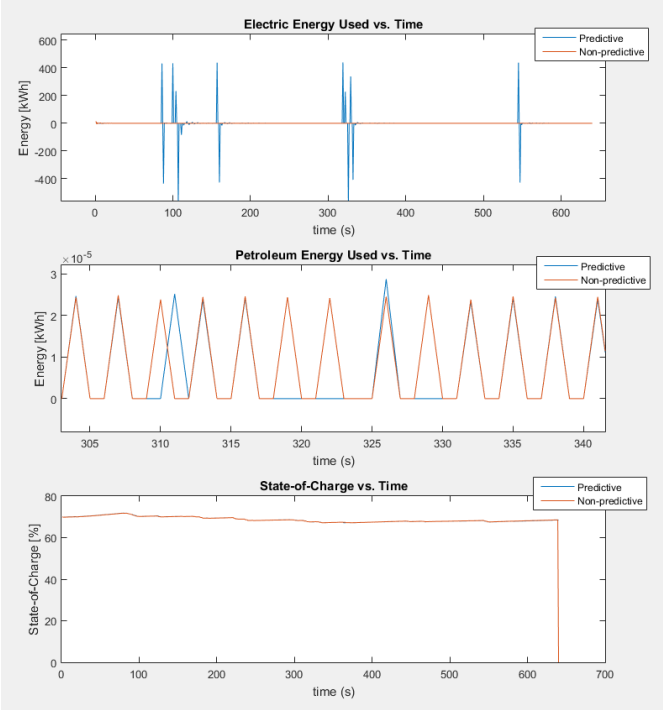

Figure 6: Prediction Controller Effects on Powertrain

Increases in electric energy flow occurs during predicted charge-gaining events to account for loss of power from the generator and increased regenerative braking torques. A zoomed-in view of petroleum energy usage is provided as well to demonstrate skipped generator charging cycles during charge gaining events. Most importantly, both controllers have similar effects on the state-of-charge of the traction batteries exemplifying how petroleum energy is supplemented by scavenging changes in vehicle potential and kinetic energy using regenerative braking.

This study presents a strategy of decreasing petroleum energy consumption of a thermostatic powertrain controller implemented in a series plugin, hybrid-electric vehicle by predicting where a driver will go and how a driver will drive using a hidden Markov model and a regression tree. Simple force vector analysis incorporating predicted speed and route is used to identify a charge-gaining event 160 meters ahead of the vehicle where there may be a decline or deceleration. These changes in potential and kinetic energy of the vehicle are scavenged using regenerative braking to supplement energy output of the series generator. To test this strategy, a vehicle, a city, and drivers are simulated in Matlab and Simulink along with both the Driver Prediction algorithms and the thermostatic control strategy. Results show that this type of predictive strategy decreases petroleum energy consumption by 8.4 % on average over a normal thermostatic propulsion controller.

3.2 Final Development Updates to Proof-of-Concept

Updates are made to the proof-of-concept design to employ the fundamental machine-learning algorithms in a real-world scenario.

3.2.1 Data Collection

Real-world road data is required for final testing. The Boost ASIO TCP/IP package is used to establish endpoint connections, request data, and read server JSON responses. Ordered collection of GPS waypoints representing road profiles are queried from Open Street Maps, stored as spline control points, and splined using a supporting Boost package to improve curvature resolution when identifying intersections. Road elevations are queried from Mapzen and are not splined. All JSON data is parsed using the Boost JSON parser and property-tree data structure.

3.2.2 Data Serialization

Road and intersection data as well as route prediction, speed prediction and trip GPS measurements are stored locally through a combination of JSON and binary serialization. Storage of data is needed to improve the boot-up time of Driver Prediction by reducing the need to re-query commonly utilized city data and to retain learned driver data from previous trips. JSON data written and read is done so using the Boost JSON parser and property-tree data structure. Binary serialization is written and read using C++11 standard library output file streams and is aggregated in Eigen matrices.

3.2.3 Neural Network Speed Prediction

The proof-of-concept Regression Tree Speed Prediction was substituted for a multi-input, multi-output neural network to improve run-time efficiency and memory complexity. Figure 8.

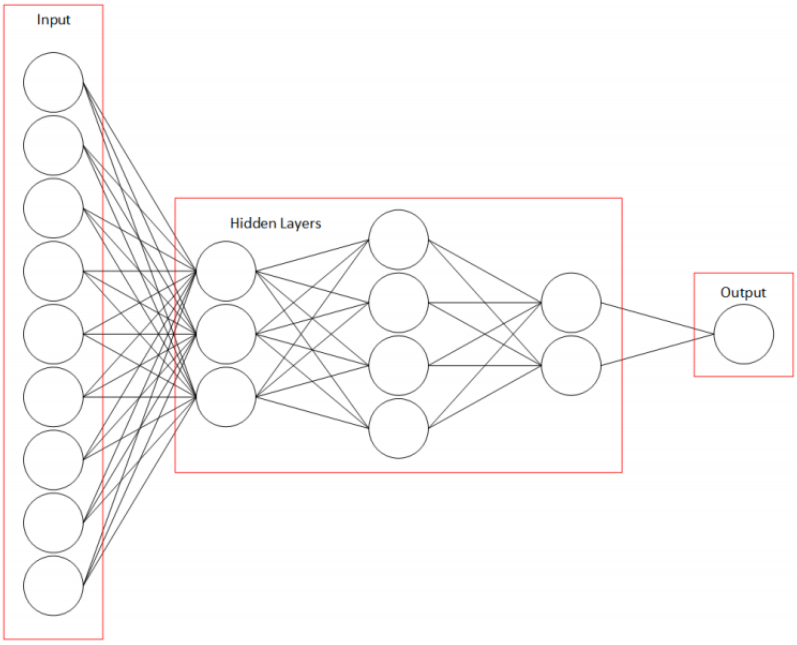

Figure 7: Neural Network Design

The network design chosen includes an input layer, any number of hidden layers and an output layer. Connections from within a layer or non-adjacent layers is forbidden. An input vector is presented to the network by setting the states of the inputs. Then the states of the units in each layer are determined by applying equations (14) and (15) from connections of previous layers. All layers have their states set in parallel.

The total input, x_j, to unit, j, is a linear function of the output, y_i. of the units that are connected to j and of the weights, w_{ij}, on these connections defined as

$$x_j = \sum_i y_i w_{ij}. \tag{14}$$

Units are given biases by introducing an extra input to each unit which always has a value of 1. The weight on the extra input is called the bias and is equivalent to the threshold of the opposite sign and is treated just like the other weights.

A unit has a real-valued output, y_i, which is a non-linear sigmoid function of its total input,

$$y_i = \frac{1}{1 + e^{-x_j}}. \tag{15}$$

The aim for each input vector is that the resultant/actual output vector produced by the network be the same or sufficiently close to the desired output vector. Assuming there is a fixed, finite set of input and output vectors (time-series speeds), the total error in the performance of the network with a particular set of weights is computed by comparing the actual and desired output vectors for every case. The total error, E, is defined as

$$E = \frac{1}{2}\sum_c \sum_j (y_{j,c} - d_{j,c})^2. \tag{16}$$

Where c is an index over cases (input-output pairs), j is an index over output units, y is the actual state of an output unit and d is the desired state. To minimize E by gradient descent it is necessary to compute the partial derivative of E with respect to each weight in the network. For a given input-output pair, the partial derivatives of the error with respect to each weight are computed in two passes. The forward pass determines unit states for all layers using equations (14) and (15) and input from units in previous layers. The backward pass which propagates derivatives from the output layer to the input layer is more complicated.

The backward pass starts by computing dE/dy for each of the output units. Differentiating equation (16) for a particular case, c, and suppressing the index c gives

$$dE/dy_j = y_j - d_j. \tag{17}$$

The chain rule is then applied to compute

$$dE/dx_j = dE/dy_j * dy_j/dx_j. \tag{18}$$

Differentiating equation (15) to get dy_j/dx_j and substituting gives

$$dE/dx_j = dE/dy_j * y_j(1 - y_j). \tag{19}$$

This shows how a change in the total input x will affect the error. The total input is just a linear function of the states from the previous layer units and is also a linear function of the weights on the connections. For a weight, w_{ji}, from the i^{th} unit in the previous layer to the j^{th} unit in the current layer the derivative is

$$dE/dw_{ji} = dE/dx_j * dx_j/dw_{ji} = dE/dx_j * y_i. \tag{20}$$

And for the output of the i^{th} the contribution of dE/dy_i resulting from the effect of i on j is

$$dE/dx_j * dx_j/dy_i = dE/dx_j * w_{ji}. \tag{21}$$

Taking into account all connections emanating from unit i

$$dE/dy_i = \sum_j dE/dx_j * w_{ji}. \tag{22}$$

Knowing how to compute dE/dy for all units in the last layer, the same procedure can be completed for all previous layers accumulating dE/dw layer-by-layer. This version of gradient descent is simple in comparison to other methods that use second derivatives and changes each weight proportionally to the accumulated dE/dw. The accumulated gradient is defined as

$$\Delta w = -\varepsilon \, dE/dw \tag{23}$$

for some error due to back-propagation, ε. The back-propagation method is improved using an acceleration method in which the current gradient is used to modify the velocity of the point in weight-space instead of its position.

$$\Delta w(t) = -\varepsilon \, dE/dw \, (t) + \alpha \Delta w(t-1) \tag{24}$$

The variable t is incremented by 1 for each layer and α is an exponential decay factor that determines the relative contribution of the current gradient and earlier gradients to the weight change [7].

3.2.1 GPS Odometer

Aside from measuring latitude and longitude to place the vehicle on the correct road within the stored road network for Driver Prediction, a GPS odometer is used to track the distance the vehicle has traveled. Differential GPS measurements and the Haversine formula are used to approximate distance traveled "as the crow flies" continuously along a given route and is defined as

$$a = sin^2\left(\frac{\Delta\varphi}{2}\right) + cos(\varphi_1) * \cos(\varphi_2) * sin^2\left(\frac{\Delta\lambda}{2}\right). \tag{25}$$

The variable a is the Haversine, λ is longitude and φ is latitude [8]. The arc-length c is calculated in radians as

$$c = 2 * atan2\left(\sqrt{a}, \sqrt{1-a}\right). \tag{26}$$

Setting *R*, the radius of the Earth in meters, equal to 6371000.0, distance traveled in meters, *d*, is

$$d = R * c. \tag{27}$$

Odometer data is used to queue route and speed prediction when the vehicle has traveled a distance equal to the prediction interval distance. [9]

3.2.2 Vehicle Diagnostics

Collecting Onboard Diagnostics Information (OBD) in real-time is used to calculate the actual energy consumed by the vehicle so it can be compared against predicted energy consumption. Actual energy consumption is calculated using the reported air-fuel ratio and O_2 measurements over serial communication,

$$m_{fuel} = {}^{m_{o2}}/_{\sigma}. \tag{28}$$

The variable m_{fuel} is fuel mass flow, σ is air-fuel ratio and m_{o2} is oxygen mass flow. Mass flow of fuel is calculated continuously over a trip. Total energy, E_{trip}, using the energy density of Octane 91 pump is defined as

$$E_{trip} = \beta * \sum_{i} m_{fuel,i} * t_i. \tag{29}$$

The energy density of fuel, β, is 0.0127 kWh/g, i is an index across all fuel mass flow measurements taken in a trip and t is the recorded time duration between measurements [10]. Vehicle diagnostics is also used to read engine load and vehicle speed.

3.2.3 Road B-Splines

A first-order b-spline with non-uniform knot vectors is fit to control points stored as nodes described in the Node section via the supporting Eigen driver package where the output *C(u)* is defined by the following equations.

$$C(u) = \sum_{i=0}^{n} N_{i,p}(u) * P_i. \tag{30}$$

The value n is the number of control points used, p is the degree of the spline, u is a knot or interval dividing the domain of the b-spline and $N_{i,p}$ is defined as follows.

$$N_{i,0}(u) = \begin{cases} 1, & u_i < u < u_{i+1} \\ 0, & otherwise \end{cases} \tag{31}$$

And

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} * N_{i,p-1} + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} * N_{i+1,p-1}(u). \quad (32)$$

Each control-point of the b-spline is associated with a basis function defined by equation (31) and equation (32). [11]

3.3 Development Resources and Code Repositories

The following resources were used to develop the proof-of-concept design in Matlab and Simulink and the final implementation in C++.

Table 5: Software Development Resources

Development Iteration	Tool	Description
Proof-of-Concept SIL Model	Matlab	Version: 2015a (academic use) Toolboxes: Prediction API (regression-tree for speed prediction), Parallel Computing (to run multiple simulations across all cores available), MATLAB (for object-oriented implementation with single run-all script and debugging)
	Simulink	For vehicle model
Final Implementation	Xcode	Version: 8.2.1
	Build Target	64-bit intel x86_64
	Compiler	Apple LLVM 8.0
	Dependencies	Boost 1.58.0, eigen 3.2.4, libC++ (LLVM C++ Standard Library with C++11 support)
	C++ Language Dialect	GNU++11 [-std=gnu++11]

All version control is performed using Github. The proof-of-concept repository can be found at https://github.com/UWEcoCAR/innovation_model. The final implementation repository can be found at https://github.com/UWEcoCAR/predictive_thermo_controller.

4. SOFTWARE IMPLEMENTATION

Driver Prediction is a powertrain optimization algorithm to improve the energy efficiency of a vehicle as a function of the drivers' driving habits. This is accomplished without hampering the performance of the vehicle or otherwise alerting the driver to the existence of the control strategy. It is a standalone code-base that ingests map, GPS and speed data to predict driver intent, well in advance, and then to forecast the amount of kWh the vehicle stands to gain or lose over the predicted route.

The control strategies independence from driver input requires it to read basic diagnostics from the vehicle and latitude and longitude from GPS to predict the drivers' end destination, route and speed over the predicted route. The majority of data percolated through the strategy is contained

and managed by a custom map class that blends array and map data structure functionality for easy data manipulation. Data management is used to serialize all pertinent information so that the algorithm may power cycle with its flashed controller without loss of connection. It is also used to gather new data as the vehicle explores unknown map territories.

4.1 System Top-Level Diagram

Below is a top level diagram depicting all major classes and inner system relevance.

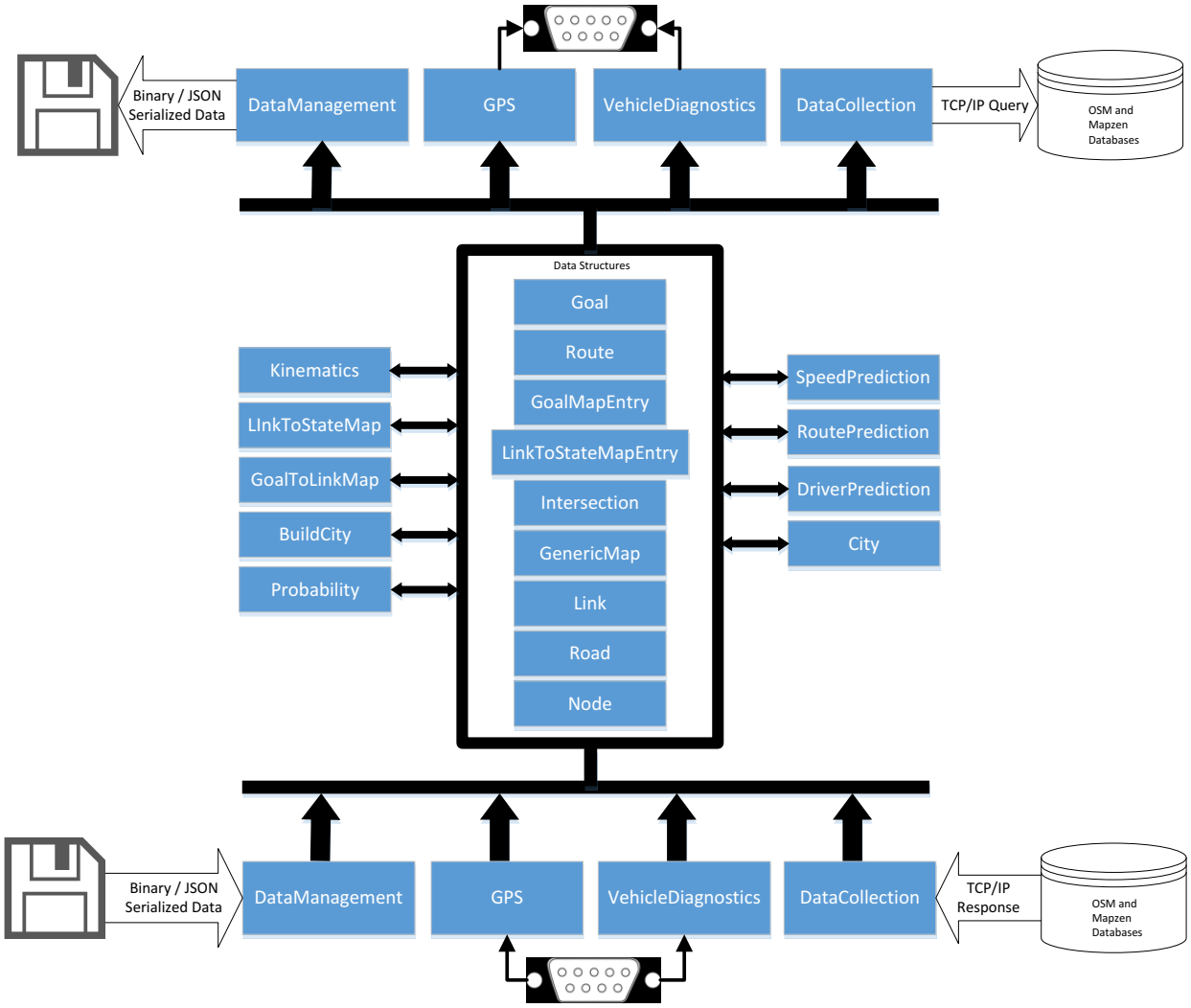

Figure 8: Top-Level System Diagram

The above top-level system diagram depicts all major classes in the Driver Prediction algorithm and their relative significance to the functionality of the system. All classes within the ‘data structures’ box have low levels of manipulative ability relative to the rest of the system. Rather, they contain vital data such as driver habits and road network data. Data busses are shown in black and are available to all classes capable of manipulating data that are not located inside the ‘data structures’ box. Data is inputted and outputted from the system frequently using the ‘GPS’ and ‘VehicleDiagnostics’ classes and upon start-up and shut down by the ‘DataManagement’ and ‘DataCollection’ classes located at the top and bottom of the diagram.

4.2 System Functional Decomposition Diagram

Below is a functional decomposition of the system.

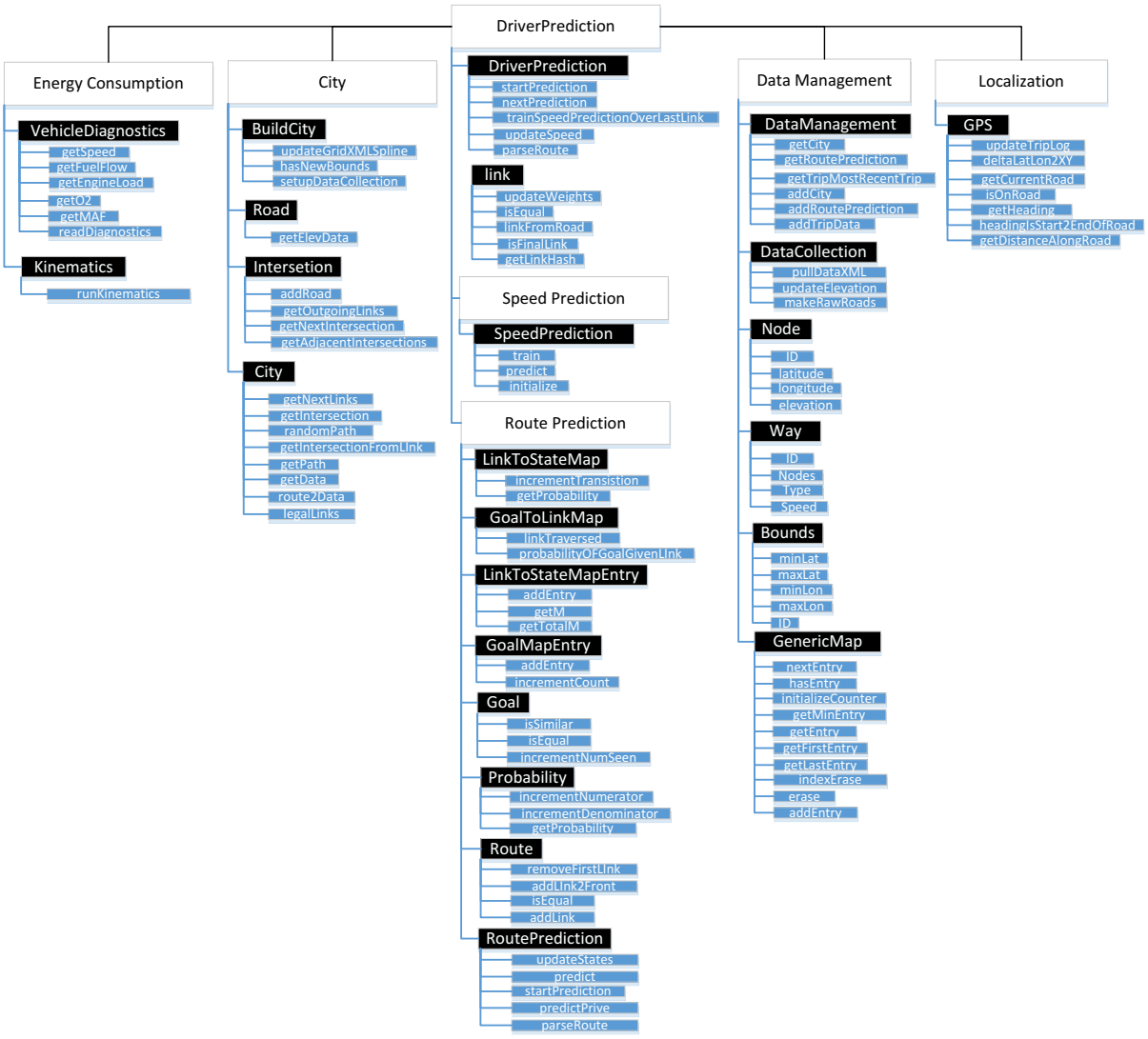

Figure 10: Functional Decomposition

The above functional decomposition diagram demonstrates the varying levels of software module encapsulation where items in white boxes are collections of classes, items in black boxes are classes, and items in blue boxes are class functions. The five major class collections are critical to system functionality and add definition to the Driver Prediction algorithm, yet have inner-dependencies as well. For instance, Driver Prediction, comprised of speed and route prediction, relies on the generic map class for ordering route prediction states and observations. Classes in data management collection serialize and de-serialize all vital Driver Prediction data. Both rely on the city class collection to interface with the locally stored road network data and to gather more. The only stand-alone class collections are the energy consumption and localization categories that rely primarily on inputting serial data from the vehicle and GPS module.

4.3 System Class Diagram

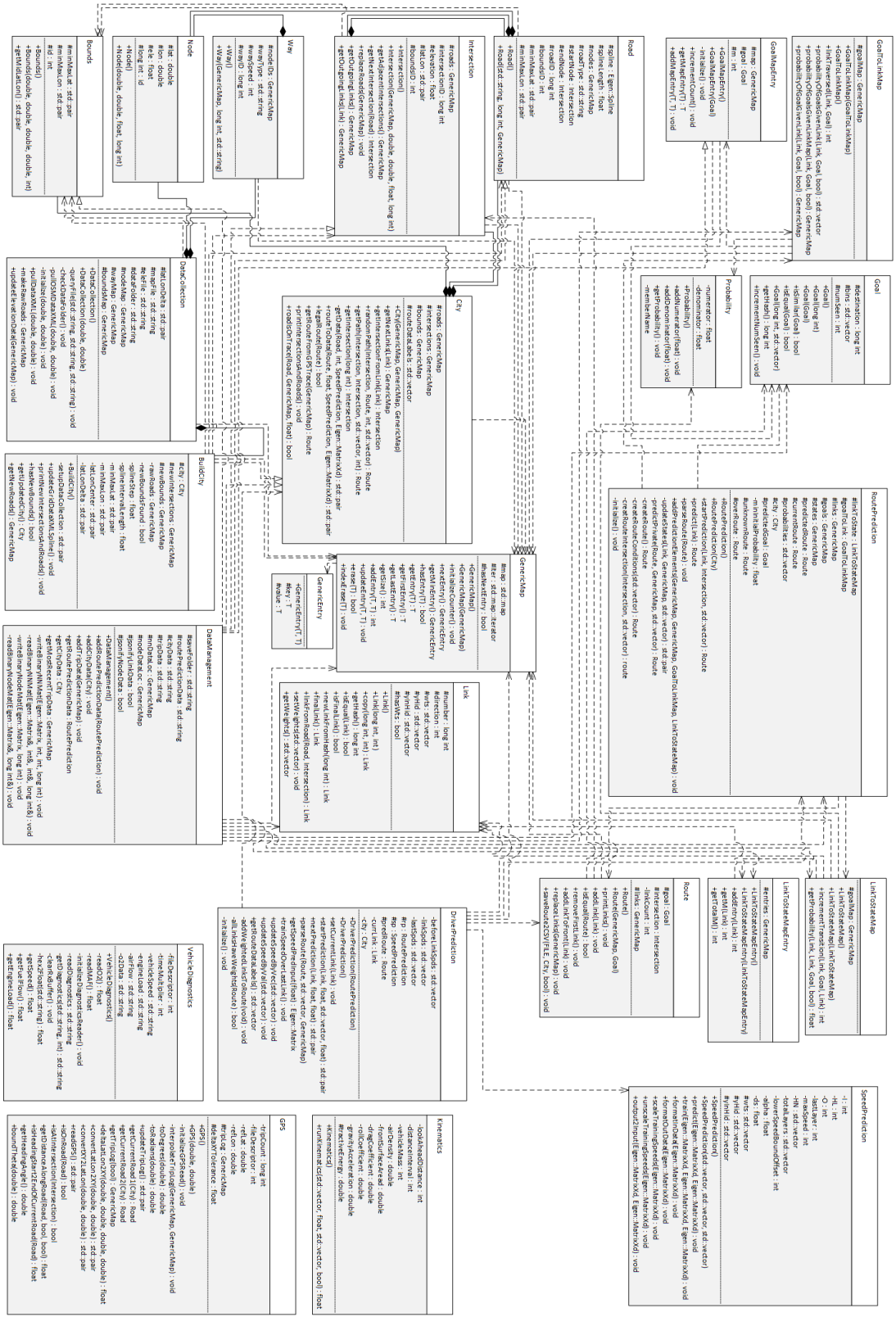

Figure 11: System Block Diagram

Inner-connectivity of the Driver Prediction software are shown by dashed and solid lines with vary levels of dependencies.

4.4 Software Modules

Each software module implementation is further described in this section. Where applicable, the submodule inputs and outputs publically accessible through getter and setter functions are defined.

4.4.1 Main

Main is the highest level of integration amongst all software modules combining output from Driver Prediction, GPS, road network interface, and vehicle diagnostics classes to localize the vehicle, predict route and speed, and calculate tractive energy over the prediction look-ahead distance.

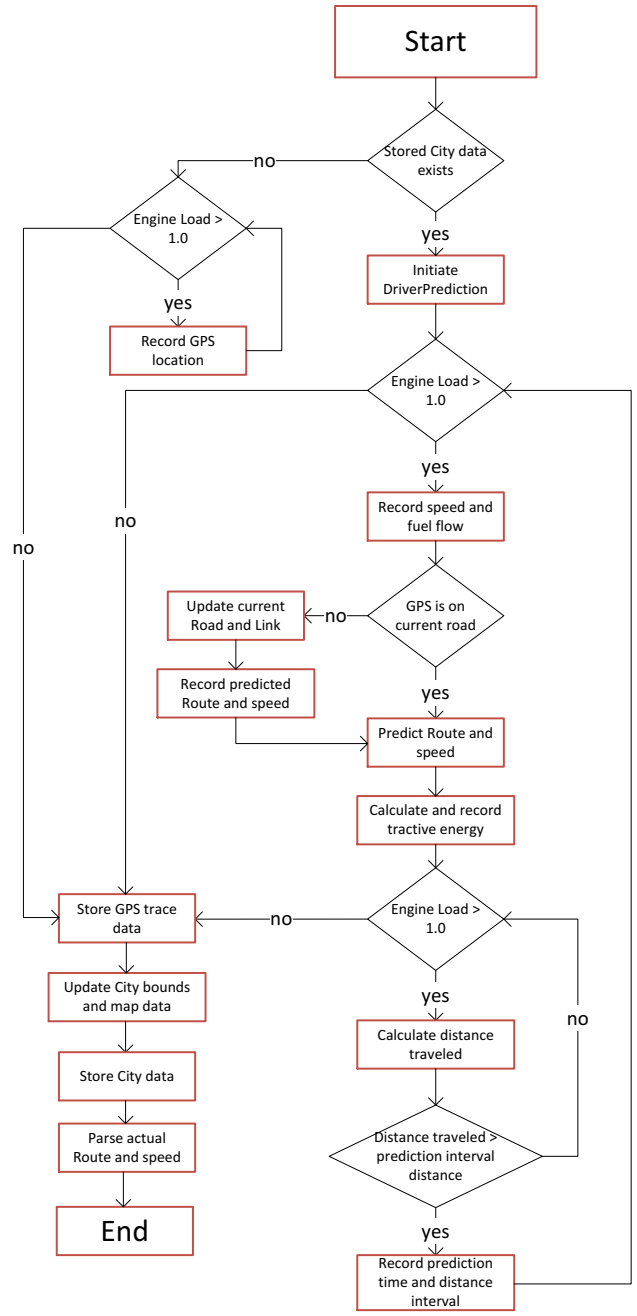

Figure 92: Main Function Activity Diagram

4.4.1.1 Initialization

First, the main classes used by the function are initialized, including the city, route prediction, speed prediction, Driver Prediction, vehicle diagnostics, GPS, and kinematics classes. Current road and current link pointers are set to 'NULL' representing the current atomic road section upon which the vehicle is located. A collection of float variables is also initialized including the distance the vehicle has travelled over the current road segment, the current conditions vector, the current vehicle speed, the total distance the vehicle has travelled over the entire power-on cycle and the prediction interval distance of the speed prediction class. A semaphore describing

vehicle heading in relation to the start intersection location on the current road segment is also initialized. Finally, vector containers of actual speed, fuel flow, execution time duration of the main logic loop, actual speed road segment labels, and predicted tractive energy are initialized.

4.4.1.2 Logic Loops

Upon start up, route and speed prediction and city data is de-serialized, and if serialization files needed for the Driver Prediction algorithm to predict are unavailable, the algorithm enters a simple logic loop of recording GPS measurements throughout the duration of the vehicle's power-on cycle where measurements are taken approximately every 1.0 seconds. However, if serialization files are available, the main logic loop of the Driver Prediction algorithm is entered. In both loops, further execution is contingent upon vehicle engine load remaining greater than one percent where engine load is defined in the VehicleDiagnostics section.

If the main logic loop is entered, meaning that Driver Prediction data is available to run the algorithm as intended, a semaphore is set signalling the subsequent prediction is the first and another variable is set to mark the start time of an execution cycle of the main logic loop, and the current GPS location is recorded. The main logic loop is officially entered. The vehicle speed and fuel flow is recorded from the vehicle diagnostics class and pushed to the end of the actual speeds vector container. These measurements are taken for visual demonstration in a post process step.

4.4.1.3 Route Prediction Update Step

GPS is then invoked to assert the current road variable, initially set to NULL, as the closest atomic road segment to the current vehicle location. If this condition initially fails, and if in further cases also fails, the current road is updated also using the GPS class. Given the updated current road, the heading semaphore is updated to associate current vehicle heading to the start intersection location of the current road. The current link is also updated as a function of the heading semaphore. If the prediction is the flagged as the first, the variable storing distance along the current road segment us updated using the GPS class, and prediction data is collected using the current vehicle location and heading Driver Prediction class. The semaphore signalling the first prediction is set to false, and if prediction data gathered from the Driver Prediction class is populated, the data is stored in speed prediction labels, buffer and csv. In this case, both route and speed prediction is performed due the update in the current road marking of the pass of an intersection.

4.4.1.4 Speed Prediction Update and Kinematics Step

The current road ID is then stored to the list of all road IDs, the variable containing the distance along the current road section is updated using the GPS class and the next bout of prediction data is gathered using the Driver Prediction class. In this case, only speed prediction is performed because the current road is unchanged. If prediction data gathered from the Driver Prediction class is populated, output is then placed in the kinematics class to calculate the tractive energy the vehicle will consume over the prediction look-ahead distance.

4.4.1.5 Main Logic Stall-Loop

The next logic loop is installed to assert whether the travel distance of the vehicle has exceeded the prediction interval distance of the speed prediction class. This is done before executing the next cycle of the main logic loop, similarly queueing off of the engine load percentage to continue the next stall cycle. First, the current GPS location is measured, and using the previous

GPS measurement, the Haversine distance is calculated between the two measurements. If the travel distance is greater than the prediction interval distance of the speed prediction class, a ratio is calculated and defined by

$$\frac{fmod(totalDistance, ds) + dist_i}{ds}. \tag{33}$$

Where $dist_i$ represent the travel distance between the current and previous GPS measurements, *totalDistance* represents the total travel distance of the vehicle during the current power-on cycle and *ds* is the prediction interval distance of the speed prediction class. The *fmod()* function yields the floating-point remainder of the first argument divided by the second. The resulting ratio is then used to buffer the actual speed measurement by adding the actual measured speed to the respective buffer a number of times equal to the integer value of the ratio. The stall loop is then broken.

The final commands of the main logic loop, record the end time of the logic loop, calculates the time delta between the start and end measurement, and stores the delta and the time buffer.

4.4.1.6 Data Serialization and Training Step

Once the engine load percentage drops below 1.0 percent, signalling the end of the power-on cycle, the build-city class is initiated. An interpolated trip log recorded by the GPS class is then serialized, and the current road network data managed by the city class is updated by the build-city class. The road network is serialized and a route over the previous trip is generated. All data container vectors are saved to csv, including the actual route, the actual route speed measurement link labels, the buffered actual speed, fuel flow, predicted tractive energy, and executions times of the main logic loop. The generated route of the previous trip log is then parsed by the Driver Prediction class to update the route and speed prediction models. Finally, the Driver Prediction class containing the route and speed prediction class data is serialized and the program ends. This function also outputs all predicted routes and speeds, actual routes, prediction interval times, interpolated trip GPS measurements, actual energy consumed by the vehicle and predicted energy consumed by the vehicle in csv format.

4.4.2 BuildCity

The build-city class is responsible for updating road network data with new roads and intersections whenever the vehicle travels outside the bounds of the known road network locally stored. As described in the Data Serialization and Training Step section, the build-city class updates a road network at the end of a power-on cycle where the predictions have already occurred.

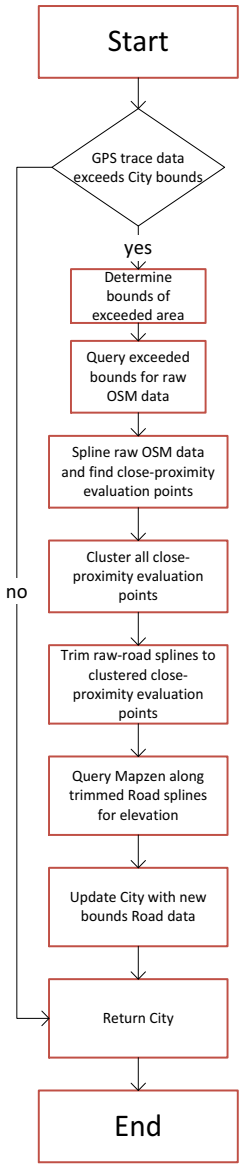

Figure 103: Activity Diagram for The BuildCity Class

4.4.2.1 Assessing Road Network Bounds and Trip Logs

Upon calling the 'updateCity' function of the build-city class, the serialized road network data is de-serialized to access all roads, intersections and bounds. If road network data exists as serialized data, the most recent serialized trip log is also de-serialized. The trip log is iterated over, comparing each GPS measurement to the inputted bounds of the road network data. If the trip log measurement is found to reside within the known road network bounds, the measurement is erased, and a new trip log measurement is assessed. Otherwise, the measurement is added to a collection of measurements that are found to reside outside the bounds of the known road network.

Once the iterative cycle completes over all GPS measurements in the most recent trip log, the collection of measurements outside of the known road network bounds is assessed. If the

collection is populated, it is also iterated over to determine the minimum and maximum latitude and longitude measurements updating locally stored min/max fields for every GPS measurement that exceeds them. Finally, the latitude and longitude minimum and maximum deltas are calculated, and a semaphore is set if new bounds are found.

If new bounds are found, a data collection class is initialized and a collection of raw road segments is queried using the center of the minimum and maximum latitude and longitude measurements calculated. A GPS class and an intermediary container of generated intersections is also initialized.

4.4.2.2 Identifying Close-Proximity Raw Road Spline Evaluations

The raw roads returned by the data collection class are iterated over to identify spline evaluation points in close proximity resembling intersections through an $O(n^4)$ operation. First a loop across all raw road segments is initialized, and then using the raw road spline length, and the desired evaluation interval distance of five meters, a spline s-value interval is calculated and defined as

$$sValueInterval = \frac{5m}{rawRoadLength}. \tag{34}$$

The s-value interval is then used to evaluate the raw road spline from s-values [0, 1.0). The two previously described steps are initiated again within the nested loop making the operation $O(n^4)$ and acheiving two spline evaluations points in latitude and longitude. The Haversine distance is then computed between the two evaluations points, and if the distance is less than approximately seven meters and the evaluated roads are not the same, the other evaluation point is added to the intermediary collection of close-proximity evaluation points. After all other road spline evaluation points have been compared to the initial evaluation point, the close-proximity evaluation points are averaged to create an intersection asserting that only one evaluation point from a given road may be averaged. The roads the close-proximity spline evaluations points belong to are also collected and added to the intersection as connecting roads. The intersection is then added to the intermediary collection of intersections.

4.4.2.3 Clustering Intermediary Intersections

The next step employs an $O(n^3)$ clustering algorithm. First an intermediary collection of clustered intersections, the output of the clustering step, is created and the last collection of non-clustered intersections is iterated over where the next iteration cycle is contingent upon the collection of non-clustered intersections being populated. Another collection of intersections closest to the current non-clustered intersection from this first loop is created, and the current intersection is added. The first nested loop is created over the non-clustered intersections not including the current intersection. The second nested loop is created over the collection of closest intersections evaluating the proximity of the closest intersection to intersection from the first nested loop. If the proximity between intersections is found to be less than 13.0 meters, the intersection from the first nested loop is added to the collection of closest intersections, and the first nested loop is restarted. If the first nested loop iterates across all non-clustered intersection without interruption, meaning all close-proximity intersections have been found, the collection of close intersections is iterated over and similar to the previous section averages all latitudes and longitudes, aggregates all connecting roads, and erases the closest intersections from the list of

non-clustered intersections to create the clustered intersection. The clustered intersection is added to the collection of clustered intersections and the first loop continues.

4.4.2.4 Forming Atomic Road Segments

The formation of atomic road segments is a more sophisticated process than the two steps prior. An atomic road segment is defined as a section of road with intersections at both ends and no intersections in between. The process begins by aggregating all existing intersections from the stored road network with the clustered intersections from the previous step, so that connections may be found between new and old road network data.

Figure 11: Build City Example with Starting Intersection and Untrimmed Connecting Roads

The algorithm begins by creating the output collection of final intersections. The first loop is created iterating across all clustered intersections from the previous step. The first action of the outer loop is collecting the connecting roads of the given intersection and creating a new collection of trimmed connecting roads. The collection of connecting roads is then iterated over in a nested loop. For each connecting road a collection of nodes is formed and the associated spline is iterated over using the s-value interval defined by equation 30 in a second nested loop. For each road spline evaluation point, the aggregated collection of existing road network and clustered intersections is iterated over in attempt of finding an intersection in close proximity to the evaluation point in a third nested loop. In this lowest level loop, distance between the current intersection and connecting road evaluation point is assessed for close proximity using the GPS class. If the intersection and connecting road evaluation point are considered close, the intersection section is stored as either the start or end intersection of the atomic road segment and an associated semaphore is set.

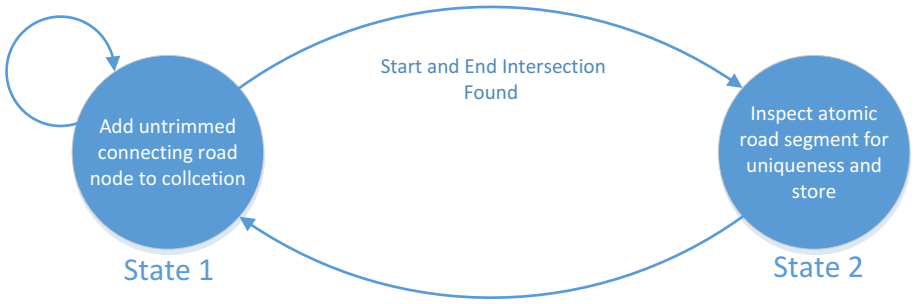

Figure 12: Flip Flop Logic of Identifying Atomic Road Segments

Flip-flop logic is used to form and identify atomic road segments, where the first intersection found in close proximity to a connecting road spline evaluation point is considered the start intersection and the next intersection found in close proximity to a different evaluation point is considered the end intersection. Until both intersections are found along the untrimmed connecting road, nodes of the road are stored. These are to be used as the nodes for the atomic road segment.

Once an atomic road segment is identified, it is asserted that one of the associated intersections is the current intersection from the outer loop. This is done to control road formation about the current intersection because, as it will be further described, it is hard to differentiate between trimmed and untrimmed road sections associated with a single intersection. It is easier to replace all untrimmed connecting roads with trimmed connecting roads. If neither of the start or end intersections is the current intersection, the semaphores and containers for the start and end intersections are reset and the second nested loop is continued. Otherwise, the adjacent intersection of the current atomic road segment is retrieved from the output collection of all intersections with trimmed connecting roads, if it exists. If the adjacent intersection exists, the trimmed road sections are iterated over to identify if the one of the trimmed connecting roads has a start and end intersection pair that matches the current intersection pair of the atomic road segment under inspection. If there exists a match, the trimmed connecting road from the adjacent intersection is added to the list of connecting roads of the current intersection and a second nested loop is continued.

If there is no intersection match between one of the trimmed connecting roads of the adjacent intersection or the adjacent intersection does not exist in the output collection of intersections with trimmed connecting roads, logic to create a new atomic road segment is entered. The nodes stored in the first state of the flip-flop road formation logic are splined, and the nodes and the spline are used to create a new road. The new road is then added to the collection of trimmed road for the current intersection from the outer loop. The semaphores and containers for the start and end intersections and the intermediary collection of nodes are reset and the second nested loop is continued.

Once the first nested loop runs to completion across all untrimmed connecting roads, the collection is replaced by the collection of trimmed connecting roads for the current intersection in the outer loop, and the current intersection is added to the output collection of intersections with trimmed connecting roads.

4.4.2.5 Elevation Update Post-Process

Finally, once the outer loop runs to completion across all clustered intersections, the output collection of intersections with trimmed roads is iterated across. For each connecting road, the elevation profile is updated by the data collection class. The output intersections and roads are added to the city class to be returned.

4.4.2.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the build-city class.

Table 6: BuildCity Getter Fields

Variable Name	Access Type	Description
rawRoads	Getter	Collection of splined raw roads that have not been trimmed between two intersections
newInts	Getter	Collection of new intersections that formed by finding raw road spline that intersect
newBounds	Getter	New bounds defined as min/max lat/lon of territory unknown to the road network
maxLat	Getter	Maximum latitude of new boundary
minLat	Getter	Minimum latitude of new boundary
maxLon	Getter	Maximum longitude of new boundary
minLon	Getter	Minimum longitude of new boundary
latCenter	Getter	Center latitude of new boundary used for querying new road data
lonCenter	Getter	Center longitude of new boundary used for querying new road data
latDelta	Getter	Delta between min and max latitude used for querying new road data
lonDelta	Getter	Delta between min and max longitude used for querying new road data

The build-city class is comprised solely of getters. It is self-contained and when constructed executes to completion in one function call; the end-result being an updated city based off of the most recent trip GPS boundaries.

4.4.3 Kinematics

The kinematics class is used to assess tractive energy using the road-load equation across the inputted speed and elevation time-series data and is relatively unsophisticated. The kinematics class is one of a handful of classes that does not have class fields accessible through getter and setter functions. Instead, a single function is used, ‘runKinematics’, that accepts the predicted speed and elevation over the predicted route and returns the tractive energy gained over the predicted route using the road-load equation.

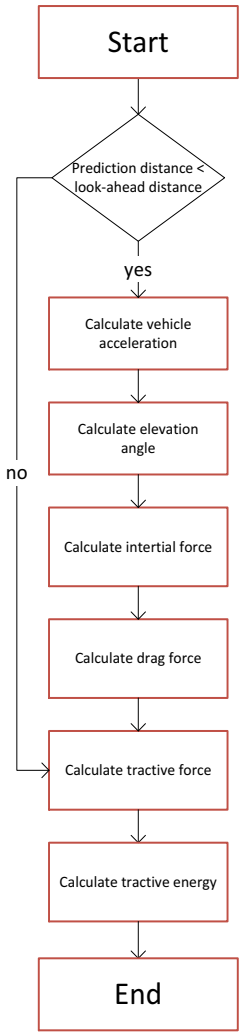

Figure 13: Activity Diagram of The Kinematics Class

4.4.3.1 Kinematics Loop

A large loop is used to assess tractive forces at every distance interval along the inputted speed and elevation traces representing all speed and elevation measurements along the predicted route. In every iteration of the loop, the acceleration from the current to the next speed value is calculated using equation (12). The road grade angle, θ_{ele}, from the current to the next elevation value, z_i, is also calculated,

$$\theta_{ele} = atan2(z_{i+1} - z_i, ds). \tag{35}$$

The value, *ds*, is the distance between elevation measurements, five meters. Using the acceleration and road grade angle, the road load equation is calculated using equation (13). The environmental parameters are set using the following values.

Table 7: Environmental Parameters

Parameter	Value	Units
Vehicle mass	1508.95	kg
Air density	1.10	kg/m^3
Front vehicle surface area	3.0	m^2
Drag coefficient	0.3	n/a
Rolling resistance coefficient	0.06	n/a
Acceleration due to gravity	9.81	m/s^2

A cumulative tractive energy variable is then added to by the tractive force exhorted on the vehicle at the current iteration multiplied by the prediction interval distance, *ds*. Once the iteration completes over the inputted speed and elevation time series, the cumulative tractive energy is returned.

4.4.4 VehicleDiagnostics

The vehicle diagnostics class is responsible for gathering data from the ELM327 v2.0 OBDII interpreter. Inputs the class vary between requesting speed, fuel flow and engine load. The vehicle diagnostics class does not have class fields accessible through getters and setters. Three functions are used to output particular diagnostics information, ‘getSpeed’, ‘getFuelFlow’, and ‘getEngineLoad’.

The unique I/O of the class comes from serial communication that ingests Onboard Diagnostics Information (OBD) II converted to RS-232 at 38400 baud from an ELM327 v. 2.0 OBDII interpreter. There is no parity, stop bit, or flow control, and data is received in 8-bit increments. This includes vehicle speed in m/s, engine load percentage, air-fuel ratio and oxygen mass flow in g/s. Updated OBDII information shall be provided to the system approximately every 3.0 seconds.

The class also requests data over the same serial bus. OBDII requests shall be outputted every 3.0 seconds.

Table 8: OBDII Data Request Messages

Request	Bytes
Engine Load Percentage	"01 04\r"
Speed	"01 0D\r"
Air-Fuel Ratio	"01 10\r"
Oxygen Mass Flow	"01 24\r"

The class also configures the OBDII interpreter by sending the following messages.

Table 9: OBDII Configuration Messages

Configuration	Bytes
Set All to Default	"AT D\r"
Print the Version ID	"AT I\r"

Echo Off	"AT E0\r"
Linefeeds On	"AT L1\r"
Headers Off	“AT H0\r”
Perform Slow Initiation	“AT S1\r”
Allow Long (>7 bytes) Messages	“AT AL\r”
Set Protocol to Automatic	“AT SP0\r”
Connect to OBDII	“0100\r”

Interpreter configuration are only sent once upon start-up.

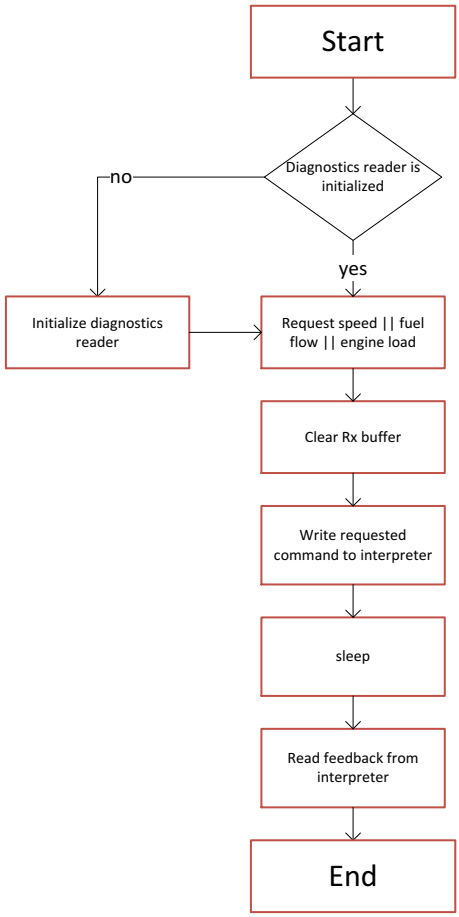

Figure 14: Activity Diagram of Vehicle Diagnostics Class

4.4.4.1 Diagnostics Configuration

Diagnostics configuration is performed only once to initialize the serial port used to communicate between the OBDII interpreter and the device executing the Driver Prediction algorithm. First, a serial port connection is established using the C++11 standard library open/read/write drivers. A Termios structure is created, and memory is set aside to configure the opened serial port. Read/write abilities are declared and the message structure is defined as having eight bits with one stop bit. No parity is used nor other forms of flow control. Finally, a one-second timeout duration is specified between Tx/Rx transmissions. The serial port control is

established and the interpreter is then configured. Table 9: OBDII Configuration Messages demonstrates the messages and message order to configure the interpreter where the carriage return is used to distinguish messages.

4.4.4.2 Requesting and Reading Interpreter Data

Two functions are used to interface with the interpreter and pull vital diagnostic information for the rest of the code-base. The 'getDiagnostics' function accepts a command from one of the publicly accessible data-request functions to be sent over serial and a time duration multiplier to allow the interpreter to read the request and return the desired information. First, the Rx buffer of the serial port is cleared using an infinite loop that only breaks until data is not yielded from the Rx buffer. Next the command is written to the serial bus. Action is stalled for a duration, t_{sleep}, calculated as a function of the time multiplier, *m*, and the request message, *cmd*, length defined as follows:

$$t_{sleep} = 25 * length(cmd) * m. \tag{36}$$

After the stall duration, the message is read using the 'readDiagnostics' function. Data is read using an infinite loop that breaks when the data is yielded by the Rx buffer as hex. If the interpreter returns "NO DATA", a zero value is returned.

4.4.4.3 Engine Load Percentage

Engine load percentage is used as an indicator the engine is still on to queue the transition between power-on and power-off cycles. It is assumed the engine is always on, when the Driver Prediction algorithm is executed. Engine load is defined as follows for reference [12].

$$engineLoad = \frac{currTorque}{maxTorque(RPM) * \frac{barometricPressure}{29.92} * \sqrt{\frac{298}{currTemp + 273}}}. \tag{37}$$

The values, *currTorque* and *currTemp*, represent the current engine torque and atmospheric temperature respectfully. Whereas, *maxTorque()* and *barometricPressure*, are engine torque as a function of revolutions per minute *(RPM)* and barometric pressure respectfully. The command message for engine load percentage to the OBDII interpreter can be found in Table 8: OBDII Data Request Messages. Response to the request message transmits in hex and thus must be converted to base-10 and manipulated to calculate the actual percentage as follows:

$$engineLoad = {}^{A}/_{2.55}. \tag{38}$$

The value, *A*, represents the first and only return byte from the OBDII interpreter.

4.4.4.4 Vehicle Speed

Vehicle speed is used to form historical speed trace vectors to be ingested by the speed prediction class to predict speed over the predicted route. The command message for vehicle speed in kilometers per hour to the OBDII interpreter can be found in Table 8: OBDII Data Request Messages. Response to the request message transmits in hex and thus must be converted to base-10. Finally, the value is returned in meters per second.

4.4.4.5 Mass Fuel Flow

Mass fuel flow is used to calculate the actual amount of energy consumed by the vehicle in a given trip using equation (29). The mass fuel flow is calculated using the air-fuel ratio and oxygen mass flow in grams per second readings from the OBDII interpreter through equation (28). Thus, two requests are used. The command messages for the air-fuel ratio and oxygen mass flow to the OBDII interpreter can be found in Table 8: OBDII Data Request Messages. Responses to the request messages transmit in hex and thus must be converted to base-10 and manipulated to calculate the actual percentage as follows:

$$airFuelRatio = \frac{2}{65536}(256 * A + B). \tag{39}$$

And

$$massAirFlow = \frac{(256 * A + B)}{100}. \tag{40}$$

The variable, *A*, is the first response value, and the variable, *B*, is the second. Again mass fuel flow is returned in grams per seconds as defined by equation (28).

4.4.5 City

The city class stores and interfaces with the known road network roads, intersections, and bounds. It provides a tool box of functions to gather predicted speed and route data over the predicted route and functions offering graph theory functionality. The most import function of the class is ‘routeToData’ which accepts a route and returns concatenated arrays of predicted speed and elevation values.

The ‘getNextLInks’ function is primarily used by route predictions and return a set of links adjacent to the argument link with respect to its direction. Similarly, ‘getIntersectionFromLink’ inputs a link and returns an intersection adjacent to the argument link with respect to its direction.

Algorithms relying heavily on graph theory are ‘getRandomPath’ and ‘getPath’. ‘[G]etRandomPath’ accepts an intersection. an initial route, and a length value and returns a random route concatenated to the initial route. ‘[G]etpath’, however, accepts a start and end intersection and returns the shortest path between the two using Dijkstra’s algorithm.

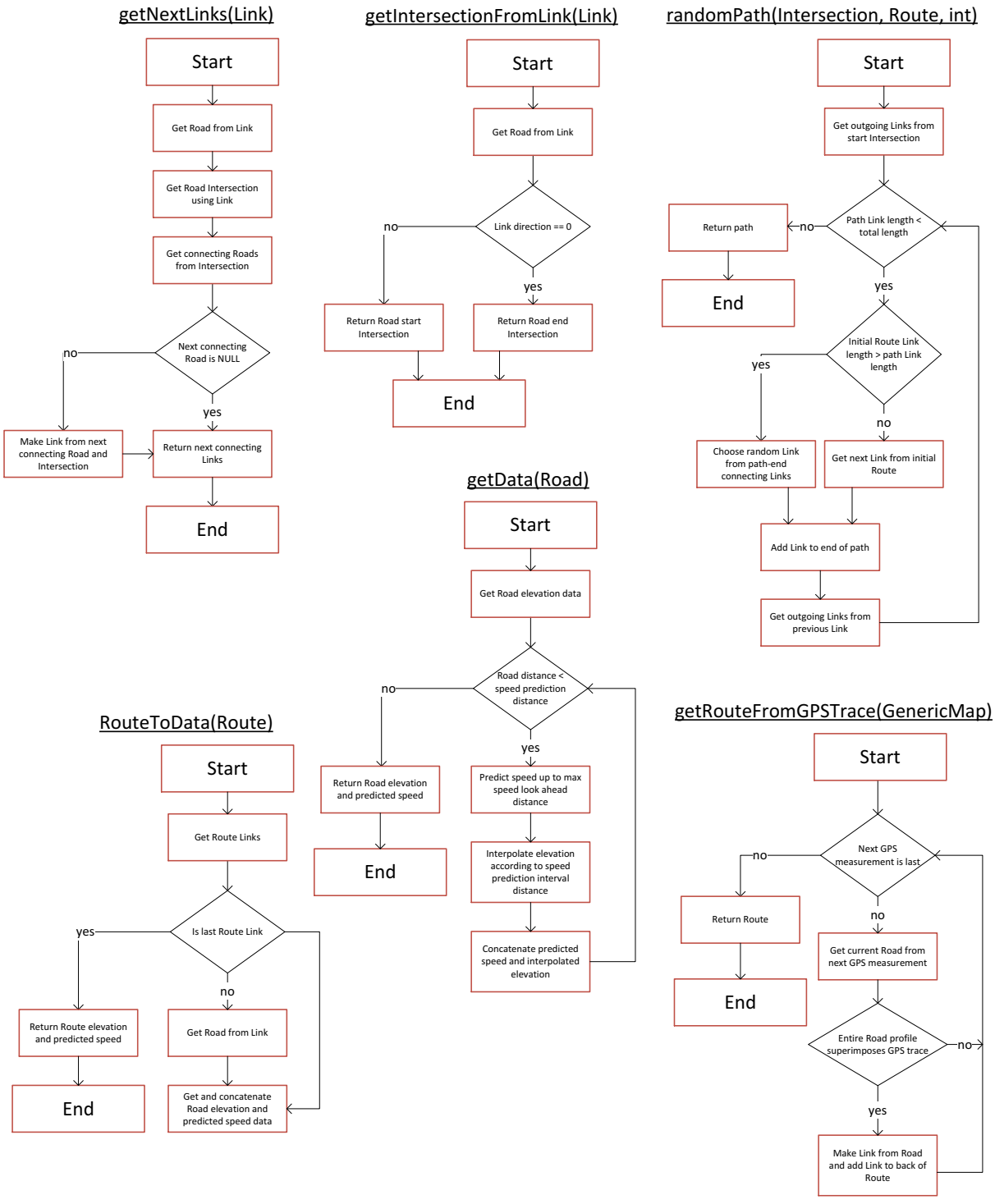

Figure 15: Activity Diagram for The City Class

4.4.5.1 getNextLinks

The ‘getNextLinks’ function inputs a link and, using the directionality of the link, returns adjacent links from the road network. First, the associated road to the argument link is retrieved

from the collection of all roads in the road network. The intersection at the end of the link is also retrieved using the 'getIntersectionFromLink' function. Then, using the returned link intersection, the collection of all connecting roads of the intersection is retrieved and iterated over. For each road not equal to the associated road of the argument link, a new link is created using the link class and inputted to a collection of links to be returned. When all connecting roads are iterated over, the adjacent links are returned.

4.4.5.2 getIntersectionFromLink

The 'getIntersectionFromLink' function is relatively simple. First the associated road to the argument link is retrieved from the road network. Then, depending upon the directionality of the link, either the start or end intersection of the associated road is returned.

4.4.5.3 randomPath

The 'randomPath' function is used primarily for testing route prediction over a simplified road network to hand-verify functionality. The function is not used routinely in the Driver Prediction algorithm, but is vital for development and testing. First, the outgoing links from the argument intersection are gathered using the 'getOutgoingLinks' function, and all links are copied to a container to prevent data lose when the link container is mutilated. Two other data containers are created to store outputted random path links and to store all intersections the random path has crossed. This is done so that no duplicates are included.

Next a loop is initialized over the desired route length. If the inputted partial route has a populated link collection, then a nested loop is created to verify the copied collection of outgoing links has at least one link equal to the i^{th} link of the inputted partial route. The variable, i, is controlled by the current iteration of the ouster loop. If there is a matching link in the copied collection of outgoing links to the i^{th} link in the inputted partial route, then the matching link is added to the collection of links in the outputted random path, and the following logic is skipped.

If the inputted partial route does not share a matching link with the copied collection of outgoing links, however, then a new nested loop is created over the copied collection of outgoing links. For each outgoing link, the associated intersection is retrieved using the 'getIntersectionFromLink' function. A second nested loop is created over the collection of passed intersections. If there is a matching intersection in this collection to the intersection associated with the current outgoing link, then the associated link is removed from the copied collection of outgoing links. A break point is placed in the outer loop in case all outgoing links have been removed. However, if more than one outgoing link exists after the nested loops have iterated to completion, meaning the associated intersections have not been passed, then a link is chosen at random using the 'rand' function of the C++11 standard library. The link is then added to the collection of links in the outputted random path.

Whether or not a match exists between i^{th} link of the inputted partial path and the copied collection of outgoing links, the associated intersection to the next link in the random path is added to the collection of passed intersections. The copied collection of outgoing links is updated using the 'nextLinks' function and the outer loop iterates. Finally, a route is created using the outputted random path links and the goal associated with the last random link found, and the route is returned.

4.4.5.4 routeToData

The ‘routeToData’ function is used to return speed prediction and elevation time-series data along the inputted predicted route using approximate five meter intervals. The function starts by clearing the link labels used to associate speed and elevation prediction data with link identifying numbers. The containers holding speed and elevation time-series data outputted by the function are initialized, and the distance along the first link of the input route is set to the argument distance inputted to the function. The loop iterating across all route links is started to concatenate predicted speed and elevation data link-by-link.

First, a road from the road network is retrieved using the current link. The inputted speed prediction class is updated with the link neural network matrix values, and the corresponding speed and elevation time-series data to the current link are retrieved using the ‘getData’ function inputting the historical speed time-series data, the distance along the current link, the speed prediction class, the link direction, and the road associated with the current link. The time-series data of the current link is concatenated onto the output speed and elevation time-series data containers, and the historical speed time-series data is left-shifted to input outputted speed data from the ‘getData’ function. The distance along the current link is set to 0.0 meters, and the outer loop continues to iterate until the final link of the route is reached. The concatenated time-series data containers are returned afterwards.

4.4.5.5 getData

The ‘getData’ function is used to return speed prediction and elevation time-series data along the inputted road segment using approximate five meter intervals. First, elevation and cumulative distance time-series data is retrieved from the argument road using the ‘getElevData’ data function, and if the argument direction semaphore is true, the elevation data is reversed. Next, variables are initialized including containers for speed and elevation to be returned, the total prediction distance, a semaphore signalling whether the prediction distance is greater than the argument road length, and intermediary containers for time-series speed data I/O for the ‘predict’ function of the speed prediction class. Lastly, historical speed data inputted to the function is scaled and offset by the ‘formatInData’ function of the speed prediction class, so that input data values ranging from zero to 100 m/s do not exceed the possible output of the objection sigmoid function used by the neural network which ranges from 0 to 1.

A loop is created with the next iteration contingent upon the total prediction distance being less than the distance of the argument road. In each iteration, the ‘predict’ function of the speed prediction class is called, inputting formatted speed prediction time-series data.

A nested loop is created over the retrieved speed prediction times series data where the prediction distance is incremented by five meters in every iteration. If the prediction distance is greater than the argument distance along the argument road but less than the argument road length, the cumulative distance values stored in the time-series data returned by the ‘getElevData’ function are assessed iteratively to find the index, *i,* where the prediction distance is first found to be less. Elevation measurements are then interpolated using the i^{th} and $i^{th} + 1$ indices of the elevation and cumulative distance time-series data containers belonging to the argument road.

$$\Delta roadElev_i = roadElev(i) - roadElev(i+1) \tag{41}$$

And

$$\Delta roadDist_i = roadDist(i) - roadDist(i+1) \tag{42}$$

And

$$\Delta pred2RoadDist_i = predDist - roadDist(i). \tag{43}$$

The $\Delta pred2RoadDist_i$ variable defines the distance between the prediction distance and the last distance measurement of the road. And finally

$$elev_{interp} = roadElev(i) + \frac{\Delta pred2RoadDist_i}{\Delta roadDist_i} * \Delta roadElev_i. \tag{44}$$

The interpolated elevation and speed values are concatenated to the output data containers after the output speed value is unformatted to resemble a speed value in meters per second. Additionally, the road identification number is concatenated to the label container belonging to the city class to accurately associate and compare predict and actual speed values for a given road. If all speed values outputted by the 'predict' function are iterated across, the intermediary input container is left-shifted and populated with output speed data that remains scaled and offset, and execution of the outer most loop continues. Once the prediction distance exceeds the argument road length, the concatenated speed and elevation containers are returned.

4.4.5.6 Getters and Setters

Below is a list of fields accessible through getters and setters for the city class.

Table 10: City Getter and Setter Fields

Variable Name	Access Type	Description
roads	Getter/Setter	Collection of all roads in road network
intersection	Getter/Setter	Collection of all intersection in road network
bounds	Getter/Setter	Collection of all bounds in road network

All fields of the city class are accessible and settable for easy serialization and de-serialization. Also, roads, intersections, and bounds may be updated when more road network data is ingested requiring convenient access.

4.4.6 Road

The road class stores vital road network information and is the graph theory equivalent to the edge. It is data structure holding spline control points that describe curvature and elevation along the road and is defined as an atomic road segment with an intersection at either end but none in between. Multiple adjacent roads constitutes a route. Most functions of this data structure are getters and setters to retrieve the set of control points, end intersections, and identifying number. The only public function to this class is 'getElevData' that returns an array of elevation measurements and their respective distances along the road segment where the measurements reside.

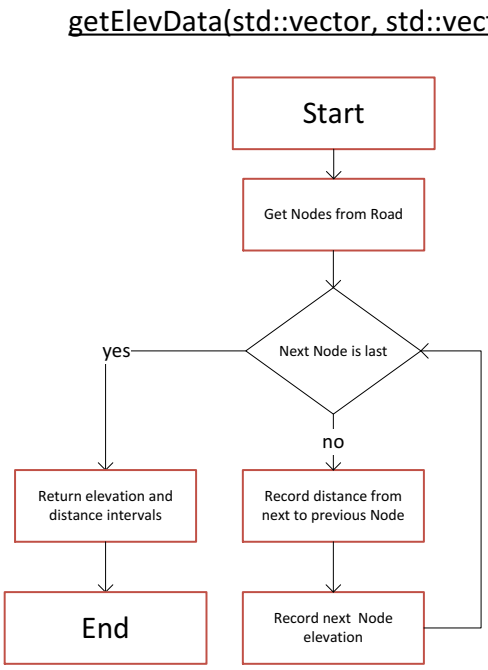

Figure 16: Activity Diagram for The Road Class

4.4.6.1 getElevData

The 'getElevData' function, however, is used to retrieve all elevation and cumulative distance time series data with approximate five meter intervals. Elevation time-series data are vectorised values stored in the spline control points, but cumulative distance data is the vectorised cumulative sum of distances between spline control points calculated using the Haversine formula from equation (25).

The function begins by instantiating a GPS class, a variable holding the cumulative sum of distance deltas and output vectors for elevation and cumulative distance time-series data. The first two control points from the collection of road spline control points are retrieved as the 'next' and 'previous' control points, and a loop is created over the remaining control points. For each loop iteration, the distance between the 'next' and 'previous' control points is calculated using the GPS class and added to the cumulative distance sum, which is concatenated to the distance vector. The elevation of the 'next' control point is concatenated to the elevation vector, and the 'next' and previous' control points are updated such that the 'previous' equals the 'next' control point and the 'next' equals the next in the collection of road spline control points. The elevation and distance vectors are returned at the end of the loop.

4.4.6.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the road class.

Table 11: Road Getters and Setters

Variable Name	Access Type	Description
spline	Getter/Setter	Spline of the road through the node control points

splineLength	Getter/Setter	Length of road in meter
nodes	Getter/Setter	Collection of GPS waypoints containing elevation and used as control points for splining
maxLat	Getter	Maximum latitude
minLat	Getter	Minimum latitude
maxLon	Getter	Maximum longitude
minLon	Getter	Minimum longitude
startIntersection	Getter/Setter	Starting intersection of the road
endIntersection	Getter/Setter	Ending intersection of the road
roadID	Getter/Setter	Identifying number of road
boundsID	Getter/Setter	Identifier of the bounds the road resides in

Most fields of the road class are accessible through getters and setters because the fields are serialized and de-serialized on every system power cycle.

4.4.7 Intersection

The intersection class is similar to the road class in that it primarily stores vital road network data with the graph theory equivalent being the vertex. The data structure contains a list of connecting roads and is defined as the unique intersection of three or more roads. Like the road class, most functions of this data structure are getters and setters for the latitude / longitude, elevation, and identifying number. However, a four graph theory function exists.

The ‘addRoad’ function accepts a road and adds it to the collection of connecting roads. The ‘getOutgoingLinks’ function can accept a link, yet does not require it. If a link is passed as an argument, it is excluded in the returned list of links connecting to the intersection. The ‘getAdjacentIntersections’ function simply returns a list of all immediately connected intersections. Whereas, the ‘getIntersection’ function accepts a road and returns the single connecting intersection at the opposite end of the road segment.

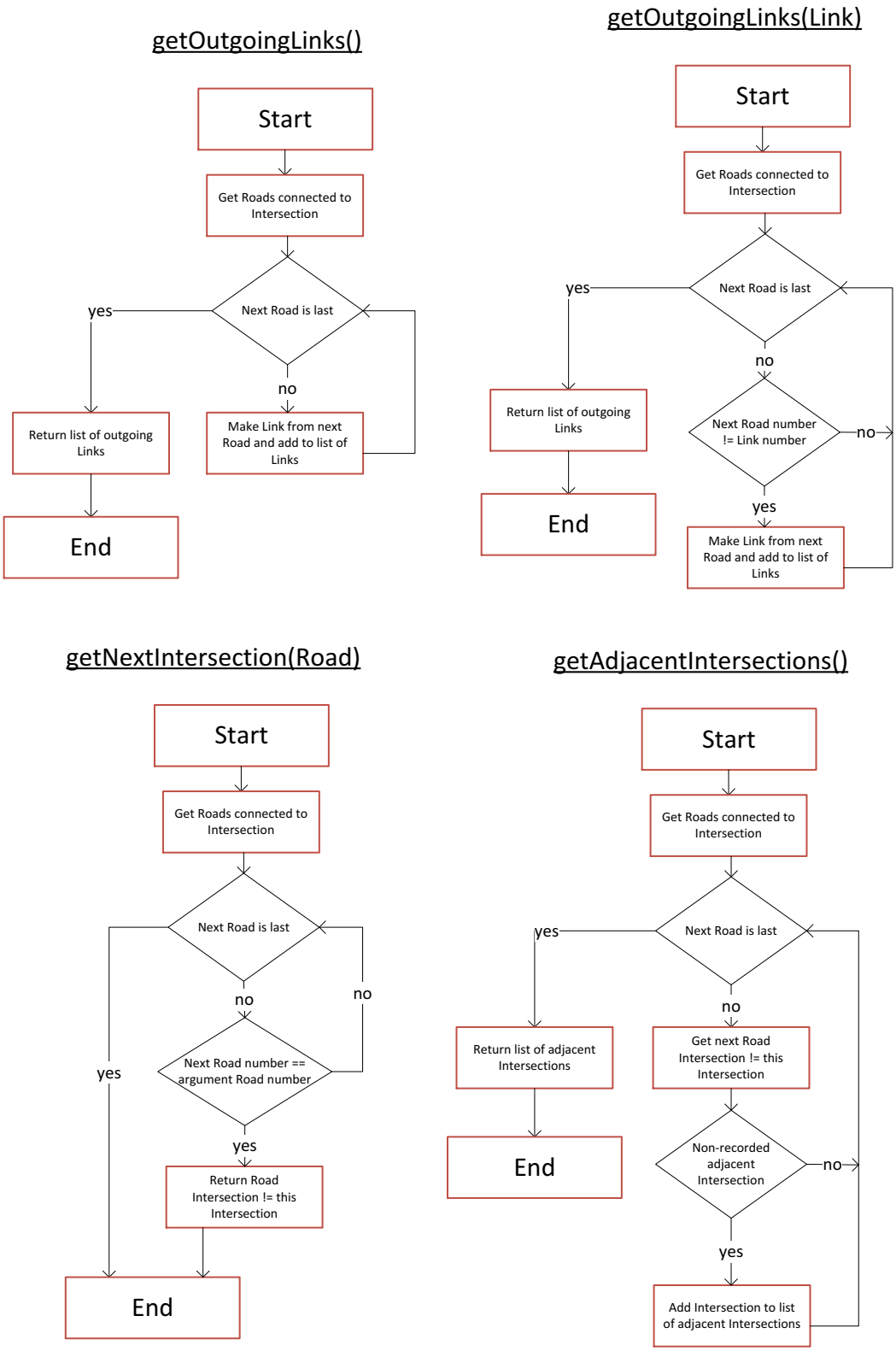

Figure 17: Activity Diagram for The Intersection Class

4.4.7.1 getOutgoingLinks

The ‘getOutgoingLinks’ function is primarily used by the route prediction class to retrieve all connecting roads of the intersection represented as links to recursively update states and state probabilities. The link class is used to represent connecting roads, for the route prediction class utilizes link hashes in associative hash maps. Additionally, the speed prediction class relies on neural network weight matrix arrays stored by the link. The function has two variations, one that accepts an argument link and one that does not.

In both function types, the connecting roads of the intersection are iterated over. For each connecting road, an outbound link is created using the ‘getLinkFromRoad’ function of the link class, the current connecting road and intersection. The resulting link is added to a collection of links to be returned. If the function passed an argument link, however, the generated link from the connecting road is not added to the collection of links to be returned if the argument and generated links share the same identifying number. The collection of output links is returned.

4.4.7.2 getNextIntersection

The ‘getNextIntersection’ function is simply made by accepting a road, and returning the opposite end intersection of the road, if the road exists in the set of connecting roads for the current intersection. The collection of connecting roads is iterated across, and if the current connecting road has the same identifying number as the argument, if/else logic is used to compare the end intersections of the connecting road and the current intersection. Simply, the connecting road end intersection with an identifying number unequal to the current intersection is returned. If no connecting road is found with the same identifying number, a void pointer is returned.

4.4.7.3 getAdjacentIntersections

The ‘getAdjacentIntersections’ function returns all opposite intersections of all connecting roads to the current intersection by iterating across all connecting roads. For each connecting road, the opposing intersection is retrieved using the ‘getNextIntersection’ function. A nested looped is created across all found adjacent intersections, and for each adjacent intersection found, the identifying numbers of the next intersection of the connecting road and the current adjacent intersection are found. If the identifying numbers match, the outer loop continues, but if no match is found, the next intersection of the current intersection is added to the collection of adjacent intersections. The second nested loop is used to ensure a set of adjacent intersections is returned without any redundancies resulting from two different connecting roads with the same intersections on either end. The collection of adjacent intersections is returned once the outer loop runs to completion.

4.4.7.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the intersection class.

Table 12: Intersection Getter and Setter Fields

Variable Name	Access Type	Description
connectingRoads	Getter/Setter	Collection of all roads connected to the intersection
ID	Getter	Identifying number of the intersection
elevation	Getter	Elevation of intersection

lat	Getter	Latitude of intersection
lon	Getter	Longitude of intersection
boundsID	Getter	Bounds ID that the intersection resides in

Similar to the road class, most fields of the intersection class are accessible through getters. This is because the fields are serialized and de-serialized on every system power cycle. The key difference between this class and the road class is that most fields are set upon class construction.

4.4.8 DataCollection

The data collection class is responsible for retrieving all road network data from the OpenStreetMap and Mapzen servers, parsing the retrieved data, and creating a raw collection of road splines that are untrimmed to atomic units. Data is transmitted over TCP/IP in which server responses are JSON/XML formatted and data is requested by a unique URL comprised of region bounds or GPS waypoints. Raw input to the class from the targeted servers consists of ordered collections of road sections, GPS waypoints along the road sections, and elevation values along the road sections. Multiple functions are used to modularize the data collection process and provide varying levels of processing to the rest of the Driver Prediction code-base.

Two public functions are utilized by this class. The ‘pullDataXML’ accepts a latitude and longitude and using the lat/lon delta fields of the class queries OSM road network data. The ‘updateElevationData’ function accepts a road and updates the spline control points with elevation data queried from Mapzen. Finally, the ‘makeRawRoads’ function makes and returns a collection of roads generated from OSM data.

This class ingests Open Street Maps (OSM) and Mapzen server response data over TCP/IP to aggregate road curvature and elevation data respectively. Updated OSM and Mapzen data is received whenever the vehicle travels outside of the bounds of the known road network

The system also outputs requests to the OSM and Mapzen servers over TCP/IP to aggregate road curvature and elevation data. Server requests are also sent whenever the vehicle travels outside of the bounds of the known road network.

```
request_stream << "GET " << getCommand << " HTTP/1.0\r\n";
request_stream << "Host: " << serverName << "\r\n";
request_stream << "Accept: */*\r\n";
request_stream << "Connection: close\r\n\r\n";

// Send the request.
boost::asio::write(socket, request);
```

Figure 18: TCP/IP Message Communicated to Endpoint

Commands (‘getCommand’) and servers names (‘serverName’) shall vary depending upon the server requested once an endpoint connection is established.

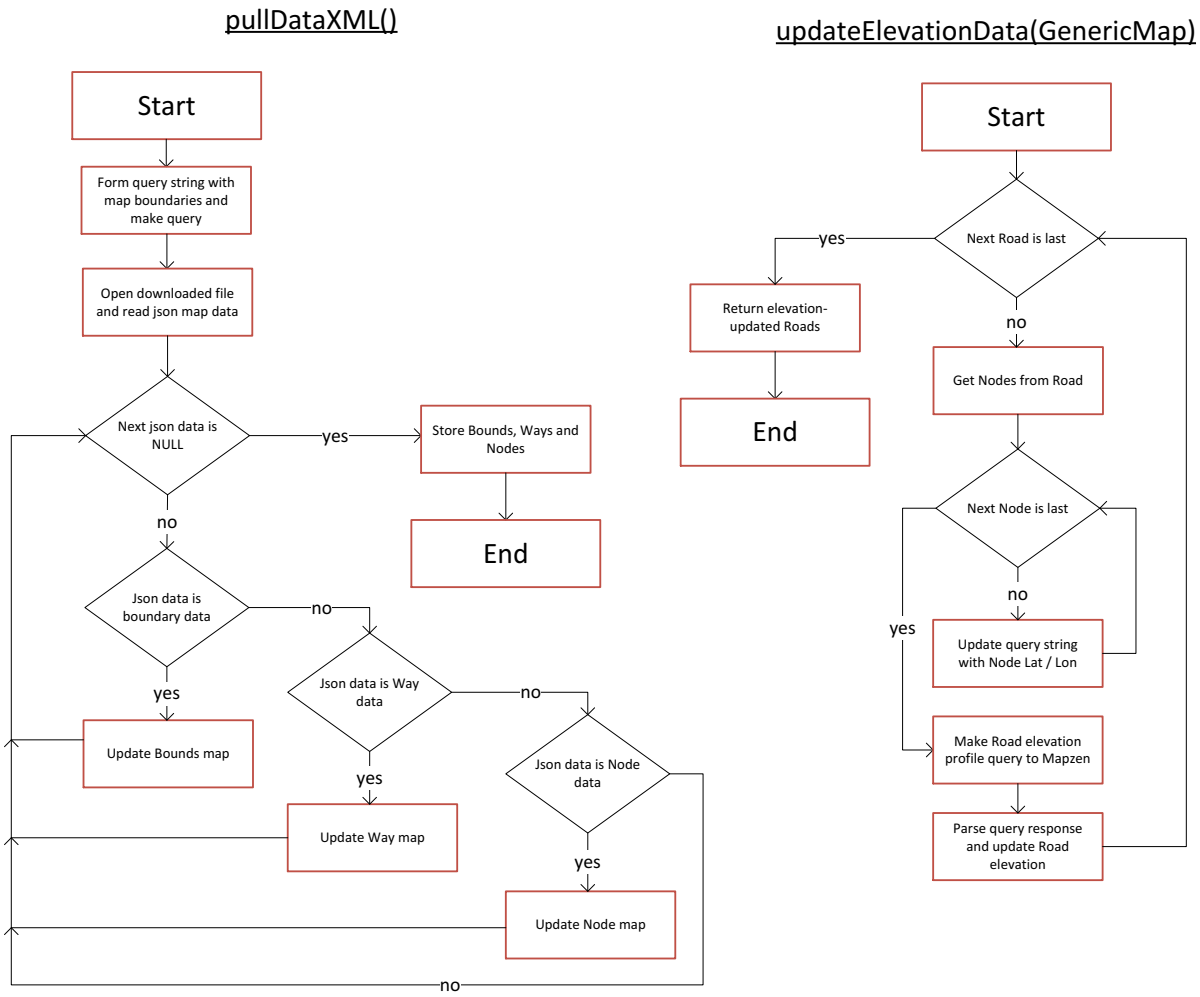

Figure 19: Activity Diagram #1 of The DataCollection Class

4.4.8.1 pullDataXML

The 'pullDataXML' function gathers road curvature data about the argument latitude and longitude inputted as a function of the latitude and longitude delta values either passed to the class constructor or set by default. The default latitude and longitude deltas used are set to 0.0002 degrees to query a square region of approximately 60x60 square nautical miles that cover the length of an average trip.

The first step to road curvature data acquisition is to build a file path using the argument latitude and longitude to see if the data needed has already been queried. If the server response data has not already been stored in the generated file path, the request is built to query the server also using the latitude and longitude arguments. The server is then queried for data using the server name and command passed as arguments to the 'queryFile' function, and the response is stored in the generated file path.

Server: “overpass-api.de”
Command: "/api/map?bbox=-122.330938,47.618074,-122.330738,47.618274000000007"

Figure 20: OSM Query Example

Above is the server used to gather data and an example command. The request is stored in the generated file path. The XML formatted response is parsed using the supporting Boost driver. Property-tree data structure stores an arbitrarily deep-nested tree structure, where each node stores keys to sub-tree nodes. The XML formatted server response file, containing road curvature data, is read using the Boost ‘read_xml’ function. It is represented in the following property-tree structure:

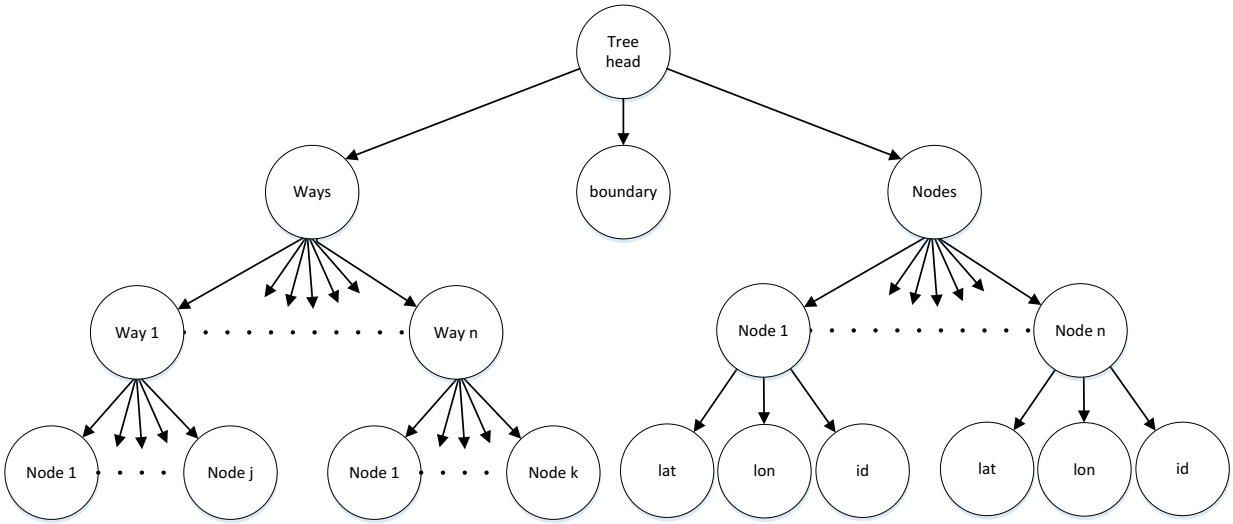

Figure 21: OSM XML Data Structure

A depth-first search is used to iteratively search each sub-tree to pull information from the lowest levels first and then across all nodes of each sub-tree level. Node data is stored in a node class and placed in a collection to be associated with a way first. Once all nodes are aggregated, they are associated with every way parsed from the property tree and used to create a collection of way classes, a structure similar to the road class. In this process, the association is a collection of node identifying numbers that are stored in the way. The primary difference between the road and way classes is that the way class only possesses a collection of keys to nodes in the collection of nodes stored by the data collection class. The function ends once all ways and nodes are parsed through the depth first search and the way and node class collections are populated with all server response data.

4.4.8.2 updateElevationData

The ‘updateElevationFunction’ receives a collection of argument roads and assembles a query of elevation data along the control points of each road to be sent to the Mapzen serve. To do so, a loop is created across all roads in the argument collection. Much of the query string remains the same and is defined in the following figure:

```
Server:     "elevation.mapzen.com"
Command:    "/height?json="
API Key:    "&api_key=mapzen-4mUVq1A"
```

Figure 22: Mapzen Query Constants

For each road, a unique JSON string is formed to retrieve the elevation data that is concatenated to the end of the command statement. This is performed using a nested loop across all nodes of the current road, forming an ordered collection of JSON objects that contain the latitude and longitude of each node control point. Once the JSON query string is formed, the intersections of the road are inspected to see if an elevation value has been associated. If not, the latitudes and longitudes of the intersections are added to the end of the JSON query string. Finally, the command, JSON query and API key strings are concatenated and passed to the 'queryFile' function with the server name. The resulting JSON-formatted response file is always overwritten to reduce the build-up of data.

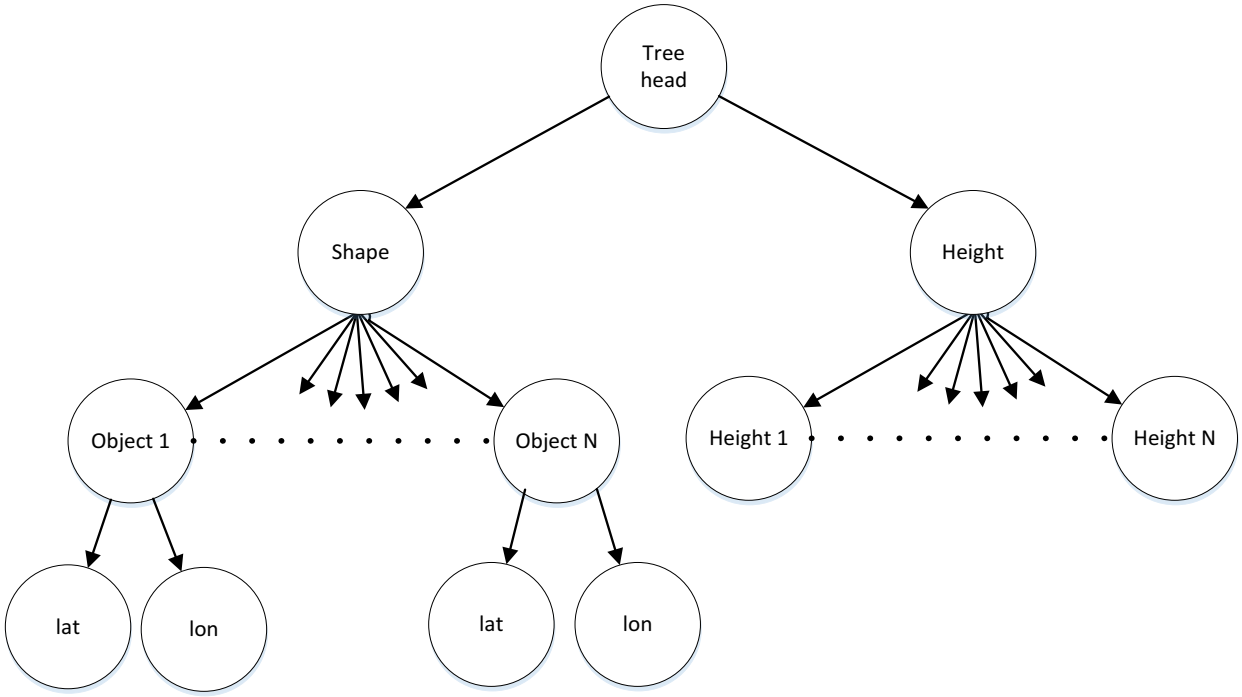

Figure 23: JSON Formatted Mapzen Response Structure

The JSON-formatted response file is read using the Boost 'read_json' function with the elevation data represented in the above property tree structure. Like the 'pullDataXML' function, data is read from the property tree in an iterative depth-first search along the sub-tree containing height data. A count of the height response values read is used to associate the corresponding road spline control point nodes. Order is preserved from the collection of control point nodes to the JSON query string and to the response data from the Mapzen server. Once all elevation measurements associated to the road spline control points have been iterated over and updated in the node data structures of the current road, the final elevation response values are associated to the road intersections if the elevations values are needed. The outer loop continues to iterate until

all roads in the argument collection have elevations associated with the spline control point nodes.

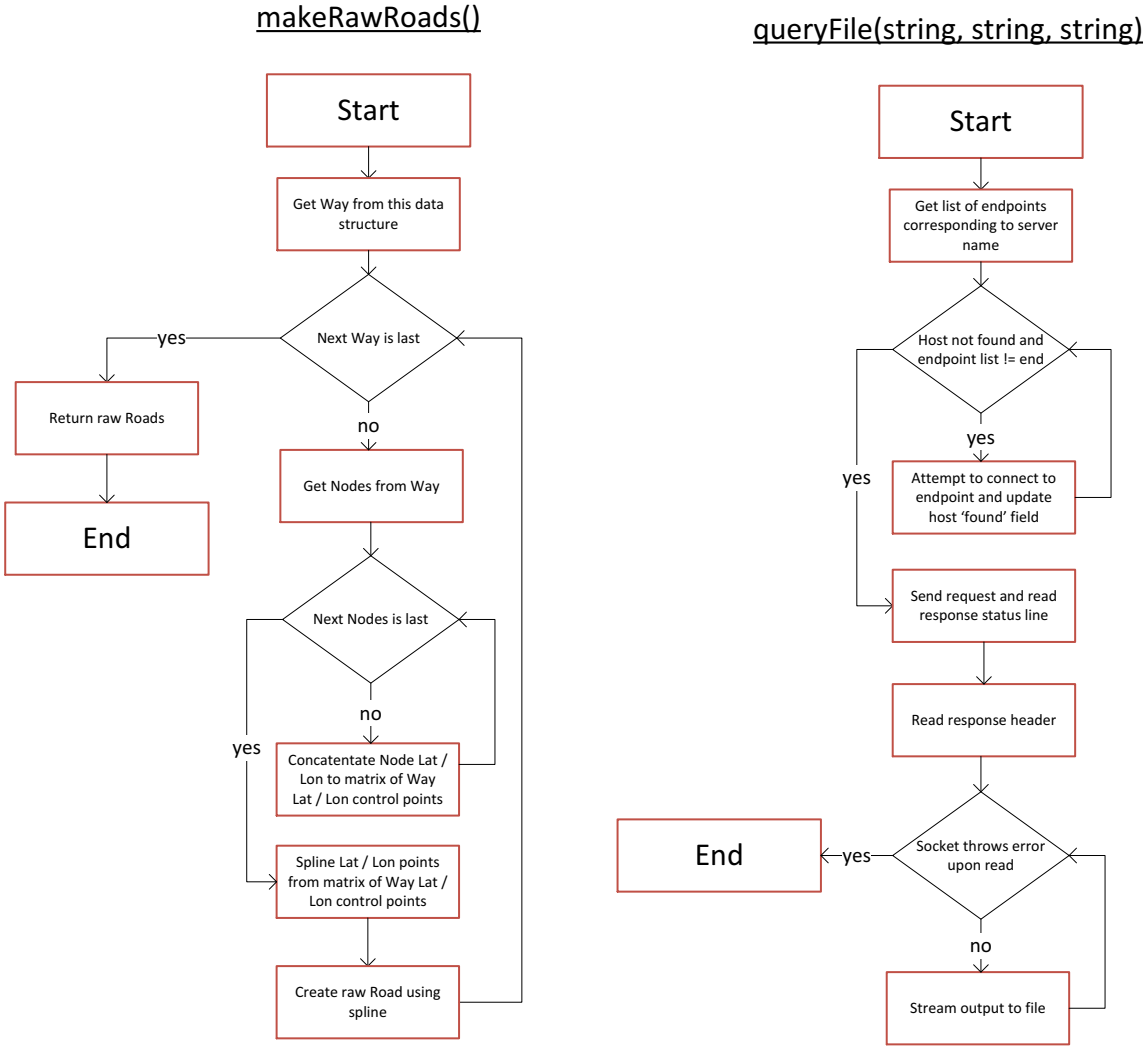

Figure 24: Activity Diagram #2 of The DataCollection Class

4.4.8.3 makeRawRoads

The 'makeRawRoads' function is used to spline the collection of way classes stored by the data collection class using approximate five meter intervals. The process is performed by instantiating a GPS class and a loop across the collection of way classes. For each way, a collection of node identifying numbers is also iterated over in a nested loop. For each identifying number, the corresponding node is retrieved from the collection of nodes stored by the data collection class and stored in a locally scoped container to the outer loop.

If the locally scoped collection of nodes has a size greater than two, logic is entered to create a road class, for at least two points are required to form the simplest straight line road. The first step forms a matrix of latitude and longitude values for each node by iterating over the locally scoped collection of nodes in a nested loop. The spline defined by equation (30) is then evaluated

on the s-value interval [0,1) to determine length of the represented road using another nested loop. Cumulative length is calculated using the Haversine formula defined by equation (25) from the GPS class. Finally, the road class is populated using the locally scoped nodes, the road distance, and road spline and added to the collection of raw roads to be returned. The collection of raw roads is returned once all nodes are splined for each way.

4.4.8.4 queryfile

The 'queryFile' function is used to form an endpoint connection with the argument server. TCP/IP is used to request data specified by the argument command and to store server response data in a local file. All network communication is performed using this function for the 'pullDataXML' and 'updateElevationData' functions. The Boost tcp::resolver class is first used to form an IO resolver instance and query instance. This initiates the resolver class for sending a query and receiving data using the argument command. Once the resolver resolves the query, a list of possible end-point connections is returned and iterated over to identify a socket capable of establishing a connection.

A socket class is instantiated, and once the successful endpoint connection is made, the socket is connected to the endpoint and the loop over all endpoints is broken. A request string is created and written to the socket defined by Figure 18: TCP/IP Message Communicated to Endpoint. The response status line is read from the socket using the 'read_until' and the argument '\r\n' is used to signal that only the first section of the response is to be read. Status message and status code fields are populated as a result to be viewed by the debugger to ensure the server name and query is performed correctly.

Finally, two sequential nested loops are used to read the response header and the resulting data into string streams to output to a locally stored file. The header is read using the 'read_until' function and the argument '\r\n\r\n' signalling the header is delimited by a blank line. The header is read by the first loop line-by-line and outputted to the local file. The socket is then read to completion one byte per read in the second loop to constitute a full character and stored to the local file.

4.4.8.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the data collection class.

Table 13: DataCollection Getter and Setter Fields

Variable Name	Access Type	Description
latDelta	Getter/Setter	Latitude delta used to query road network data
lonDelta	Getter/Setter	Longitude delta used to query road network data
nodeMap	Getter	Collection of all road control points
wayMap	Getter	Collection of all roads as presented in rawest form from OSM
boundsMap	Getter	Collection of all queried bounds

All map fields are accessible for the build-city class when updating road network data. GPS delta are settable to adjust query boundaries about the central query latitude and longitude.

4.4.1 Node

Below is a list of fields accessible through getters and setters for the node class.

Table 14: Node Getter Fields

Variable Name	Access Type	Description
lat	Getter	Latitude
lon	Getter	Longitude
ele	Getter	Elevation
id	Getter	Identifying number of node

The node class is simple with all fields initialized in the constructor and easily recreated upon de-serialization. No public class functions exist other than the getters.

4.4.2 Bound

Below is a list of fields accessible through getters and setters for the bound class.

Table 15: Bound Getters

Variable Name	Access Type	Description
maxLat	Getter	Maximum latitude of boundary
minLat	Getter	Minimum latitude of boundary
maxLon	Getter	Maximum longitude of boundary
minLon	Getter	Minimum longitude of boundary

The bound class is also simple with all fields initialized in the constructor and easily recreated upon de-serialization. No public class functions exist other than the getters.

4.4.3 Way

Below is a list of fields accessible through getters and setters for the way class.

Table 16: Way Getter Fields

Variable Name	Access Type	Description
nodeIDs	Getter	GenericMap of nodes
wayType	Getter	Type of road way represents (residential, arterial, etc.)
waySpeed	Getter	Speed limit of road way represents
ID	Getter	Road identifying number

The way class is also simple with all fields initialized in the constructor. No public class functions exist other than the getters.

4.4.4 DataManagement

The data management class is used to store all vital data to the city and route prediction classes. It is recreated upon restart of the controller used to execute the Driver Prediction algorithm. It is

also used to store all GPS measurements recorded throughout the duration of a trip. Data management must be able to execute to completion within one second. This is don upon startup and shutdown to recreate or store all vital classes and data structures.

This class also has no fields accessible through getters and setters. Instead, this class serializes data to disk / drive and de-serializes data to memory. Objects in memory or on disk / drive are serialized to and from JavaScript Object Notation (JSON) and binary. These objects include road networks, route prediction, speed prediction and trip data. Serialized road network data is represented as road curvature (in latitude / longitude) and elevation profiles, intersection latitudes and longitudes, and identifying labels for roads and intersections. Serialized route prediction data is represented as LinkToStateMaps, associating link-to-link transitions given the actual goal, and GoalToLinkMaps, associating the number of times a link is observed for the actual goal, links observed and goals observed. Serialized speed prediction data consists of all neural network weight, activation and output matrices. Lastly, serialized trip data consist of raw GPS measurements taken over the course of a trip. All data managed in this class is the minimum amount needed to recreate its respective class.

The public functions of the class are used for serialization. The ‘addCity’, ‘addRoutePrediction’, and ‘addTripData’ functions accept the respective input and serialize vital data to the inputs. The ‘getCity’, ‘getRoutePrediction’, and ‘getMostRecentTripData’ de-serializes data and returns their respective data types.

4.4.4.1 addRoutePrediction

The ‘addRoutePrediction’ function is used to store all vital data structures and classes belonging to the route prediction class. It is also used to store all associated link neural network matrix arrays used by the speed prediction class associated with the links stored in the argument. A combination of JSON and binary formatted output files store data locally to disk, and all learned driver data is serialized within this function as a result. The serialization process utilizes the property-tree data structure and populates the structure from the bottom up for the stored links, goals, link-to-state map, and goal-to-link map.

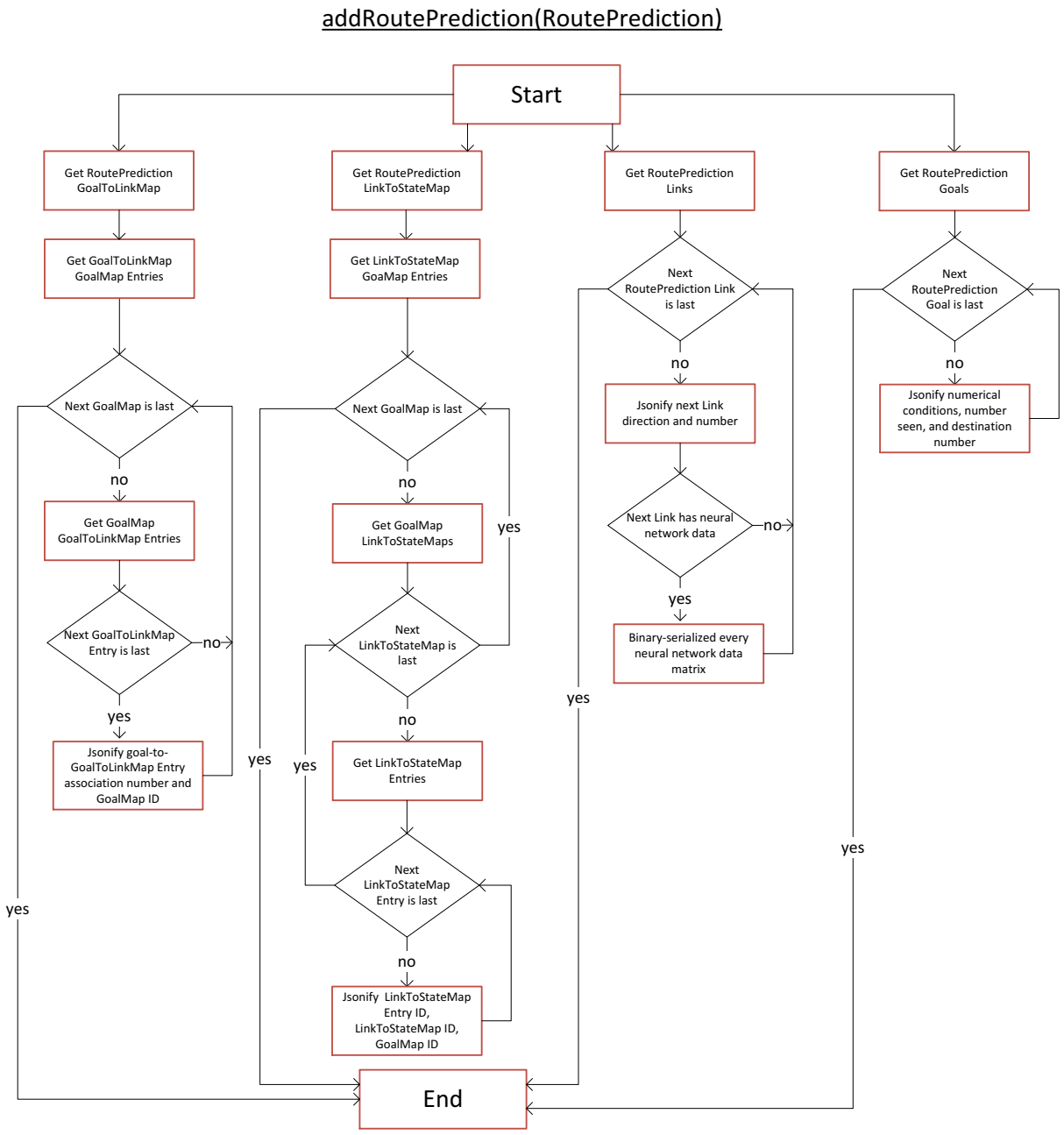

Figure 25: Activity Diagram for addRoutePrediction Function

The function begins by iterating across the collection of previously travelled links stored in the argument route prediction class. For each link, the direction and number is stored in a property-tree structured to be converted to a JSON-formatted output file containing all other vital class and data structure fields. This is used to recreate the argument route prediction class. If the link possesses neural network matrix arrays, then these data structures are serialized to a binary output file using a nested loop structure. The first loop iterates along the various types of neural network matrix arrays, input activation matrices, weight matrices, and output matrices. For each

type, a nested loop iterates along the indices of the matrix array, and each matrix is serialized to the binary output file using the 'writeBinaryNNMat' function which inputs the output file path stored locally on disk, the current matrix, the matrix type, the matrix index, and the hash of the link of the associated matrix.

The next step is to serialize the collection of previously seen goals of the route prediction class. Similarly, a loop is created over the collection, and for every goal, the identifying number and number of times the goal has been previously seen is inputted into the property-tree structure to be converted to the JSON-formatted output file. A nested loop is also created to input the vectorised starting conditions of the goal to the property-tree.

Next, the link-to-state map 3D hash table is serialized in a large nested loop structure. The first loop in the structure iterates across all goal map entries in the link to state map. For each goal map entry, the stored link-to-state map entries are iterated across in a nested loop, and for each link-to-state map entry, the collection of links is iterated across in a second iterated loop. For each link in the current link-to-state map entry, the property-tree to be converted to the JSON-formatted output file is updated with the goal map entry goal identification number, the link-to-state map identification number, the link identification number and the number of times the link has been entered into the current link-to-state-map entry.

The final process to serializing the argument route prediction class is serializing the goal-to-link map 2D hash table using another nested loop structure. The collection of goal map entries is iterated over, and for each goal map entry, the collection of links is iterated over in a nested loop. For each link, the property-tree to be converted to the JSON-formatted output file is updated with the goal map entry identification number, the link identification number, and the number of times the link has been entered into the goal map entry.

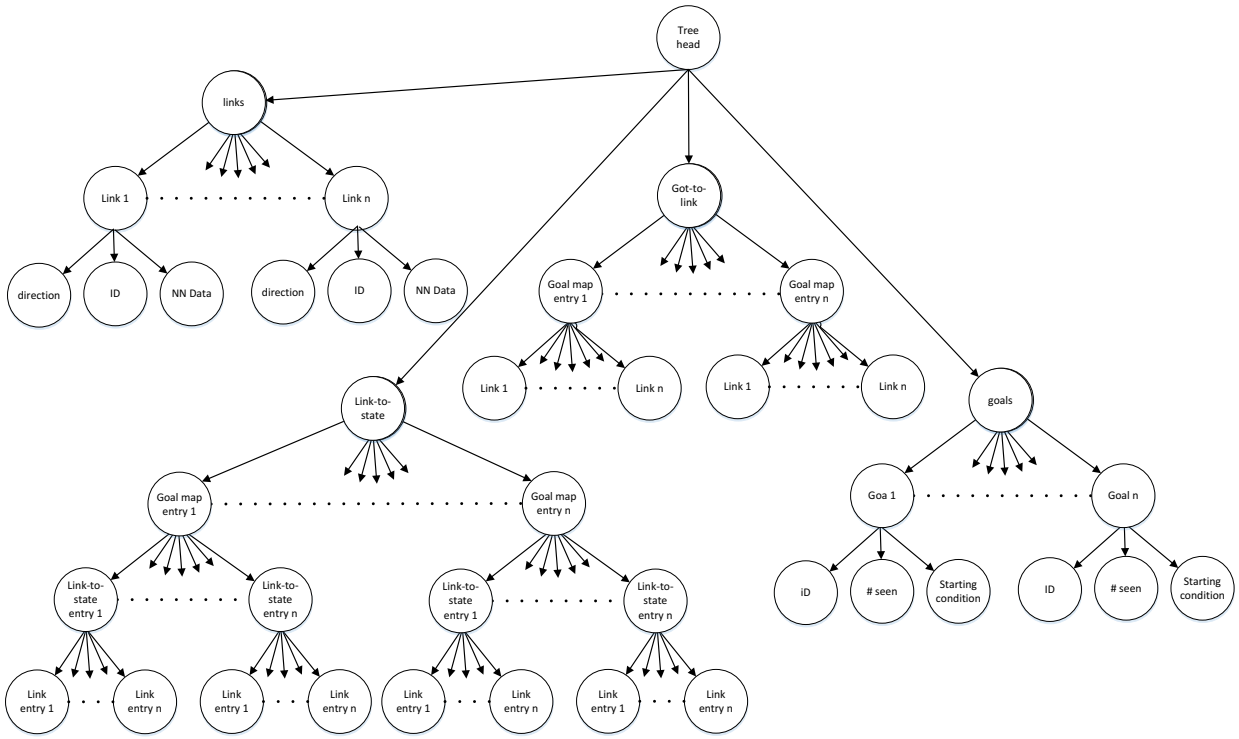

Figure 26: Property-Tree Structure of Serialized Route Prediction Data

At the end of the function, four sub-trees exist for the link and goals. Link-to-state map and goal-to-link map classes are stored by the route prediction class. These sub-trees are joined creating a single property-tree, and written to a JSON-formatted output file stored locally on disk.

4.4.4.2 addCityData

The ‘addCityData’ function is used to serialize the roads, intersections and bounds comprising the road network through a combination of JSON and binary formatted output files. Throughout the class, the property-tree data structure is used to compile data from the bottom-up for the road, intersection and bounds class collections.

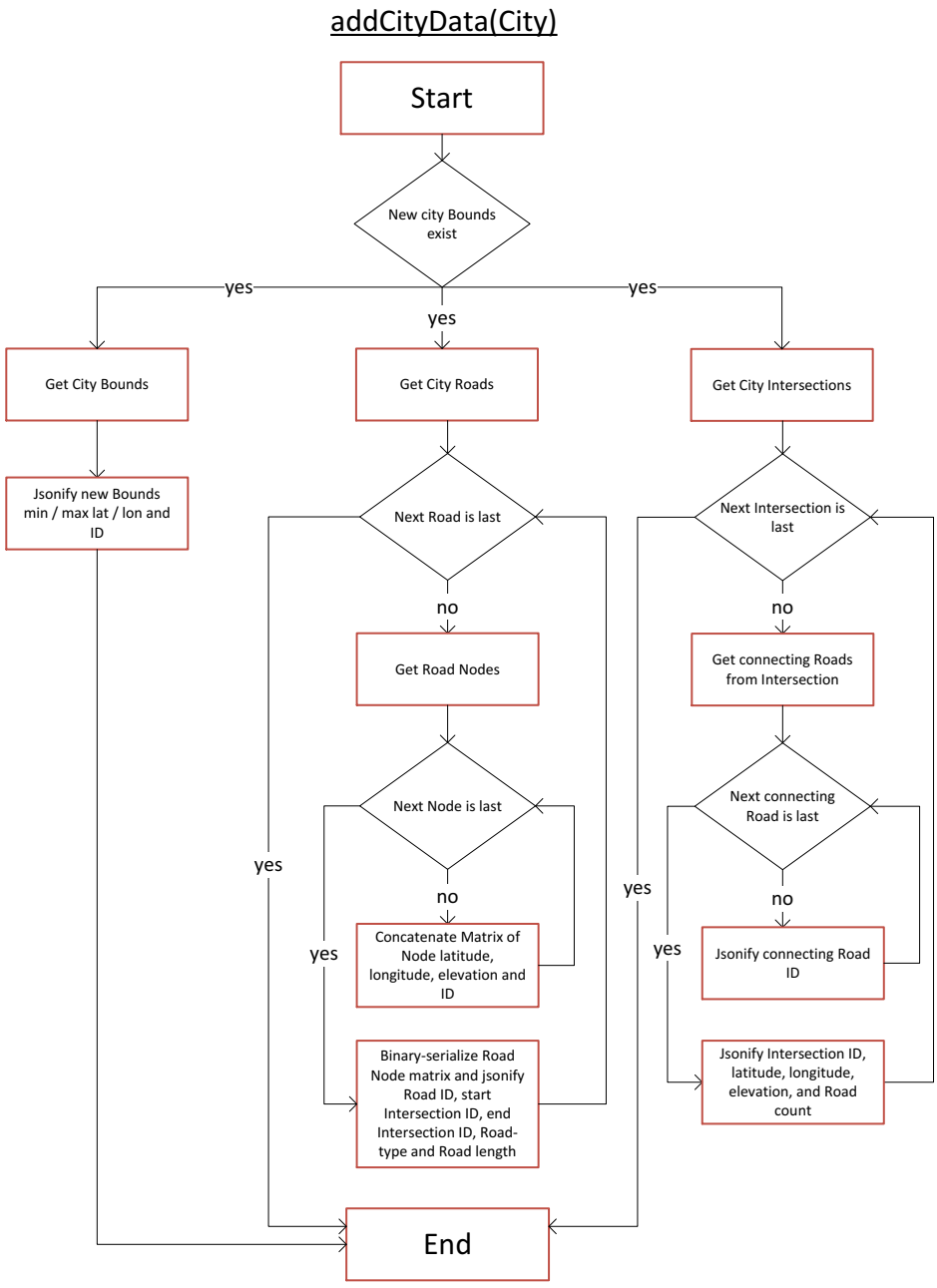

Figure 27: Activity Diagram of The addCityData Function

The process begins by iterating across the collection of atomic road sections stored by the argument city class. For each road, the property-tree to be converted to the JSON-formatted output file is updated with the road identification number, the start intersection identification number, the end intersection identification number, the road type, and the road spline length. The collection of nodes for each road, however, is iterated over to form a matrix containing the latitude, longitude, elevation, and identification number of each node in a nested loop. Once all road node data is represented in the matrix, the matrix is serialized to the binary-formatted output

file stored locally on disk using the ‘writeBinaryNodeMat’ function that inputs node matrix and road identification number as arguments.

The next serialization step is to iterate across the collection of intersections of the argument class. For each intersection, the property-tree to be converted to the JSON-formatted output file is updated with the identification number, the number of connecting roads, the elevation, the latitude, and the longitude. Additionally, all identification numbers of the connecting roads are added to the intersection JSON object by iterating across the collection in a nested loop.

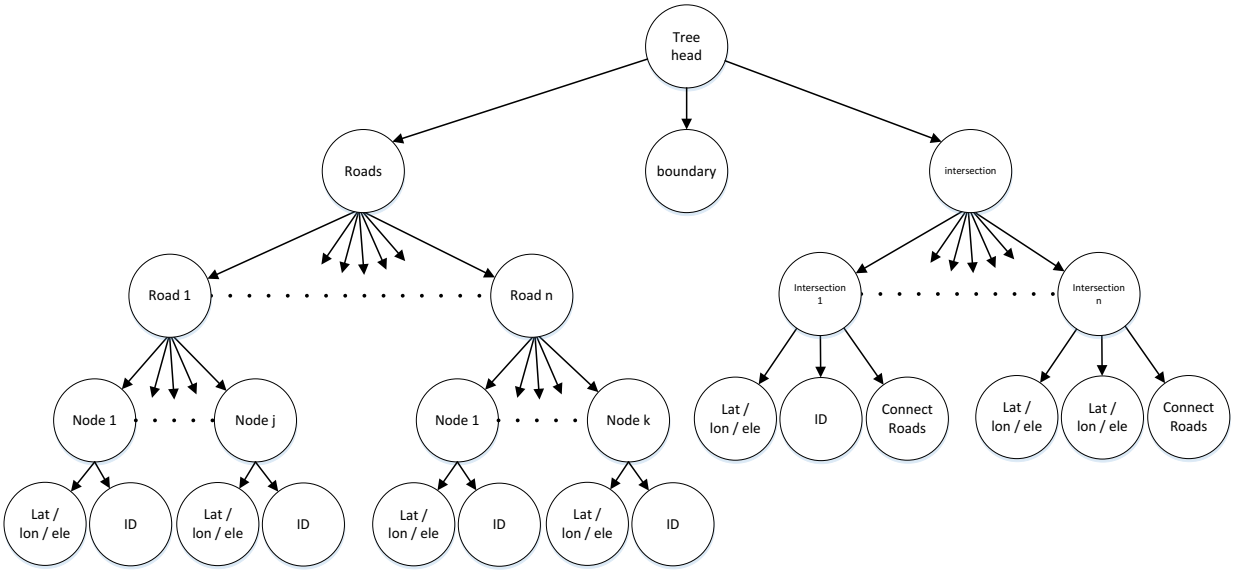

Figure 28: Property-Tree Structure of Serialized Road Network Data

Finally, the property-tree to be converted to the JSON-formatted output file is updated with the latitude and longitude boundaries of the stored road network. The resulting bounds, road, and intersection sub-trees are joined into a single property tree and written to the JSON-formatted output file locally stored on disk.

4.4.4.3 addTripData and getMostRecentTripData

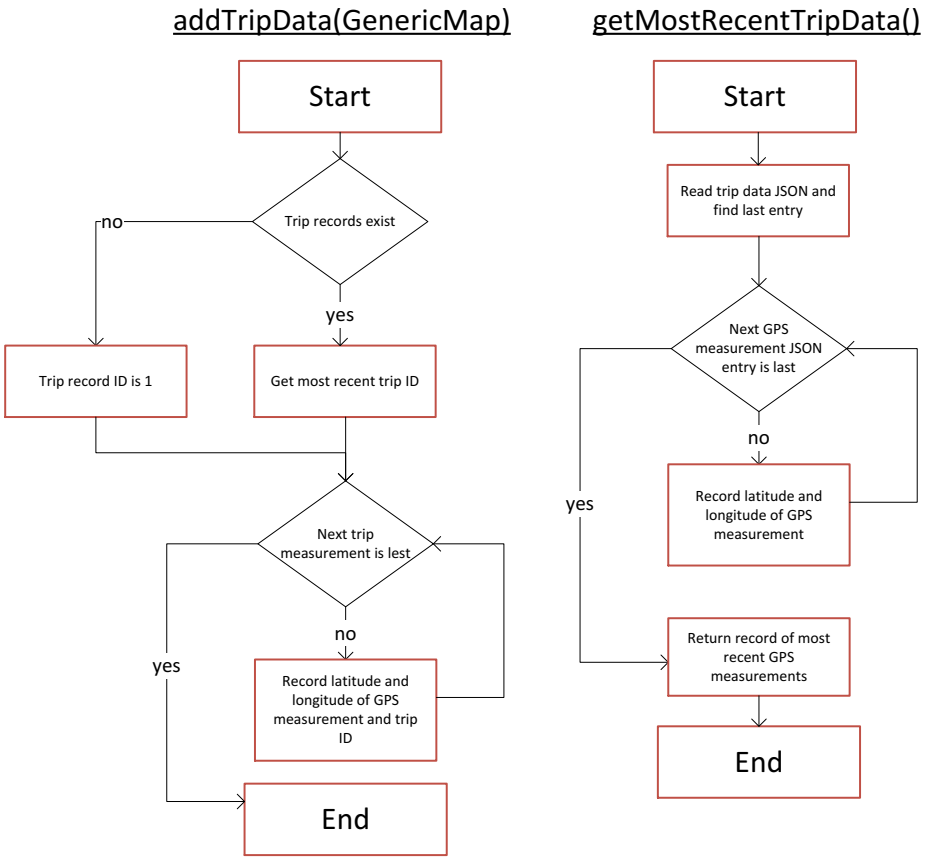

Figure 29: Activity Diagrams of the addTripData and getMostRecentTripData Functions

The ‘addTripData’ function is used to serialize the collection of GPS measurements taken periodically over the most recently completed trip. The property-tree to be converted to the JSON-formatted output file is updated by iterating across the argument collection of GPS measurements and recording the corresponding latitude and longitude. The trip number is also recorded and placed in the property-tree. Once the loop runs to completion, the property-tree is written to the JSON-formatted output file locally stored on disk.

The ‘getMostRecentTripData’ function de-serializes the last recorded trip data from a file saved to disk. The JSON structure is read and returned as a property tree using the Boost ‘read_json’ function. The structure of the property tree is provided below.

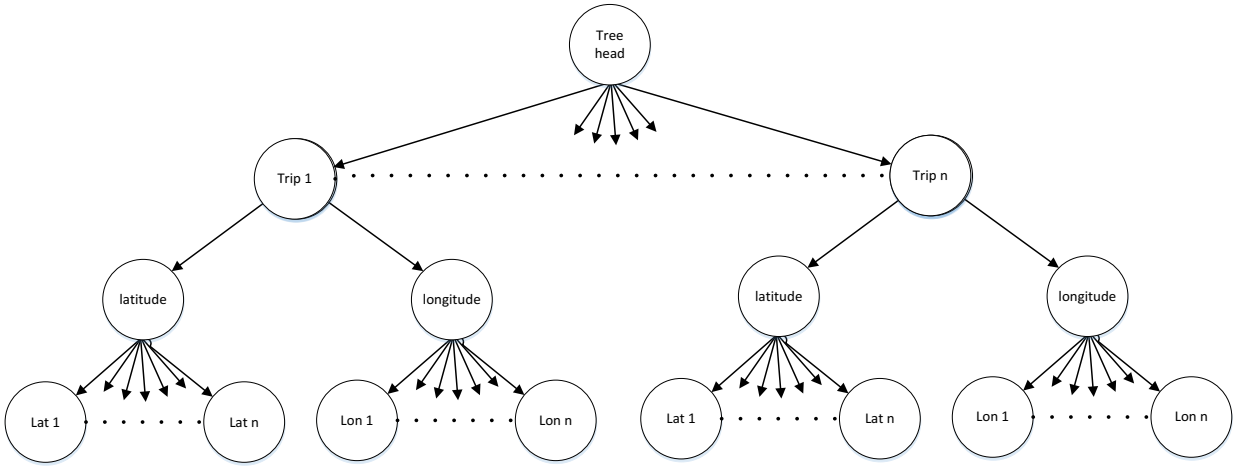

Figure 30: Trip-Log Property Tree Structure

A depth-first search using a nested loop structure aggregates two separate latitude and longitude collections with counter variables as the keys to each value in the collections. The counter allows the latitude and longitude values to be associated to one another and maintains order amongst the measurements. Once latitude and longitude measurements are associated, the paired-values are stored in another collection until the final and most recent trip is parsed.

4.4.4.4 getRoutePrediction

The ‘getRoutePrediction’ function reads serialized data from JSON and binary formatted files stored on disk to recreate a route prediction class previously saved. It is also used to populate the collection of links stored by the class with neural network matrix arrays used for speed prediction.

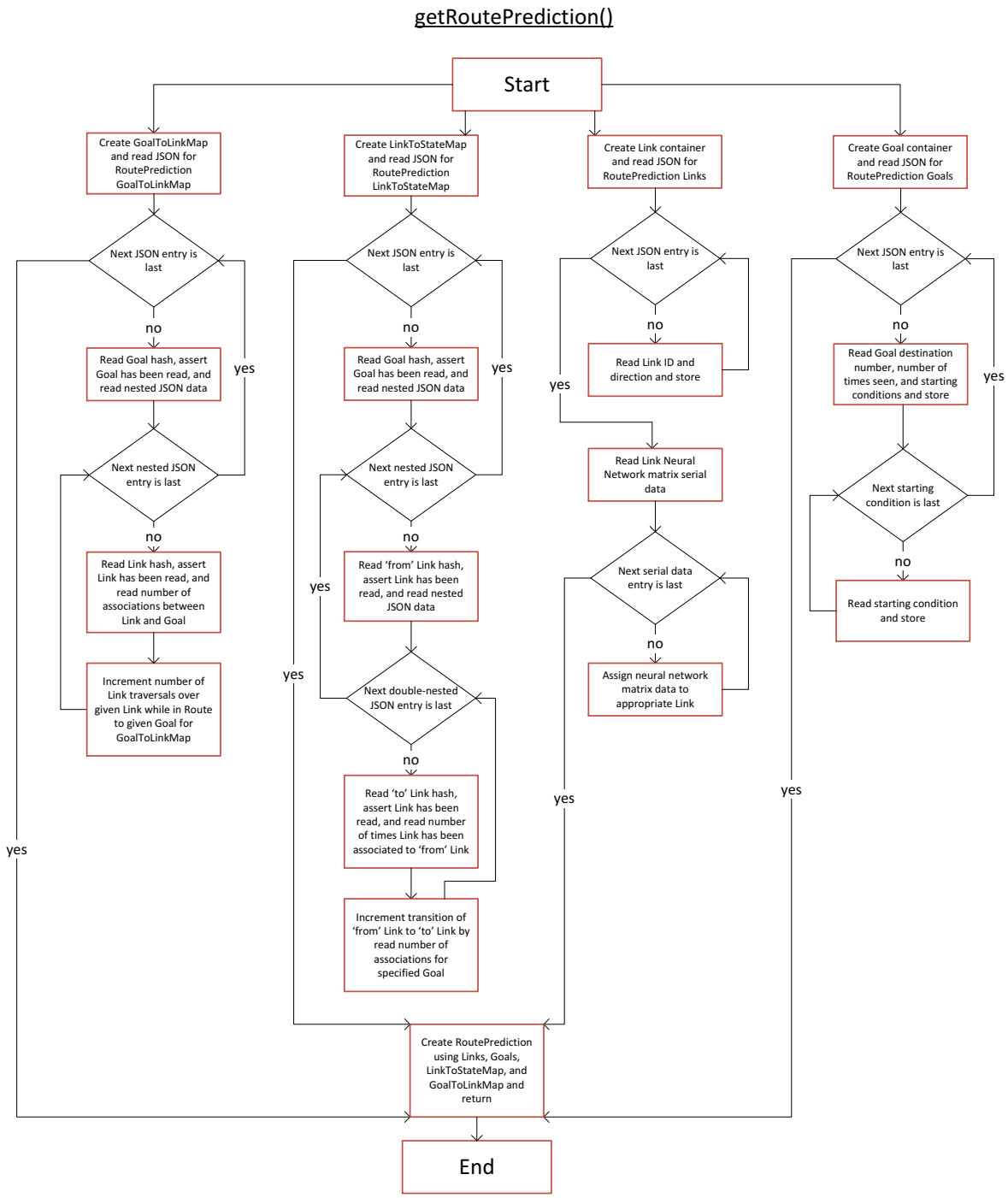

Figure 31: Activity Diagram for The getRoutePrediction Function

The first step to processing the serialized data, is to read the JSON-formatted filed stored to disk using the 'read_json' Boost function. The returned property-tree structure is defined in Figure 26, and similar to parsing data from the data collection class, a depth-first search via a nested loop structure is used to populate collections of links, link-to-state map entries, goal-to-link map entries and goals.

First, links are populated using a depth-first search over the entire link-subtree seen in Figure 26. For each JSON object representing a link, the link identifying number and direction are stored in locally scoped variables to construct a new link, and the new link is stored in an intermediary collection of links.

Once the JSON-formatted serialization file is completely parsed for link data, the binary-formatted serialization file containing neural network matrix arrays for speed prediction is read to associate stored data to the collection of links. Another intermediary collection of neural network matrix arrays is created to associate the arrays to link hashs. The binary-formatted serialization file is parsed in a loop with the next iteration contingent upon whether the next line read is the end-of-file. For each iteration of the loop, a new matrix is read using the 'readBinaryNNMat' function and a neural network matrix, the type of matrix (activation, weight, or output), the layer it belongs to, and the link hash it is associated with is returned. The returned values are used to matrix in the intermediary collection of matrices. Once the JSON and binary formatted files are parsed for link data, the intermediary collection of matrices is iterated over, and the stored matrices are associated with the intermediary collection of links. Links are updated with neural network values using the 'setWeights' function of the link class.

Next, goals are parsed in a depth-first search along the goal sub-tree defined in Figure 26 also using a nested loop structure. For each JSON object representing a goal, the identifying number, the number of times the goal has been seen, and the conditions are stored in locally scoped variables and used to construct a new goal class to be placed in a collection of goals.

Link-to-state map entries are parsed algorithmically, similar to links and goals along the link-to-state sub-tree defined in Figure 26. However, a link-to-state map is instantiated to store data from the link-to-state map sub-tree, as opposed to a generic collection. For each JSON object representing a link-to-state map entry, the starting link, transition link, and end goal identifying numbers are stored in locally scoped variables and used to retrieve the corresponding classes from the intermediary collection of goals and links already parsed. For each set of links and end-goal, the set is passed to the 'incrementTransistion' function of the link-to-state map in a loop set to iterate the number of times equal to the observed associations between transition links and end goal.

Finally, the goal-to-link map entries are parsed along the goal-to-link sub-tree defined in Figure 26. This is similar to the link-to-state map, for a goal-to-link map class is used to aggregate serialized data, as opposed to a generic collection. For each JSON object representing a goal map entry, the link and end goal hashes are stored in locally scoped variables and used to retrieve the corresponding classes from the intermediary collection of goals and links already parsed. For each link and goal pair, the pair is passed to the 'linkTraversed' function in a loop set to iterate the number of times equal to the observed associations between the link and end goal.

The intermediary collection of goals and links are added to a route prediction class. The link-to-state map and goal-to-link map are also passed to the route prediction class which is returned once the JSON and binary formatted serialization files are parsed completely.

4.4.4.5 getCityData

Similar to the 'getRoutePrediction' function, the 'getCityData' function relies on JSON and binary formatted serialization files. These are used to de-serialize city data to recreate the class using the stored road network data, comprised of roads, intersections, and bounds.

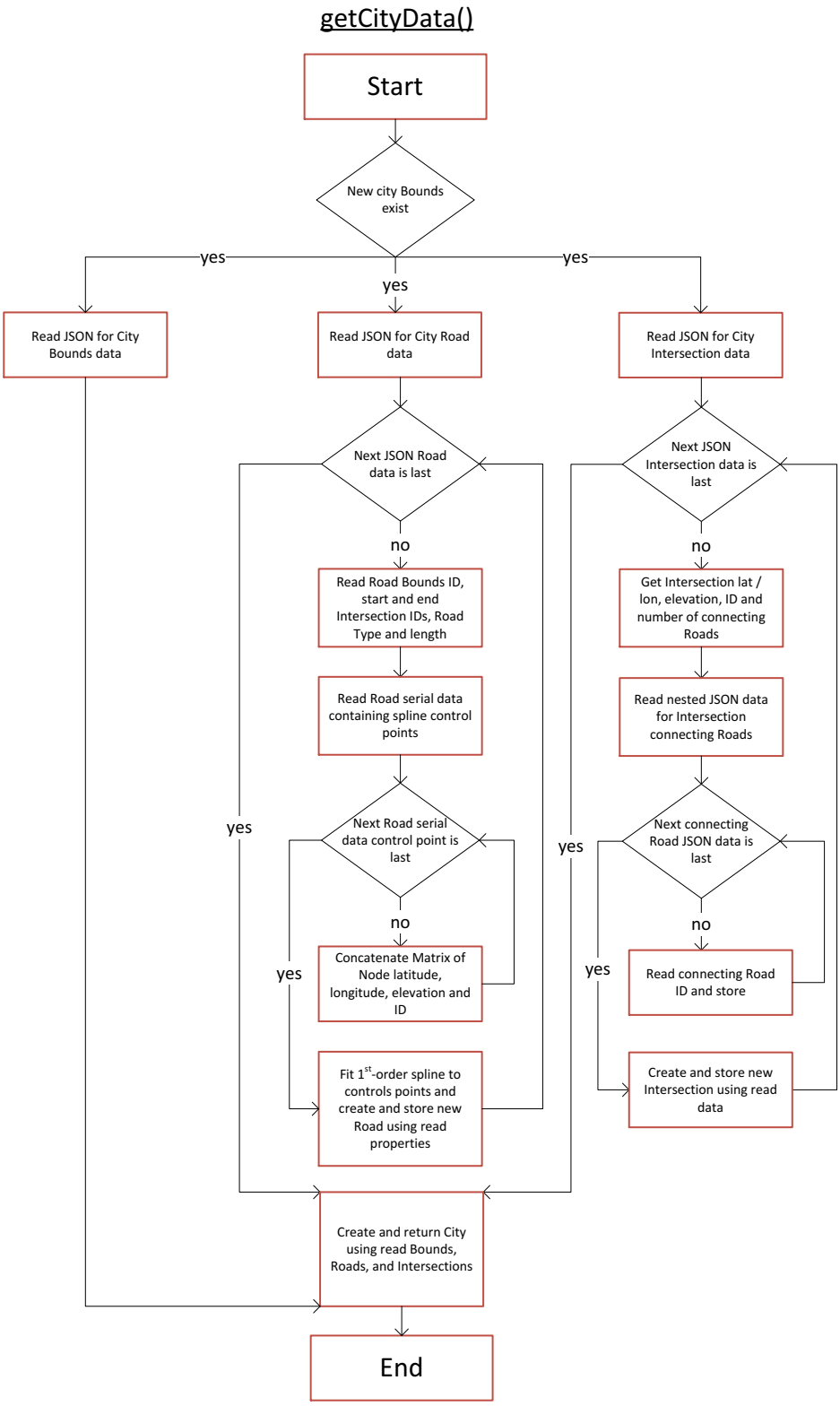

Figure 32: Activity Diagram for The City Class

Also similar to the parsing of all property-tree structures used by the Driver Prediction algorithm, the returned property-tree from the JSON-formatted serialization file containing city data is parsed with a depth-first search in a nested loop structure. The property-tree structure of the city data is defined in Figure 28.

First, the binary-formatted serialization file containing all road node data is read in a loop with the next iteration contingent upon whether the next line read is the end-of-file. For each iteration of the loop, a matrix representing the collection of nodes for a given road is read using the 'readBinaryNodeMat' function. For each row of the matrix, a node class is recreated by parsing the elevation, latitude, longitude, and identification number into locally scoped values to be passed to the node constructor, and the new node is added to a locally scoped collection. The collection of road nodes is then stored in a larger collection behind a key value equal to the road identification number to be associated with other road data stored in the JSON-formatted serialization file containing city data.

The remaining road data is parsed along the road sub-tree depicted in Figure 28 and stored in the JSON formatted serialization file. This file contains all city data collected using a depth-first search in a nested loop structure. For each JSON object representing a road, the starting intersection identification number, ending intersection identification number, road identification number, and road type are stored in locally scoped variables and passed to a road constructor. Lastly, the collection of already parsed road nodes is used to retrieve the set of nodes associated with the newly constructed road, and the road is added to a larger collection of roads.

Next, intersections are parsed along the intersection sub-tree depicted in Figure 28. This is done in a fashion algorithmically similar to the road class. For each JSON object representing an intersection, the elevation, latitude, longitude, identification number, and connecting road identification numbers are stored in locally scoped variables. From the collection of roads, the connecting road identification number are used to retrieve the corresponding classes, which are added to sub-set collection of roads of the connecting intersection. Finally, the intersection constructor is called, passing the parsed data to a collection of connecting roads, and the new intersection is passed to a collection of intersections.

Bounds representing the outer parameter of known road data are parsed last. These contain minimum and maximum latitude and longitude values. The city class constructor is called last and passes the collections of roads, intersections and bounds.

4.4.5 DriverPrediction

The Driver Prediction class relies on speed and route prediction classes. It is used to predict a route and time-series speed values over the predicted route from the current vehicle location to the predicted end destination. Logic to determine which machine learning algorithm to employ is a function of the current location. The training processes for each algorithm are abstracted by the class as well as logic to buffer actual speed data to perform speed prediction.

Three speed buffers are maintained by the Driver Prediction class. The first contains a number of historical speed values equal to the input size of the speed prediction class from the end of the previous link to train speed prediction over link transitions. This is performed to keep predicted speed values continuous across a predicted route. Another buffer contains all historical speed measurements across the current link stored by the Driver Prediction class to train speed prediction over the link and once it transitions to a new link. Finally, a queue of historical speed

values equal to the size of the speed prediction input size is maintained to provide speed prediction time-series input to predict future time-series speed values.

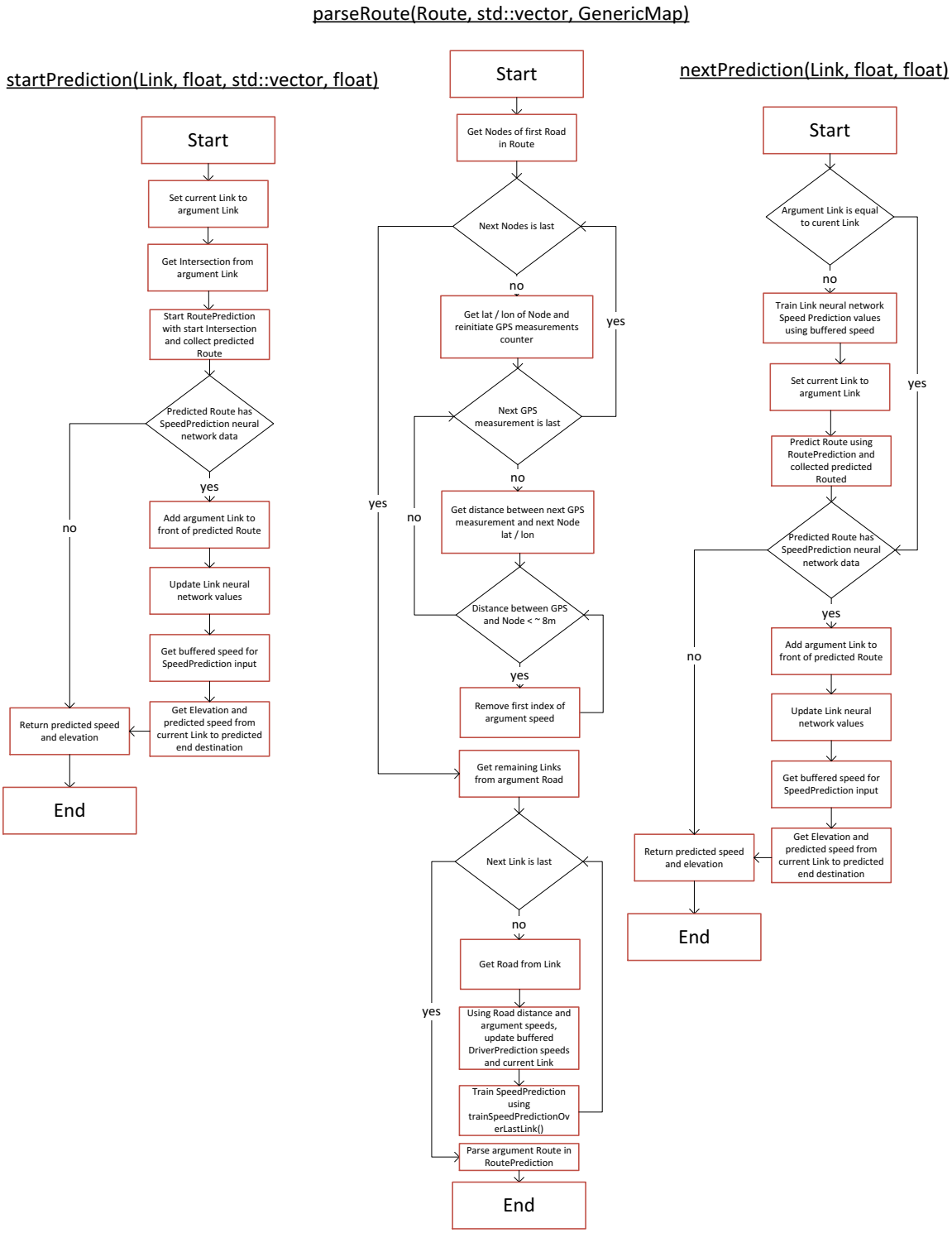

Figure 33: Activity Diagrams for The startPrediction, parseRoute, and nextPrediction Functions

4.4.5.1 startPrediction

The ‘startPrediction’ function initializes the route prediction class. It attempts to gather speed prediction data over the predicted route, assuming neural network matrix arrays exist for all links along the predicted route. The current link stored by the Driver Prediction class is updated by the argument link and is also used to determine the starting intersection of the predicted route. Both values are passed to the ‘startPrediction’ function of the route prediction class to retrieve the first predicted route after the start-up of the Driver Prediction algorithm. The predicted route stored by the Driver Prediction class is then updated to the predicted route and retrieved from the ‘startPrediction’ function. If the predicted route is not equivalent to the unknown, or over routes used by the route prediction when states are improperly updated, or the predicted route has come to an end, speed is predicted. To do so, historical speed is formed from the speed buffers stored by the Driver Prediction class using the ‘getSpeedPredInput’ function and the ‘routeToData’ function is employed over the predicted route from the argument distance along the argument link to return the predicted speed and elevation time-series data.

4.4.5.2 nextPrediction

The ‘nextPrediction’ function is used for every route and speed prediction after the ‘startPrediction’ function is called. If the argument link is not equal to the current link stored by the Driver Prediction class, either the predicted route is incorrect, or the vehicle has travelled the first atomic road segment of the predicted route requiring the route to be updated.

To update the predicted route, the previous link neural network matrix arrays are updated using the ‘trainSpeedPredictionOverLastLink’ function. This is done if the predicted route is not the unknown or over route used by the route prediction class. The updated predicted route is retrieved from ‘nextPrediction’ function of the route prediction class, and the current link stored by the Driver Prediction class is updated to the argument link. The Driver Prediction predicted route is also updated.

Regardless of whether the argument link matches the current link stored by the Driver Prediction class, speed prediction is always performed. Buffered actual speed data is retrieved from the ‘getSpeedPredictionInput’ function and the ‘route2Data’ function is employed along the predicted route from the argument distance along the argument link to return the predicted speed and elevation times-series data.

4.4.5.3 parseRoute

The ‘parseRoute’ function is use to update neural network matrix arrays for all links in the argument route using the ‘trainSpeedPredictionOverLastLink’ function and inputted speed time-series data over the previously travelled route. The function also updates the link-goal associations in the route prediction class.

The first step to training the abstracted machine learning algorithms stored by the Driver Prediction class is clearing all speed buffers. This is essential because the argument speed buffer is used as the time-series data to train speed prediction. Next, the distance of the argument speed buffer is computed by multiplying the length of the buffer by the speed prediction interval distance. This is done to store the travel distance of the vehicle and associate ordered subsets to be formed for each road segment. Link-by-link training is performed in a loop across all route links where the associated road to the current link is retrieved, and the road distance is used to create a subset from the argument speed buffer proportional to the length of the current road. The

values in the subset are inputted to the speed buffers stored by the Driver Prediction class using the 'updateSpeedsByVal' function called within a nested loop across all values in the subset. The current link is also stored by the Driver Prediction class and is updated to the current link within the loop across all route links. Training is done using the 'trainSpeedPredictionOverLastLink' function. Lastly, route prediction is trained using the corresponding 'parseRoute' function.

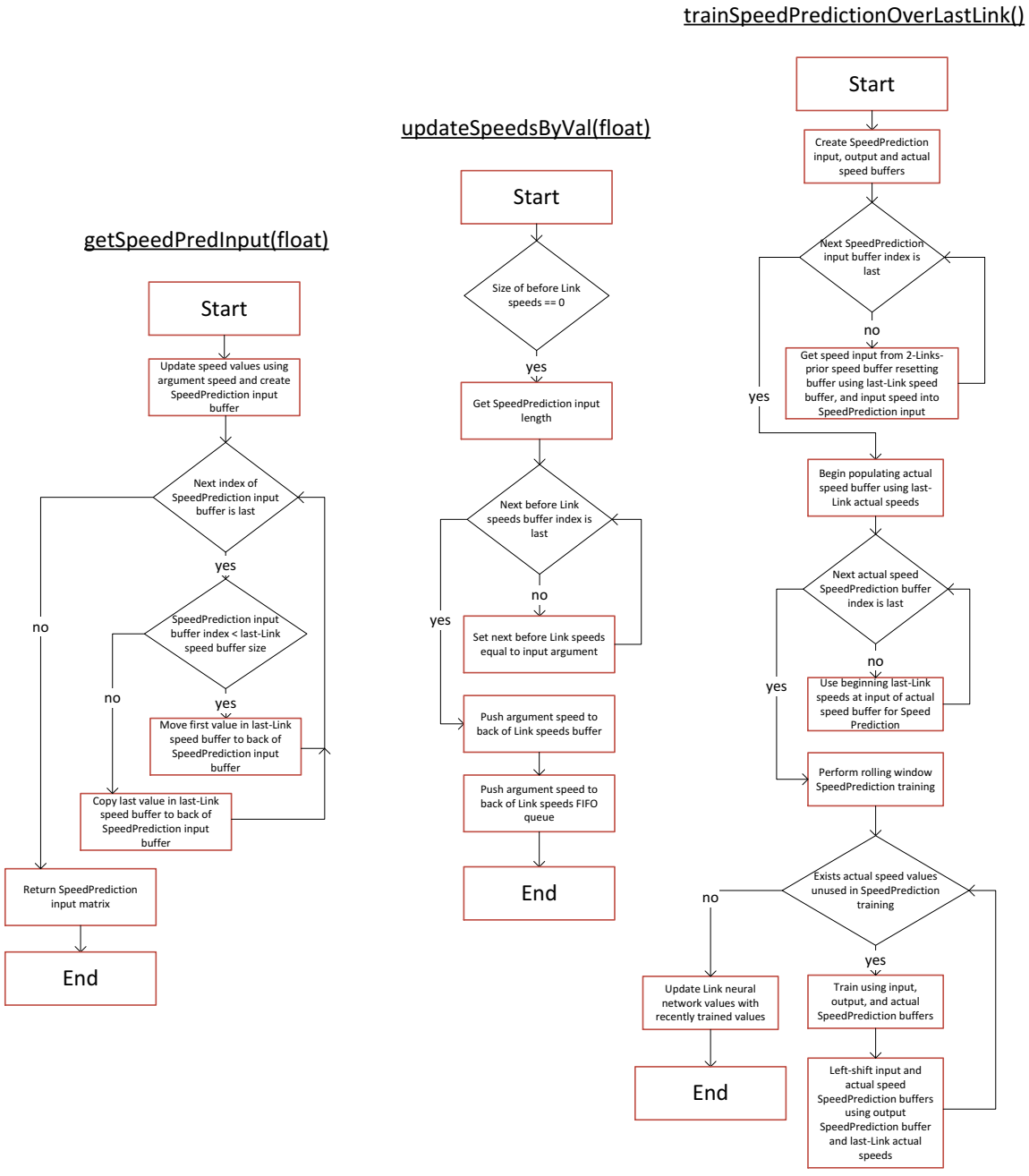

Figure 34: Activity Diagram for getSpeedPredInput, updateSpeedByVal and trainSpeedOverLastLink Functions

4.4.5.4 getSpeedPredInput

The ‘getSpeedPredInput’ function is used to convert queued historical speed data to row-matrix format to input into the speed prediction class, update buffered data stored by the Driver Prediction class with the argument speed value, and handle cases where the input speed size of the speed prediction class is greater than the amount of queued historical speed data stored by the Driver Prediction class. The function begins by passing the argument speed value to the ‘updateSpeedsByVal’ function to update the buffered speed data stored by the Driver Prediction class. A row-matrix then is created with size equal to the input size of the speed prediction class also stored by the Driver Prediction class. A loop across the row matrix is initialized. If the row matrix index is less than the size of the queued historical speed measurements with capacity equal to the input size of the speed prediction class, the first value of the queued historical speed values is placed at the index of the row-matrix, and the first values of the queued historical speed values is moved to the rear of the queue. If the row-matrix size is greater, however, the value from the rear of the queue is added to the current index of the row-matrix.

4.4.5.5 updateSpeedsByVal

The ‘updateSpeedsByVal’ function is used to update all three speed buffers stored by the speed prediction class. The buffer containing historical speed values over the previous link, if uninitialized, is set to store the same argument speed value at every index. This only occurs at the beginning of a trip. This buffer is updated in the ‘trainSpeedPredictionOverLastLink’ function using the last speed values of the current link. Next, the buffer containing all historical speed values over the current link is updated with the argument link concatenated to the end. Lastly, the queue of the most recent historical speed measurements equalling the size of the speed prediction input is updated with the argument speed value also concatenated to the end. If the size of the queue is equal to the size of the speed prediction input, the first value is erased.

4.4.5.6 trainSpeedPredictionOverLastLink

The last function, ‘trainSpeedPredictionOverLastLink’, is used to update the current link stored by the Driver Prediction class with neural network matrix arrays produced by training the speed prediction class over the buffered speed data also stored by the Driver Prediction class.

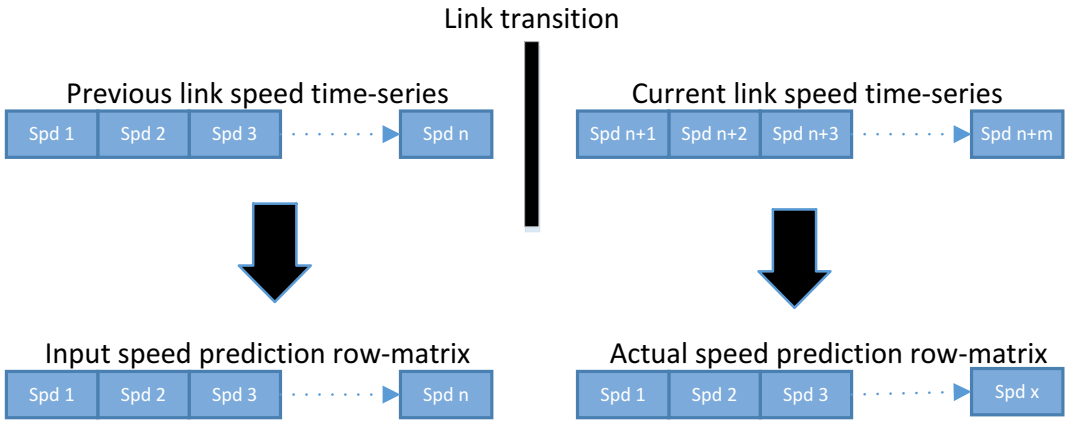

Figure 35: Initialization of Intermediary Speed Prediction Row Matrices

The process begins by creating intermediary row-matrix containers. These are to be passed and retrieved from the speed prediction class as the rolling window training process takes place, and

over the buffered historical speed measurements associated with the current link. A loop over the speed prediction input matrix populates the indices of the row-matrix with historical speeds from the end of the previous link. It then replaces the current index of the speed buffer with the first value from the first-in-first-out queue of historical speeds. The first value of the queue is then placed at the rear. The current link neural network matrix arrays are retrieved and inputted in the speed prediction class stored by the Driver Prediction class.

The next row-matrix is populated representing the desired or actual speed time-series values equal in length to the output of the speed prediction class. This is done to assess error of the predicted values. A loop is created over the row-matrix of actual speed time-series values, and if the buffer of historical speed values over the current link is still populated, the first value is placed into the current index of the row-matrix representing actual speed values. It is then erased from the buffered historical speed data. Otherwise, if the buffer of historical speed values is empty, a 0-value is placed in the current index of the row-matrix representing actual speed values.

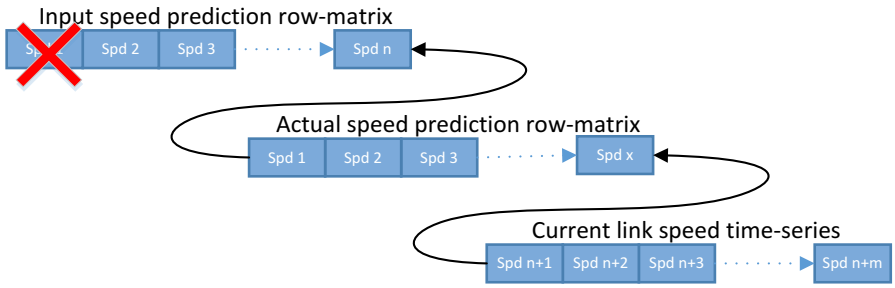

Figure 36: Rolling Window Update Step of Intermediary Speed Prediction Row Matrices

Once the actual and input speed matrices are populated, they are passed to the 'predict' and 'train' functions for the third output matrix to be populated and used to train the neural network matrix arrays. This process is repeated in a loop over the remaining current link speed values yet to be trained. For each iteration of the loop, values in the input and actual row matrices are left-shifted to input new values from the historical speed data of the current speed link. This done until the buffer is empty.

4.4.5.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the Driver Prediction class.

Table 17: DriverPrediction Setter Fields

Variable Name	Access Type	Description
beforeLinkSpds	Setter	Array of last recorded speeds from previous link used as historical speeds to train speed prediction from start of current link
linkSpds	Setter	Array of recorded speeds of current link used to train speed prediction over current link
lastSpds	Setter	First-in, first-out queue used to input speed values for speed prediction
routePrediction	Setter	Route prediction class

speedPrediction	Setter	Speed prediction class
predRoute	Setter	Predicted route generated by route prediction
currLink	Setter	Current link the vehicle is on
city	Setter	Used to get elevation and predicted speed data over predicted route

Vehicle speed arrays, the predicted route, the current link, and the city are used internally to the Driver Prediction class and thus only set. Route and speed prediction variables are set upon construction.

4.4.6 Link

The link class describes a direction along an atomic road segment within the road network. It also stores a set of neural network matrix arrays for speed prediction. The direction stored by the link is one where the trajectory along the associated road segment is from the start to end intersection. It is zero otherwise. The identifying number of the link is equal the identifying number of the associated road.

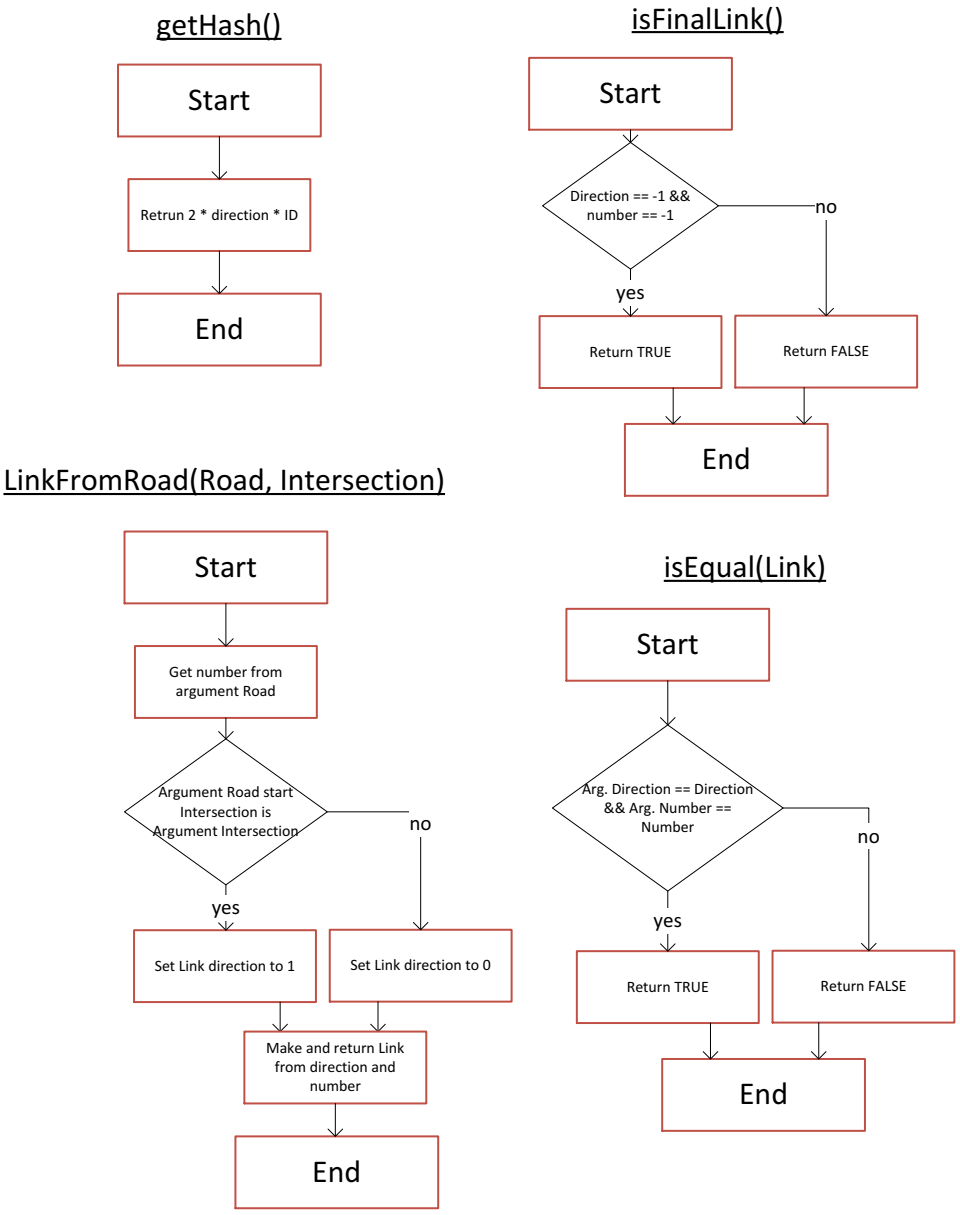

Figure 37: Activity Diagrams for The getHash, isFinalLink, linkFromRoad, and isEqual Functions

4.4.6.1 getHash

The hash of a link is defined as

$$linkHash = 2 * num_{dir} + num_{id} \tag{43}$$

where the *dir* and *id* subscripts relate to the link orientation with respect to the associated start and end intersections and identifying number. The 'getHash' function returns a hash of the link direction and identifying numbers to be stored, as key values, in the associative multidimensional hash maps of the route prediction class.

4.4.6.2 isFinalLink

The final link is used to identify the end of routes and is defined as a normal link with the direction and identifying numbers equal to -1. The value, -1, is used because direction or identifying numbers are always positive. The ‘isFinalLink’ function is used to compare the links values to -1 and return the Boolean equivalence.

4.4.6.3 linkFromRoad

Typically, links are associated back to roads by retrieving the corresponding road within the road network using the link identifying number. However, in the case where a link is to be made by a road, the ‘linkFromRoad’ function accepts a road and an intersection as arguments. The link direction is discerned by whether or not the argument intersection is the road start intersection, and the direction and road identifying numbers are passed to a link constructor be returned.

4.4.6.4 isEqual

The ‘isEqual’ function is used simply to compare the link in which the function is evoked from and an argument link. The Boolean equivalence of link direction and identifying numbers is returned.

4.4.6.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the link class.

Table 18: Link Getter and Setter Fields

Variable Name	Access Type	Description
number	Getter/Setter	Road segment number of the link represents
direction	Getter/Setter	Determines orientation of link with respect to starting intersection
hasWeights	Getter	True if neural network weights are associated with link
wts	Getter/Setter	Array of weight matrices for speed prediction
yInHid	Getter/Setter	Array of input vectors for speed prediction
yHid	Getter/Setter	Array of output vectors for speed prediction

All vital fields are accessible and settable to serialize link data and de-serialize / update link data when recreating the route prediction class on startup and updating link neural network weights through training speed prediction.

4.4.7 GPS

The GPS class offers a localization toolbox of functions to calculate the heading, the nearest road to the vehicle, the road orientation with respect to heading, and odometer data. It also stores a log of time-series GPS measurements along the most recently traveled road by communicating with a GPS receiver over RS232.

The system also ingests GPS information over RS-232 at 115200 baud from a uBlox EVK-M8N GPS receiver. There is also no parity, stop bit, or flow control, and data is received in 8-bit increments. Messages are received in National Maritime Electronics Association (NMEA)

Global Positioning Latitude / Longitude (GPLL) format. Updated GPS information is provided to the system approximately every 3.0 seconds.

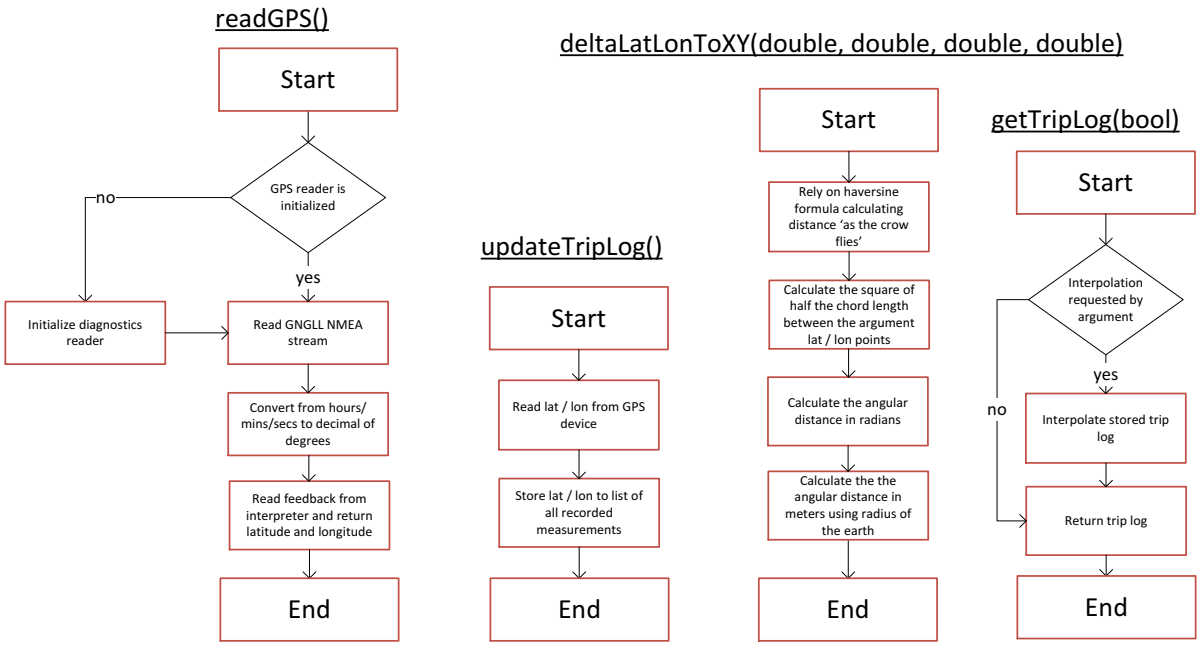

Figure 38: Activity Diagrams for The readGPS, updateTripLog, deltaLatLonToXY, and getTripLog Functions

4.4.7.1 readGPS

The 'readGPS' function communicates with a ublox NEA-M8L evaluation kit GPS receiver to retrieve the current latitude and longitude over RS232. These values are used for calculating heading, updating odometer data and updating the trip log stored by the GPS class.

Before data is read from the GPS receiver, the serial bus establishing connection must be initialized. A serial port connection is established using the C++11 standard library open/read/write drivers. A Termios structure is created, and memory is set aside to configure the opened serial port. Only read abilities are declared and the message structure is defined as having eight bits with one stop bit. No parity is used nor other forms of flow control. Finally, a one-second timeout duration is specified between Tx/Rx transmissions. Because the GPS receiver constantly sends GPS data, data is only read and not written unlike the vehicle diagnostics module.

Once the serial bus is initialized, data is read using an indefinite loop which only breaks once the serial Rx buffer is populated. The NMEA string inputted is GPGLL formatted defined below with latitude and longitude values in hours-minutes-seconds.

```
  $GPGLL,4916.45,N,12311.12,W,225444,A,*1D

Where:
     GLL          Geographic position, Latitude and Longitude
     4916.46,N    Latitude 49 deg. 16.45 min. North
     12311.12,W   Longitude 123 deg. 11.12 min. West
     225444       Fix taken at 22:54:44 UTC
     A            Data Active or V (void)
     *iD          checksum data
```

Figure 39: GPGLL NMEA Format

The NMEA sentence is read and parsed to retrieve the latitude and longitude values in hours-minutes-seconds using a string stream and the comma delimiter. It is then converted to degrees to accommodate the trigonometry used by the Haversine distance formula using modulo operations. Lastly, a pair containing the formatted latitude and longitude is returned.

4.4.7.2 updateTripLog

The ‘updateTripLog’ function is called approximately every five meters along the current route. It is used to keep a record of the path from the start to end destination. The process simply stores the returned latitude and longitude from the ‘readGPS’ function in a collection stored by the GPS class.

4.4.7.3 deltaLatLonToXY

This function is used to calculated the distance in meters between the argument points provided in latitude and longitude degrees. The Haversine formula is employed to define in the GPS Odometer section. The resulting arc distance in returned.

4.4.7.4 getTripLog

This function is used to retrieve the stored GPS measurement log stored by the GPS class. If specified by the argument bool, it then interpolates the log for better waypoint resolution used when training Driver Prediction. If interpolation is required, a loop over the log entries calculates the middle latitude and longitude point between the current and next log entries in each iteration. The new point is saved with the current log entry to a new collection and returned once the loop has run to completion.

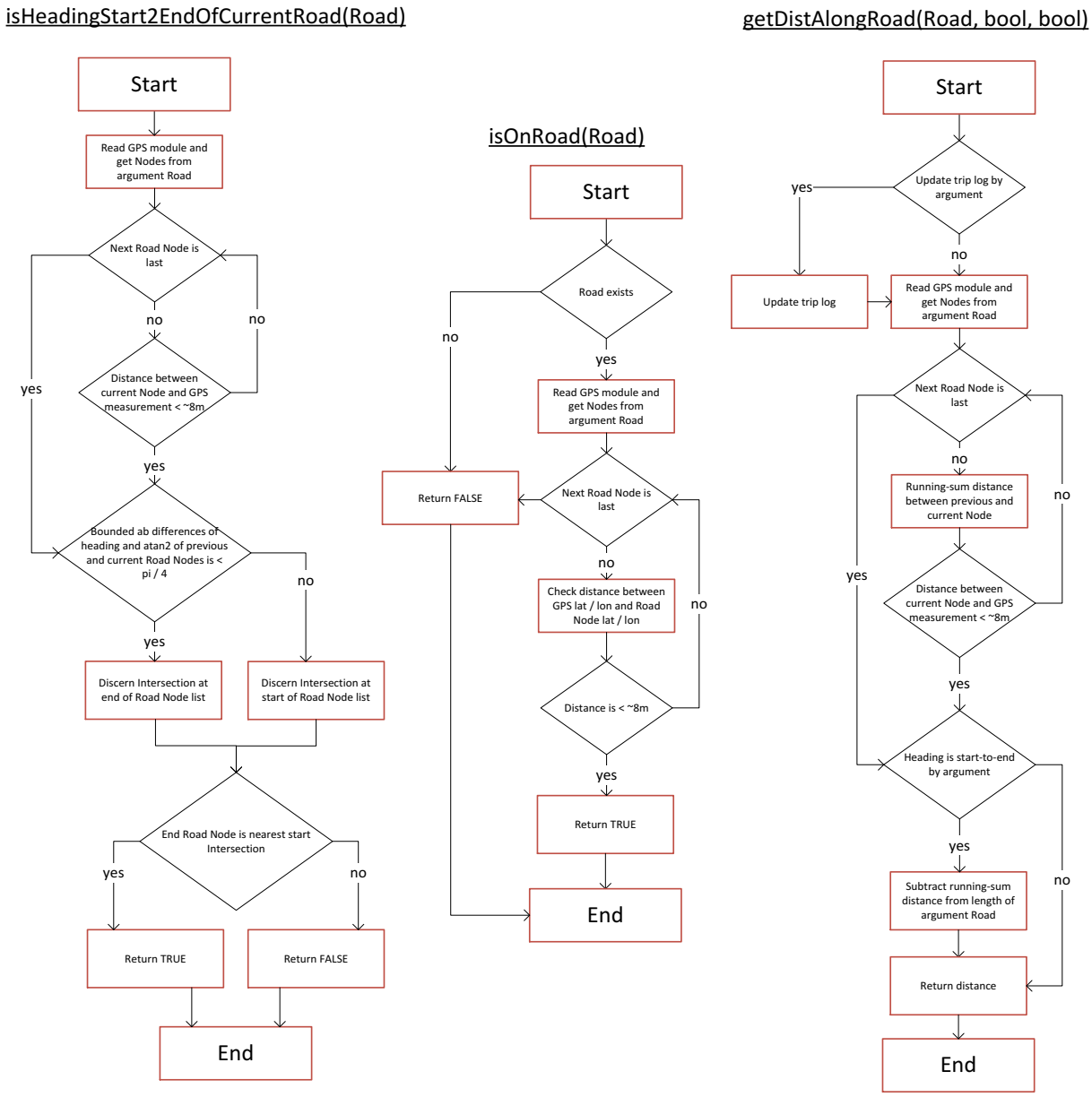

Figure 40: Activity Diagrams for the isHeadingStart2EndOfCurrentRoad, isOnRoad, and getDistanceAlongRoad Functions

4.4.7.5 isHeadingStart2EndOfCurrentRoad

This function determines the vehicle direction along the current road to associate the correct link and a set of neural network matrix arrays used for route and speed prediction, respectively. To do so, the nodes of the argument link are iterated across in a loop to determine the closest node to the current vehicle location using the' deltaLatLonToXY' and 'readGPS' functions. For iteration of the loop, locally scoped variables are updated from the current and previous latitudes and longitudes of the road nodes. Once the closest node is found to the vehicle location, the loop breaks, and the deltas between the current and previous latitudes and longitudes is calculated to

determine the heading of the of the road nearest the current vehicle location. The road heading is calculated using atan2. Next, the vehicle heading is retrieved using the 'getHeading' function.

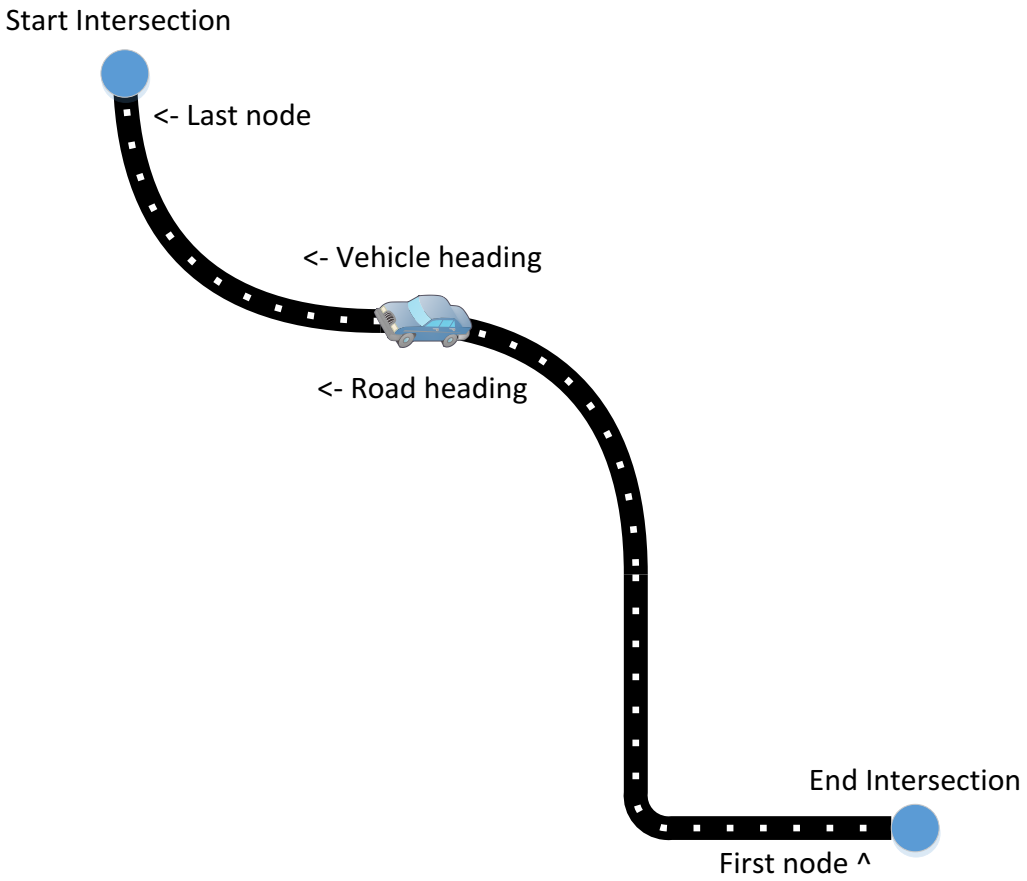

Figure 41: Road Segment Travel Orientation Example

The vehicle heading is compared to the road heading to determine which end of the road the vehicle is traveling towards. If the headings align, the end of the node collection is used to determine proximity to the start intersection. If they do not align, the start of the node collection is used to determine proximity to the start intersection. If the end node selected in which the vehicle is traveling towards is nearest the start intersection, a false Boolean is returned. Otherwise, a true Boolean is returned.

4.4.7.6 isOnRoad

This function is used to determine if there exists a node on the argument road that is within approximately seven meters of the current vehicle location. It is used to indicate transitions from one road segment to another. Only the argument road nodes are checked for proximity to current vehicle location instead of all roads in the road network. This is done to minimise run-time complexity to O(n) as opposed to $O(n^2)$.

The process is executed in a loop over all nodes of the argument road after the current vehicle location is retrieved using the 'readGPS' function. The 'deltaLatLon2XY' function is used to compute the distance between each node and the current vehicle locations. This is a threshold determination compared against the seven-meter minimum distance requirement. If a distance

less than seven meters is found, a true Boolean is returned. Otherwise, a false Boolean is returned.

4.4.7.7 getDistanceAlongRoad

The final toolbox function offered by the GPS class is the 'getDistanceAlongRoad' function used to determine distance along the road from one end to the current vehicle location with respect to the current direction of travel. Distance along the argument road segment is used as a point of reference to queue speed prediction.

The function begins by retrieving the current vehicle location from the 'readGPS' function. A loop is then used to iterate across all argument road nodes to calculate distance between the current and previous node and update a variable holding the cumulative distance between successive nodes. The loop is broken when the current node is within seven meters of the current vehicle location. Cumulative and proximity distances are calculated using the 'deltaLatLon2XY' function.

Lastly, a Boolean representing the direction of the travel of the vehicle along the argument road is retrieved by passing the argument road to the 'isHeadingStart2EndOfCurrentRoad'.

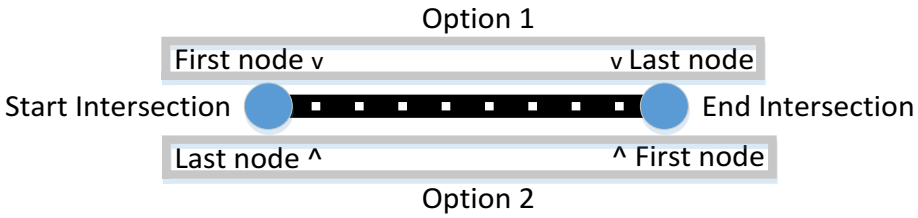

Figure 42: Possible Orientation of Road Nodes with Respect to Start and End Intersection

The orientation of the road node order with respect to the start and end intersections is also determined by calculating the distances between the start node and start intersection and the start node and the end intersection using the 'deltaLatLon2XY' function. Another Boolean is set to describe the orientation by asserting the distance between the start node and the start intersection is less than the distance between the start node and the end intersection. The XOR of the Booleans describing the vehicle orientation and node order orientation with respect to the start and end intersection is taken, and if true, the returned distance is the cumulative distance subtracted from the road distance. Otherwise, the cumulative distance is returned.

4.4.7.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the GPS class.

Table 19: GOS Getter and Setter Fields

Variable Name	Access Type	Description
tripLog	Getter/Setter	Log of all GPS measurements along the course of a trip
refLat	Getter/Setter	Reference latitude used to base distance calculations off of
refLon	Getter/Setter	Reference longitude used to base distance calculations off of

All class fields are accessible and settable, ‘tripLog’ being the most important for its GPS record of the current trip.

4.4.8 GenericMap

The generic map class is the lowest level data structure used within the code base to store ordered and unordered collections and provide a tool box of functions to access and manipulate the stored data. The class is employed to give total developer-control over various types of collections used and marries the functionality of the traditional map and array data structures. The class ultimately serves as a template wrapper to the map class provided by the C++11 standard library package.

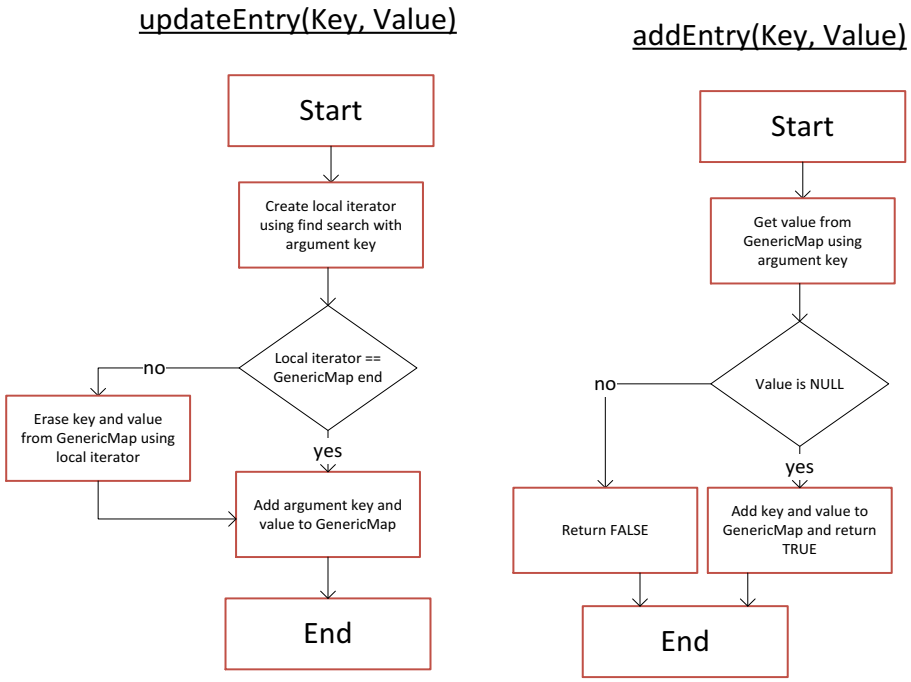

Figure 43: Activity Diagrams for The updateEntry and addEntry Functions

4.4.8.1 updateEntry

The ‘updateEntry’ function is used to modify an existing entry within the generic map and is most often evoked by the multidimensional hash maps within the route prediction class to update association numbers between links and states and links and goals. The argument key is used to look up the associated value in the map, and if it exists, the value is erased and replaced with the argument values at the argument key index. If a value does not exist at the argument key, however, the argument value is added in the map at the argument key index.

4.4.8.2 addEntry

The ‘addEntry’ function adds the argument value to the map at the argument key and is the most-direct and commonly used way of updating the map.

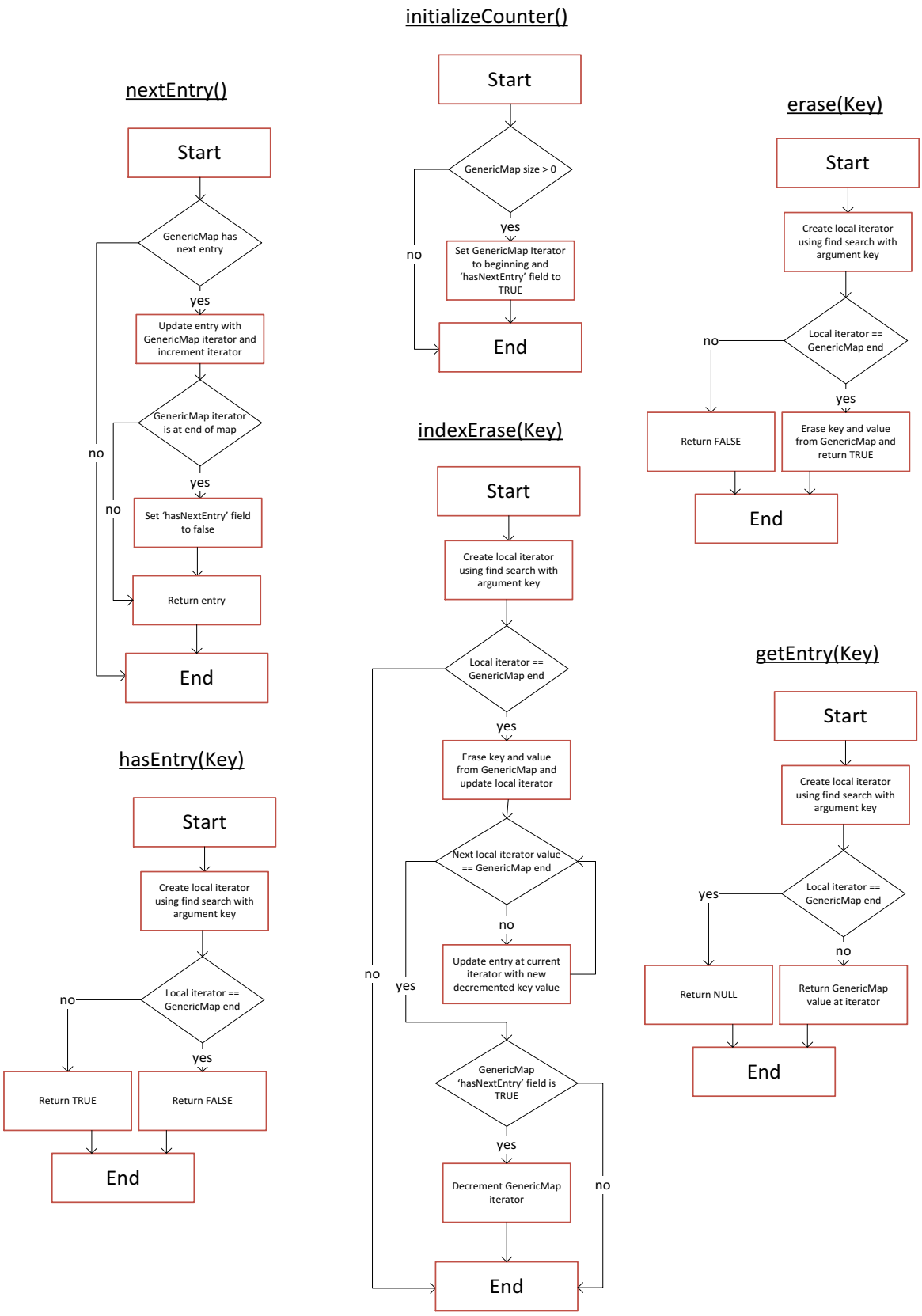

Figure 44: Activity Diagrams for The nextEntry, initializeCounter, erase, hasEntry, indexErase, and getEntry Functions

4.4.8.3 nextEntry

The ‘nextEntry’ function is used to iterate across the collection of stored data by returning the key-value pair stored in a generic entry class at the next position of the iterator stored by the generic map class. If a next value exists, it is returned, and the iterator is incremented. But if the iterator is equal the last value of the map, a semaphore stored by the generic map class is set signalling a next entry does not exist, and a void pointer is returned.

```
genericMap->initializeCounter();
GenericEntry<K, V>* nextEntry = genericMap->nextEntry();

while(nextEntry != NULL)
{
        K key = nextEntry->key;
        V value = nextEntry->value;
        .
        .
        .
        .
        .
        .
        .
        nextEntry = genericMap->nextEntry();
}
```

Figure 45: Example of Iterating Across Unordered Collection of Data

4.4.8.4 initializeCounter

This function is used to reset the iterator and the semaphore signalling the iterator of the Driver Prediction class is at the last value in the generic map so that the collection of values may be iterated over. This is done by setting the iterator equal to the first value stored in the map and setting the end-of-map semaphore to false.

4.4.8.5 erase

This function is used to remove a key-value pair at the argument key value. A locally scoped iterator is retrieved using the ‘find’ function of the C++11 map class, and if the iterator is equal to the map end, a false Boolean is returned. Otherwise, the pair is erased using the retrieved locally scoped iterator and the ‘erase’ function of the C++11 map class, and a true Boolean is returned.

4.4.8.6 hasEntry

The ‘hasEntry’ function is used to determine if a key-value pair exists at the argument key of the map. A locally scoped iterator is retrieved also using the ‘find’ function of the C++11 standard library. If the locally scoped iterator is equal to the key value of the last value stored in the generic map, a false Boolean is returned. Otherwise, a true Boolean is returned.

4.4.8.7 indexErase

The ‘indexErase’ function is used to erase a key-value pair from an ordered collection of data stored in the generic map class and update the remaining values such that key-continuity is

maintained. The process begins by retrieving a locally scoped iterator from the ‘find’ function, and similar to the ‘erase’ function, the value is erased if the iterator is not equal to the last value stored in the generic map. However, the iterator is then incremented in a loop until equal to the last value stored in the generic map. For each iteration, the key-value pair at the iterator location is stored in a locally scoped variable, the pair is erased using the ‘erase’ function, and the pair is then re-added using the ‘addEntry’ function with a decremented key value.

4.4.8.8 getEntry

The last tool box function of the generic map class retrieves the key-value pair at the argument key position, if it exists, in the generic map. The ‘find’ function of the C++11 standard library is used to retrieve a locally scoped iterator at the location of the argument key. If the locally scoped iterator is not equal to the value stored at the end of the generic map, it is returned as a generic entry. Otherwise, a null pointer is returned.

4.4.8.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the generic map class.

Table 20: GenericMap Setter Fields

Variable Name	Access Type	Description
map	Setter	Template where object entries are added through a setter
iter	Setter	Incremented through a setter
hasNextEntry	Getter	False if iter is at end of map. Otherwise true

The GenericMap template class is used as a universal data structure for ordered and unordered sets of information. As a result, the map and iterator are not accessible, but only settable as information is added to, erased from and manipulated through the data structure. The ‘hasNextEntry’ field is used when iterating through the map.

4.4.1 GoalMapEntry

Below is a list of fields accessible through getters and setters for the goal map entry class.

Table 21: GoalMap Getter

Variable Name	Access Type	Description
Map	Getter	Template map used to associate a link hash to either a LinkToStateMapEntry or a integer
Goal	Getter	Goal the goal map entry pertains to
M	Getter/Setter	Used to keep track of the number of associations between the key and value

The template map and goal is accessible through a getter for data serialization. Upon de-serialization, the map and goal are reconstructed using publicly accessible function within the class.

The ‘addMapEntry’ function accepts of key and value of any type to be stored in the local map. Conversely, the ‘getMapEntry’ function accepts a key and returns the value at the key.

4.4.2 Probability

The probability class is used by the link-to-state map class to calculate probabilities over the number of associations recorded by the three-dimensional map. Probabilities are calculated by totaling the number of associations between the transition-from link and a given goal over the total number of states for the same given goal. The only public function of this class is ‘getProbability’ which simply returns the numerator divided by the denominator.

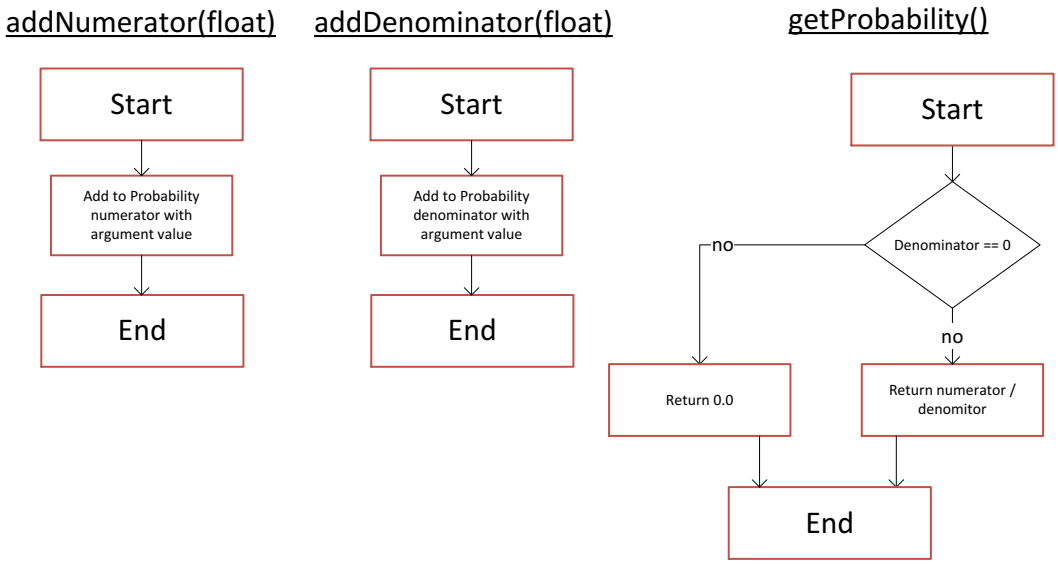

Figure 46: Activity Diagram for The addNumerator, addDenominator and getProbability Functions

4.4.2.1 addNumerator and addDenominator

These functions simply update the numerator and denominator fields by adding the respective argument values to them. Argument values are used to pass the number of associations stored in the link-to-state map.

4.4.2.2 getProbability

This function returns the numerator field divided by the denominator field stored by the class if the denominator is not equal to zero. Otherwise, zero is returned.

4.4.2.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the probability class.

Table 22: Probability Setters Fields

Variable Name	Access Type	Description
numerator	Setter	Added to through a setter
denominator	Setter	Added to through a setter

The probability function fields are settable to reflect updated ratios of link-to-link transitions or link observations over the number of goals observed.

4.4.3 Route

The route class is used to store an ordered collection of connected links that represent a trip taken by a driver. It also stores the end goal of the trip representing the final intersection or destination. The class is either updated in real-time by previously traveled links or generated as the succession of most probable link transitions in the route prediction class.

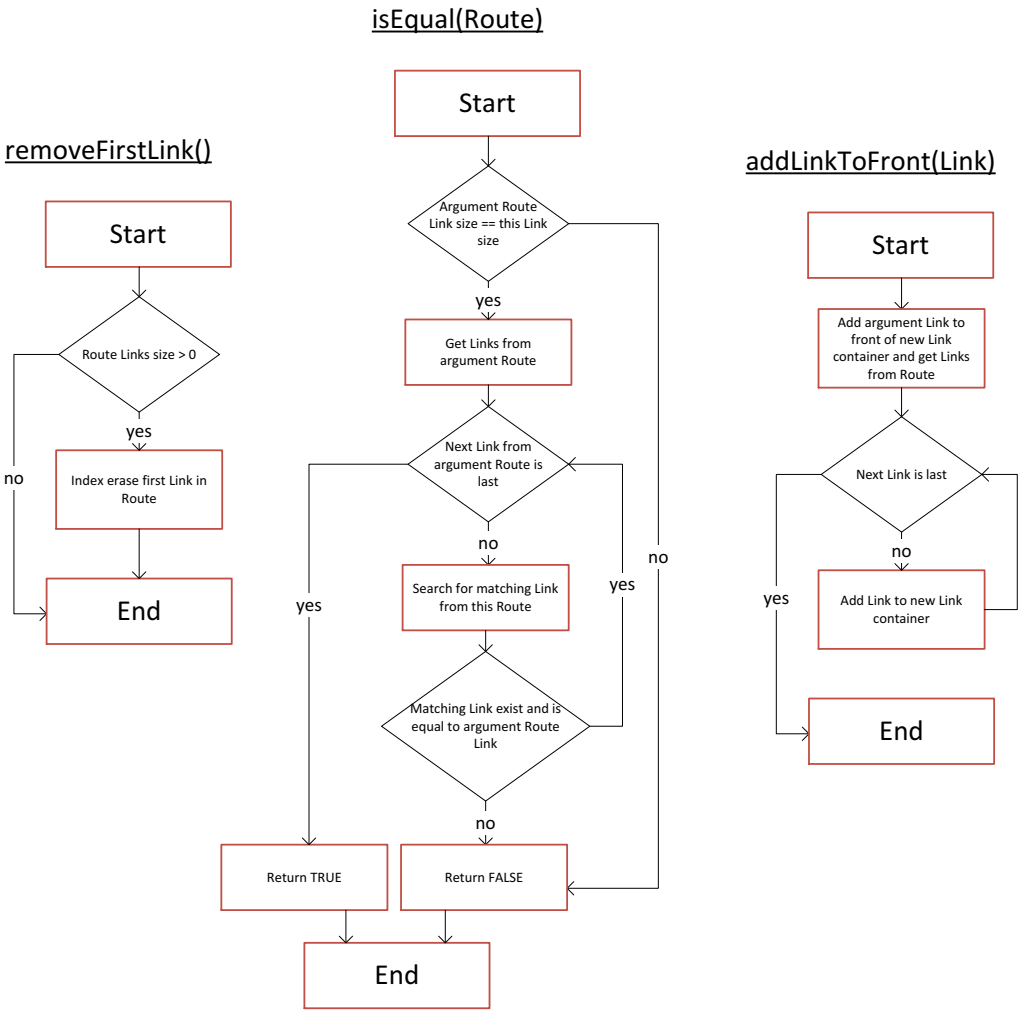

Figure 47: Activity Diagram of The Route Class

4.4.3.1 removeFirstLink

This function simply removes the first link in the ordered collection of links stored by the class using the ‘indexErase’ function of the generic map class. The function is called when the predicted route from the previous state remains correct for the updated state. As a result, only the first link of the predicted route must be removed after it has been travelled over.

4.4.3.2 isEqual

This function is used to compare two routes by comparing links stored and proceeds by initially comparing sizes of link collections. If the collections are equal in size, a loop is created over the argument route links and the links stored by the class. For each iteration, the i^{th} links of the two route are compared using the 'isEqual' function of the link class. If a pair of links is found un-equal or the link collections are unequal in size, a false Boolean is returned. Otherwise, a true Boolean is returned.

4.4.3.3 addLinkToFront

This function is used to add the argument link to the front of the ordered collection of links stored by the class. It is used by the Driver Prediction class to add the current link to the front of the predicted route, for the predicted route starts at the next intersection. The current link is needed to perform speed prediction from the current vehicle location over the predicted route. The process is done by initializing a loop over the stored links, and in every iteration, the i^{th} link is stored in a locally scoped collection of links using an incremented key value. The argument link is added to the front of the locally scoped collection, and the current collection of links stored by the route class is replaced by the locally scoped collection.

4.4.3.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the route class.

Table 23: Route Getter and Setter Fields

Variable Name	Access Type	Description
Links	Getter/Setter	Ordered collection of links representing the road segments traversed by the route
Goal	Getter	The end-destination goal of the route
Intersection	Getter	The end-destination intersection of the route

All fields are accessible for quick look-up. Links are settable for the first link of the route is typically removed as the route is traversed. Routes are not serialized.

4.4.4 LinkToStateMap

The link-to-state map class is used to associate links to a state comprised of an adjacent link associated to an end goal in a three-dimensional hash map. Associations are stored for every route travelled and across all links in the route. The structure of the link-to-state map is as follows:

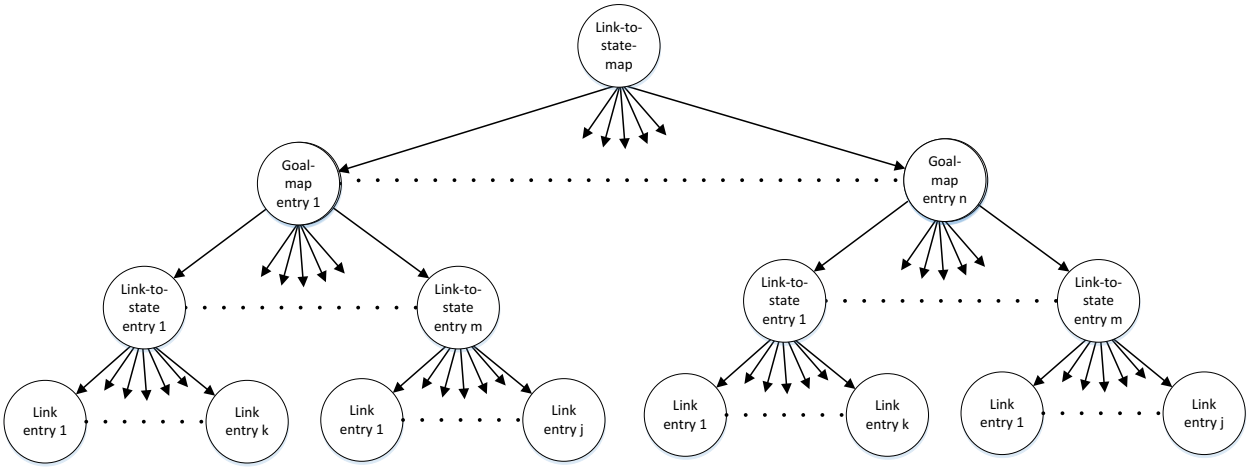

Figure 48: Link-to-State Map Structure

Every goal map entry contains a goal hash and a collection of link-to-state map entries. Each link-to-state map entry contains an identifying link hash and a collection of key-values pairs of other link hashes and association numbers. The goal map entry goal hash and the link entry link hash represent a state. The link-to-state map entry link hash represents the transition-from link. It is also considered as the link to the previous state.

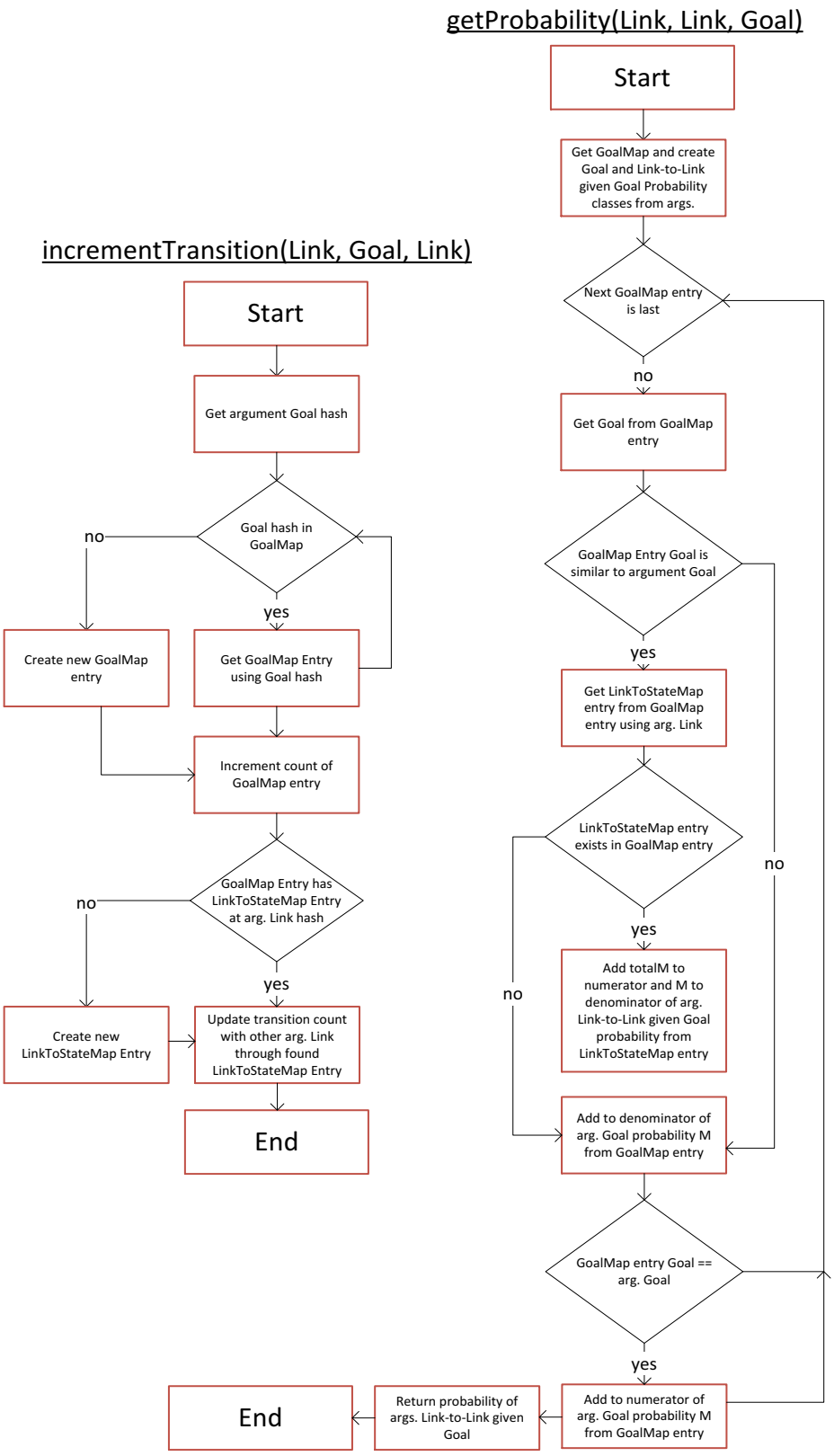

Figure 49: Activity Diagram for The incrementTransistion and getProbability Functions

4.4.4.1 incrementTransition

This function is used in the training step of route prediction to update the three-dimensional associative hash map along the links of a route. It accepts a link from the previous state and a state comprised of an adjacent link and an end goal. The purpose is to increment the number of associations and record between the link of the previous state and the next state. This is done so that the probabilities of the states along a predicted route can be calculated.

This process is completed by retrieving the hash from the argument goal and using the value to retrieve the corresponding goal map entry from the link-to-state map. If the goal map entry does not exist, a new one is made using the argument goal hash. The hash of state link is then retrieved and used to find the corresponding link-to-state map entry from the goal map entry. If the link to state map entry does not exist, it is created. Finally, the 'addEntry' function of the link-to-state map entry is passed to the link of the previous state to increment the association between the previous state link and the next state. The association number is then returned.

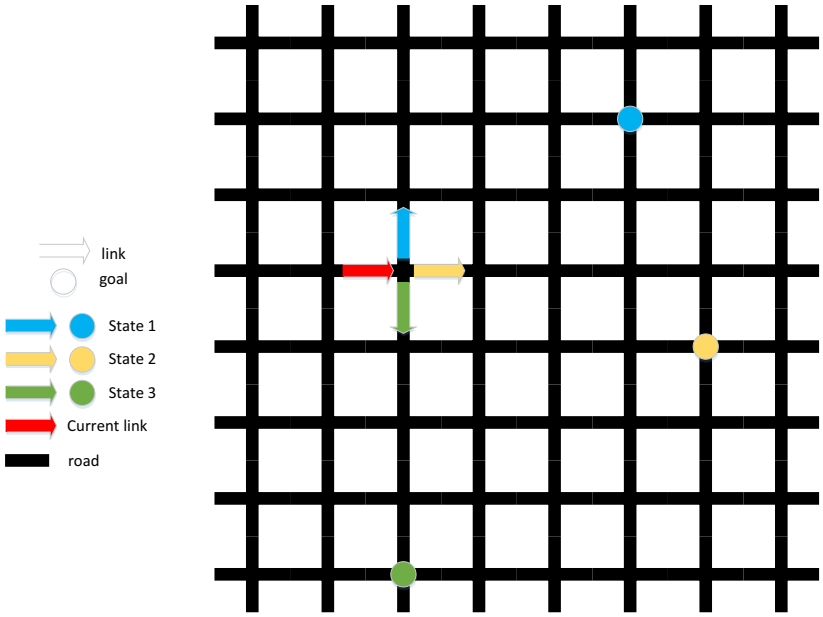

Figure 50: Link-to-State Transition Example in Road Grid

The above diagram depicts the three states provided for the current link. The link from the previous state over a road grid exemplifies how the links and states relate to a realistic driving scenario.

4.4.4.2 getProbability

This function is used to determine the probability of a given state as a function of the previous state link. Since the n^{th} observation is dependent on the $(n^{th} - 1)$ observation, it only partially defines how the Hidden Markov Model is implemented in route prediction. The remaining definition is described in the RoutePrediction section with respect to calculating the first-order Markov chain as a joint probability of multiple observations.

The function operates by iterating across all goal map entries stored by the link-to-state map class. For each goal map entry, if the argument goal is equal to or similar to the goal stored by

the goal map entry (determined by the ‘isEqual’ and ‘isSimilar’ functions), the link-to-state map entry (with the matching hash of the argument link representing the previous state link) is retrieved. Goals with similar starting conditions (indicated by the ‘isSimilar’ function) are used in calculating the link-to-state probability to add an additional predictor other than link-to-state associations. This has proven to work well with respect to prediction accuracy. The probability is updated by adding the total number of associations between the next state link and goal to the numerator and the total number of associations for the next state goal to the denominator. This is done by calling the ‘getM’ and ‘getTotalM’ functions of the link-to-state map class, respectively. Once the loop runs to completion, the probability is returned.

4.4.4.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the link-to-state map class.

Table 24: LinkToStateMap Getter Field

Variable Name	Access Type	Description
goalMap	Getter	3D hash map containing goal map entries that contain link to state map entries

The 3D hash map is accessible through a getter for data serialization. Upon de-serialization, the hash map is reconstructed using publicly accessible functions within the class.

4.4.5 LinkToStateMapEntry

The link-to-state map entry class is a one-dimensional hash map of link hashes and association numbers as key-value pairs. The class is utilized by the link-to-state map class to store link hashes and the number of times the link has been associated with a goal to calculate state probabilities.

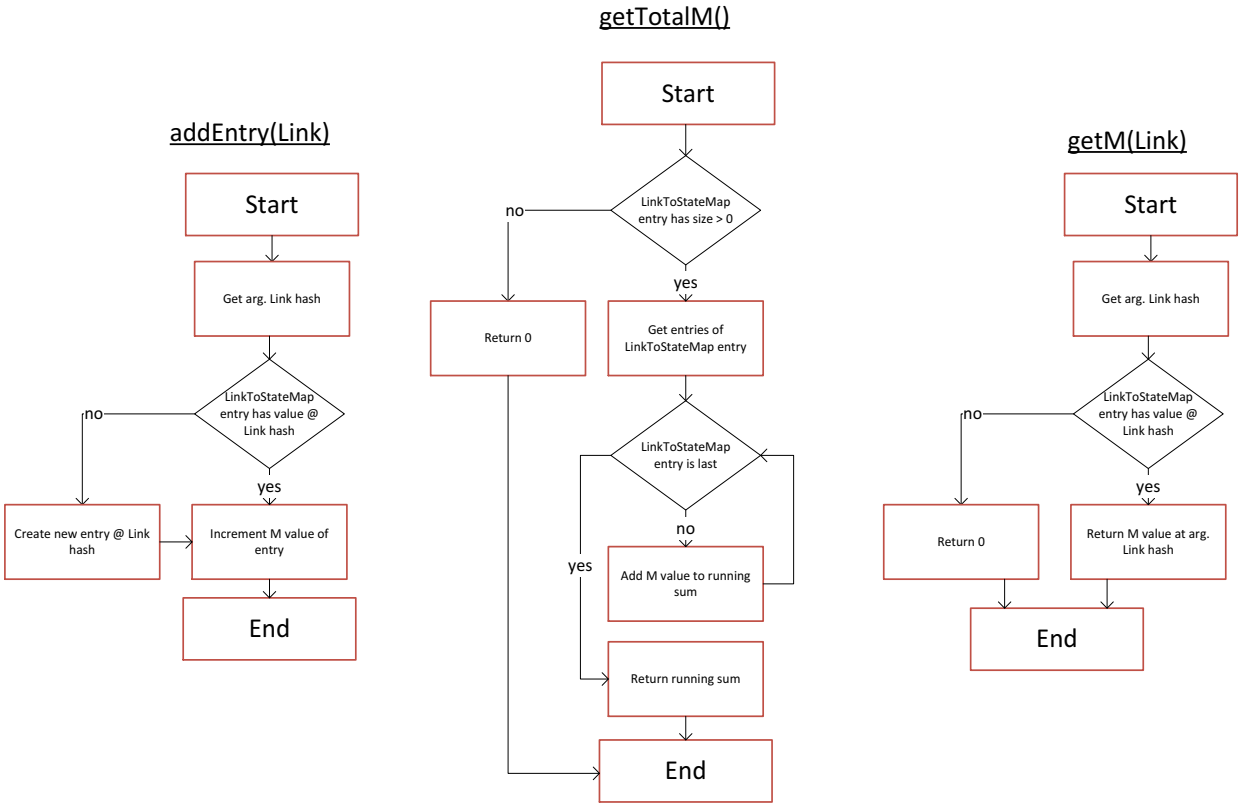

Figure 51: Activity Diagrams of the addEntry, getTotalM, and getM Functions

4.4.5.1 addEntry

The ‘addEntry’ function is used to add a link hash and association number as a key-value pair to the collection of pairs stored by class. The argument link hash is used to retrieve the corresponding entry from the collection. If the entry exists, it is updated using the ‘updateEntry’ function of the generic map class with an incremented value. If the entry does not exist, a new one is created using the link hash and an association value of one. The association number is returned at the end of the function.

4.4.5.2 getTotalM

This function is used to sum the association numbers and return the total. To do so, a loop is created over all entries in the collection stored by the class, and a cumulative sum is updated in each iteration. The sum is returned once the loop runs to completion.

4.4.5.3 getM

The last function is used to return the number of associations corresponding to the argument link. If there exists a key-value pair stored at the argument link hash, the value is returned. Otherwise, zero is returned.

4.4.5.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the link-to-state map entry class.

Table 25: LinkToStateMapEntry Getter Field

Variable Name	Access Type	Description
Entries	Getter	Hash map containing link hashes and the number of times they have been observed

The hash map is accessible through a getter for data serialization. Upon de-serialization, the hash map is reconstructed using publicly accessible functions within the class.

4.4.6 GoalToLinkMap

The goal-to-link map is a two-dimensional hash map used to store the associations between goals and links. The structure of the map is defined below.

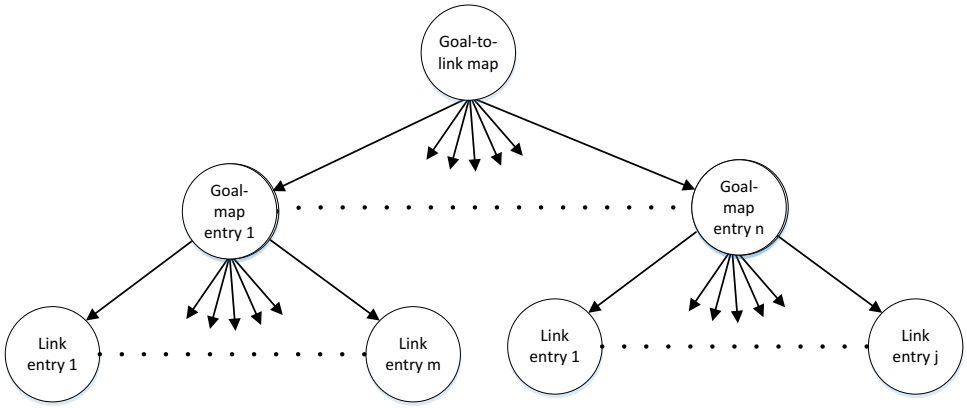

Figure 52: Goal-to-Link Map Structure

Similar to the link-to-state map, every goal map entry stores a goal hash and a collection of link entries. Every link entry stores a link hash and a number of times the corresponding link has been associated to the goal of the parent goal map entry. The probability calculated form the goal-to-link map used to constrain the state probability distribution at a given observation, *n*. For example, if the next state in an observation has high probability, but the link of the previous state has a low goal-to-link probability with next state goal due to lack of associations, the state probability distribution is constrained.

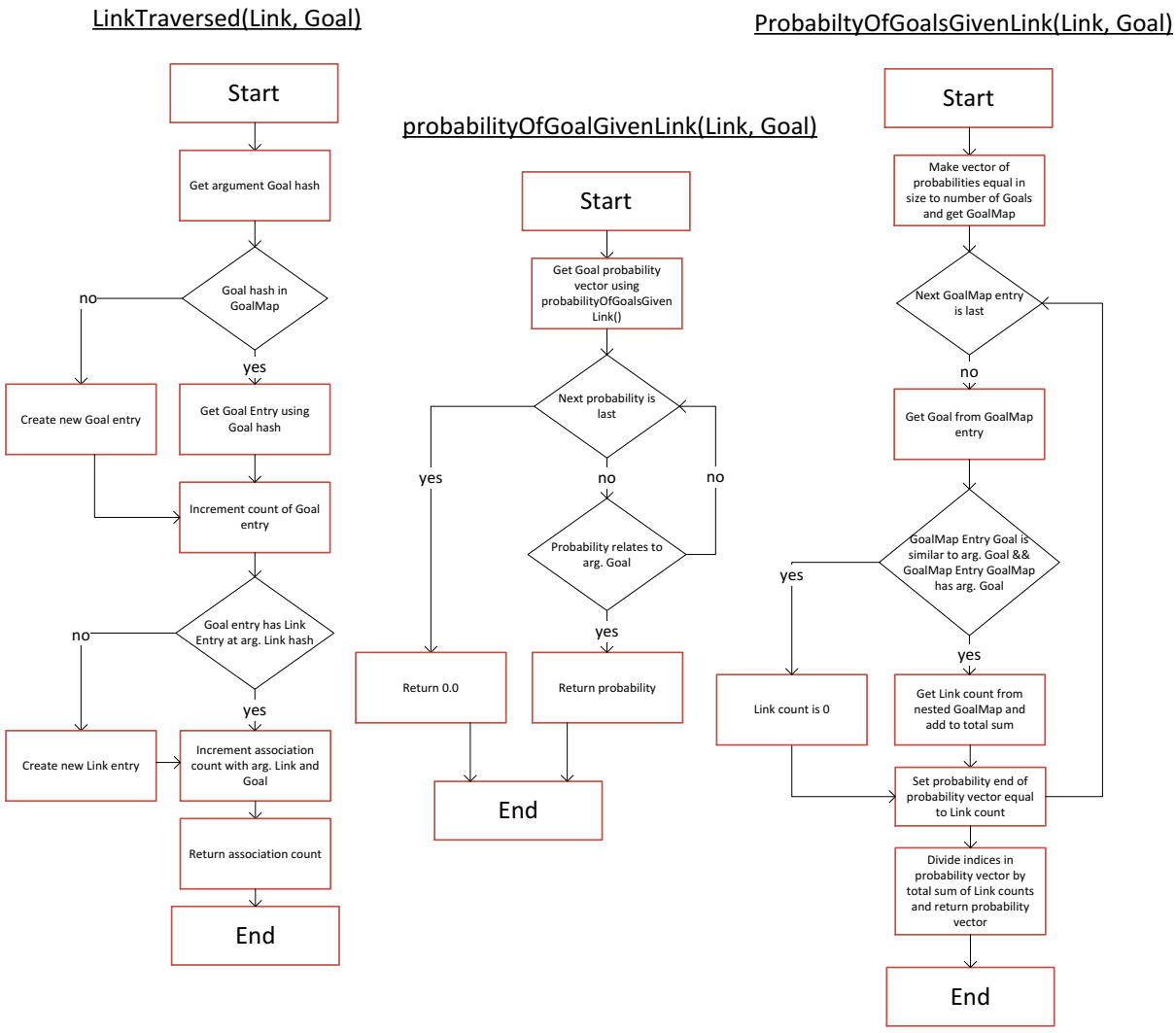

Figure 53: Activity Diagram for the linkTraversed, probabilityOfGoalGivenLink, and probabilityOfGoalsGivenLink Functions

4.4.6.1 linkTraversed

This function is used in the training step of route prediction to store associations between goals and links similar to the 'incrementTransition' function of the link-to-state map class. Updates are made to the two-dimensional hash map stored by the class for every link in a route by incrementing the association number between a link and stored by a route.

The process is completed by using the hash of the argument goal to retrieve the corresponding goal map entry stored by the class. If the goal map entry does not exist, a new one is created. The link-to-state map entry corresponding to the argument link hash is retrieved from the goal map entry. If the link-to-state map entry exists, the association number is incremented using the 'updateEntry' function of the generic map class. Otherwise, a new link-to-state map entry is added to the goal map entry at the argument link hash with a value of one. Lastly, the association number is returned.

4.4.6.2 probabilityOfGoalsGivenLink

This function is used to return the likelihood of all goals representing the end destination as a function of all associations stored by the class and the argument link. It is used by the 'probabilityOfGoalGivenLink' function to return the probability of a single goal. The function is executed by iterating across goal map entries of the class. For each iteration, the existence of the argument link within the goal map entry is established and the goal map entry goal is assessed to see if it is similar to the argument goal. Again, the 'isSimilar' function is used to broaden the predictors beyond goal-to-link associations to include starting conditions. If the case statement is true, locally scoped variables containing the number of times the argument link are associated with the goal map entry goal. A cumulative sum of argument link associations are also updated using the 'getMapEntry' of the goal-to-link map entry class. The hash of the goal map entry goal and the count of associations to the argument link are stored in two-dimensional array to assess a probability for each goal entry goal after the loop runs to completion. Finally, the count of associations between the goal map entry goals and the argument links in the two-dimensional array are averaged using the cumulative sum of associations between the argument goal and link. The array is then returned.

4.4.6.3 probabilityOfGoalGivenLink

The last function of the goal-to-link map class is used to retrieve the probability of the argument goal representing the end destination as a function of the argument link. To do so, a two-dimensional array containing all goal map entry goals stored by the class and respective probabilities is retrieved from the 'probabilityOfGoalsGivenLink' function. A loop over the returned array is used to identify the matching argument link hash to one stored in the array. If a match is found, the corresponding probability is returned. Otherwise, if no match is found, a probability of zero is returned.

4.4.6.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the goal-to-link map entry class.

Table 26: GoalToLInkMap Getter Field

Variable Name	Access Type	Description
goalMap	Getter	2D hash map containing goal map entries of link hashes and the number of times they have been observed for an end goal

The hash map is accessible through a getter for data serialization. Upon de-serialization, the hash map is reconstructed using publicly accessible functions within the class.

4.4.7 GoalMapEntry

The goal map entry is a class used to store link-to-state-map entries for the link-to-state map class and goal-to-link associations for the goal-to-link map class. Because link-to-link transitions and single links associate with a goal in the implemented Hidden Markov Model, the goal map entry class is designed as a template to store various data types in relation to a goal. As a result, the class inherits the same functionality as the generic map class with the only distinction being that the goal map entry also stores a goal.

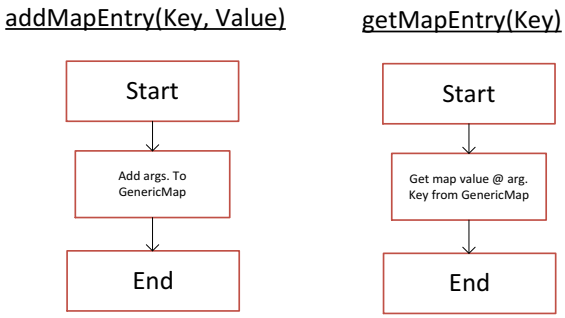

Figure 54: Activity Diagram of the addMapEntry and getMapEntry Functions

4.4.7.1 addMapEntry

This function serves as a wrapper and, in effect, the argument key and value are passed to the 'addEntry' function of the generic map class.

4.4.7.2 getMapEntry

Similary, this function is another wrapper. It returns the retrieved value of the 'getEntry' function from the generic map class after being passed to the argument key.

4.4.7.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the goal map entry class.

Table 27: GoalMap Getter

Variable Name	Access Type	Description
Map	Getter	Template map used to associate a link hash to either a LinkToStateMapEntry or a integer
Goal	Getter	Goal the goal map entry pertains to
M	Getter/Setter	Used to keep track of the number of associations between the key and value

The template map and goal is accessible through a getter for data serialization. Upon de-serialization, the map and goal are reconstructed using publicly accessible function within the class.

4.4.8 Goal

The goal represents an end destination in a route and is used to associate with links and link transitions. The goal class shares an identifying number with a corresponding intersection, a vector of starting conditions and a count of the number of times the goal has been observed as the actual end destination.

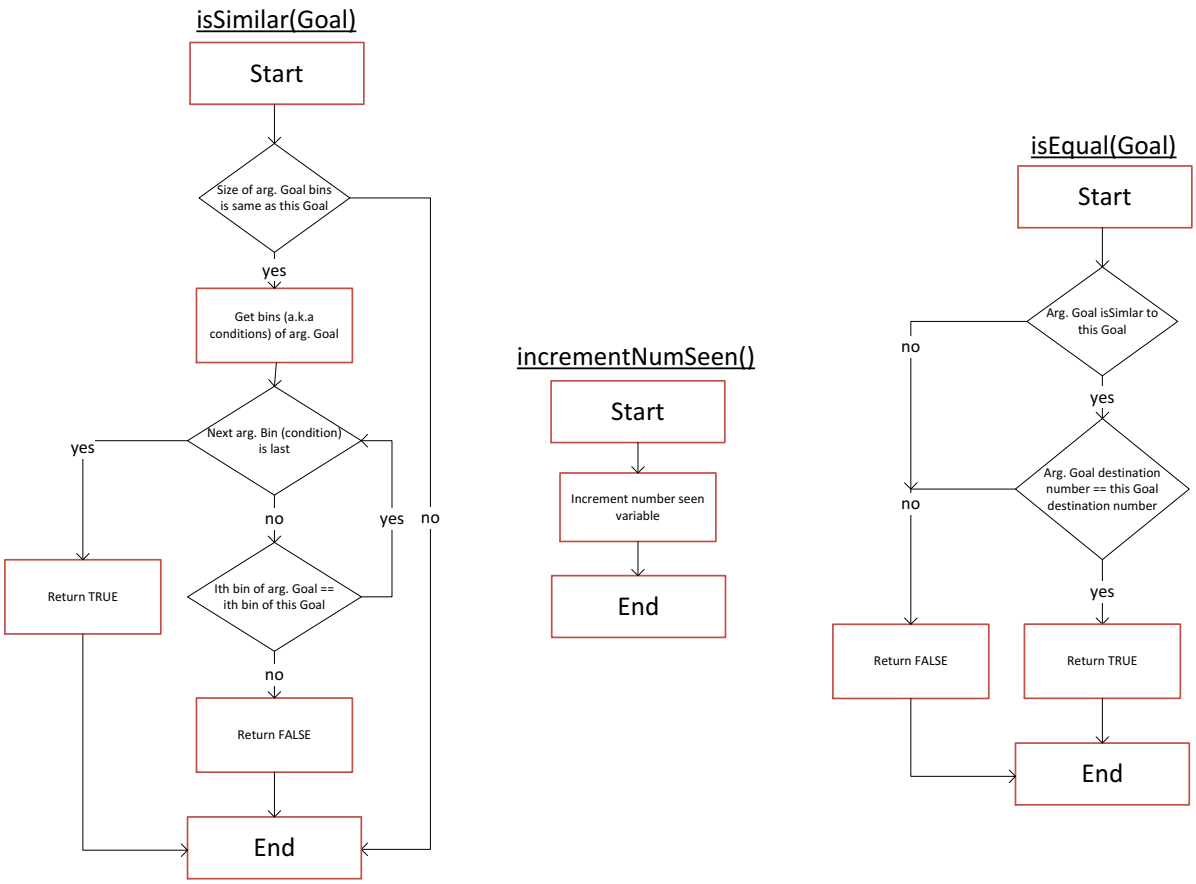

Figure 55: Activity Diagram of the isSimilar, incrementNumSeen and isEqual Functions

4.4.8.1 isSimilar

This function is used to compare starting conditions of two goals. It is used by the goal-to-link and link-to-state maps to create a subset of known goals to associate links and link-to-link and transitions with each other. The function operates by asserting the starting conditions vectors are equal in size. If they are, it iterates over the vectors to compare index values. If a pair of index values mismatch or the starting condition vectors are unequal in size, a false Boolean is returned. Otherwise, a true Boolean is returned once the loop runs to completion.

4.4.8.2 incrementNumSeen

This function is used in the training step of the Hidden Markov Model. It is implemented to increment the number of times the goal has been observed as the end destination. This value is used by the route prediction class as *a priori* of the implemented Hidden Markov Model, $p(g) = 1$, to bias the prediction of subsequent links.

4.4.8.3 isEqual

The ‘isEqual’ function is used to compare the identifying numbers, number of times seen and starting conditions of two goals using the ‘and’ operation. The starting conditions are assessed using the ‘isSimilar’ function.

4.4.8.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the goal class.

Table 28: Goal Getters and Setters Fields

Variable Name	Access Type	Description
Destination	Getter	The identifying number of the intersection the goal is set to
Bins	Getter	The conditions (weather, time, etc.) associated with the goal
numSeen	Getter/Setter	The number of times the goal has been observed

The destination and bins fields are accessible through getters for serialization and referencing. The 'numSeen' field is settable to increment the number of times the goal has been observed.

4.4.9 RoutePrediction

The route prediction class implements a first-degree Hidden Markov Model further described in the Route Prediction section. It is used to calculate Markov chains representing an ordered collection of connected road segments over the known road network. In this application, the hidden component of the model is the drivers' intended destination and route.

A typical Markov model is comprised of six elements: 1) a finite set of states, 2) a finite set of actions, 3) a transition function which is the probability of transitioning to a state given an action, 4) a finite set of observations, 5) an observation function which is the probability of receiving an observation after the system ends up in a state given an action, and 6) an initial state distribution. For the implemented model, a state is a link associated with a goal. Because actions are not represented explicitly, they are defined using the transition function. An observation is the current GPS location of the vehicle within the known road network. The observation function is deterministic, meaning it is assumed there is no error in localization. The initial state distribution is the probability distribution across the most probable goal.

The transition function is used to describe a movement from one state to another as a function of an action. It is assessed using two probabilities rather than the single probability of the given state transition. The probability of a state given a link is multiplied by the probability of the state goal given the same link. Specifically, given the driver's state, the next link is predicted, and the drivers' goal is predicted given the predicted next link.

Given the ability to predict the next link, the complete route can be predicted several different ways: 1) the goal probability distribution can be used to predict the next link until the most probable goal or an arbitrary goal is reached, 2) because the goal probability is calculated as a function of the next link probability given the current state, the most probable route can be determined irrespective of the most likely goal, and 3) the system can be biased with a predicted end goal *a priori*i to effect the subsequent prediction of the Markov chain.

The following route prediction algorithm employs a combination of the three strategies above. To generate the predicted route, a recursive forward algorithm adapted to the dual-probability transition function is employed to update state probabilities distributions, choose the next most probable state, and repeat using the update function,

$$p(s_{t+1})_{i,j} = \sum_{k} p(g_j | l_{t+1,i}) * p(|l_{t+1,i}| < l_t, g_{t,k} >) * p(s_t)_k. \quad (45)$$

Above, k defines the number of previous states containing the previously predicted link, l_t, $l_{t+1,i}$ is the i^{th} next potential link, g_j is the j^{th} goal observed, and $g_{t,k}$ is the k^{th} previously predicted state goal.

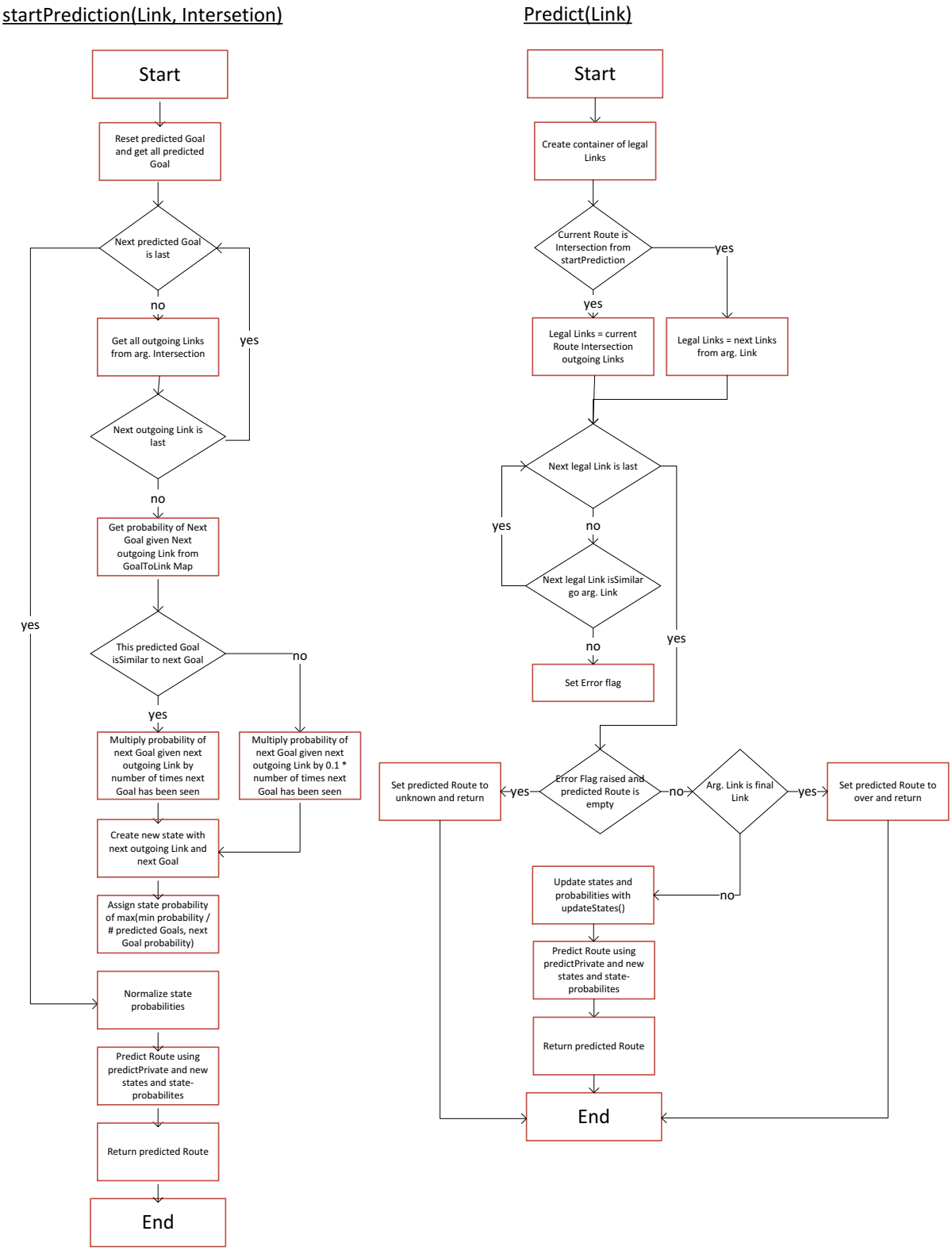

Figure 56: Activity Diagrams for the startPrediction and Predict Functions

4.4.9.1 startPrediction

The ‘startPrediction’ function is used to create the initial state distribution and employ the recursive forward algorithm to generate the predicted route. To do so, a collection of links is retrieved from the ‘getOutgoingLinks’ function using the argument intersection and link. This function is different than the alternative prediction functions in the class in that it accepts an intersection. This is done to establish a starting point for prediction. Lastly, the vector of state probabilities stored by the class is initialized equalling the number of goals observed, multiplied by the number of connecting links retrieved.

The initial state distribution is created by selecting the most probable goal as a function of starting conditions to define *a priori* and bias the subsequent generation of the Markov chain. The associated next link to the most probable goal is used for the recursive process remaining in route prediction. The process is completed using a loop over all goals observed and a nested loop over all links connected to the argument link. For each goal-link pair, the probability is assessed using the ‘probabilityOfGoalGivenLink’ function of the goal-to-link map stored by the class. If the goal is not similar to the argument starting conditions determined by a dummy goal and the ‘isSimilar’ function, the calculated goal-link probability is penalized with a 0.1 multiplier. The link-goal state is stored in the collection of the class, and the state probabilities vector is updated corresponding values.

Once all state probabilities have been assessed, the values are averaged, copied, and the copied state probabilities are passed to the ‘predictPrivate’ function including a copy of the states to generate the remainder of the predicted route recursively. Copies of the state and state probabilities are provided for the recursive formation of the predicted route mutilates argument parameters and the states and probabilities stored by the class are used in subsequent calls to the prediction functions. The predicted route retrieved from the ‘predictPrivate’ function is finally returned.

4.4.9.2 predict

The ‘predict’ function is the other publically accessible prediction function other than ‘startPrediction’ and is used to execute pre and post process steps to generating the Markov chain after the initial state distribution has been created. Similar to the ‘startPrediction’ function, the connecting links to the argument link are retrieved using the ‘getNextLinks’ function of the city class or the ‘getOutgoingLinks’ function of the intersection class depending on whether the predicted route has an associated intersection stored by the route prediction class. A loop over the retrieved link is used to assert the argument link exists in the returned collection, for the argument link represents the next link taken from the previous point the state and state probabilities were calculated from and must by connected. If the argument link is found to connect, states are updated with the argument link and the resulting states and state probabilities are stored by the route prediction class.

If the argument link is equal to the first link of the predicted route, the first link is removed, and the resulting route is returned. Otherwise, copies of the stored states and state probabilities are made and passed to the ‘predictPrivate’ function to recursively generate the predicted route. The predicted route is then returned.

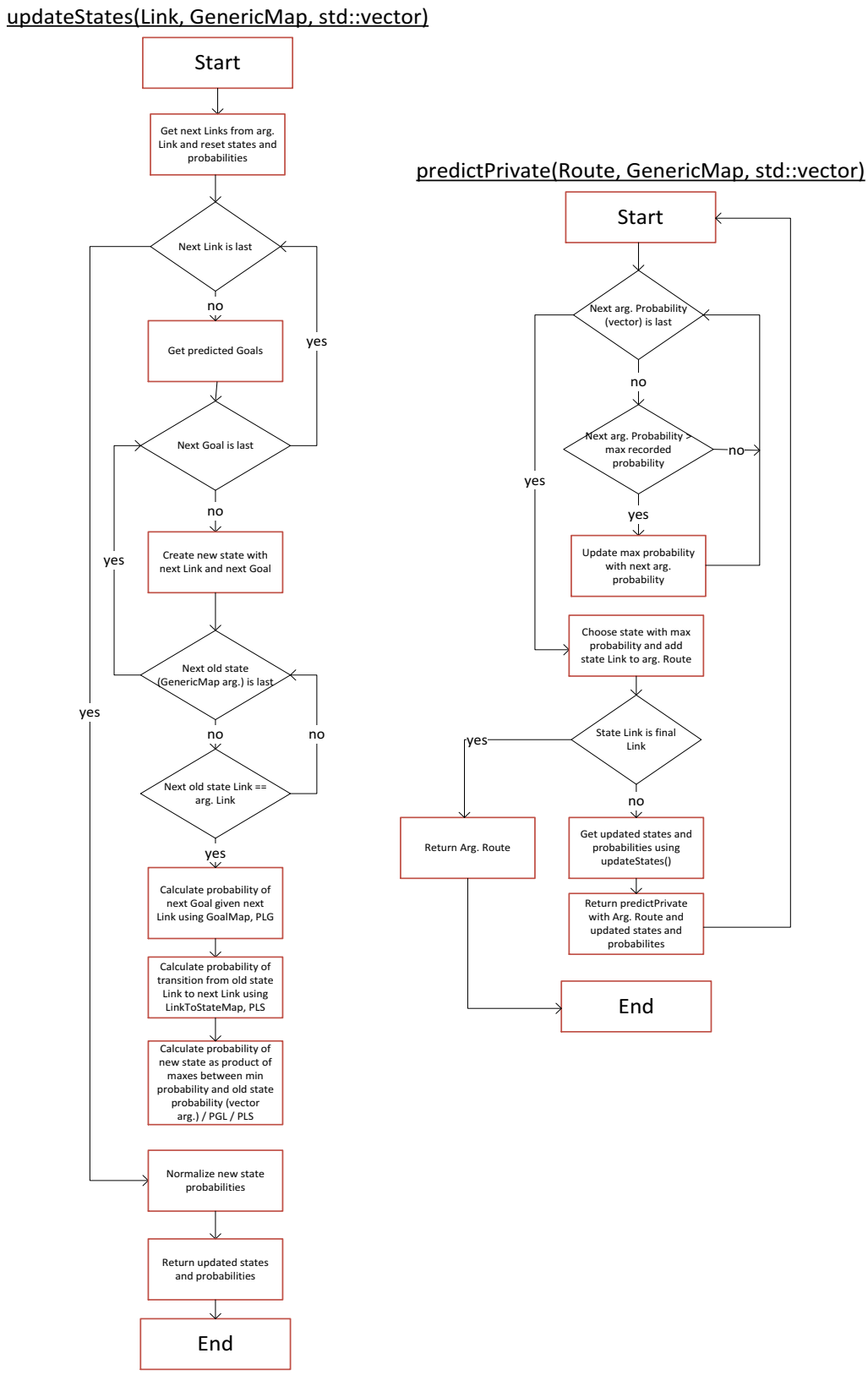

Figure 57: Activity Diagrams for the updateStates and predictPrivate Functions

4.4.9.3 predictPrivate

The ‘predictPrivate’ function is a recursive function used to generate the Markov chain. It uses a base case of the most probable state containing the final link indicated by the ‘isFinalLink’ function to exit recursion. The function operates by finding the most probable state in a loop over the argument state probabilities. It does this by updating the index of the state probability vector that contains the largest probability. If no probabilities are returned, the unknown route is returned. Otherwise, the most probable state is retrieved from the argument collection of states using the max-probability index. The link of the most probable state is added to the end of the argument route. If the route does not exist, signalling the most probable state is first, a route is created.

If the most probable state does not contain the final link, the ‘updateStates’ function is called using the argument states and state probabilities and most probable state link as arguments. The returned states and state probabilities are then passed to the ‘predictPrivate’ function in the recursive step. The updated argument route is returned.

4.4.9.4 updateStates

This function is used to update state and state probabilities as defined by equation (45) and is performed by getting a collection of connecting links to the argument link using the ‘getNextLinks’ function. A collection of new states and probabilities is initialized equal to the number of observed goals multiplied by the number of connecting links retrieved. The collection of connecting links is iterated across in a loop. In a nested looped, the collection of observed goals is also iterated across. In a second nested loop, the collection of previous states provided as an argument is finally iterated across to update the probabilities of the new states.

For each previous state, if the link is equal to the argument link, equation (45) is calculated using the ‘probabilityOfGoalGoalGivenLink’ function of the goal-to-link map, the ‘getProbability’ of the link-to-state map and the previous state probability passed as an argument. The goal-link pair is stored as a link and the calculated probability is stored in the new state probabilities vector. The result is a probability assessed for every connecting link as a function of every observed goal. Lastly, the new state probabilities are averaged and the returned with the new states.

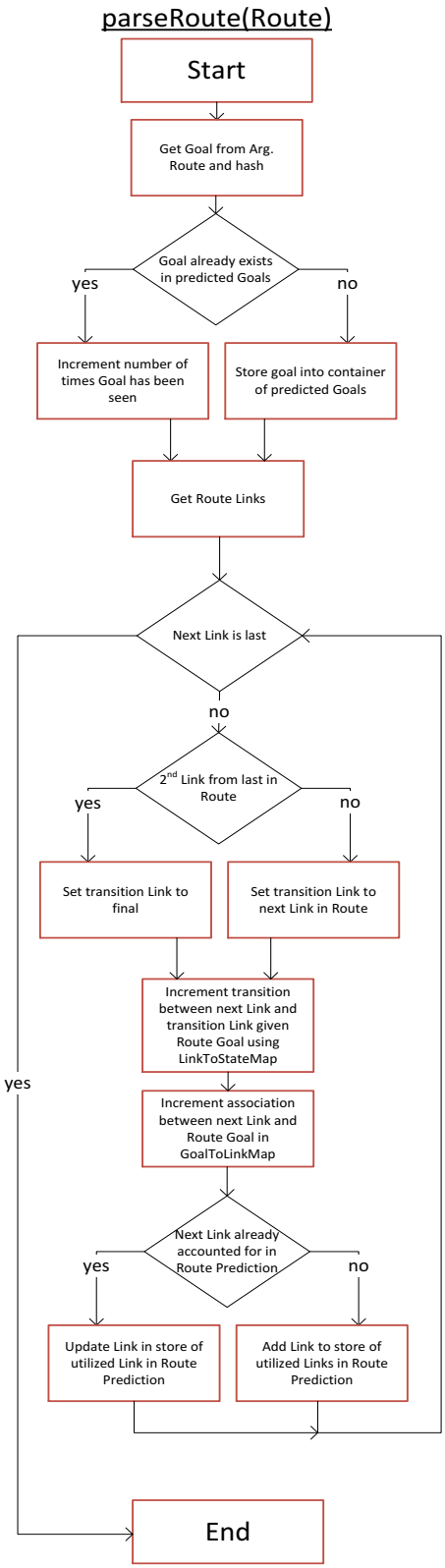

Figure 58: Activity Diagram for the parseRoute Function

4.4.9.5 parseRoute

This function is used to update the multidimensional hash maps stored by the class. It is the only way of training the Hidden Markov Model by updating link-state and link-goal associations and the collections of links and goals observed. To do so, the goal is retrieved from the argument route, hashed and used to determine if the goal has been observed. If it has, the number of times the goal has been seen is incremented. Otherwise, the number of times the goal has been seen is set to one and stored in the collection of observed goals using the goal hash as the key and the goal pointer as the value.

Next, the ordered collection of links of the argument route is iterated over in a loop. For each link, the next link is retrieved if the count of iterations is less than the size of the collection. If the count of iterations is equal to the size of the collection, the next link is set to the final link. The link-to-state map is updated by passing the current and next links and the argument route goal to the 'incrementTransition' function. The goal-to-link map is updated by passing the current link and argument route goal to the 'linkTraversed' function. Lastly, once the current link is added to the collection of observed links using the 'addEntry' function of the generic map class, the learning process finalizes and all links of the argument route are iterated over.

4.4.9.1 Getters and Setters

Below is the list of fields accessible through getters and setters for the route prediction class.

Table 29: Route Prediction Getter and Setter Fields

Variable Name	Access Type	Description
linkToState	Getter/Setter	Contains all associations between link-to-link transitions and an end goal.
goalToLink	Getter/Setter	Contains all associations between a link and an end goal
Links	Getter/Setter	Contains all observed links
Goals	Getter/Setter	Contains all observed goals
City	Getter/Setter	Used to determine next links or intersection from arguments links and intersections in public functions
currentRoute	Getter	Contains historically accurate ordered collection of links for a given trip up to current location
predictedRoute	Getter	Contains ordered collection of links comprised predicted route from current location to predicted end destination
predictedGoal	Getter	Predicted end destination

The most important variables to the class are the multi-dimensional hash maps and collections of links and goals. These fields have setter capabilities to recreate route prediction from serialized data without loss of learned driver routes.

4.4.10 SpeedPrediction

The speed prediction class implements a Multi-Input, Multi-Output Neural Network further described in the Neural Network Speed Prediction section. It is used to predict time-series speed as a function of historical time-series speed measurements. The neural network is comprised of six elements: 1) units, 2) states, 3) biases, 4) hidden states, 5) weights, and 6) activation

functions. These units make up the architecture of the neural network which is depicted in Figure 7: Neural Network Design using multiple layers of units with each connected to every other unit in the adjacent layers. The states represent the values calculated at each unit using the associated weights, hidden states, and activation function. The bias is used to shift the sigmoid curve of the activation function in the domain with respect to the weights inputted so that the output range is more variable in relation to an input. Hidden states are states of the hidden units between the input and output layers. The weights describe numerically the strength between units. Lastly, the activation function used is the sigmoid function and defined in equation (15).

A standard feed forward prediction algorithm is used to cascade state values from the input layer through the hidden layers to the output layer of the network using the sigmoid activation function to produce a predicted output. State values are calculated by multiplying the previous layer states by the transpose of the current layer weights for each unit in which the products are inputted into the sigmoid function.

Training, on the other hand, is more sophisticated and relies on gradient descent to update weight values as a function of the cascaded state delta between the desired output and states of the output layer, and from the output layer back through the remainder of the network.

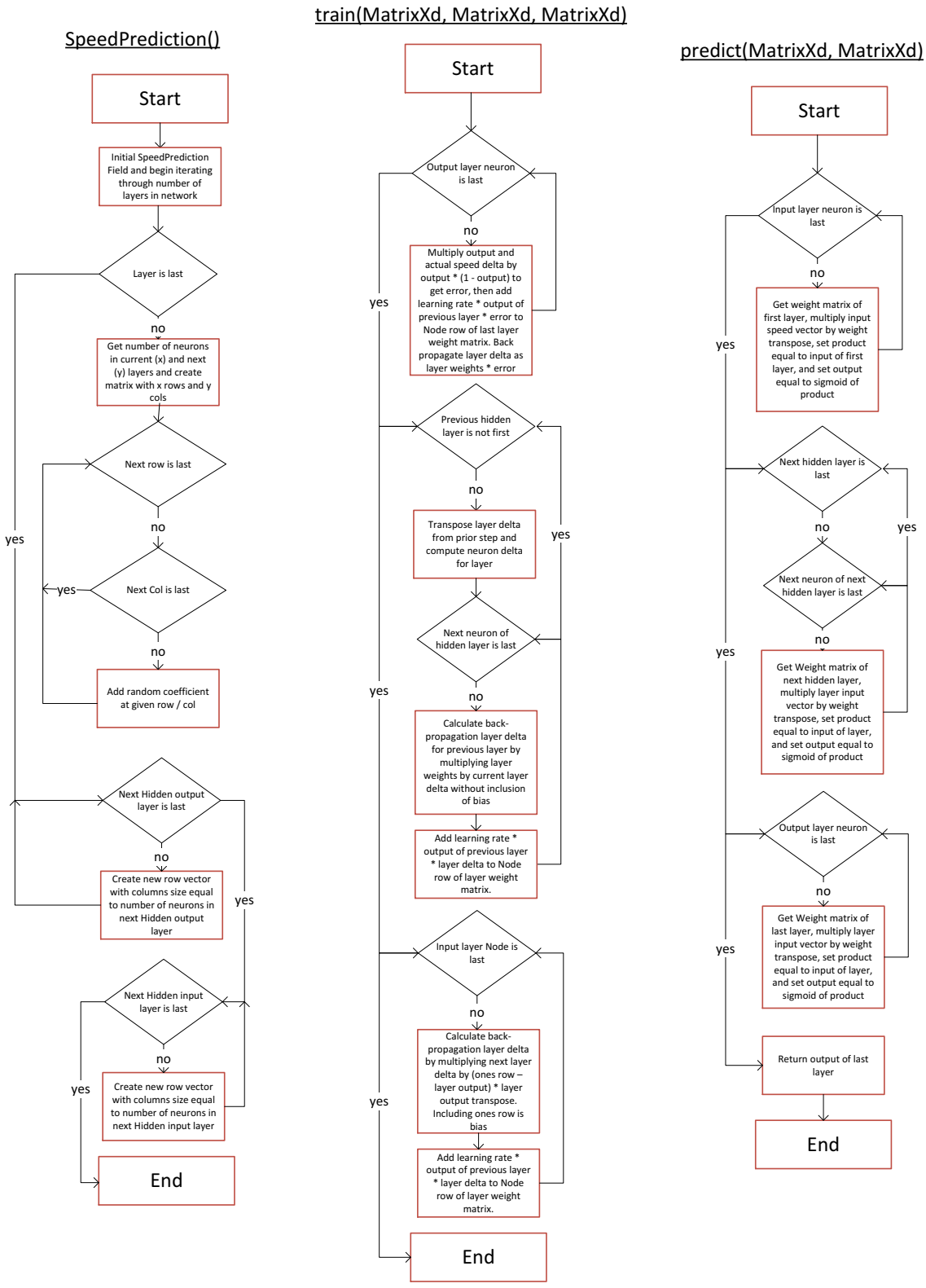

Figure 59: Activity Diagrams for the SpeedPrediction, predict, and train Functions

4.4.10.1 SpeedPrediction

The constructor of this class is used to initialize the weight matrices with random scaled values to be trained on later. The function also initializes the vector arrays of the input and state values to the hidden layers. Weight matrices are initialized in two nested loops over the rows and columns of a weight matrix used for two adjacent layers. In the outer most loop, the number of rows for a given weight matrix is equal to the number of units in the current layer. The number of columns is equal to the number of units in the next layer, plus one, to account for the inclusion of the bias term. Values on the range of [-.05 and .05] are placed into each cell of each matrix using the 'rand' function of the standard C++11 library.

The input and state vector arrays are also initialized with respective loops. One less input vector is needed due to the direct input of the scaled historical time-series speed measurements. The length of the input vectors is equal to the length of each layer, also plus one, to account the bias term. The state vector array is equal in length to the number of layers used and vector length is equal to the length of the corresponding layer.

The weight and state vector arrays are vital to the network for feed-forward and back-propagation algorithms. The input vectors, on the other hand, are not necessary and used as temporary storage containers for the products of the previous layer states and the current layer weight vectors.

4.4.10.2 predict

This function is used to employ the feed-forward prediction algorithm. The following image depicts the process by which the state vectors are updated through the hidden layers:

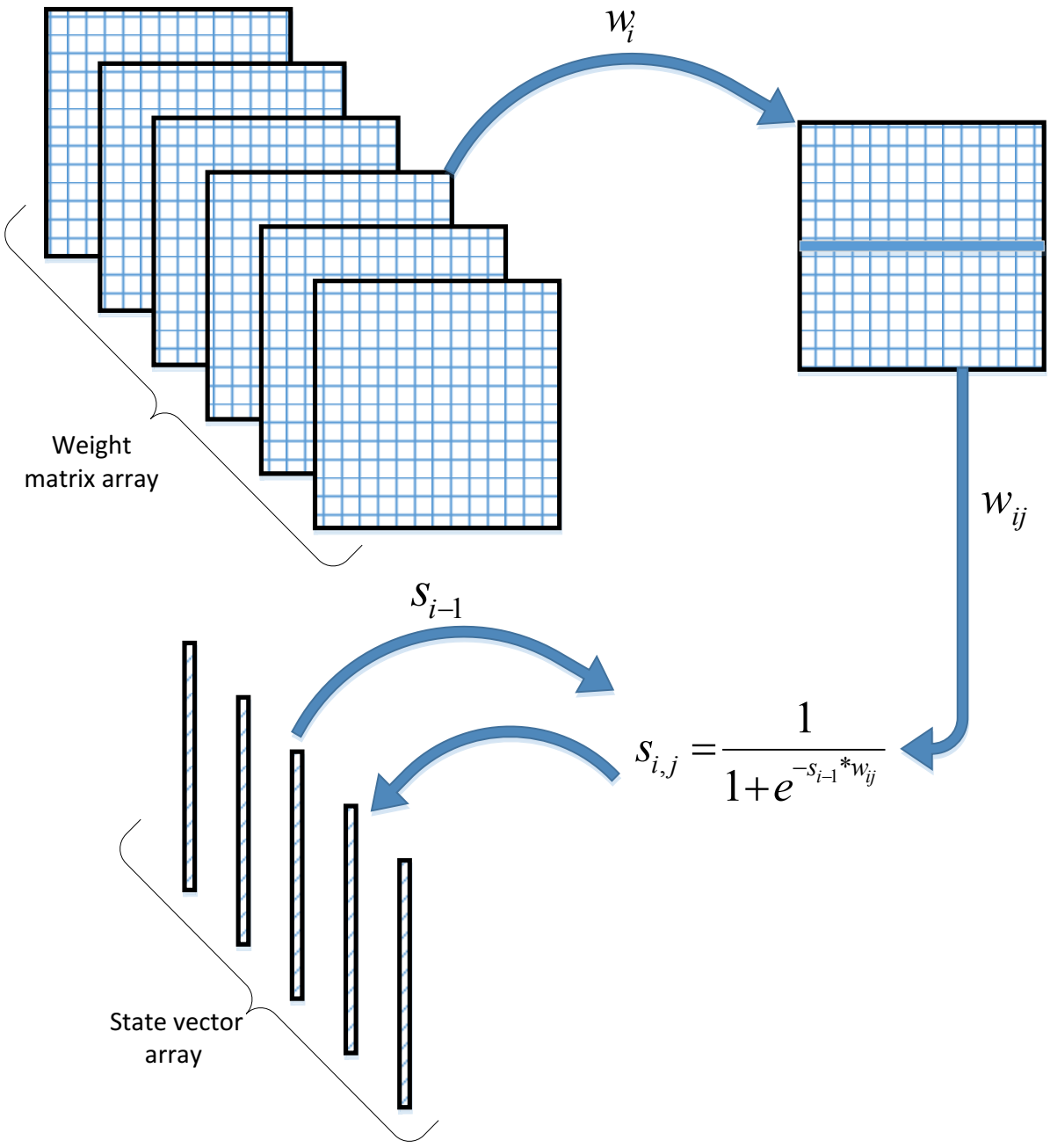

Figure 60: Feed-Forward Speed Prediction

Forward prediction through the hidden layers is performed in a nested loop. The outer loop iterates through the weight, input and state vectors, and the inner loop iterates through the rows of each weight matrix, input, and state vectors representing the number of units in each layer. For each iteration of the inner loop, the previous state vector is multiplied by the current row of the weight matrix selected. This value is stored in the same row or the input vector. The value stored in the input vector is then inputted into the sigmoid activation function, and the resulting product is placed in the same row of the current state vector. At the end of each iteration cycle of the

inner loop, the last row-value of the current state vector is set to one as the bias for future feed-forward predictions, for this value is overwritten. The input vector is not necessary but used as an intermediary container to check calculations using the debugger.

The only exception to the feed-forward process described above is the utilization of the historical time-series measures and the population of the output vector and the input and output layers, respectively. At the input layer, the historical time-series measurement is used as the previous state vector. At the output layer, the current state vector is the output vector.

4.4.10.3 train

Back propagation training using gradient descent is a three-step process. First, the rows of the output layer matrix (one for each output unit) are assessed in a loop using equation (17). The desired output and the output vector of the system produce the error gradient with respect to the output of the system. Additionally, the error gradient with respect to the output unit is calculated using equation (19) and with respect to the row of the output layer weight matrix using equation (20). The error gradient with respect to the output layer weight matrix row is saved to be used in the next back propagation step. Lastly, the output layer weight matrix row is updated using equation (24) in which the cumulated error gradient of the weight matrix for the next back propagation step is calculated as the previous state vector, multiplied by the error gradient with respect to the output unit from equation (19).

Once the error gradient with respect to the output layer weight matrix is calculated across the output units, the gradient is propagated back across the layers of the network using equation (24) in another loop. For each layer iterated over, a new error gradient with respect to the units is calculated using equation (19) and the error gradient with respect to the weight matrix from the previous back propagation step. The error gradient with respect to the current layer weight matrix is then calculated for the next back propagation step multiplying the current layer weight matrix by the error gradient with respect to the units of the current layer defined by equation (20). Finally, equation (24) is used to update the weights of the current layer; the cumulated error gradient of the layer weight matrix for the next back propagation step is calculated as the previous state vector multiplied by the error gradient with respect to the current layer units from equation (19). This process is repeated until the all layer weight matrices are updated except for the first layer in which the same process is repeated; and the previous state vector is replaced by the historical time-series speed measurements.

4.4.10.1 Getters and Setters

Below is a list of fields accessible through getters and setters for the speed prediction class.

Table 30: SpeedPrediction Getter and Setter Fields

Variable Name	Access Type	Description
wts	Getter/Setter	Array of weight matrices for each layer
yHid	Getter/Setter	Array of output vectors for each layer
yInHid	Getter/Setter	Array of input vectors for each layer

All vital fields to the speed prediction class are accessible and settable to quickly swap weight matrix and I/O vector arrays into speed prediction as speed is predicted over a route comprised of multiple links.

5. HARDWARE IMPLEMENTATION

Below is a diagram depicting hardware used and connections made. GPS measurements are reported continuously from the ublox EVK-M8N. OBDII vehicle diagnostics are reported upon request from the ELM327 V2.0.

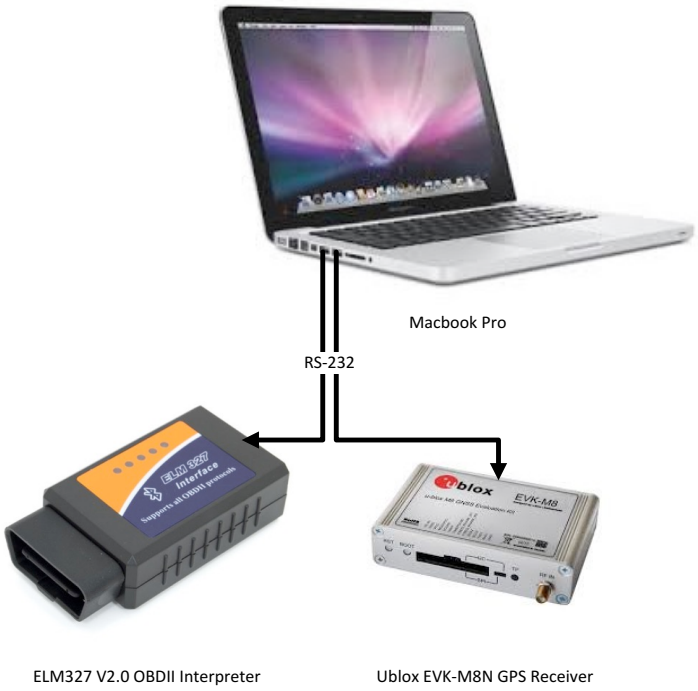

Figure 61: Top-Level Connectivity

The majority of data inputted to the Driver Prediction is provided by an OBDII interpreter and a GPS receiver. Broadcast rate of the ublox GPS receiver is fixed at 330 ms and measurements rely solely on satellite triangulation. There is no sensory fusion of a wheel-tick odometer or the onboard Intertial Measurement Unit even though the receiver is capable of using both. Power to both devices is provided by USB.

5.1 Run-Time Performance

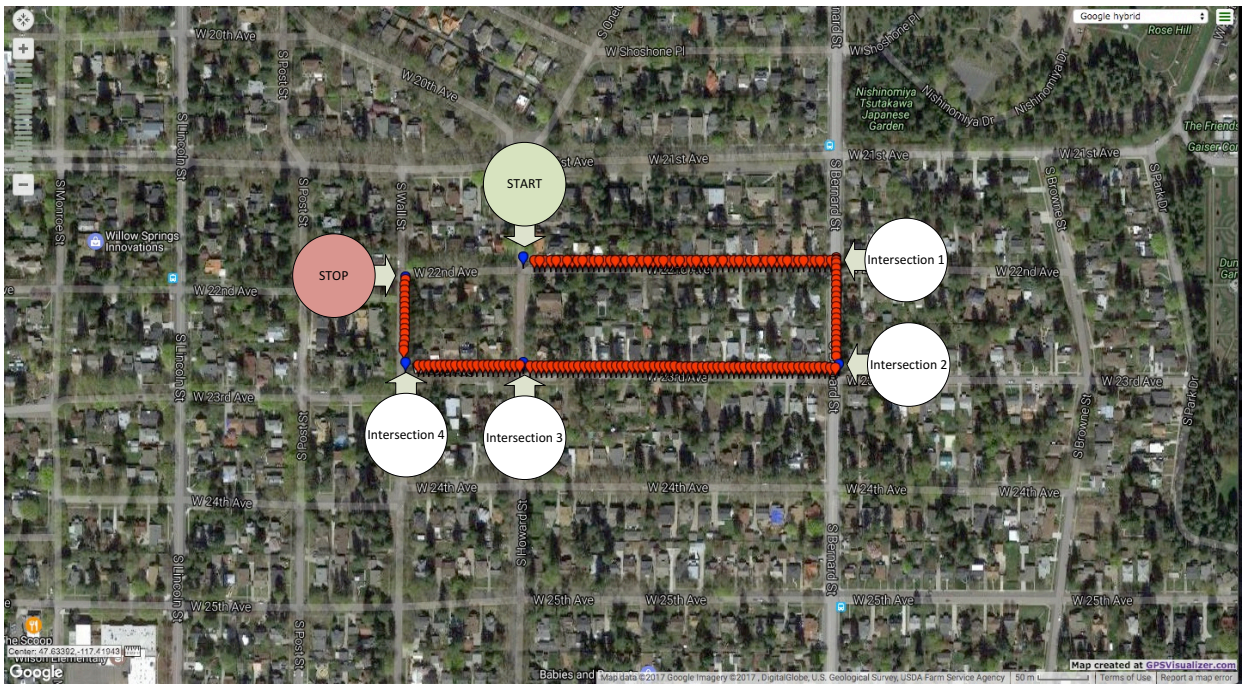

Figure 62: Route Used to Assess Run-Time Performance

Below are metrics of run-time performance of the algorithm over a short route in a rural neighborhood produced by Xcode. Major computations take place at intersections when route prediction states update in an $O(n^4)$ operation within a recurrent function. Direction of travel along the route is marked by the start and stop indicators.

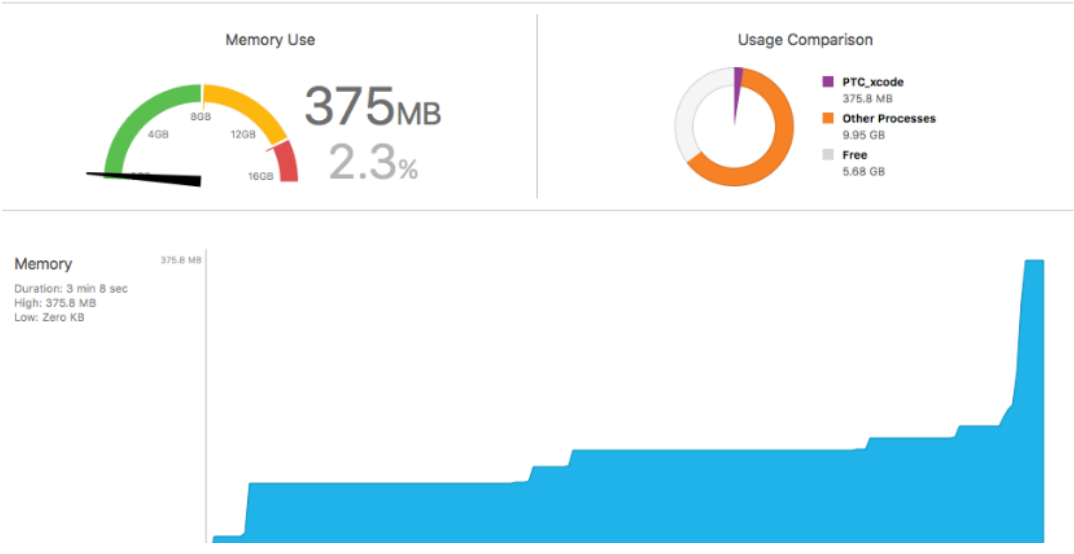

Figure 63: Memory Usage Over Short Route in Rural Neighborhood

Memory usage increases greatly at the beginning of the algorithm as data is de-serialized and slowly increase as memory is dynamically allocated to the heap to update route prediction states

at each intersection. The ‘Memory Use’ and ‘Usage Comparison’ charts are snap shots of memory utilization at the time the photo is taken. However, the blue diagram depicts memory usage over time.

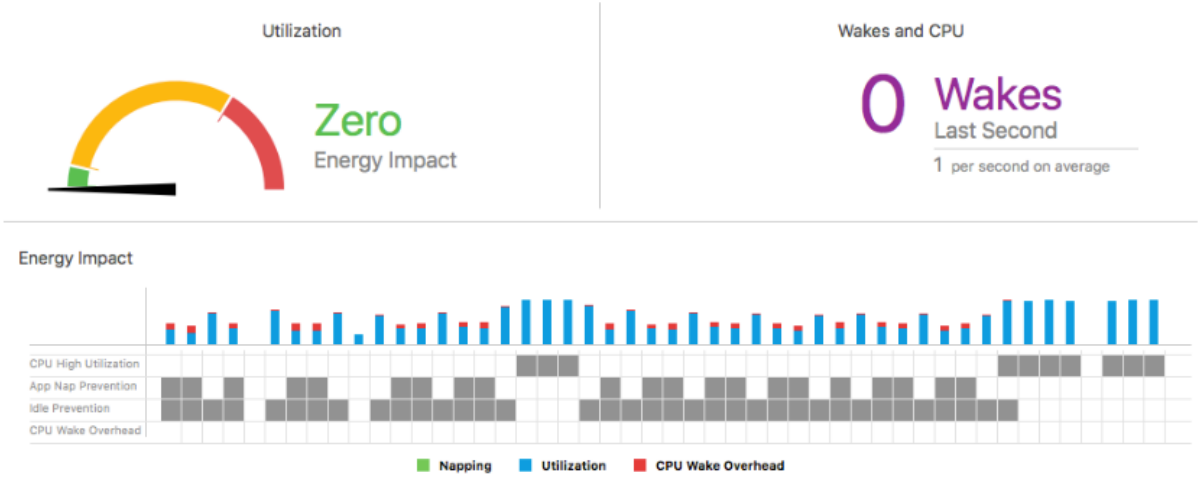

Figure 64: Energy Impact over Short Route in Rural Neighborhood

The above depicts the activity of the algorithm over the same small rural route. The algorithm is not put into ‘App Nap’, a mode utilized by OS X to reduce energy consumption, as it is continuously executing. As a result, no wakes are used, and the system is never placed in an idle state. Through most of the run-time execution, CPU wake overhead is available except for when states are updated at intersections.

Figure 65: Data Written at End of Driver Prediction Algorithm

The above diagram depicts the amount of data written to disk at the end of execution before shutdown occurs. For reasons unknown metrics of data read is not provided, but is expected to be the same or comparable to data written if a new route is trained. The amount of data read and written is a function of the size of the known road network, the number of trips taken, and the number of routes stored by the route and speed prediction algorithms. Serialized data represents a known road network of approximately 24x35 blocks in which approximately 12.5 % of the known road network consists of routes used to train route and speed prediction.

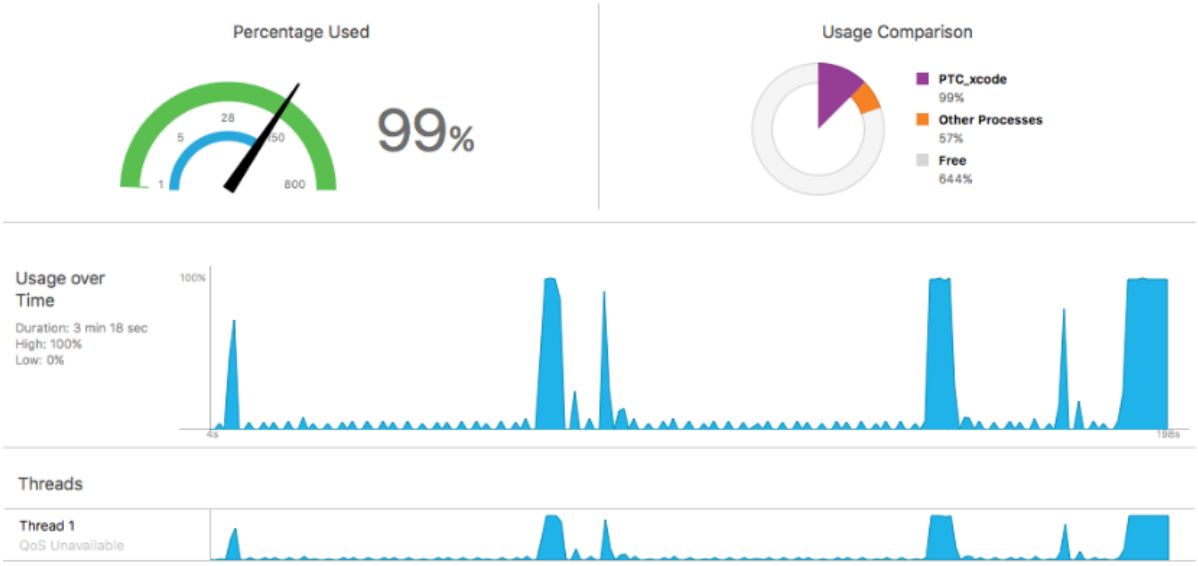

Figure 66: CPU Usage Over Shor Route in Rural Neighborhood

Above is a diagram depicting CPU usage over time of the same short rural route. Similar to memory metrics, the CPU usage over time depicts the large processing requirement of route prediction at intersections. The small periodic spikes in CPU usage is the processing of speed prediction over five meters along the predicted route.

6. TEST PLAN

The following section is divided into two categories, software modules and system testing, to define low-level module specifications and cases that are verified in unit-tests and system-level specifications and cases that are verified in road testing.

6.1 Software Module Test Plan

Each software module has a unique unit test to assess proper functionality as an independent entity where the module is not used in a larger system. For each module, a unit test is written and includes ‘assert’ functions of the C++11 standard library to ensure output is the intended function of the input. Unit tests are commented-out in the ‘main’ function and uncommented whenever a change is needed that can be verified without drive-testing. This greatly improves code-base development efficiency.

6.2 Software Module Test Specification and Cases

The following will list what is to be tested as well as quantify different test limits for each software module.

6.2.1 RoutePrediction

Below is a list of test specifications for all data I/O from the route prediction class.

Table 31: RoutePrediction Test Specifications

Access Function	Variable Name	I/O	Specification
startPrediction	linkTaken	I	Current link the vehicle is located on
	startIntersection	I	Intersection in which route initiated from
	predictedRoute	O	Predicted route from the next intersection to the predicted end destination
predict	linkTaken	I	Current link the vehicle is located on
	predictedRoute	O	Predicted route from the next intersection to the predicted end destination
parseRoute	route	I	Actual route the vehicle took

The ‘startPrediction’ function is responsible for initializing states and state-probabilities associated with the route prediction class, generating the predicted route, and recursively updating local states and state probabilities for each route link. A successful ‘startPrediction’ execution returns a legitimate route that is or is similar to the actual route without fault. The ‘predict’ function operates similarly to ‘startPrediction’, but relies on previously updated states from the last predicted route and link taken. Before recursively generating the predicted route, however, steps are taken to assess the legality of transitioning from the previous to the current link as a function of the connections made in the road network. Similarly, a successful execution of the ‘predict’ function is one that returns a legitimate route that is similar or equal to the actual route without fault. For both prediction functions, state probabilities are assessed as a function of link-to-state and link-to-goal associations stored in the LinkToStateMap and GoalToLinkMaps fields of the route prediction class. Finally, the ‘parseRoute’ function accepts a traversed route and updates the LinkToStateMap, GoalToLinkMap, Links, and Goals classes belonging to the route prediction class. Successful execution of the ‘parseRoute’ function updates all aforementioned fields without fault.

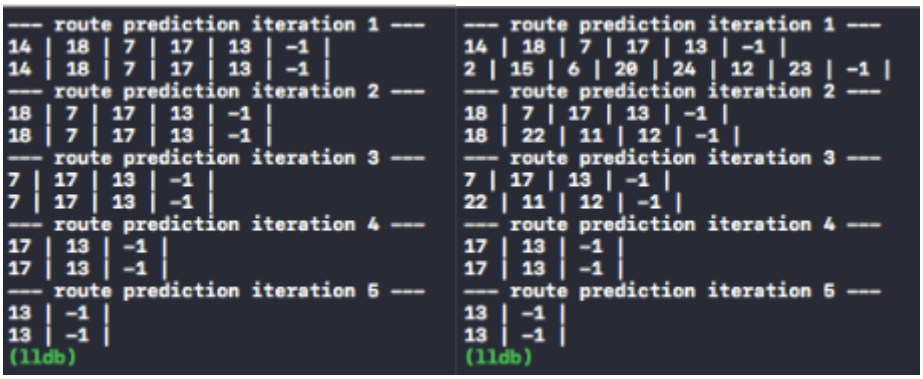

Figure 67: Trained Versus Untrained Route Prediction Unit Test Output

The publicly accessible functions of the route prediction class are tested using the following methods. The ‘parseRoute’ function is tested by creating a random collection of links stored in a route and passing the random route into the ‘parseRoute’ function. As the function iterates through the route links, the link-to-state and link-to-goals associations are checked for

correctness using the debugger. The ‘startPrediction’ and ‘predict’ functions are tested using a mock road network simplified for hand validating the correctness of the functions. In both cases, a randomly generated route in the mock road network is created, parsed and then iterated over using the stored links, iteratively feeding the links of the random route into the ‘startPrediction’ and ‘predict’ functions to monitor the correctness of the predicted route. A simple debugger is used to verify states and state probabilities are updated correctly and link-to-state and link-to-goal associations are used to correctly to assess probability. Meanwhile an ‘assert’ is used to ensure the predicted route is correct before completely iterating over the actual route.

6.2.2 LinkToStateMap

Below is a list of test specifications for all data I/O from the link-to-state map class.

Table 32: LinkToStateMap Test Specifications

Access Function	Variable Name	I/O	Specification
incrementTransition	li	I	Transition link
	lj	I	Current link
	gj	I	Predicted goal
getProbability	li	I	Transition link
	lj	I	Current link
	gj	I	Predicted goal
	pl	O	Probability of link-to-link transition given goal

The ‘incrementTransition’ function is used to increment the association between a link-to-link transition and an end goal using a three-dimensional hash map. The first dimension of the hash map stores the goal hash. The second level stores the transition link hash. The third level stores the current link hash with the count of associations between the transition and current links. If an association does not already exist, it is added to the LinkToStateMap hash map. The ‘getProbability’ function, on the other hand, iterates across all transition link entries stored under the goal hash passed to the function. For each transition link associated with the argument goal, the number of times the current link is observed is added over the total number of transition link observations, and the probability is returned after iteration is complete.

The ‘incrementTransition’ function is tested by creating a set of links and goals and through a unit test, asserting the hash map stored in the LinkToStateMap is updated correctly. This is done as different combinations of transition links, current links, and goals are passed to the function. The ‘getProbability’ function is tested similarly using the same set of links and goals. The returned probabilities are asserted to be true by pre-generating the probabilities for each of the different argument combinations.

6.2.3 LinkToStateMapEntry

Below is a list of test specifications for all data I/O from the link-to-state map entry class.

Table 33: LinkToStateMapEntry Test Specifications

Access Function	Variable Name	I/O	Specification
addEntry	li	I	Input link
	m	O	Count of link observations
getM	li	I	Input link
	m	O	Count of link observations
getTotalM	m	O	Total count of link observations

The ‘addEntry’ function simply increments the count associated with the argument link, and if the link does not already exist in the hash map, a new entry is added. In both cases, the number of observations is returned. The ‘getM’ function returns the number of observations stored for the associated hash map. Finally, the ‘getTotalM’ function returns the sum of all link observations stored in the map.

The tests for the publicly accessible functions in this class are performed using a unit test. The ‘addEntry’ and ‘getEntry functions are tested by simply inputting a random link an unspecified number of times and asserting the returned observation count of the link is correct. A similar test method is used for the ‘getTotalM’ function for a collection of links.

6.2.4 GoalToLinkMap

Below is a list of test specifications for all data I/O from the goal-to-link map class.

Table 34: GoalToLInkMap Test Specification

Access Function	Variable Name	I/O	Specification
linkTraversed	link	I	Current link
	goal	I	Current goal
	count	O	Number of times link is traversed to goal
probabilityOfGoalGivenLink	link	I	Inputted link
	goal	I	Predicted end goal
	prob	O	Probability of end goal given link

The ‘linkTraversed’ function operates similarly to the ‘incrementTransition’ function of the ‘LinkToStateMap’ class. Both attempt to associate two or more objects while keeping track of the number of associations; the exception being that the ‘GoalToLinkMap’ has two degrees of dimensionality in comparison to the three dimensions of the ‘LinkToStateMap’. The ‘linkTraversed’ function stores a goal hash at the first level and a link hash with the number of associations between the link and the goal at the second level. The ‘probabilityOfGoalGivenLink’ function then returns the probability of the goal given the argument link by dividing the number of goal-link observations by the total number of link observations in the hash map.

The class is also tested using unit tests. The ‘linkTraversed’ function is tested using a collection of links and goals to assert random associations of links and goals are incremented correctly within the class hash map using a pre-generated association map. The ‘probabilityOfGoalGivenLink’ function is tested similarly in that a collection of links and goals with pre-generated goal-to-link probabilities are asserted against the output of the function.

6.2.5 GoalMapEntry

Below is a list of test specifications for all data I/O from the goal map entry class.

Table 35: GoalMapEntry Test Specification

Access Function	Variable Name	I/O	Specification
addMapEntry	key	I	Look-up for value in map
	value	I	Value stored in map
getM	key	I	Look-up for value in map
	value	O	Value stored at look-up in map

The ‘GoalMapEntry’ class is capable of accepting different data types and data pointers, yet the two data types entered into the class routinely are ‘LinkToStateMapEntry’ pointers and long integers. As a result, the class is developed and tested to support only these two data types despite its ability to support others. The ‘addMapEntry’ function stores values to the one dimensional hash map using the inputted key and value and increments a count of all values. The ‘getM’ function returns the total count of values stored in the hash map.

The ‘addMapEntry’ class is tested using a collection of ‘LinkToStateMapEntries’ and random long integers representing link hashes. Values are added to the hash map through the ‘addMapEntry’ function. Functionality is tested via a unit test where a getter is used to access the hash map and the ‘getM’ function returns the value count to assert both are properly updated.

6.2.6 Goal

Below is a list of test specifications for all data I/O from the goal class.

Table 36: Goal Test Specifications

Access Function	Variable Name	I/O	Specification
isSimilar	Goal	I	Comparison goal
	isSim	O	Boolean of whether or not comparison goal is similar
isEqual	Goal	I	Comparison goal
	isEqual	O	Boolean of whether or not comparison goal is equal

The goal class represents an intersection and is used by the route prediction class as end points for predicted routes. The ‘isSimilar’ function is used to compare starting conditions of the class against the inputted goal. Similarly, the ‘isEqual’ function is used to compare the goal destination numbers as well as their similarity using the ‘isSimilar’ function.

Both functions are tested using a small collection of goals and simple asserts to ensure condition vectors and goals are considered similar or equal. This is a function of the way this small collection of goals is constructed.

6.2.7 Route

Below is a list of test specifications for all data I/O from the route class.

Table 37: Route Test Specifications

Access Function	Variable Name	I/O	Specification
addLink	link	I	Link to be added to the end of link collection
isEqual	route	I	Route to be compared against
	isEqual	O	Boolean of whether or not comparison goal is equal

The route class contains an ordered collection or links representing road segments a driver travels over from a start to an end destination. The ‘addLink’ function places the argument link at the end of the ordered collection of links stored by the route. The ‘isEqual’ function compares the order, size and individual links contained in the ordered list of links belonging to the argument link.

The ‘addLink’ function is tested by adding a random collection of links to two test routes, and then using the ‘isEqual’ function, the two test routes are compared to be equal. To ensure the ‘isEqual’ function works as intended, a third test route is created with a different collection of links and is compared to be different against one of the initial test routes.

6.2.8 Probability

Below is a list of test specifications for all data I/O from the probability class.

Table 38: Probability Test Specifications

Access Function	Variable Name	I/O	Specification
getProbabilty	prob	O	Simply numerator over the denominator

The probability class is very simple and used by the ‘GoalToLinkMap’ and ‘LinkToStateMap’ classes to return state and end goal probabilities. There are two fields to the class, ‘numerator’ and ‘denominator’, with setters that increment the fields. The only function, ‘getProbability’, simply returns the numerator divided by the denominator.

The test for the ‘getProbability’ function is also simple by recreating known ratios in the class and asserting returned probabilities that are equal to the decimal-value of the known ratios.

6.2.9 SpeedPrediction

Below is a list of test specifications for all data I/O from the speed prediction class.

Table 39: SpeedPrediction Test Specifications

Access Function	Variable Name	I/O	Specification
predict	in	I	Input row vector of historical speed
	out	O	Output row vector of predicted speed
train	in	I	Input row vector of historical speed
	act	I	Actual row vector of vehicle speeds
	out	I	Output row vector of predicted speed

The speed prediction class consists of a multi-input, multi-output neural network to perform rolling window time-series predictions of vehicle speed. All I/O to the following functions represent times series data. The ‘predict’ function takes a row vector of historical vehicle speeds, scales the values and forward propagates layer outputs to produce a time-series output row vector of predicted vehicle speed, where layer outputs are a function of the previous layer sigmoid activations. The ‘train’ function accepts the same two row vectors as well as another row vector representing the actual speed the vehicle traveled over the prediction distance. In a step called back propagation, error deltas from the last layer are propagated backwards using gradient descent to update weight matrices for each layer. The tests for speed prediction are deemed successful visually by comparing the actual and predicted vehicle speed traces over a randomly generated speed trace. The absolute error of the predicted and actual speed traces is also calculated where acceptable error is less than 5 m/s.

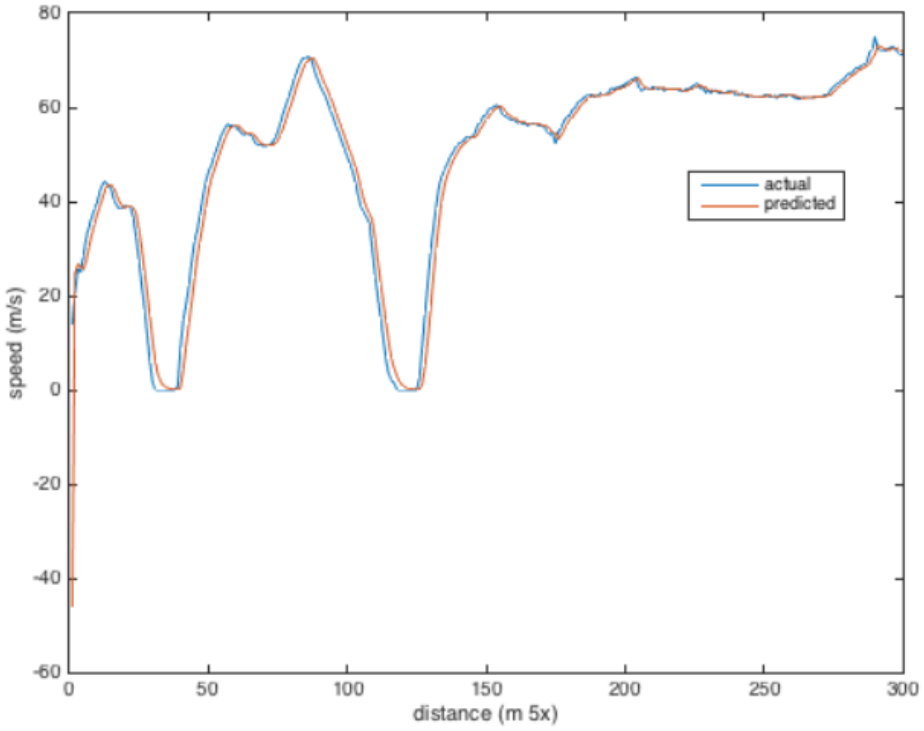

Figure 68:SpeedPrediction Unit Test Output

Using a loop within a unit test, the actual random speed trace is iterated over where historical speed data is inputted and predicted data is compared against actual speed data in a rolling window fashion. In every iteration, the 'predict' and 'train' functions are called. All the while, predicted and actual speed is calculated and the absolute error of each predicted speed is calculated. A successful predicted speed has similar, if not equivalent, profile to the actual speed.

6.2.10 GenericMap

Below is a list of test specifications for all data I/O from the generic map class.

Table 40: GenericMap Test Specifications

Access Function	Variable Name	I/O	Specification
nextEntry	entry	O	Next entry from current position of iterator
getMinEntry	entry	O	Entry with smallest look-up index
hasEntry	key	I	Look-up index for desired value
	hasEntry	O	Boolean of whether or not the entry exists
getEntry	key	I	Look-up index for desired value
	value	O	Value stored at look-up index
getFirstEntry	value	O	Value stored at the beginning of map
getLastEntry	value	O	Value stored at the end of map
addEntry	key	I	Look-up index to store value at
	value	I	Value to be stored at look-up index

updateEntry	key	I	Look-up index of value to be altered
	value	I	New value to be added at look-up index
indexErase	key	I	Ordered value to be erased at look-up index
erase	key	I	Value to be erased at look-up index

The generic map class is used to store ordered and unordered collections of data irrespective of data type and is used as the main organization data structure throughout the entirety of the code base. The ‘nextEntry’ function is used to iterate over the values stored in the class by returning the next value at the current iterator position and then incrementing the iterator. If the iterator is at one value before the end, a flag is set to signal the iterator being at the end of the map. If no next value exists, nothing is returned. The ‘getMinEntry’ function returns the values stored at the smallest look-up index, where the ‘hasEntry’ function returns the Boolean existence of the value at the specified look-up index. Similarly, the ‘getEntry’ function returns the value stored at the specified look-up index and returns nothing if nothing exists. The ‘getFirstEntry’ and ‘getLastEntry’ functions return the first and last values in the map. Values are added to the map using ‘addEntry’ function by place inputted values at the specified look-up index. Similarly, the ‘updateEntry’ function replaces the current value stored at the specified look-up index with the argument index. Finally, ‘indexErase’ and erase remove values at the specified look-up index. The only difference is that the ‘indexErase’ function updates the look-up indices to maintain continuity.

Tests for the publicly accessible functions of the generic map class are performed using a unit test and simple assertions. In all test cases, the generic map is populated with random key-value pairs. The ‘nextEntry’ function is tested by asserting the number of values returned is equal to the number of key-value pairs initially created and that all key-values pairs are accounted for in the return values signaling successful iteration from the beginning to the end of stored values. The ‘getMinEntry’ function is tested by asserting the key return is the smallest of the initial key-value pairs. Output form the ‘hasEntry’ and ‘getEntry’ functions which are asserted against the initial set of key-value pairs. For example, both functions must return false or nothing if an argument key does not exist in the initial key-value set; and both must return true if the argument key does exist. The ‘getFirstEntry’ and ‘getLastEntry’ are asserted to return the minimum and maximum key-value pairs of the initial set. The ‘updateEntry’ function is tested by passing a new value and asserting the returned value of the ‘getEntry’ function matches the initially passed value. Similarly, the ‘erase’ function is tested by passing a populated key and then asserting the output of the ‘hasEntry’ function with the same key is false. Finally, the ‘indexErase’ function must pass the same test case as the ‘erase’ function. However, the generic map is then iterated over, asserting that each key value returned from the ‘nextEntry’ function has a difference of one from the previous key. By default, the ‘addEntry’ function is deemed operational by the successful completion of all aforementioned test cases.

6.2.11 GenericEntry

The generic entry class does not have publicly accessible functions to set or update class fields. As a result, the class fields are made public. This class is used as an intermediary container to return key-value pairs from the generic map class. Because of the class’s simplicity, test specifications and test cases are not included. Successful operation of the generic map test cases indicates successful operation of the generic entry.

6.2.12 DriverPrediction

Below is a list of test specifications for all data I/O from the Driver Prediction class.

Table 41: DriverPrediction Test Specifications

Access Function	Variable Name	I/O	Specification
startPrediction	currentLink	I	Current link the vehicle is on
	distance	I	Distance along the link the vehicle has traveled
	speed	I	Current speed of the vehicle
	predData	O	Predicted speed and elevation values along the predicted route
nextPrediction	currentLink	I	Current link the vehicle is on
	distance	I	Distance along the link the vehicle has traveled
	speed	I	Current speed of the vehicle
	predData	O	Predicted speed and elevation values along the predicted route
parseRoute	route	I	Route the vehicle has traveled
	spd	I	Vectors of recorded speeds
	trace	I	Ordered collection of recorded GPS measuremnts

The Driver Prediction class is used to encompass the route and speed prediction classes. This class provides a single interface to output predicted speed over the predicted route as well as the elevation over the predicted route. The ‘startPrediction’ function inputs the current link representing the current road segment the vehicle is traveling, the distance along the link the vehicle has traveled and the current speed at which the vehicle is traveling. These values are used to initialize the route prediction class belonging to the Driver Prediction class. Then using the predicted route, the ‘startPrediction’ function pulls the predicted speed and elevation over the predicted route using the ‘routeToData’ function offered by the city class. The predicted speed and elevation are returned to be ingested by the kinematics class. The ‘nextPrediction’ class operates similarly. However, this function checks to ensure the current link is the same as the link passed to the class in the previous call for prediction data. If the argument link is the same, the function does not update the predicted route and returns predicted speed and elevation. If the argument link is not the same, route prediction is performed using the new link, the weights for the previous link are updated by training speed prediction with recorded speed values, and the predicted speed and elevation is returned over the new predicted route. Finally, ‘parseRoute’ ingests an entire trip-worth of prediction training data to train route and speed prediction link-by-link using the actual vehicle speed trace recorded over the trip.

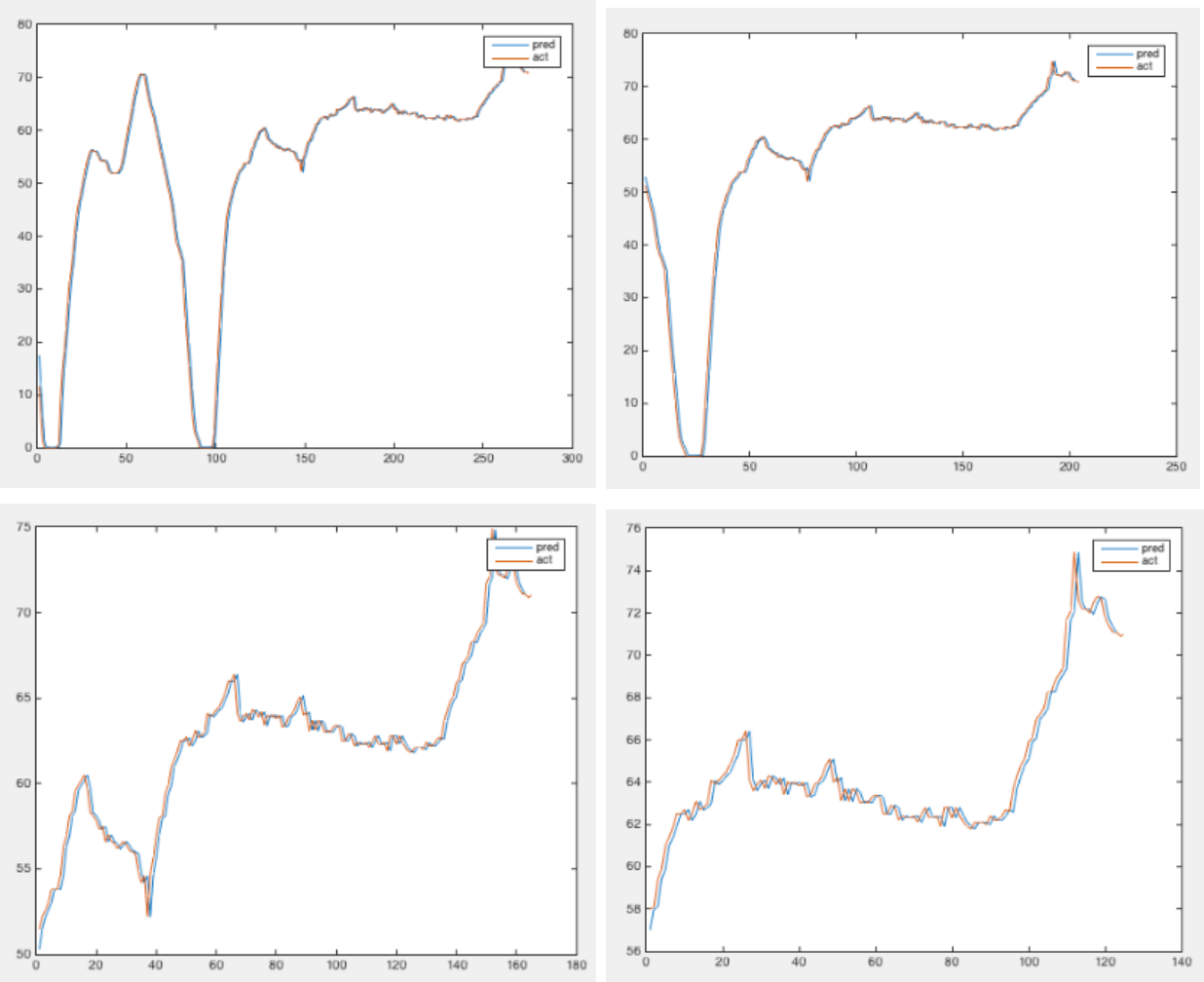

Figure 69: SIL DriverPrediction SpeedPrediction Output (m/s = y-axis | m = x-axis) as Vehicle Travels Predicted Route

The testing of this class serves a software-in-the-loop (SIL) test bench for the majority of the code base. It requires a road network and fully functional route and speed prediction, serialization, city and build-city classes. All three functions of the classes are tested using a large SIL unit test serving as a combination of the route and speed prediction test cases. First, a speed trace is comprised using concatenated EPA drive schedules [13]. Random paths through an existing road network are generated and inputted into the 'parseRoute' function along with the speed traces and a GPS trace representing the randomly generated path to train the route and speed prediction algorithms. Then one of the randomly generated routes is selected as the actual route. Using a large loop, the links of the actual route and associated speed values are fed into the 'startPrediction' and 'nextPrediction' functions with the appropriate travel distances in rolling window fashion. Functionality is assessed by calculating error of predicted and actual speed trace values and accuracy of the predicted route.

6.2.13 Link

Below is a list of test specifications for all data I/O from the link class.

Table 42: Link Test Specifications

Access Function	Variable Name	I/O	Specification
getHash	hash	O	Hash of the direction and identifying number
isEqual	link	I	Comparison link
	isEqual	O	Boolean of whether comparison link is equal
linkFromRoad	road	I	Road segment link represents
	intersection	I	Intersection at end of road to determine link direction
	newLink	O	Returned link as a function of arguments
isFinalLink	isFinal	O	Boolean of whether link has -1 number and direction

The link class serves as the connection between the route and speed prediction classes and ultimately represents a road segment. The 'hash' function simply returns the identifying number of the link plus twice the direction number. The 'isEqual' function returns the Boolean equivalence of the input link direction and identifying numbers. The 'linkFormRoad' function accepts a road and intersection. The resulting link has the same identifying number as the road and has a direction number equation to whether or not the argument intersection is the start intersection of the argument road. The 'isFinalLink' function compares the link identifying number and direction to -1. The 'finalLink' is used at the end of routes to signal route prediction when the predicted route has reached the last road segment.

The publicly accessible classes are assessed using simple asserts in a unit test with a small collection of test links. The 'getHash' function is tested against a pre-computed hash of a test link. The 'isEqual' function is tested by assessing output using two equal and unequal links. The 'linkFromRoad' function is tested by asserting the direction of the resulting link matches the Boolean equivalence of the argument intersection and the start intersection of the argument road. Finally, the 'isFinalLink' function is asserted to return true when a link is constructed with direction and identifying numbers set to -1. It is false otherwise.

6.2.14 Node, Bound, Way

These classes are similar to the generic entry class in that all fields are publicly accessible and do not offer publicly accessible classes other than getters and setters. As a result, test specifications and cases are not provided.

6.2.15 City

Below is a list of test specifications for all data I/O from the link class.

Table 43: City Test Specifications

Access Function	Variable Name	I/O	Specification
getNextLinks	link	I	Link used to find connecting links at end
	adjacentLinks	O	Connecting links at end of argument link
getIntersectionFromLink	link	I	Link used to find intersection
	intersection	O	Intersection pointed to by the link
getRandomPath	intersection	I	Starting intersection of the random route

	initialRoute	I	Partial route to be concatenated
	length	I	Length of route outputted by function
	randRoute	O	Resulting output route
getPath	startInt	I	Starting intersection of resulting route
	endInt	I	End intersection of resulting route
	shortRoute	O	Resulting route
routeToData	route	I	Route to be iterated over to produce output
	predData	O	Predicted speed and elevation

The city class offers a tool box of functions to interface with a road network. The 'getNextLinks' function accepts a link, finds the associated road, and using the argument link direction, accesses all connecting roads from the appropriate road intersection. The returned value is a collection of links generated from each connecting road. The 'getIntersectionFromLink' function accepts a link, finds the corresponding road, and using the argument link direction, returns the appropriate road intersection. The 'getRandomPath' function is used primarily for training route prediction. It accepts a start intersection, a partial route, and a specified length of the resulting output route. The output is generated iteratively by adding a random link that has not already been added to the end of the input route using the 'getNextLinks' function. When the random route has reached the specified length, it is returned. The 'getPath' function is similar, but relies on Dijkstra's algorithm to find the shortest pathway between the two argument intersections. The most important function of the class is 'routeToData'. It accepts a route the vehicle is predicted to travel and, using the weights associated with each link, returns two vectors: one vector of predicted speed and a second vector containing concatenated elevation traces of the road segments.

The city class is tested on a function-by-function basis. The 'getNextLinks' function is tested by generating a test link from a random road and it's start intersection. Output of the function, a collection of links, is tested against the connecting roads of the start intersection. It is associated to the random road by comparing identify numbers of the returned links and connecting roads. Similarly, the 'getIntersectionFromLink' function is tested by generating a test link from a random road and the road's start intersection. Output of the function, an intersection, is tested against the random road's start intersection. The 'getRandomPath' and 'getPath' function outputs are verified by asserting all links in the returned routes are adjacent. This is accomplished by iterating along the links of the returned route and using the 'getNextLinks'. Additionally, the 'getRandomPath' output is verified to start at the specified start intersection and to contain the input route using iterative link-by-link comparisons against the input route and the 'isEqual' function offered by the link class. Conversely, the output of the 'getPath' function is asserted to contain the specified start and end intersections at the start and end of the returned route. The 'routeToData' function is tested using step-by-step analysis in the debugger through the SIL test described in the Driver Prediction test case.

6.2.16 BuildCity

Below is a list of test specifications for all data I/O from the build-city class.

Table 44: BuildCity Test Specification

Access Function	Variable Name	I/O	Specification
updateGridDataXMLSpline	city	O	The updated city

The build-city function adds road network data whenever the vehicle travels out of the known boundaries of locally stored road network data. The 'updateGridDataXMLSpline' function performs the entire update process by identifying the exceeded bounds of the most recent trip data, querying new raw road data using the data collection class, and then parsing the raw data to create atomic road segments to be added to the existing road network.

A unit test is created for the build-city class. However, the functionality of the class is verified visually by comparing the road coverage of the atomic road segments superimposed on a map. Additionally, the connecting roads associated with each intersection and the start and end intersections of each road segment are hand-verified to be true upon visual inspection of the superimposed map. Finally, elevation traces associated with each road segment are spot checked against known hills in the map.

Figure 70: Intersection Properties

The above image depicts a pop-up dialogue for an intersection describing the connecting road identifying numbers, elevation, lat/lon and intersection identifying number. Proper formation of intersections should have all connecting road identifying numbers in the pop-up dialogue for all adjacent roads to the intersection.

Figure 71: Road Properties

The above figure depicts the pop-up dialogue of a road with the start and end intersection identifying numbers, the spline control point (node) elevation, lat/lon, and the road identifying number. Proper formation of a road correctly lists the start and end intersection identifying numbers and the start and end of the road. Additionally, points on the map where hills are known to be should be accurately reflected in the road elevations trace.

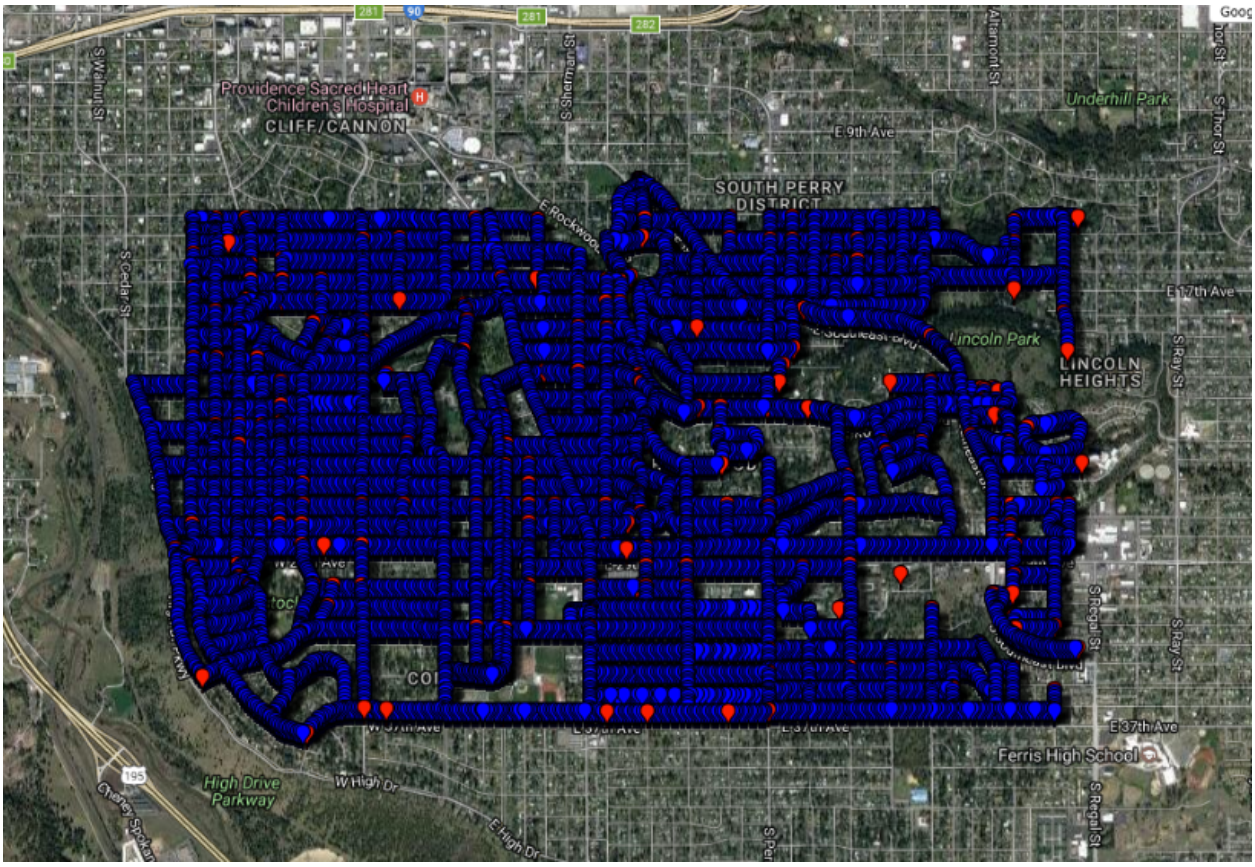

Figure 72: Road Coverage of BuildCity Over South Hill of Spokane, WA

The above image depicts road network coverage over the South Hill of Spokane, WA. Successful generation of a new road network has approximately 98% coverage of real road data represented in the road network. It should also have at least 95% accuracy in identification of road start and end intersections and of intersection connecting roads.

6.2.17 Road

Below is a list of test specifications for all data I/O from the road class.

Table 45: Road Test Specifications

Access Function	Variable Name	I/O	Specification
getElevData	predData	O	Elevation and elevation distance measurements

The road class holds all control points to define the road curvature profile and elevation trace. It also contains a start and end intersection. The ‘getElevData’ returns the elevation data along the control points between the start and end intersections. For every elevation measurement, there is also a distance component between the measurement and the start intersection along the curvature profile of the road.

The ‘getElevData’ function is tested by ensuring the returned elevation data represents the entire length of the road and that all measurements are approximately 5 meters apart.

6.2.18 Intersection

Below is a list of test specifications for all data I/O from the intersection class.

Table 46: Intersection Test Specification

Access Function	Variable Name	I/O	Specification
addRoad	road	I	New connecting road to the intesection
getOutgoingLinks	link	I	Adjacent link to intersection
	connectingLinks	O	Connecting roads of intersection represented as links
getAdjacentIntersections	intersections	O	All other intersections of all connecting roads
getIntersection	road	I	Connecting road of intersection
	intersection	O	Other intersection of argument road

The intersection class represented a junction of atomic road segments. The ‘addRoad’ function accepts a road and adds it to the collection of connecting roads for the intersection. The ‘getOutgoingLinks’ function iterates through all connecting roads and makes a list of links using the identify numbers of each connecting road and the orientation of the start intersection for the connecting road with respect to the intersection. The list is then returned. Similarly, the ‘getAdjacentIntersections’ function iterates through all connecting roads making a list of other intersections. The list is then returned. Finally, the ‘getIntersection’ function accepts a road to be queried against the list of connecting roads. It the argument road exists in the list of connecting roads, then the other intersection is returned. Otherwise, nothing is returned.

All tests of the intersection class are performed with a collection of pre-generated roads with one intersection equal to the test intersection. The ‘addRoad’ function adds a road to the test intersection and then utilizes the getter function of the connecting roads to assert the road has been added correctly. The ‘getOutgoingLinks’ function is tested by making a test collection of links from the pre-generated collection of roads, adding all roads to the test intersection with the ‘addRoad’ function, and asserting the test function has an output equivalent to the test collection of links. The ‘getAdjacentIntersections’ function is tested by asserting the list of return intersections accounts for all road intersections not equal to the test intersection. Finally, the ‘getIntersection’ function is tested by asserting the returned intersection is equal to the argument road intersection that is not equal to the test intersection.

6.2.19 Kinematics

Below is a list of test specifications for all data I/O from the kinematics class.

Table 47: Kinematics Test Specification

Access Function	Variable Name	I/O	Specification
runKinematics	speed	I	Input vector of predicted speed
	elevation	I	Input vector of elevation over the

			predicted route
	tractiveEnergy	O	Tractive force of each input integrated over the length of the input

The ‘runKinematics’ function is used to input predicted speed and elevation over the predicted route into the road-load equation such that times-series output can integrated over the look-ahead distance of the vehicle to produce the total tractive energy the vehicle energy will use or gain in the look-ahead distance.

The function is tested using varying combinations of positively and negatively sloped speed and elevation traces. For each output of the ‘runKinematics’ function, the sign of the tractive energy sign is asserted to be positive or negative as a function of the sloped speed and elevation combination inputted.

6.2.20 VehicleDiagnostics

Below is a list of test specifications for all data I/O from the vehicle diagnostics class.

Table 48: VehicleDiagnostics Test Specification

Access Function	Variable Name	I/O	Specification
getSpeed	speed	O	Speed in m/s
getFuelFlow	fuelFlow	O	Mass fuel flow in g/s
getEngineLoad	load	O	Engine load as a percentage
getDiagnostics (private)	cmd	I	Command hex associated with specific command
	time	I	Amount of time allowed between read and write
readDiagnostics (private)	msg	O	String message read from the Rx serial buffer

The ‘getSpeed’, ‘getFuelFlow’ and ‘getEngineLoad’ functions all return the diagnostic values for their named functions and utilize the ‘getDiagnostics’ function. The ‘getDiagnostics’ function takes command from the public diagnostics function and outputs the command onto a serial-bus connected to the OBDII interpreter. After waiting the specified number of microseconds, the raw response value is returned by the ‘readDiagnostics’ function, the value is converted to a float data type and returned. Lastly, ‘theReadDiagnostics’ function performs a serial read on the Rx buffer of the serial bus and returns the raw value.

The chain of command is tested by calling the public diagnostic functions in unit test during a test drive, where the response values are returned and reasoned to be valid or not.

6.2.21 GPS

Below is a list of test specifications for all data I/O from the vehicle diagnostics class.

Table 49: VehicleDiagnostics Test Specification

Access Function	Variable Name	I/O	Specification
getHeadingAngle	angle	O	Arc-tangent of differential GPS
isHeadingStart2EndoOfCurrentRoad	road	I	Current road
	isStart2End	O	Boolean of whether current heading point from start to end evaluation of road spline
getDistanceAlongRoad	road	I	Current road
	dist	O	Distance along current road in meters
isOnRoad	road	I	Variable argument road
	isOnRoad	O	Boolean of whether argument road is closest to current GPS location
deltaLatLon2XY	lat/lon	I	First GPS measurement
	lat/lon	I	Second GPS measurement
	dist		Haversine distance between measurements

The GPS class is another toolbox of functions used to compare road network data against GPS measurements and to store a log of all GPS measurements for a given trip. The 'getHeadingAngle' function is used to compute the arc-tangent of differential GPS measurements taken while the vehicle is in motion. The return value is the heading angle in radians. The 'isHeadingStart2EndOfCurrentRoad' function accepts a road, computes the current distance along the argument road, computes the heading, and then determines if the heading angle is in the same or opposite directions of the nearest road evaluation points of the current GPS location. The 'getDistancAlongRoad' function is used to compute the distance between the current GPS location and the start intersection along the curvature profile of the road. The 'isOnRoad' function is used to assess whether or not the argument road is the nearest road to the current GPS location. Finally, the 'deltaLatLon2XY' function takes two GPS measurements to compute the distance between the two measurements using the Haversine formula.

The GPS function is tested using an existing road network with the GPS module connected during a road test. The 'getHeading' function is tested by reporting the heading calculated to console and asserting the measurement differs by approximately 180 degrees upon maneuvering U-turns. The 'isHeadingStart2EndOFCurrentRoad' function is tested by also reporting the output to console and asserting the value also flips upon maneuvering U-turns. The 'getDistanceAlongRoad' class is tested by reporting the calculated value to console and asserting the value increases as the vehicle motions from one road intersection to the other. The 'isOnRoad' function is tested by driving across multiple roads and asserting the reported value to console changes to false upon leaving the current road. Lastly, the 'deltaLatLon2XY' function is tested by measuring the actual travel distance of the vehicle and comparing it against the reported meters-calculation.

6.2.22 DataCollection

Below is a list of test specifications for all data I/O from the data collection class.

Table 50: DataCollection Test Specification

Access Function	Variable Name	I/O	Specification
pullDataXML	lat/lon	I	Center of query region
updateElevationData	road	I	Road to be updated
	road	O	Updated road
makeRawRoads	rawRoads	O	Non-atomic road segments

The data collection class is used to query all outside information needed for updating the road network. The ‘pullDataXML’ function creates query regions about the input parameters using the lat/lon deltas passed to the class upon construction and places queries with the ‘queryFile’ function. Data is read using the Boost json parser library and stored in the node, bounds, and way intermediary classes using generic map containers. The ‘updateElevationData’ function accepts a road and, like the ‘pullDataXML’ function, creates a query for elevation for each spline control point in the argument road. It also queries data using the ‘queryFile’ function. Data is read using the Boost json parser library to update the existing road spline control points. Finally, the ‘makeRawRoads’ function takes all intermediary classes of non-atomic road data, and creates a collection of non-atomic roads.

The ‘pullDataXML’ function is tested by visually inspecting the raw roads coverage after querying for road network data of, in this case, the Greenlake area in Seattle, WA. The data is viewed after envoking the ‘makeRawRoads’ function where the intermediary data classes are used to create non-atomic roads and road data is written to csv to be viewed by the GPSVisualizer. Lastly the ‘updateElevationData’ is spot checked by comparing road sections with known hills to stored elevation traces generated by the function.

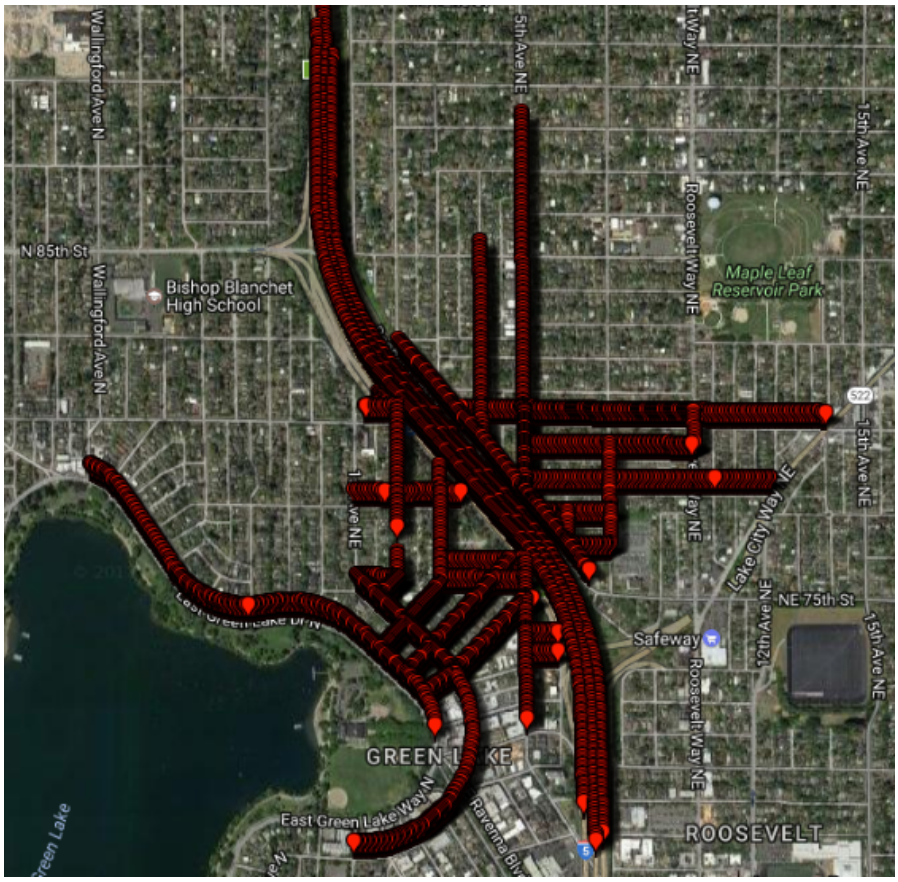

Figure 73: Raw Road Coverage of Data Collection

The above picture shows raw road coverage. Successful querying of road data is considered as having approximately 98% of real roads represented by road network data.

6.2.23 DataManagement

Below is a list of test specifications for all data I/O from the data management class.

Table 51: DataManagement Test Specification

Access Function	Variable Name	I/O	Specification
addCity	city	I	City to be serialized
addRoutePrediction	routePrediction	I	RoutePrediction to be serialized
addTripData	trip	I	Trip log to be serialized
getCity	city	O	City generated by serial data
getRoutePrediction	routePrediction	O	RoutePrediction generated by serial data
getMostRecentTripData	trip	O	Tip log generated by last serialized data

The data management class is vital to storing all Driver Prediction data, so that the embedded device or pocket PC used to run the algorithm may power cycle without the loss of learned driver data. The ‘addCity’, ‘addRoutePrediction’, and ‘addTripData’ functions are used to serialize their respective arguments using JSON or binary output such that the arguments can be recreated

on reboot. Conversely, the ‘getCity’, ‘getRoutePrediction’, and ‘getMostRecentTripData’ functions are used to recreate their output data types from serialized data. Both serialization and de-serialization are to take place within one second of start-up and shut-down.

Data serialization and de-serialization for the ‘addCity’ and ‘getCity’ classes are tested by visually inspecting serialization files for all vital values.

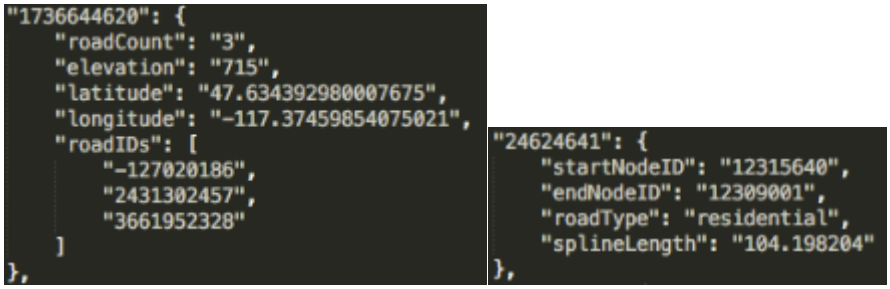

Figure 74: Intersection and Road Data Structures Stored as JSON

The above figures do not include road curvature and elevation profile data that is stored as binary data. Successful inclusion of this data to save and recreate the city class is accomplished when execution of the ‘addCity’ and ‘getCity’ functions is done without error and where the all of the returned city roads and intersections have pertinent and accurate data for route prediction over the generated road network.

Similarly, the ‘getRoutePrediction’ and ‘addRoutePrediction’ functions are tested by visually inspecting serialization files for all vital values.

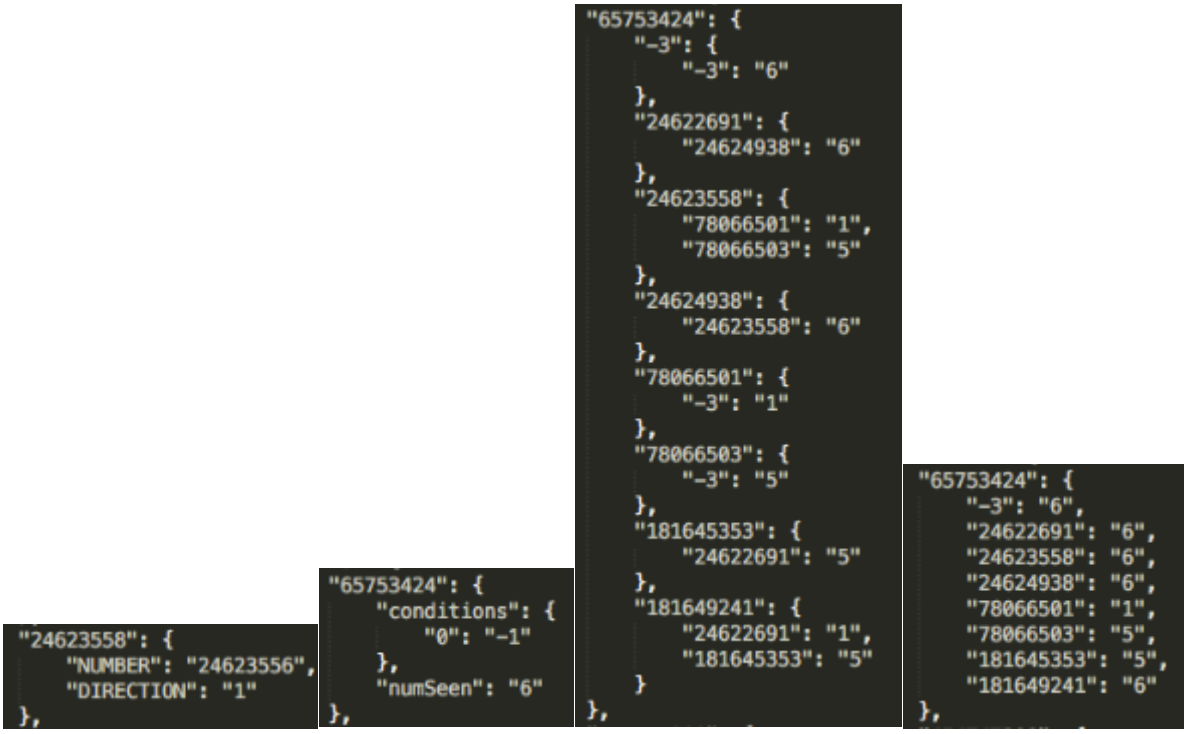

Figure 75: Link, Goal, LinkToStateMap Entry, and GoalToLinkMap Entry Serialization Example.

Like the road class, the link class does not include neural network matrices in JSON, but rather binary. Successful inclusion of this data to save and recreate the RoutePrediction class is accomplished when execution of the ‘getRoutePrediction’ and ‘addRoutePrediction’ functions is done without error and when the returned RoutePrediction links have all pertinent data and speed prediction over the links without fault.

6.3 System Test Plan

The system test procedure is performed while driving and assesses successful prediction of route and speed to calculate energy used by the vehicle over the predicted route. A randomly selected set of training routes within a known road network spanning 24x35 blocks are traversed a variable number of times to train the route and speed prediction algorithms. A random subset of the training routes is then used to assess the performance of route and speed prediction to calculate energy usage over the predicted route. With each traversal of a given test route, the route and speed predictions are expected to improve.

6.4 System Test Specifcations and Cases

Specifications of the software modules operating as a system are provided to define proper operation of the algorithm over any given route. The system must be able to learn a route, and successfully predict the route as well as provide a predicted speed over the predicted route from the current vehicle location to the predicted end destination the next instance the route is driven. The system must be able to distinguish between a road and an intersection to queue the prediction algorithms. It must also be able to determine correctly the road the vehicle is traversing and the intersection it may be heading towards. Data must be read from the vehicle diagnostics port and GPS receiver to understand speed, fuel flow and engine load and current location. Data must be stored to and read from disk to account for the shutdown and start-up of the algorithm on a periodic basis. As a result, shutdown and start-up must be performed quickly. New road data must be ingested and parsed and used to update the existing road network whenever the vehicle travels outside the bounds of the known road network. Energy must be calculated using the predicted speed and elevation time-series data over the predicted route. Vital data to assess the successful operation of the algorithm must be saved for every execution. Lastly, the system must be able to measure odometer values to assist in queuing machine learning algorithms.

Successful operation of the aforementioned test specifications is ensured by referencing the debugger during on-road testing to ensure every specification is met. Break points are strategically set at each critical software module and checked repeatedly until a confidence in each module’s successful operation within the system is achieved.

7. ANALYSIS OF RESULTS

Analysis of results is performed over a known road network spanning 24x35 block on the south hill of Spokane, Washington. The route map is provided below and shows where the test routes reside in the map. Training routes, a larger set of routes in which the test routes are a subset of, are not shown but are randomly dispersed across the road network to add randomness to route predictions and are traversed only once before testing over the test routes.

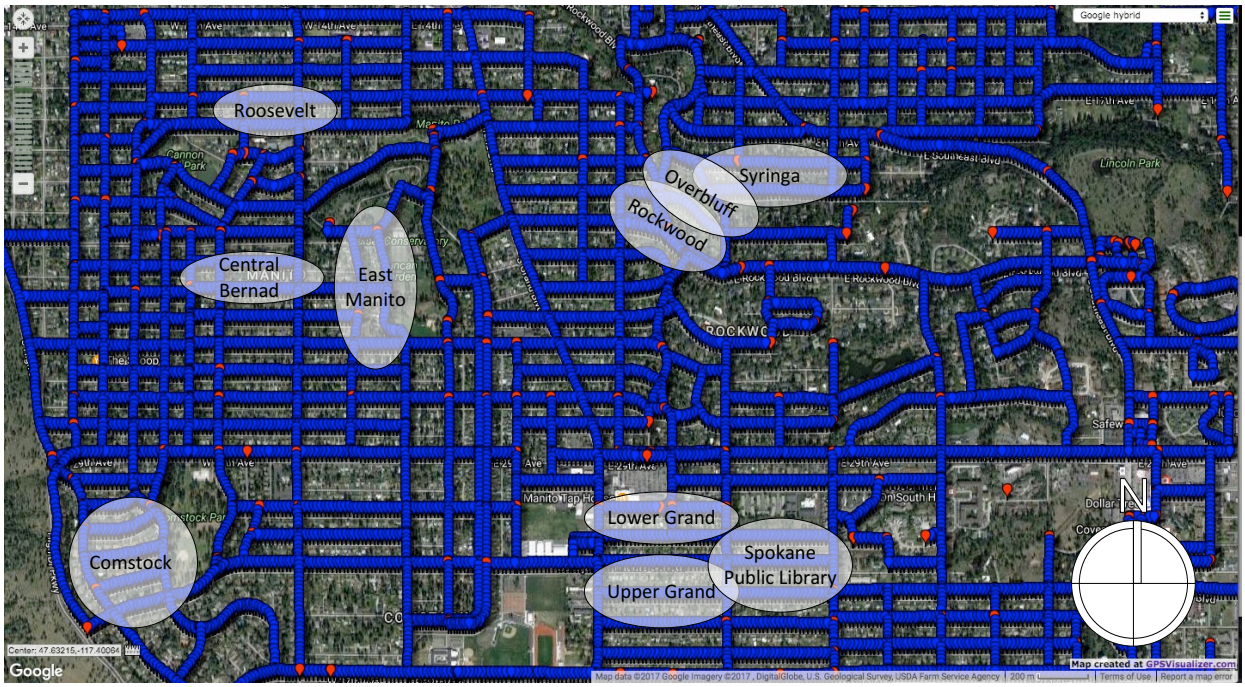

Figure 76: Test Route Map

The test routes are chosen at random locations in the map but are completely comprised of rural neighborhood streets with varying elevation profiles to test predicted energy as a function changing elevation as well as speed. Each test route is traversed a various number of times to assess speed prediction accuracy along the test routes as well as route prediction accuracy. Traffic, pedestrians and random shifts in speed along a given test route introduce randomness to speed prediction as well.

The following sections provide the accuracy of energy prediction as a function of route and speed prediction. Each section details results and error analysis of a test route and includes a map of the route and a data snap shot of the actual and predicted speed and energy time-series profiles as the elevation profile. Five tables are also included with averages calculated across all traversals of the given test route.

The ‘Accuracy of Cumulative Energy Calculations’ table compares the predicted and actual net energy consumed over the entire route as well as during charge-gaining events when energy consumed is predicted to be negative. Net and charge-gaining energy consumption over the predicted route is assessed to study their potential effective use in optimizing powertrain management algorithms for hybrid-electric vehicles. For example, net energy consumption can be used to maintain state-of-charge of a traction battery most efficiently by providing only the net predicted energy consumption to the battery before reaching the predicted end-destination where a grid charger may be located. This could improve upon fuel-efficiency of thermostatic and load-following control strategies for neither assess long-term driver intent and only operate within a relatively small time span of vehicle information. Charge-gaining consumption, on the other hand, can be used to optimize shorter look-ahead distances along the predicted route to assess more efficiently when to operate the gas portion of a hybrid powertrain.

The ‘Accuracy of Independent Predicted Energy Calculations’ table assesses the average and standard deviation of all individual time-series calculations for the predicted net and charge-gaining energy consumption. This information is used to assess the precision of the individual predicted energy values with respect to actual consumption.

The ‘Accuracy of Time-Series Speed Predictions’ table assesses the average and standard deviation of all individual time-series calculations for the predicted speed along the predicted route. This information is used to assess the precision of the individual predicted speed values with respect to actual values.

The ‘Route Travel Details’ table simply depicts the number of times the test route is tested over and the net distance traversed as a result. Most travel distances are the same for a given test route. However, there are cases where the start and ending locations along the start and end road segments differ.

Finally, the ‘Route Prediction Accuracy’ depicts the number of times the route predicted is not equal to the test route due to randomness introduced from the training routes. All energy and speed prediction results and error analysis, however, are based on route prediction predicting the test route. Only the times in which route prediction predicts the test route are used to increment the number of times the test route is traversed.

7.1 West Manito Route

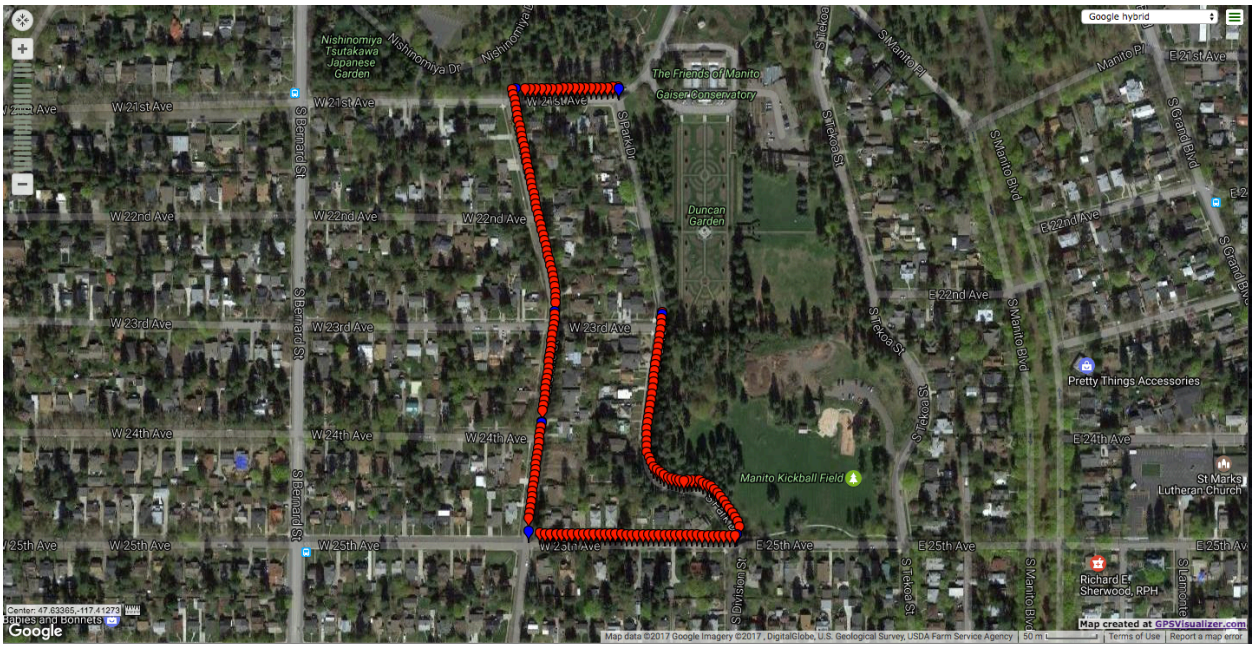

Figure 77: West Manito Route

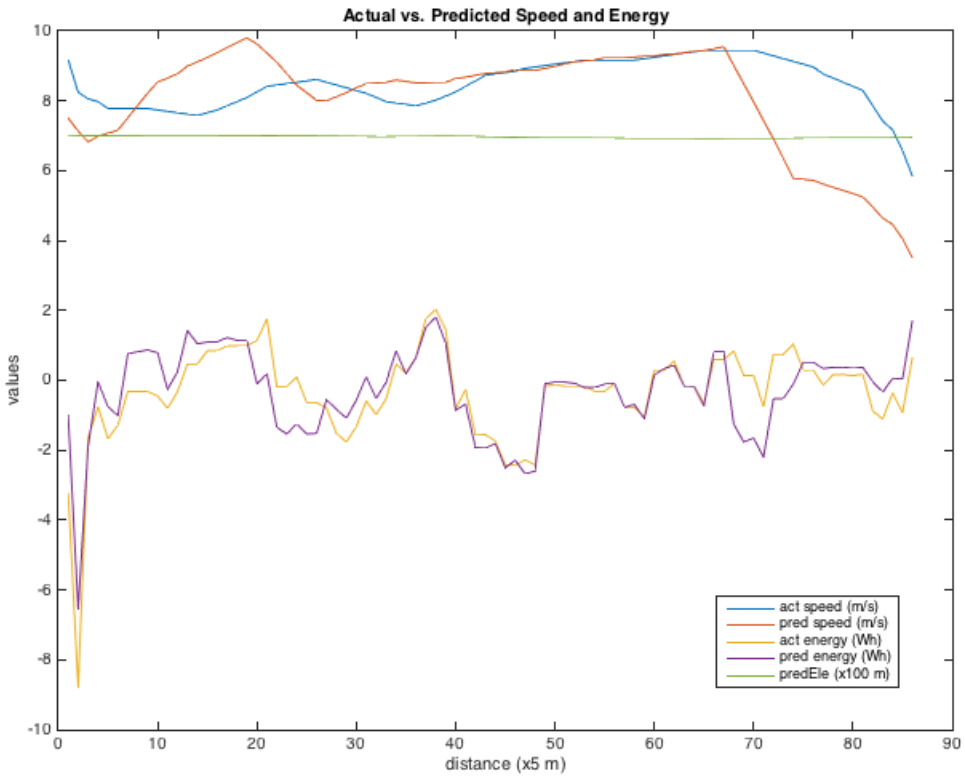

Figure 78: West Manito Route Data Snap Shot

Table 52: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-45.7	-32.8	28.3%
Average Net Energy Consumed (Wh)	-18. 4	-9.7	47.1%

Table 53: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.643
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.626
Absolute Average of Predicted Energy Delta (Wh)	0.579
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.600

Table 54: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.52
Absolute Standard Deviation of Predicted Speed Delta (m/s)	1.25

Table 55: Route Travel Details

Metrics	Value
Number of Times Route Traversed	8
Total Travel Distance Over Route (km)	13.57

Table 56: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	89%

7.2 Spokane Public Library Route

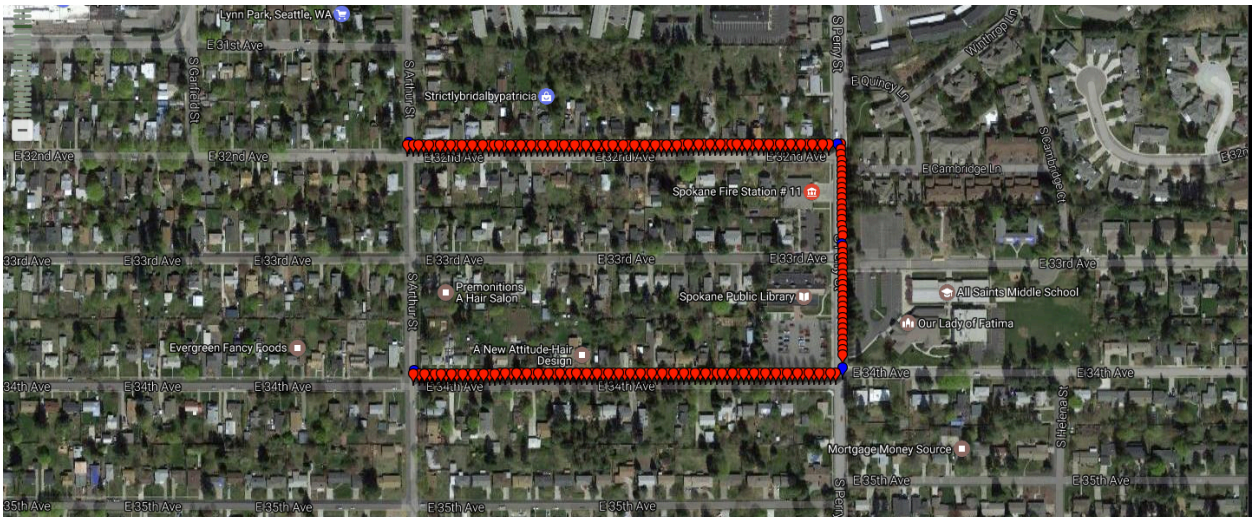

Figure 79: Spokane Public Library Route

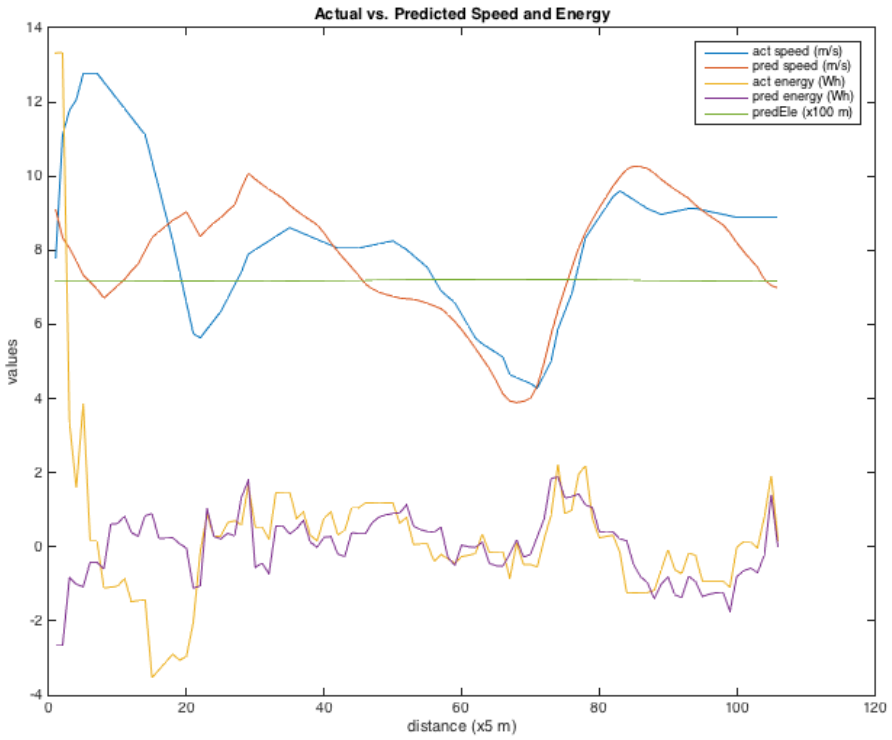

Figure 80: Spokane Public Library Route Data Snap Shot

Table 57: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-41.9	-25.3	39.6%
Average Net Energy Consumed (Wh)	6.71	18.2	63.1%

Table 58: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.767
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.845
Absolute Average of Predicted Energy Delta (Wh)	0.660
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.786

Table 59: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	2.18
Absolute Standard Deviation of Predicted Speed Delta (m/s)	1.57

Table 60: Route Travel Details

Metrics	Value
Number of Times Route Traversed	7
Total Travel Distance Over Route (km)	15.28

Table 61: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	89%

7.3 Rockwood Route

Figure 81: Rockwood Route

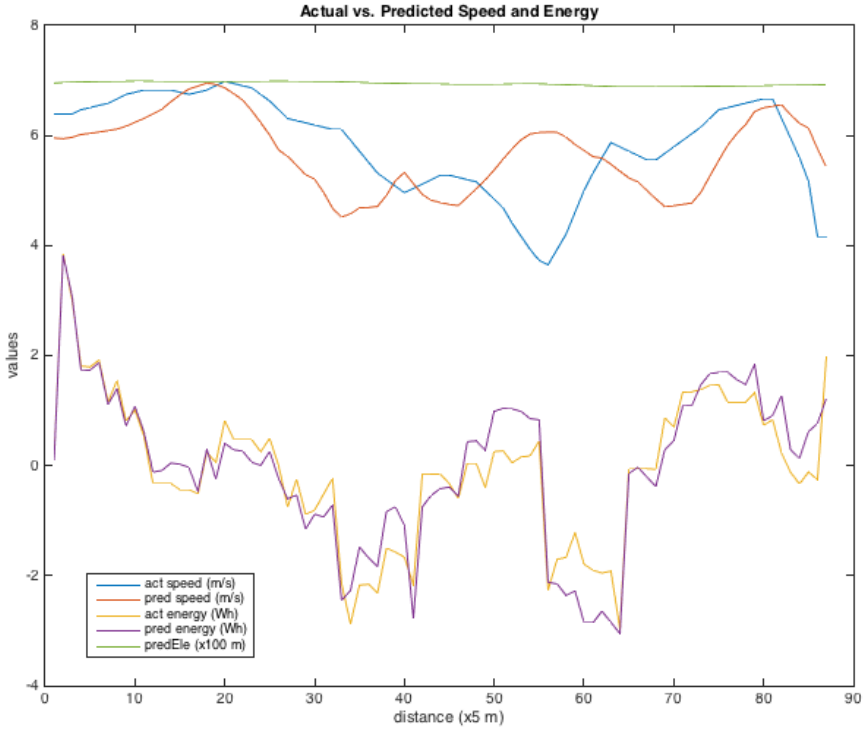

Figure 82: Rockwood Route Data Snap Shot

Table 62: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-51.8	-39.5	23.7%
Average Net Energy Consumed (Wh)	-9.92	-12.7	21.9%

Table 63: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.609
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.499
Absolute Average of Predicted Energy Delta (Wh)	0.542
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.466

Table 64: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.39
Absolute Standard Deviation of Predicted Speed Delta (m/s)	1.01

Table 65: Route Travel Details

Metrics	Value
Number of Times Route Traversed	14
Total Travel Distance Over Route (km)	24.10

Table 66: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	94%

7.4 Roosevelt Route

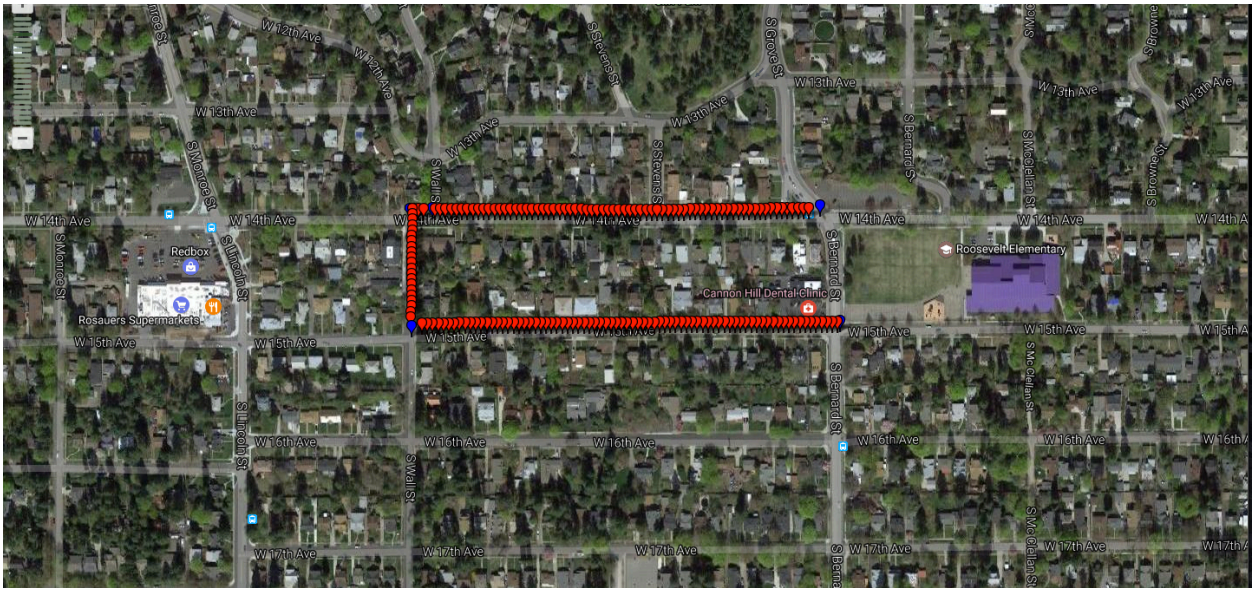

Figure 83: Roosevelt Route

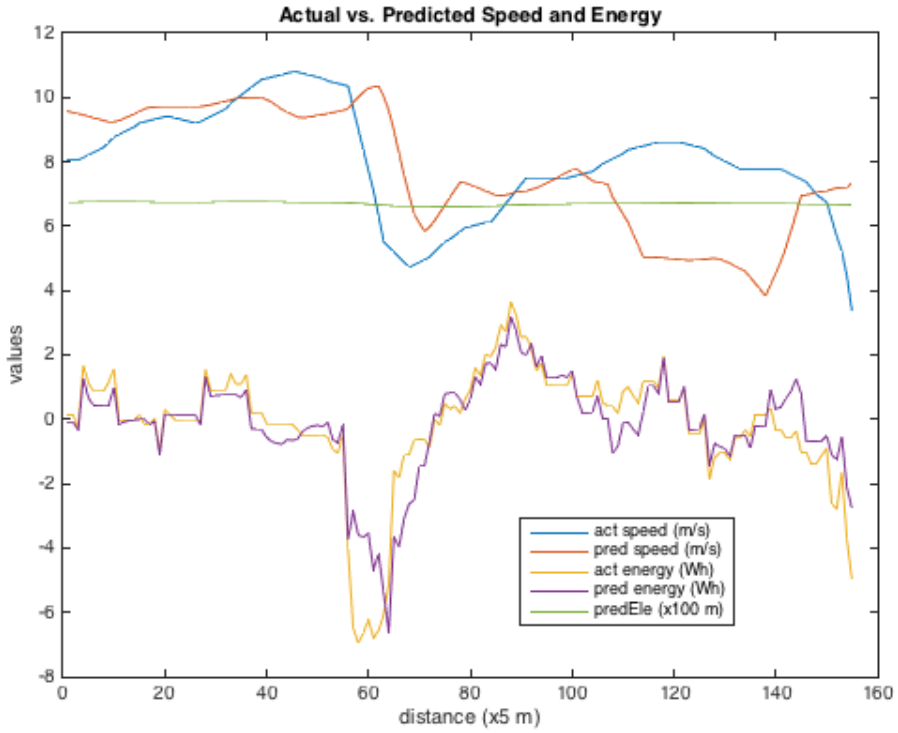

Figure 84: Roosevelt Route Data Snap Shot

Table 67: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-71.7	-76.0	5.7%
Average Net Energy Consumed (Wh)	-31.2	-30.9	1.0%

Table 68: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.598
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.703
Absolute Average of Predicted Energy Delta (Wh)	0.518
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.615

Table 69: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.57
Absolute Standard Deviation of Predicted Speed Delta (m/s)	1.06

Table 70: Route Travel Details

Metrics	Value
Number of Times Route Traversed	9
Total Travel Distance Over Route (km)	10.26

Table 71: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	91%

7.5 Overbluff Route

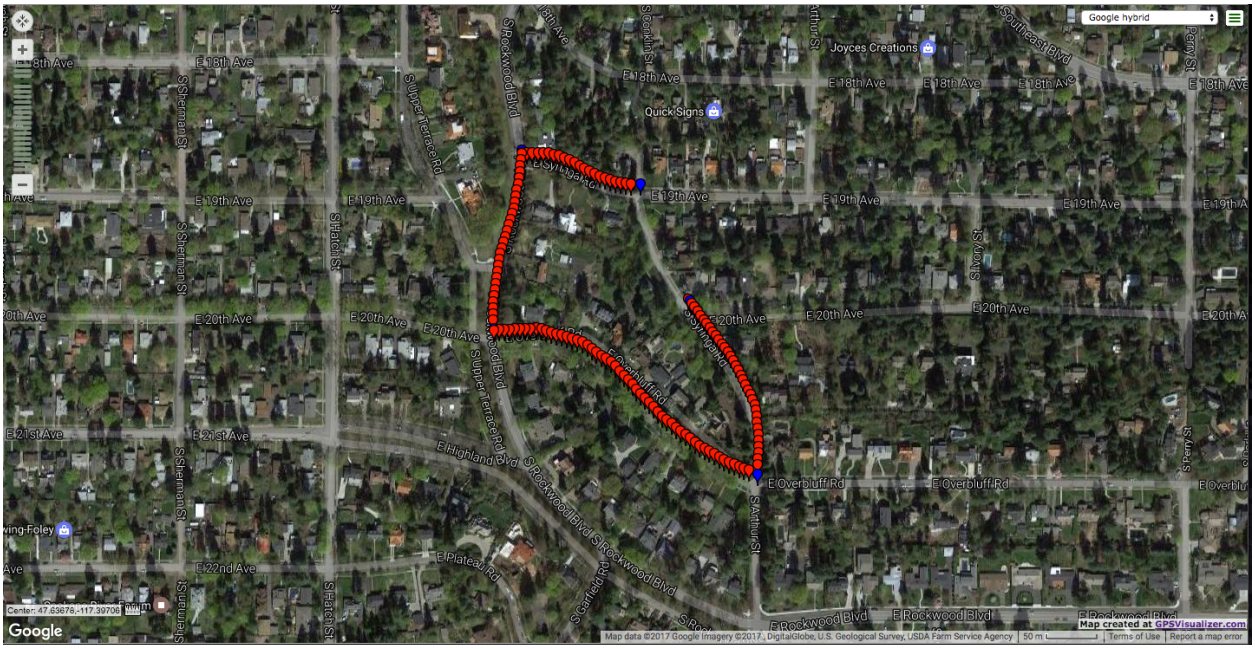

Figure 85: Overbluff Route

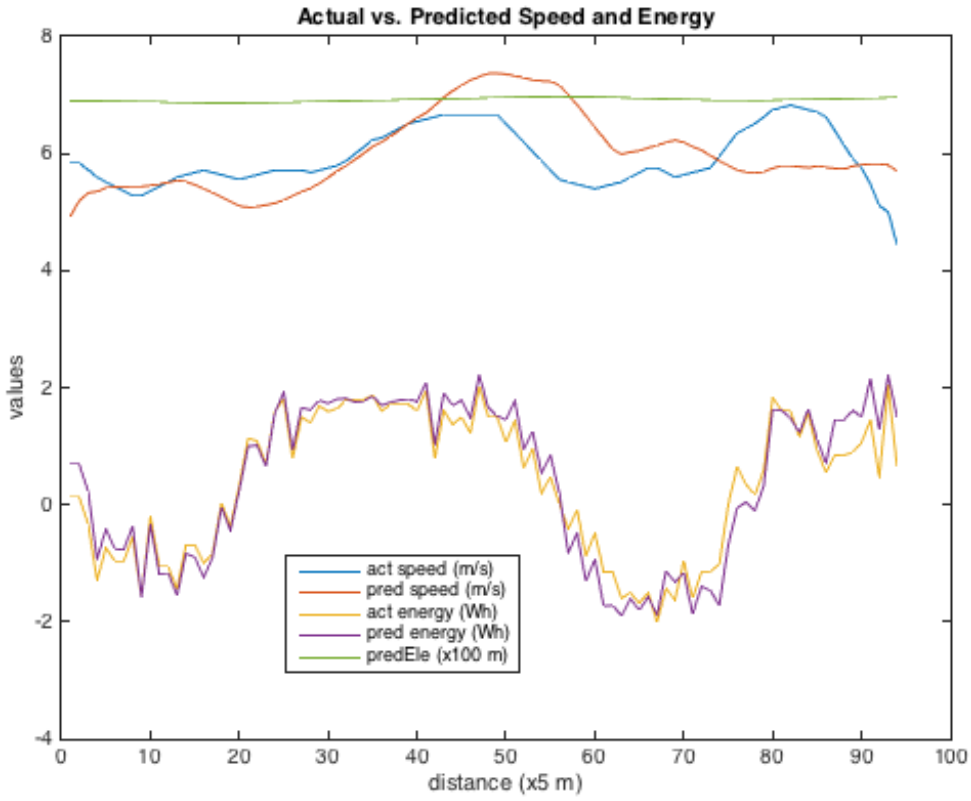

Figure 86: Overbluff Route Data Snap Shot

Table 72: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-48.84	-35.13	28.1%
Average Net Energy Consumed (Wh)	60.39	66.76	9.60%

Table 73: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.669
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.886
Absolute Average of Predicted Energy Delta (Wh)	0.519
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.693

Table 74: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.36
Absolute Standard Deviation of Predicted Speed Delta (m/s)	1.12

Table 75: Route Travel Details

Metrics	Value
Number of Times Route Traversed	19
Total Travel Distance Over Route (km)	29.29

Table 76: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	97%

7.6 Syringa Route

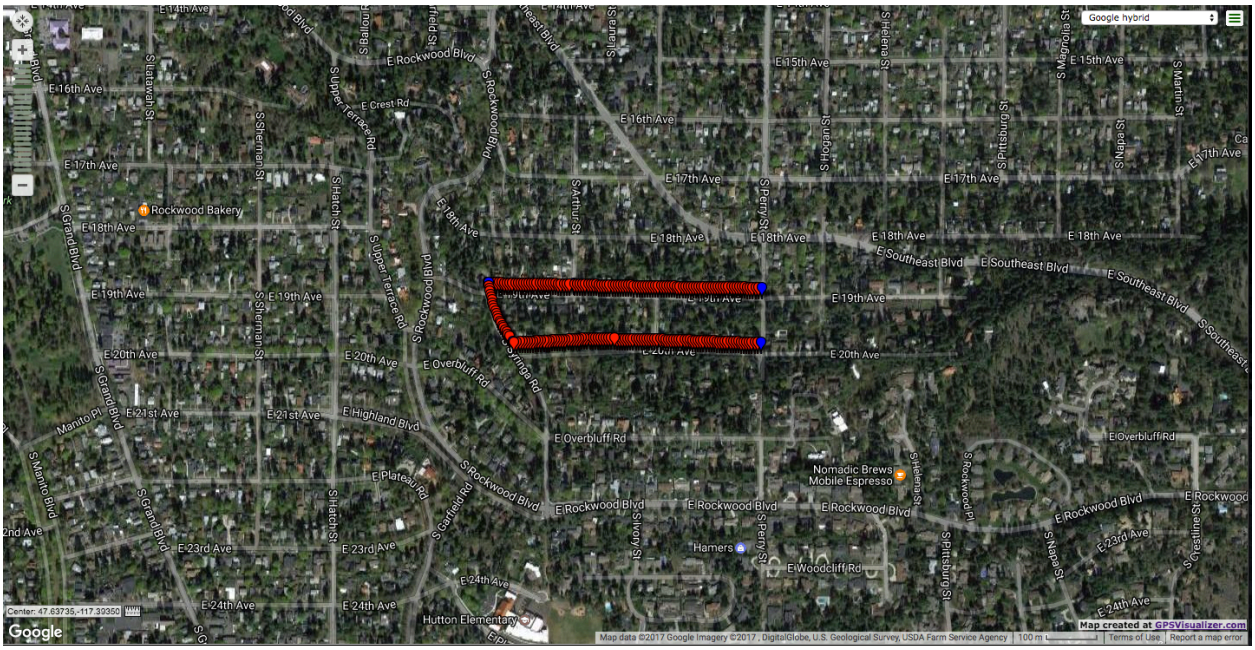

Figure 87: Syringa Route

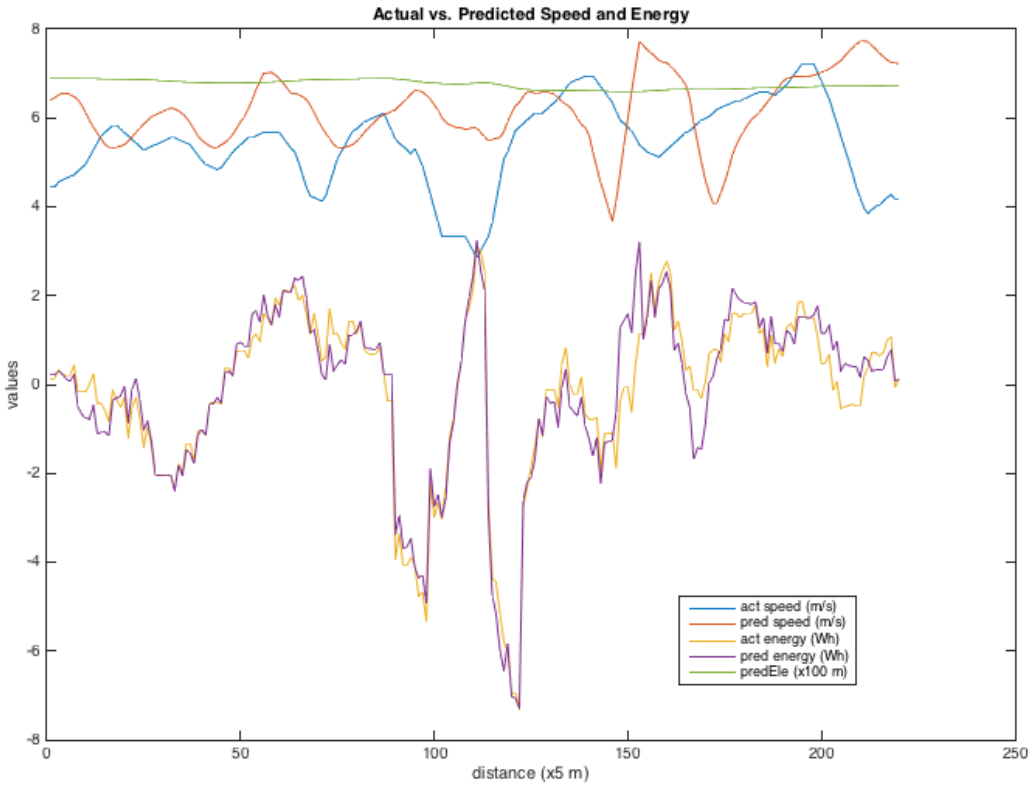

Figure 88: Syringa Route Data Snap Shot

Table 77: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-147.6	-140.4	4.88%
Average Net Energy Consumed (Wh)	-61.21	-57.94	5.39%

Table 78: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.355
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.305
Absolute Average of Predicted Energy Delta (Wh)	0.333
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.284

Table 79: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.55
Absolute Standard Deviation of Predicted Speed Delta (m/s)	0.98

Table 80: Route Travel Details

Metrics	Value
Number of Times Route Traversed	15
Total Travel Distance Over Route (km)	33.50

Table 81: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	95%

7.7 Central Bernard Route

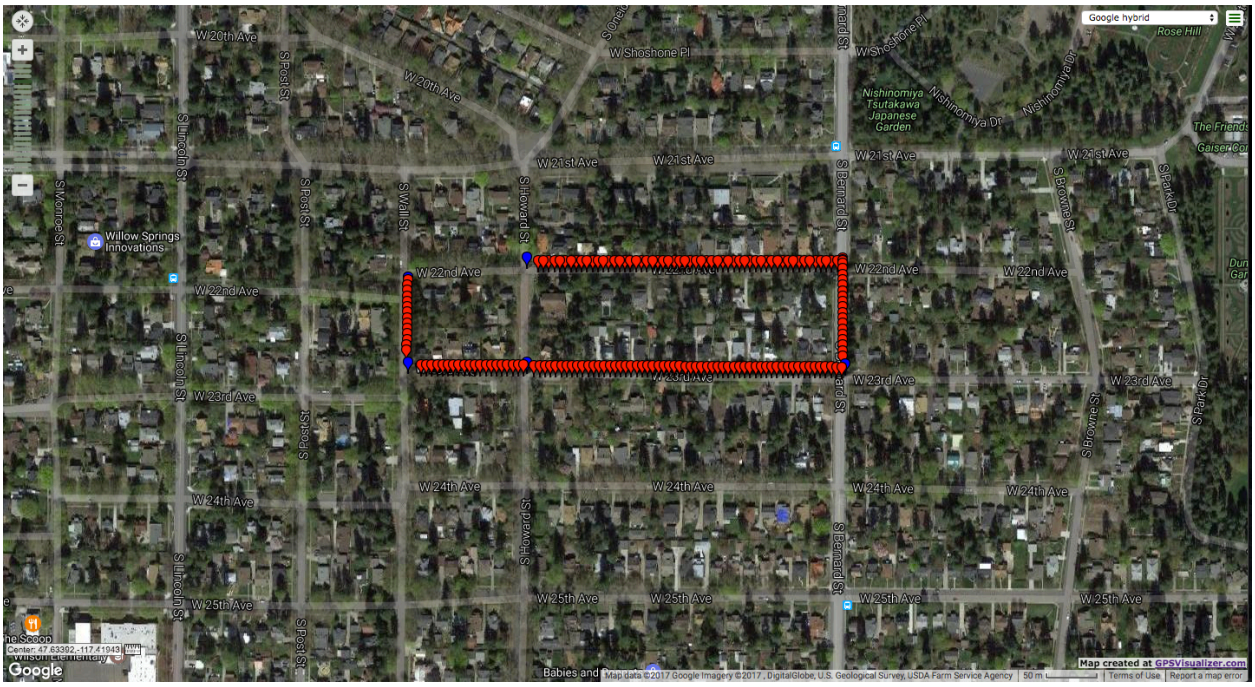

Figure 89: Central Bernard Route

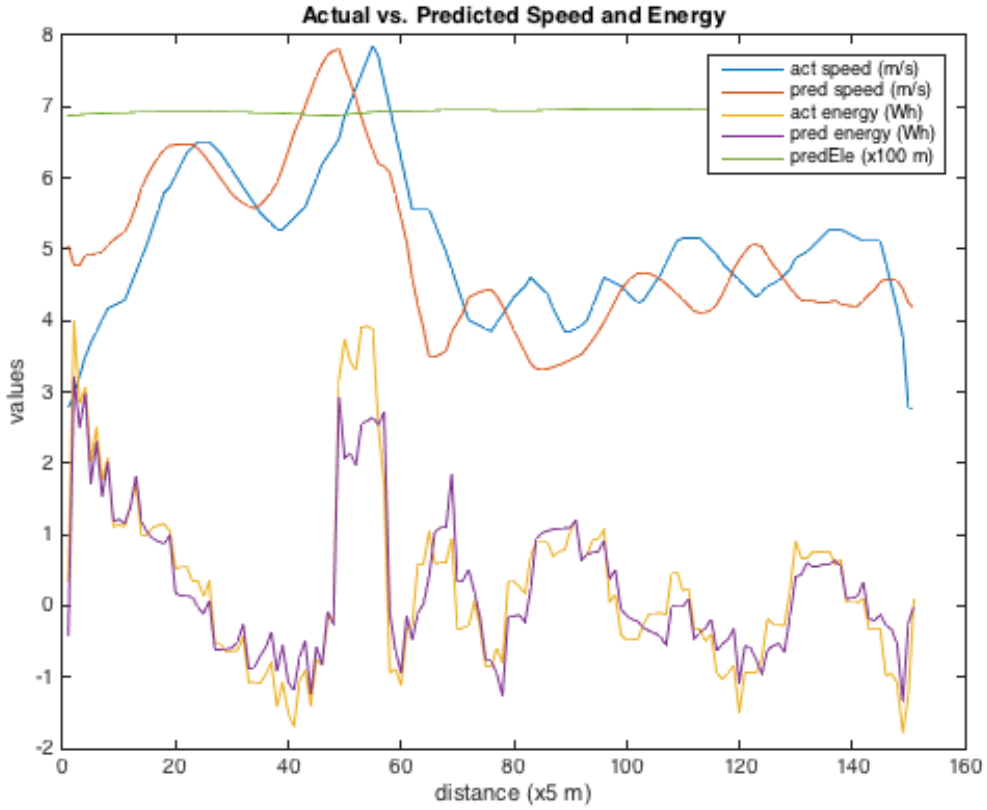

Figure 90: Central Bernard Route Data Snap Shot

Table 82: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-37.95	-33.28	12.4%
Average Net Energy Consumed (Wh)	31.22	31.21	0.00%

Table 83: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.389
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.343
Absolute Average of Predicted Energy Delta (Wh)	0.411
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.425

Table 84: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.00
Absolute Standard Deviation of Predicted Speed Delta (m/s)	0.88

Table 85: Route Travel Details

Metrics	Value
Number of Times Route Traversed	9
Total Travel Distance Over Route (km)	18.75

Table 86: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	93%

7.8 Upper Grand Route

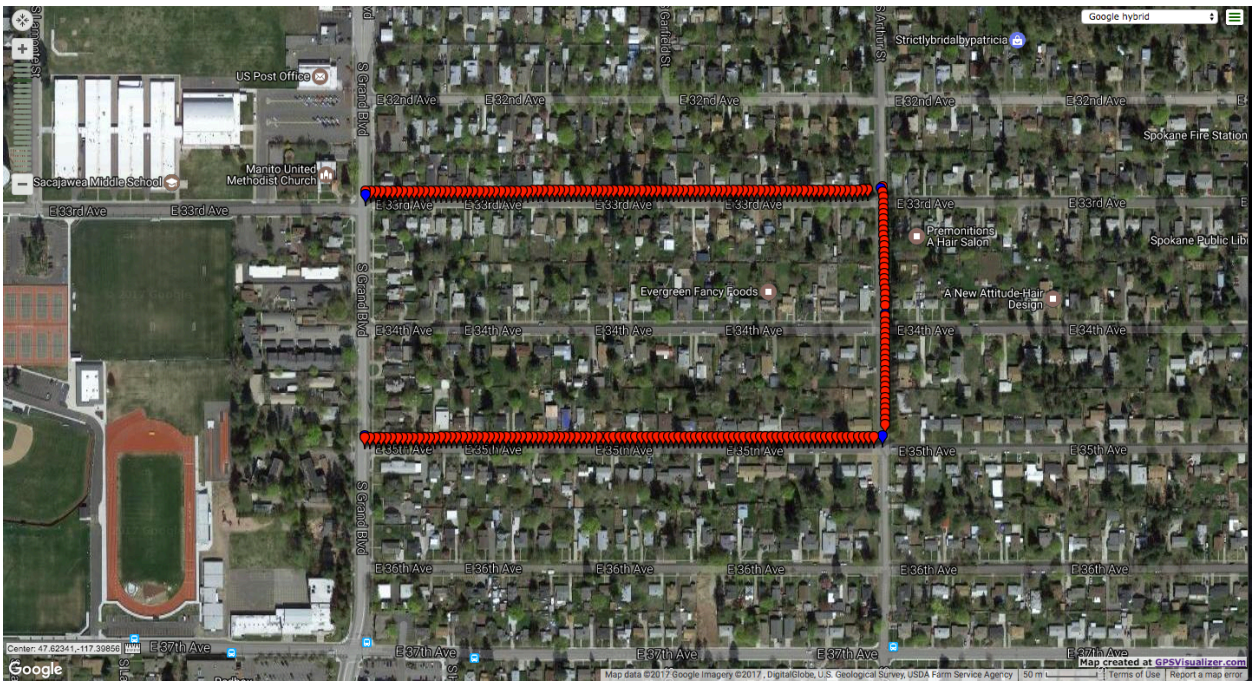

Figure 91: Upper Grand Route

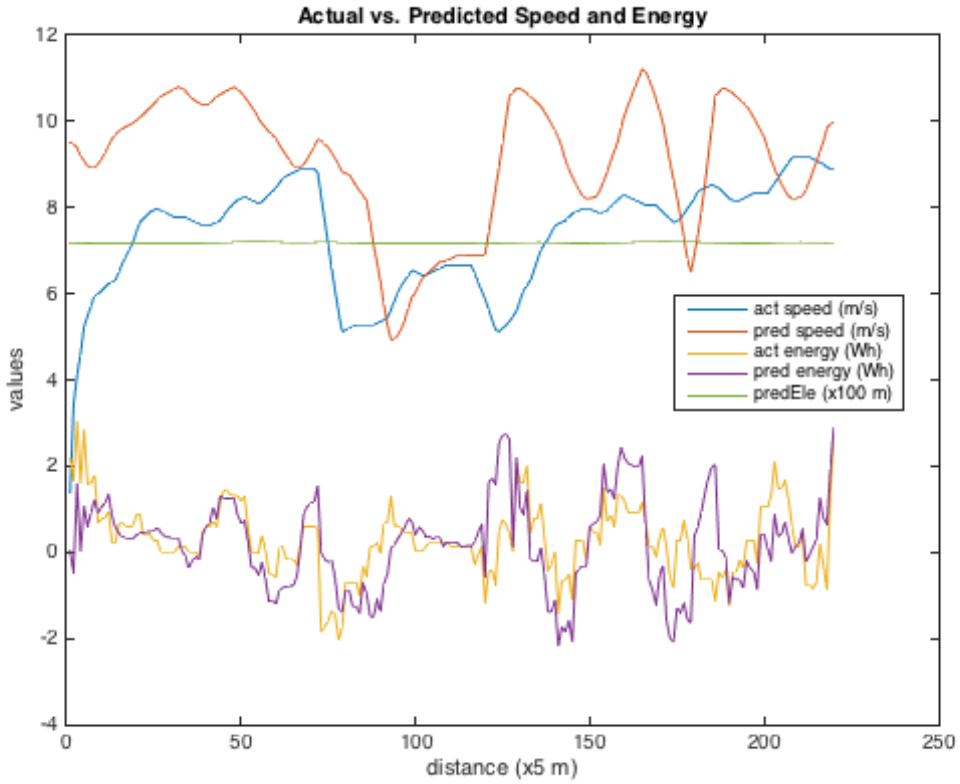

Figure 92: Upper Grand Route Data Snap Shot

Table 87: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-49.9	-10.7	78.5%
Average Net Energy Consumed (Wh)	28.3	34.9	18.9%

Table 88: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.834
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.547
Absolute Average of Predicted Energy Delta (Wh)	0.784
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.726

Table 89: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.46
Absolute Standard Deviation of Predicted Speed Delta (m/s)	1.22

Table 90: Route Travel Details

Metrics	Value
Number of Times Route Traversed	17
Total Travel Distance Over Route (km)	32.90

Table 91: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	94%

7.9 Lower Grand Route

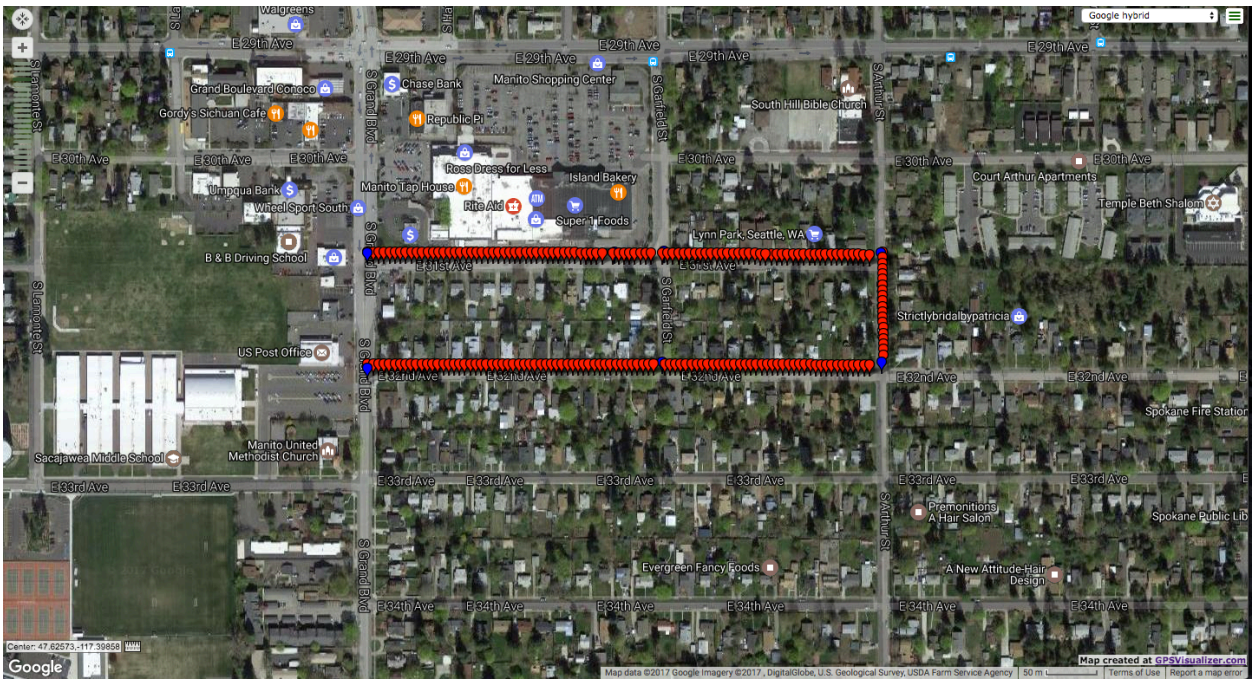

Figure 93: Lower Grand Route

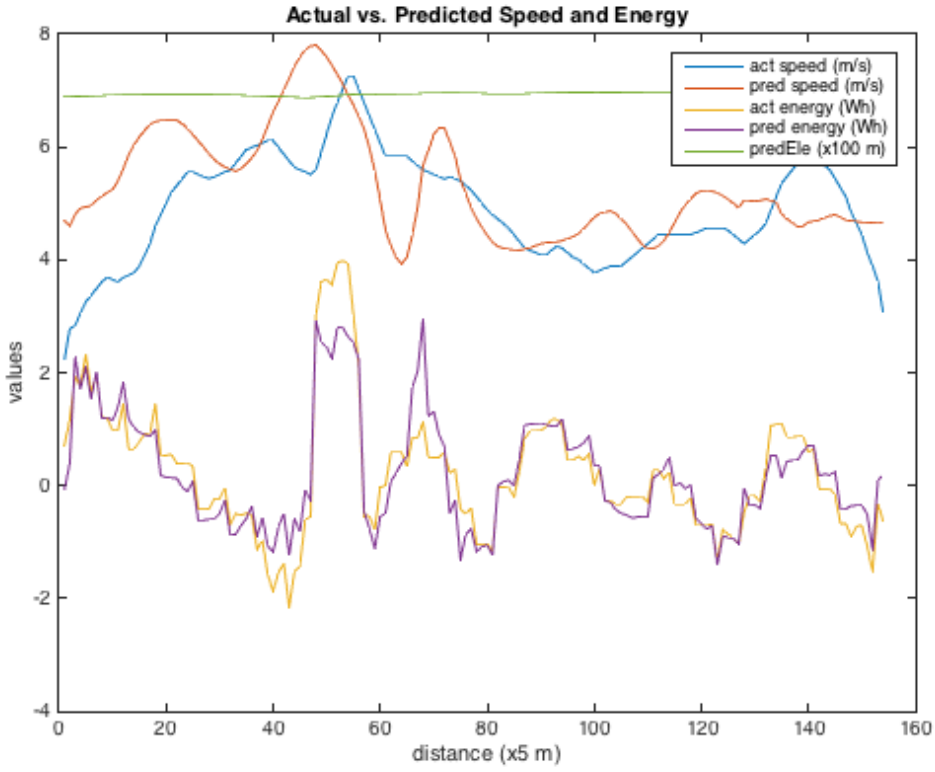

Figure 94: Lower Grand Route Data Snap Shot

Table 92: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-15.84	2.77	0.00%
Average Net Energy Consumed (Wh)	30.58	38.55	20.6%

Table 93: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.716
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.673
Absolute Average of Predicted Energy Delta (Wh)	0.625
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.628

Table 94: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.94
Absolute Standard Deviation of Predicted Speed Delta (m/s)	1.27

Table 95: Route Travel Details

Metrics	Value
Number of Times Route Traversed	33
Total Travel Distance Over Route (km)	76.53

Table 96: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	99%

7.10 Comstock Route

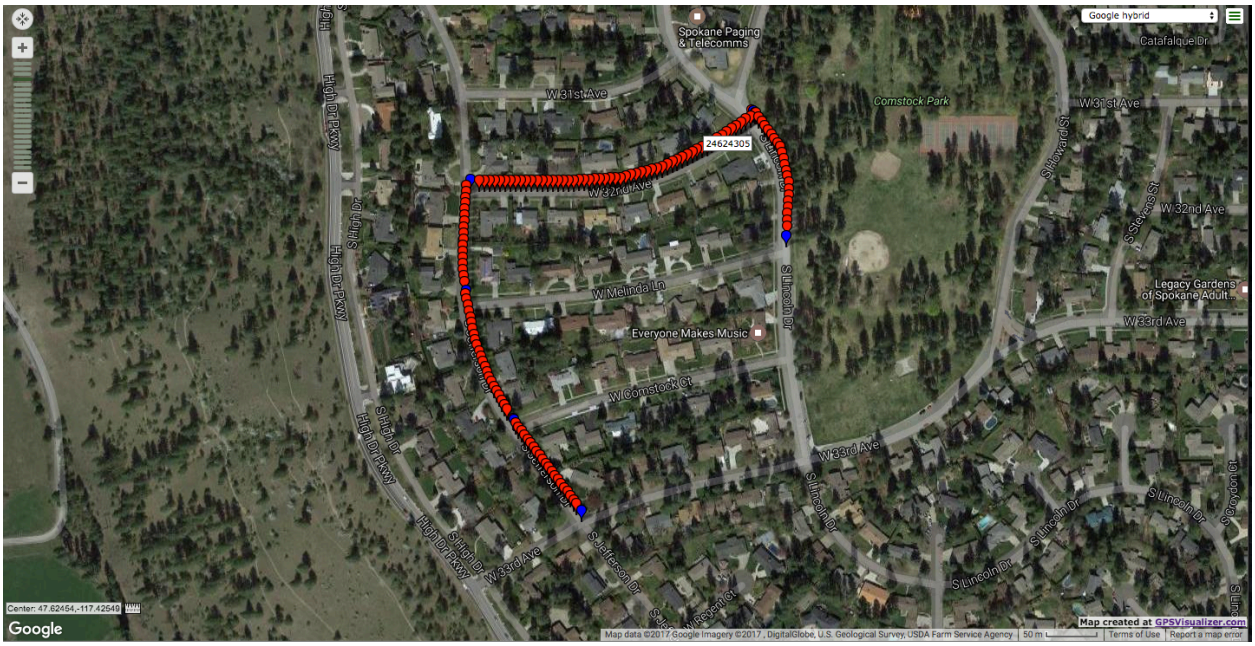

Figure 95: Comstock Route

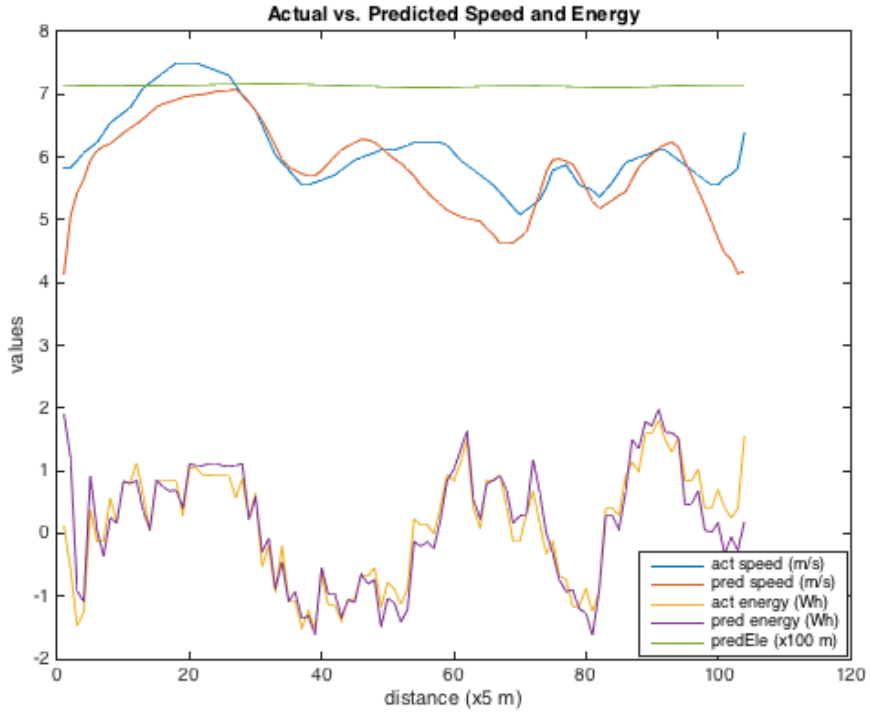

Figure 96: Comstock Route Data Snap Shot

Table 97: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-24.33	-24.31	0.00%
Average Net Energy Consumed (Wh)	17.11	16.47	4.01%

Table 98: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.377
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.259
Absolute Average of Predicted Energy Delta (Wh)	0.353
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.360

Table 99: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	0.80
Absolute Standard Deviation of Predicted Speed Delta (m/s)	0.62

Table 100: Route Travel Details

Metrics	Value
Number of Times Route Traversed	7
Total Travel Distance Over Route (km)	7.58

Table 101: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	85%

7.11 Aggregated Results

The aggregated results assess the traversals of all test routes.

Table 102: Accuracy of Cumulative Energy Calculations

Metrics	Predicted	Actual	Error
Average Energy Gained (Wh)	-50.97	-35.04	31.3%
Average Net Energy Consumed (Wh)	12.99	17.89	27.3%

Table 103: Accuracy of Independent Predicted Energy Calculations

Metrics	Value
Absolute Average of Predicted Gained Energy Delta (Wh)	0.639
Absolute Standard Deviation of Predicted Gained Energy Delta (Wh)	0.601
Absolute Average of Predicted Energy Delta (Wh)	0.567
Absolute Standard Deviation of Predicted Energy Delta (Wh)	0.580

Table 104: Accuracy of Time-Series Speed Predictions

Metrics	Value
Absolute Average of Predicted Speed Delta (m/s)	1.60
Absolute Standard Deviation of Predicted Speed Delta (m/s)	1.15

Table 105: Route Travel Details

Metrics	Value
Number of Times Test Routes Traversed	151
Total Travel Distance Over Route (km)	276.4

Table 106: Route Prediction Accuracy

Metrics	Value
Route Prediction Accuracy	91%

8. ANALYSIS OF ERROR

Error in energy prediction can be attributed to many factors, the two largest being error in speed prediction and lack of isolated testing contexts. An average error of 1.6 m/s with an average standard deviation of error of 1.15 m/s for all 276 km of rural city testing in which the average speed limit is 13.4 m/s (30 mph) can greatly affect calculation of the predicted energy consumption. The usage of the square of predicted speed within the road load equation only exacerbates the effect of speed prediction error in calculating energy consumption. Contributing factors to speed prediction error may include limited prevention to over-fit, sub-optimal network depth, and limited training cases. Additionally, testing contexts are not isolated to traffic and road conditions. Various traffic conditions can lead to infinite training cases exposed to speed prediction making it difficult for the underlying neural network to generalize. Differing road conditions due to snow and rain also affect the coefficient of rolling resistance and air density constants used in the road load equation introducing error to the calculation of the predicted energy consumption.

8.1 Speed Prediction

8.1.1 Network Depth for Real World Testing

One of the shortcomings to the learning procedure employed in speed prediction is that the error surface may contain local minima so that the gradient descent algorithm through back propagation is not guaranteed to find a global minimum. It has been found through much testing that the neural network learning procedure typically does not get stuck in local minima that are worse than the global minimum, but this phenomenon has been observed when the network depth is insufficient. Adding more connections between units increases the dimensionality in the weight-space providing solution paths around the barriers of local minima in the lower dimensional subspaces [7]. Despite testing various network architecture depths from SIL testing of the original Matlab code-base, the increased number of training cases in real world testing may require a reassessment of the optimal network depth.

8.1.2 Training Cases

In training, routes are traversed a various number of times with various contexts but may not include the majority of potential speed-profiles a driver may drive over a given route. As a result, it is difficult for the neural network to generalize all speed cases. The problem may be further perpetrated by the fact that training of the speed prediction algorithm is performed only once after the completion of a route, in which case insufficient weight may be placed on the pathways around local minima to generalize effectively in future traversals of the same route.

8.1.1 Prevention of Over-Fit

If the number of training cases is much greater than the number of cases trained and tested against, error associated to over-fitting may be less likely. The speed prediction algorithm, however, has limited prevention to over-fitting, two the main prevention methods being early stopping and regularization. Rather, a small learning rate coefficient is used to prevent weights from becoming so large that their value drastically affects output given small changes in input. Prolonged testing can lead to accumulated weight values defined by equation (24), but sporadic manual comparisons of speed prediction error between learned and tested speed profiles for a given route show the network still retained acceptable performance in generalization.

8.2 Non-Isolated Testing Contexts

8.2.1 Traffic

Various traffic conditions may be a major culprit in introducing an overabundance of training cases to the underlying neural network in speed prediction. As a result, a lack of isolating the time in which testing occurs during off-peak and peak traffic conditions makes it difficult for the neural network to generalize random slowdowns and stops. All traffic conditions are reflected in the all neural network weight, input and output matrices for all road segments. Repetitive testing with no traffic may yield higher false negatives in generalizing random slowdowns and stops when traffic exists, and in the opposite effect, repetitive testing with traffic may yield more noise in predictions when no traffic exists.

8.2.2 Road Surface Conditions

Finally, road surface and weather conditions fluctuated throughout testing from rain and snow, yet the coefficient of rolling resistance and air density parameters remained constant in the road load equation to calculate predicted energy consumption. This is a source of error.

9. ANALYSIS OF PROJECT SUCCESS / FAILURE

Overall, the project is a success. There is good correlation between actual and predicted speed and energy consumption and a high percentage of accuracy for route prediction. Additionally, for all but one test route, average predicted charge-gaining energy consumption is correct in that a charge-gaining event is reflected in actual energy consumption. Precision of the speed and energy consumption is also acceptable.

In comparison to the proof-of-concept SIL model, the accuracy of the predicted value is less, but this is expected as the speed traces of the proof-of-concept SIL model used white noise as opposed to highly variable speed trace profiles used in the real-world testing.

10. SUMMARY AND CONCLUSION

10.1 Summary

Overall a proof-of-concept software-in-the-loop study is performed to assess the accuracy of predicted net and charge-gaining energy consumption for their potential effective use in optimizing powertrain management of hybrid vehicles. With promising results of improving fuel efficiency of a thermostatic control strategy for a series hybrid-electric vehicle, the route and speed prediction machine learning algorithms are redesigned and implemented for real-world testing in a stand-alone C++ code-base to ingest map data, learn and predict driver habits, and store driver data for fast startup and shutdown of the controller or computer used to execute the compiled algorithm. Speed prediction is performed using a multi-layer, multi-input, multi-output neural network using feed-forward prediction and gradient descent through back-propagation training. Route prediction utilizes a Hidden Markov Model with a recurrent forward algorithm for prediction and multi-dimensional hash maps to store state and state distribution constraining associations between atomic road segments and end destinations. Predicted energy is calculated using the predicted time-series speed and elevation profile over the predicted route and the road-load equation. Testing of the code-base is performed over a known road network spanning 24x35 blocks on the south hill of Spokane, Washington. A large set of training routes are traversed once to add randomness to the route prediction algorithm, and a subset of the training routes, testing

routes, are traversed to assess the accuracy of the net and charge-gaining predicted energy consumption. Each test route is traveled a random number of times with varying speed conditions from traffic and pedestrians to add randomness to speed prediction. Prediction data is stored and analyzed in a post process Matlab script.

10.2 Conclusion

In all, the aggregated results and analysis of all traversals of all test routes reflect the performance of the Driver Prediction algorithm. The error of average energy gained through charge-gaining events is 31.3% and the error of average net energy consumed is 27.3%. The average delta and average standard deviation of the delta of predicted energy gained through charge-gaining events is 0.639 and 0.601 Wh respectively for individual time-series calculations. Similarly, the average delta and average standard deviation of the delta of the predicted net energy consumed is 0.567 and 0.580 Wh respectively for individual time-series calculations. The average delta and standard deviation of the delta of the predicted speed is 1.60 and 1.15 respectively also for the individual time-series measurements. The percentage of accuracy of route prediction is 91%. Overall, test routes are traversed 151 times for a total test distance of 276.4 km.

Future applications of the Driver Prediction algorithm may include improving conventional rule-based thermostatic, load following and hybrid control strategies for hybrid powertrain management by providing anticipatory energy consumption to the otherwise purely real-time measurements. Target state of charge of a traction battery would most likely transform into a target window in which state of charge can decrease below the traditional target in anticipation of receiving energy through or the road or from predicted end-destination grid chargers and in some cases, even exceed the target. Longevity of gas powered energy sources aboard the hybrid architectures would improve through wiser and less frequent utilization. Additional applications may include improvements to optimization-based control strategies to split dynamically power output of the gas and electric motors based on the current vehicle state for optimal efficiency. Most optimization-based implementations require information of future drives, particularly future speed. In one study, a recurrent neural network is used to predict 20 seconds of look-ahead speed as a function of previous speed [14]. However, for larger energy savings, this look-ahead time is too short and does not offer the potential to utilize a predicted end-destination charger to optimize powertrain utilization. In future work, usage of Driver Prediction in a hybrid powertrain controller will be compared against conventional rule-based and optimization-based controllers to assess energy saved.

11. REFERENCES

[1] R. Simmons, B. Browning, Y. Zhang and V. Sadekar, "Learning to Predict Driver Route and Destination Intent," in *IEEE Intelligent Trasportation Systems Conference*, Toronto, Canada, 2006.

[2] L. R. Rabiner and B. H. Juang, "An Introduction to Hidden Markov Models," *ASSP Magazine, IEEE,* vol. 3, no. 1, pp. 4-16, 1986.

[3] C.-Y. Chen and K. Grauman, "Clues from the Beaten Path:," June 2011. [Online]. Available: http://vision.cs.utexas.edu/projects/location-estimation/cvpr2011_location.htm.

[4] M. A. Weiss, Data Structures and Algorithm Analysis in Java, 3rd ed., Boston, MA: Person Education, Inc., 2012.

[5] L. Breiman, Classification and regression trees, Belmont, Calif: Wadsworth International Group, 1984.

[6] MathWorks, "fitrtree," 2016. [Online]. Available: http://www.mathworks.com/help/stats/fitrtree.html.

[7] H. W. D. G. R. Rumelhart, "Learning Representatives by Back-Propogating Errors," *Nature,* vol. 333, pp. 533-536, 9 October 1986.

[8] Veness and Chris, "Calculate Distance, Bearing and More Between Latitude and Longitude," Moveable Type Scripts, 2016.

[9] Movable Type Scripts, "Movable Type Scripts," 4 Octover 2016. [Online]. Available: http://www.movable-type.co.uk/scripts/latlong.html.

[10] W. R. Z. Werner, Hydrogen in the Energy Sector, Ludwig-Blkow-Systemtechnik.

[11] Eigen-unsupported, "Eigen-unsupported," 15 October 2016. [Online]. Available: https://eigen.tuxfamily.org/dox-devel/unsupported/classEigen_1_1Spline.html.

[12] F. O. Alessandrini and F. F. Adriano, "Consumption calculation of vehicles using OBD data," *La Sapienza.*

[13] U. E. P. Agency, "Dynomometer Drive Schedules," U.S. Environmental Protection Agency, 2016.

[14] G. P. R. S. Arsie and M. C. G. M. I., "Opti- mization of supervisory control strategy for parallel hybrid vehicle with provisional load estimate," *AVEC,* pp. 23-27, 2004.

[15] K. M. A. Chaudry and A. A. A. A., "Neuro Fuzzy and Punctual Kriging Based Filter for Image Restoration," *Applied Soft Computing,* vol. 12, no. 3, pp. 817-832, February 2013.

12. APPENDICES

12.1 main.cpp

```
/*
 * main.cpp
 *
 *  Created on: Mar 8, 2016
 *      Author: vagrant
 */

#include <assert.h>
#include <chrono>
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <iostream>
#include <sstream>
#include <stdlib.h>

#include <utility>
#include <cmath>
#include <math.h>

#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <vector>

#include "BuildCity.h"
#include "City.h"
#include "DataManagement.h"
#include "DriverPrediction.h"
#include "GPS.h"
#include "Kinematics.h"
#include "RoutePrediction.h"
#include "Route.h"
#include "UnitTests.h"
#include "VehicleDiagnostics.h"


using namespace PredictivePowertrain;

// ****************************************************************************************************************************************
// ****************************************************************************************************************************************
//                                                  DP Diagnostics Reporting
```

```
// ************************************************************************************************************************************************************
// ************************************************************************************************************************************************************

// save actual route and speed
void saveActualData(Route* actualRoute,
                    std::vector<long int>* actualLabels,
                    std::vector<float>* actualSpeed,
                    std::vector<float>* fuelFlow,
                    std::vector<float>* energy,
                    std::vector<float>* time,
                    City* city)
{
    // ROUTE
    FILE* csvRoute = std::fopen("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/DP_ACTUAL_ROUTE.csv", "w");
    actualRoute->saveRoute2CSV(csvRoute, city, true);

    // SPEED FUEL FLOW and CALCULATED ENERGY
    FILE*                                                   csvSpeedFuelFlowEnergyTime                                                                  =
std::fopen("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/DP_ACTUAL_SPEED_FUEL_FLOW.csv", "w");

    // labels
    for(int i = 0; i < actualLabels->size(); i++)
    {
        fprintf(csvSpeedFuelFlowEnergyTime, "%ld,", actualLabels->at(i));
    }
    fprintf(csvSpeedFuelFlowEnergyTime, "\n");

    // speed
    for(int i = 0; i < actualSpeed->size(); i++)
    {
        fprintf(csvSpeedFuelFlowEnergyTime, "%f,", actualSpeed->at(i));
    }
    fprintf(csvSpeedFuelFlowEnergyTime, "\n");

    // fuel flow
    for(int i = 0; i < fuelFlow->size(); i++)
    {
        fprintf(csvSpeedFuelFlowEnergyTime, "%f,", fuelFlow->at(i));
    }
    fprintf(csvSpeedFuelFlowEnergyTime, "\n");

    // calculated energy from prediction data
    for(int i = 0; i < energy->size(); i++)
    {
        fprintf(csvSpeedFuelFlowEnergyTime, "%f,", energy->at(i));
    }
    fprintf(csvSpeedFuelFlowEnergyTime, "\n");

    // time
    for(int i = 0; i < time->size(); i++)
```

```
    {
        fprintf(csvSpeedFuelFlowEnergyTime, "%f,", time->at(i));
    }

    fclose(csvSpeedFuelFlowEnergyTime);
}

// save pred data
void savePredData(DriverPrediction::PredData predData,
                  std::vector<long int> labels,
                  Route* predRoute,
                  City* city,
                  FILE* predDataFile,
                  FILE* predRouteFile)
{
    // route
    predRoute->saveRoute2CSV(predRouteFile, city, false);

    // other prediction data
    fprintf(predDataFile, "--- spd n ele ---\n");

    // pred labels
    for(int i = 0; i < labels.size(); i++)
    {
        fprintf(predDataFile, "%ld,", labels.at(i));
    }
    fprintf(predDataFile, "\n");

    // pred speed
    for(int i = 0; i < predData.first.size(); i++)
    {
        fprintf(predDataFile, "%f,", predData.first.at(i));
    }
    fprintf(predDataFile, "\n");

    // pred elevation
    for(int i = 0; i < predData.second.size(); i++)
    {
        fprintf(predDataFile, "%f,", predData.second.at(i));
    }
    fprintf(predDataFile, "\n");
}

int main()
{
    // ********************************************************************************************************************************
    // ********************************************************************************************************************************
    //                                                      Unit Tests
    // ********************************************************************************************************************************
    // ********************************************************************************************************************************
```

```
//    city_ut();
//    buildCity_ut();
//    dataCollection_ut();
//    dataManagement_ut();
//    driverPrediction_ut();
//    GPS_ut();
//    kinematics_ut();
//    routePrediction_ut();
//    speedPrediction_ut();
//    vehicleDiagnostics_ut();

// ********************************************************************************************************************************************
// ********************************************************************************************************************************************
//                                                          Get Serialized Data
// ********************************************************************************************************************************************
// ********************************************************************************************************************************************

// create data manager
DataManagement dm;

// get city data
City* city = dm.getCityData();

// get route prediction data
RoutePrediction* rp = dm.getRoutePredictionData();

// ********************************************************************************************************************************************
// ********************************************************************************************************************************************
//                                                             Initiate
// ********************************************************************************************************************************************
// ********************************************************************************************************************************************

// add city to route prediction
rp->addCity(city);

// add route prediction to Driver Prediction
DriverPrediction::PredData predData;
DriverPrediction dp(rp);

// create diagnostics model
VehicleDiagnostics vd;

// create gps model
GPS gps;

// current road vehicle is on
Road* currRoad = NULL;
Link* currLink = NULL;

// kinematics
```

```
Kinematics kin;

// get distance along road
float distAlongRoad = 0.0;

// start Driver Prediction
std::vector<float> currConditions(1);
currConditions.at(0) = -1;

// vehicle speed;
float vehSpd;
float totalDist = 0;

// vehicle speed prediction distance between predictions
float ds = SpeedPrediction().getDS();

// ensure heading is along road spline trajectory
bool headingIsStart2End = true;

// containers for actual fuel and speed
std::vector<long int> actualLabels;
std::vector<float> actualSpeed;
std::vector<float> fuelFlow;
std::vector<float> energy;
std::vector<float> time;

FILE* predDataFile = fopen("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/DP_PRED_DATA.csv", "w");
FILE* predRouteFile = fopen("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/DP_PRED_ROUTE.csv", "w");
fprintf(predRouteFile, "name, description, color, latitude, longitude\n");

// ************************************************************************************************************************************
// ************************************************************v***********************************************************************
//                                                          Run Program
// ************************************************************************************************************************************
// ************************************************************************************************************************************

// record trip data to build map
if(city == NULL)
{
    std::string line;
    while(vd.getEngineLoad() > 1.0)
    {
        auto start = std::chrono::system_clock::now();
        while(1)
        {
            auto end = std::chrono::system_clock::now();
            std::chrono::duration<double> diff = end - start;

            if(diff.count() > 1.0) { break; }
        }
```

```
        gps.updateTripLog();
    }
}
else
{
    // to start Driver Prediction
    bool isFirstPrediction = true;

    auto start = std::chrono::system_clock::now();
    std::pair<double, double> prevLatLon = gps.readGPS();

    // while car is running
    while(vd.getEngineLoad() > 1.0)
    {
        // get vehicle speed
        vehSpd = vd.getSpeed();
        std::cout << "veh speed: " << vehSpd << std::endl;
        actualSpeed.push_back(vehSpd);

        // get fuel flow
        fuelFlow.push_back(vd.getFuelFlow());

        // update current road if intersection happen
        if(!gps.isOnRoad(currRoad))
        {
            currRoad = gps.getCurrentRoad1(city);
            headingIsStart2End = gps.isHeadingStart2EndOfCurrentRoad(currRoad);

            if(headingIsStart2End)
            {
                currLink = Link().linkFromRoad(currRoad, currRoad->getStartIntersection());
            }
            else
            {
                currLink = Link().linkFromRoad(currRoad, currRoad->getEndIntersection());
            }

            std::cout << "on new road: " << currLink->getNumber() << " - " << currLink->getDirection() << std::endl;

            // start prediction
            if(isFirstPrediction)
            {
                // get distance along road
                distAlongRoad = gps.getDistAlongRoad(currRoad, true, headingIsStart2End);
                predData = dp.startPrediction(currLink, vehSpd, &currConditions, distAlongRoad);
                isFirstPrediction = false;
            }

            if(predData.first.size() > 0 && predData.second.size() > 0)
```

```
    {
        std::vector<long int> predLabels = dp.getRouteDataLabels();
        savePredData(predData, predLabels, dp.getPredRoute(), city, predDataFile, predRouteFile);
    }
}

// update current road label
actualLabels.push_back(currRoad->getRoadID());

// get distance along road
distAlongRoad = gps.getDistAlongRoad(currRoad, true, headingIsStart2End);

// predict iteratively along vehicle route
predData = dp.nextPrediction(currLink, vehSpd, distAlongRoad);

// approximate energy usage from predicted speed and elevation change
if(predData.first.size() > 0 && predData.second.size() > 0)
{
    energy.push_back(kin.runKinematics(predData.first, ds, predData.second, false));
}

// ensure vehicle has traveled prediction interval distance before next prediction
while(vd.getEngineLoad() > 1.0)
{
    std::pair<double, double> currLatLon = gps.readGPS();
    float travelDist_i = gps.deltaLatLonToXY(prevLatLon.first, prevLatLon.second, currLatLon.first, currLatLon.second);

    if(travelDist_i > ds)
    {
        // get left over dist from previous dist measurement and add to current dist measurement
        float distRatio = (std::fmodf(totalDist, ds) + travelDist_i) / ds;

        totalDist += travelDist_i;
        std::cout << "total dist traveled: " << totalDist << std::endl;

        if(distRatio > 2)
        {
            // if dist ratio is more than 2x prediction interval distance, buffer speed
            for(int i = 1; i <= distRatio; i++)
            {
                actualSpeed.push_back(vehSpd);
                actualLabels.push_back(currRoad->getRoadID());
                dp.updateSpeedsbyVal(vehSpd);
            }
        }

        std::cout << "prediction distance: " << travelDist_i << std::endl;
        std::cout << "**********************************************************************************" << std::endl;
        prevLatLon = currLatLon;
        break;
```

```
                }
            }

            auto end = std::chrono::system_clock::now();
            std::chrono::duration<double> diff = end - start;
            time.push_back(diff.count());
            start = end;
        }
    }

    // ********************************************************************************************************************************************
    // ********************************************************************************************************************************************
    //                                                  Store Data & Shutdown
    // ********************************************************************************************************************************************
    // ********************************************************************************************************************************************

    // add -1 to end of speed trace for later parsing
    actualSpeed.push_back(-1);

    // create build city module to add new data if needed
    // BuildCity bc;

    // get and store trip data
    dm.addTripData(gps.getTripLog(true));

    // update city data
    // city = bc.getUpdatedCity();

    // store city data
    dm.addCityData(city);

    // get route from city using gps trace
    Route* actualRoute = city->getRouteFromGPSTrace(gps.getTripLog(true));

    // save actual route
    saveActualData(actualRoute, &actualLabels, &actualSpeed, &fuelFlow, &energy, &time, city);

    // parse route
    dp.parseRoute(actualRoute, &actualSpeed, gps.getTripLog(true));

    // store route prediction data
    dm.addRoutePredictionData(dp.getRP());

    fclose(predRouteFile);
    fclose(predDataFile);

      std::cout << "finished Driver Prediction" << std::endl;

      return 0;
}
```

12.2 VehicleDiagnostics.cpp

```
/*
 * VehicleDiagnostics.cpp
 *
 *  Created on: Apr 17, 2016
 *      Author: vagrant
 */

#include "VehicleDiagnostics.h"

namespace PredictivePowertrain {

VehicleDiagnostics::VehicleDiagnostics()
{
    this->fd =  -1;
    this->timeMultiplierSFFEL = 3500;
    this->initializeDiagnosticsReader();

    this->vehicleSpeed = "01 0D\r";
    this->engineLoad = "01 04\r";
    this->airFlow = "01 10\r";
    this->o2Data = "01 24\r";
}

VehicleDiagnostics::~VehicleDiagnostics() {}

void VehicleDiagnostics::initializeDiagnosticsReader()
{
    this->fd = open("/dev/cu.wchusbserialfd120", O_RDWR | O_NONBLOCK);

    if(this->fd < 0)
    {
        std::cout << "Unable to open /dev/tty." << std::endl;
    }

    std::cout << "initializing vd" << std::endl;

    struct termios theTermios;

    memset(&theTermios, 0, sizeof(struct termios));
    cfmakeraw(&theTermios);
    cfsetspeed(&theTermios, 38400);

    // must stay in this particular order
    theTermios.c_cflag = CREAD | CLOCAL;      // turn on READ
    theTermios.c_cflag &= ~PARENB;            // Make 8n1
    theTermios.c_cflag &= ~CSTOPB;
    theTermios.c_cflag &= ~CSIZE;
    theTermios.c_cflag &= ~CRTSCTS;           // no flow control
```

```
    theTermios.c_cflag |= CS8;

    theTermios.c_cc[VMIN] = 0;
    theTermios.c_cc[VTIME] = 10;              // 1 sec timeout
    ioctl(this->fd, TIOCSETA, &theTermios);

    std::vector<std::string> startup(10);
    startup.at(0) = "AT D\r";
    startup.at(1) = "AT D\r";

    startup.at(2) = "AT I\r";
    startup.at(3) = "AT E0\r";
    startup.at(4) = "AT L1\r";
    startup.at(5) = "AT H0\r";
    startup.at(6) = "AT S1\r";
    startup.at(7) = "AT AL\r";

//    startup.at(2) = "AT Z\r";
//    startup.at(3) = "AT E0\r";
//    startup.at(4) = "AT L0\r";
//    startup.at(5) = "AT S0\r";
//    startup.at(6) = "AT H0\r";

    startup.at(8) = "AT SP 0\r";
    startup.at(9) = "0100\r";

    this->clearRxBuffer();

    for(int i = 0; i < startup.size() - 1; i++)
    {
        std::cout << this->getDiagnostsics(startup.at(i), 300) << std::endl;
    }

    // initialize
    std::cout << this->getDiagnostsics(startup.at(9), 30000) << std::endl;

    this->clearRxBuffer();
}

std::string VehicleDiagnostics::readDiagnostics()
{
    // define vars
    char buf[255];
    size_t res;

    while(true) // wait for read
    {
        res = read(this->fd,buf,255);

        if(res > 0)
```

```
        {
            buf[res]=0;

            std::string newMsg(buf);
            return newMsg;
            break;
        }
    }
}

float VehicleDiagnostics::getSpeed()
{
    std::string vehSpdRaw = this->getDiagnostsics(this->vehicleSpeed, this->timeMultiplierSFFEL);

    // get val
    float spdkph = this->hex2Float(vehSpdRaw.substr(6,2));

    return spdkph / 3.6;
}

float VehicleDiagnostics::getFuelFlow()
{
    float afr_i = this->readO2();
    float airFlow_i = this->readMAF();
    float fuelFlow = airFlow_i / afr_i;

    std::cout << "fuel flow: " << fuelFlow << std::endl;

    return fuelFlow;
}

float VehicleDiagnostics::readMAF()
{
    std::string airFlowRaw = this->getDiagnostsics(this->airFlow, this->timeMultiplierSFFEL);

    // parse air float
    float AF_A = this->hex2Float(airFlowRaw.substr(6,2));
    float AF_B = this->hex2Float(airFlowRaw.substr(9,2));
    float MAF = (256.0 * AF_A + AF_B) / 4;

    std::cout << "MAF: " << MAF << std::endl;

    return MAF;
}

float VehicleDiagnostics::readO2()
{
    std::string o2DataRaw_i = this->getDiagnostsics(this->o2Data, this->timeMultiplierSFFEL);

    float o2Data_A = this->hex2Float(o2DataRaw_i.substr(6,2));
```

```
    float o2Data_B = this->hex2Float(o2DataRaw_i.substr(9,2));
    float AFR = 2.0 / 65526.0 * (256.0 * o2Data_A + o2Data_B);

    std::cout << "AFR: " << AFR << std::endl;

    return AFR;
}

float VehicleDiagnostics::getEngineLoad()
{
    std::string engineLoadRaw = this->getDiagnostsics(this->engineLoad, this->timeMultiplierSFFEL);

    float engineLoad_A = this->hex2Float(engineLoadRaw.substr(6,2));
    float engineLoad = engineLoad_A / 2.55;

    std::cout << "engine load: " << engineLoad << std::endl;

    return engineLoad;
}

std::string VehicleDiagnostics::getDiagnostsics(std::string cmd, int timeMultiplier)
{
    if(this->fd < 0)
    {
        this->initializeDiagnosticsReader();
    }

    this->clearRxBuffer();

    write(this->fd, cmd.c_str(), cmd.length());

    usleep((25 * cmd.length()) * timeMultiplier);

    std::string response = this->readDiagnostics();

    if(!response.find("NO DATA") || response.size() < 6)
    {
        response = "000000000000000";
    }

    return response;
}

float VehicleDiagnostics::hex2Float(std::string hex)
{
    if(hex.find("0x") == std::string::npos)
    {
        hex = "0x" + hex;
    }
```

```
    float val;
    std::stringstream ss;
    ss << std::hex << hex;
    ss >> val;
    return val;
}

void VehicleDiagnostics::clearRxBuffer()
{
    // clear buffer remainder
    char buf[255];
    int i = 1;
    while(i > 0)
    {
        i = read(this->fd, buf, 255);
        std::string msg(buf);
    }
}

}
```

12.3 VehicleDiagnostics.h

```
/*
 * VehicleDiagnostics.h
 *
 *  Created on: Apr 17, 2016
 *      Author: vagrant
 */

#ifndef VEHICLE_DIAGNOSTICS_GPS_H_
#define VEHICLE_DIAGNOSTICS_GPS_H_

#include <utility>
#include <cmath>
#include <math.h>

#include <iostream>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <vector>

#include "../map/GenericMap.h"
```

```
namespace PredictivePowertrain {

    class VehicleDiagnostics {
    private:

        int fd;
        int timeMultiplierSFFEL;

        std::string vehicleSpeed;
        std::string engineLoad;
        std::string airFlow;
        std::string o2Data;

        float readO2();
        float readMAF();
        void initializeDiagnosticsReader();
        std::string readDiagnostics();
        std::string getDiagnostsics(std::string cmd, int timeMultiplier);

        void clearRxBuffer();
        float hex2Float(std::string hex);

    public:

        VehicleDiagnostics();
        virtual ~VehicleDiagnostics();

        float getSpeed();
        float getFuelFlow();
        float getEngineLoad();
    };

}

#endif /* VEHICLE_DIAGNOSTICS_GPS_H_ */
```

12.4 VehicleDiagnosticsUnitTest.cpp

```
/*
 *  VehicleDiagnostics
 *
 *
 */

#include "../vehicle_diagnistics/VehicleDiagnostics.h"
#include <chrono>

using namespace PredictivePowertrain;

// unit test for the SpeedPrediction class
```

```
void vehicleDiagnostics_ut()
{
    VehicleDiagnostics vd;

    // log values for duration of time
    int timeItr = 0;
    while(timeItr < 360)
    {
        std::cout << "------ " << timeItr << " ------" << std::endl;
        std::cout << "fuel flow (g/a): " << vd.getFuelFlow() << std::endl;
        std::cout << "speed (mph): " << vd.getSpeed() * 2.23 << std::endl;
        std::cout << "engine load (%): " << vd.getEngineLoad() << std::endl;

//        auto start = std::chrono::system_clock::now();
//        while(1)
//        {
//            auto end = std::chrono::system_clock::now();
//            std::chrono::duration<double> diff = end - start;
//
//            if(diff.count() > 1.0) { break; }
//        }

        timeItr++;
    }
}
```

12.5 Kinematics.cpp

```
/*
 * Kinematics.cpp
 *
 *  Created on: May 5, 2016
 *      Author: vagrant
 */


#include "Kinematics.h"


namespace PredictivePowertrain {

Kinematics::Kinematics()
{
    this->distLookAhead = 28;
    this->distInterval = 5;
    this->vehicleMass = 1508.195;
    this->airDensity = 0.3;
    this->frontSurfaceArea = 3;
    this->dragCoefficient = .3;
    this->rollCoefficient = 0.006;
    this->gravityAcceleration = 9.81;
```

```
}

float Kinematics::runKinematics(std::vector<float> predictedSpeed,
                                float ds,
                                std::vector<float> predictedElevation,
                                bool predChange)
{
    // calculate tractive energy usage over look ahead distance
    float tractiveEnergy = 0.0;
    int maxLookAhead = std::min(this->distLookAhead, (int)predictedSpeed.size()-2);
    for (int i = 0; i < maxLookAhead; i++)
    {
        // acceleration (assumes linear acceleration between speed measurements)
        float va = predictedSpeed.at(i);
        float vb = predictedSpeed.at(i + 1);
        float vehicleAcceleration_i = (std::pow(vb, 2) - std::pow(va, 2)) / (2 * ds);

        // elevation angle
        float ea = predictedElevation.at(i);
        float eb = predictedElevation.at(i + 1);
        float elevationAngle_i = std::atan2(eb - ea, this->distInterval);

        // tractive force
        float intertialForce = vehicleMass * vehicleAcceleration_i + vehicleMass * gravityAcceleration * std::sin(elevationAngle_i);
        float dragForce = 0.5 * airDensity * frontSurfaceArea * dragCoefficient * std::pow(predictedSpeed.at(i), 2);
        float rollingForce = rollCoefficient * vehicleMass * gravityAcceleration;
        float tractiveForce = intertialForce + dragForce + rollingForce;

        // tractive energy
        tractiveEnergy += tractiveForce * this->distInterval;
    }

    return tractiveEnergy;
}

}
```

12.6 Kinematics.h

```
/*
 * Kinematics.h
 *
 *  Created on: May 5, 2016
 *      Author: vagrant
 */

#ifndef OPTIMIZER_OPTIMIZER500_H_
#define OPTIMIZER_OPTIMIZER500_H_

#include <math.h>
#include <vector>
```

```
#include <utility>
#include <cmath>
#include <algorithm>

namespace PredictivePowertrain {

    class Kinematics {
    private:
        static int idx;
        int distLookAhead;
        int distInterval;
        int vehicleMass;
        double airDensity;
        double frontSurfaceArea;
        double dragCoefficient;
        double rollCoefficient;
        double gravityAcceleration;

        bool trqCut;
        bool regen;
        float tractiveEnergy;
    public:
        Kinematics();
        float runKinematics(std::vector<float> predictedSpeed,
                            float ds,
                            std::vector<float> predictedElevation,
                            bool predChange);
    };
}



#endif /* OPTIMIZER_OPTIMIZER500_H_ */
```

12.7 KinematicsUnitTest.cpp

```
/*
 * KinematicsUnitTest.cpp
 *
 *  Created on: Apr 30, 2016
 *      Author: vagrant
 */

#include "UnitTests.h"
#include "../kinematics/Kinematics.h"

#include <assert.h>
#include <iostream>
#include <fstream>

using namespace PredictivePowertrain;
```

```
void kinematics_ut() {

    int length = 1000;
    Kinematics kin;
    std::string num;

    std::vector<float> spdUpVec(length);
    std::ifstream spdUp("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/kinSpdIncrease.csv");
    for (int i = 0; i<length; i++){
        std::getline(spdUp, num, ',');
        std::stringstream fs(num);
        float f = 0.0;
        fs >> f;
        spdUpVec.at(i) = f;
    }

    std::vector<float> spdDownVec(length);
    std::ifstream spdDown("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/kinSpdDecrease.csv");
    for (int i = 0; i<length; i++){
        std::getline(spdDown, num, ',');
        std::stringstream fs(num);
        float f = 0.0;
        fs >> f;
        spdDownVec.at(i) = f;
    }

    std::vector<float> eleUpVec(length);
    std::ifstream eleUp("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/kinEleIncline.csv");
    for (int i = 0; i<length; i++){
        std::getline(eleUp, num, ',');
        std::stringstream fs(num);
        float f = 0.0;
        fs >> f;
        eleUpVec.at(i) = f;
    }

    std::vector<float> eleDownVec(length);
    std::ifstream eleDown("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/kinEleDecline.csv");
    for (int i = 0; i<length; i++){
        std::getline(eleDown, num, ',');
        std::stringstream fs(num);
        float f = 0.0;
        fs >> f;
        eleDownVec.at(i) = f;
    }

    float negTractiveEnergy = kin.runKinematics(spdDownVec, 5.0, eleDownVec, false);
    assert(negTractiveEnergy < 0.0);
```

```
    float posTractiveEnergy = kin.runKinematics(spdUpVec, 5.0, eleUpVec, false);
    assert(posTractiveEnergy > 0.0);

}
```

12.8 City.cpp

```
/*
 * CityObj.cpp
 *
 *  Created on: Jan 8, 2016
 *      Author: Amanda
 */

#include "City.h"

namespace PredictivePowertrain {

City::City() {
    this->roads = new GenericMap<long int, Road*>();
    this->intersections = new GenericMap<long int, Intersection*>();
    this->boundsMap = new GenericMap<int, Bounds*>();
    std::srand(std::time(0));
}

City::City(GenericMap<long int, Intersection*>* intersections, GenericMap<long int, Road*>* roads, GenericMap<int, Bounds*>* boundsMap) {
      this->roads = roads;
      this->intersections = intersections;
      this->boundsMap = boundsMap;
    std::srand(std::time(0));
}

City::~City()
{
    delete(this->roads);
    delete(this->intersections);
    delete(this->boundsMap);
}

int City::getRoadMapSize() {
      return this->roads->getSize();
}

int City::getInstersectionMapSize() {
      return this->intersections->getSize();
}

GenericMap<long int, Link*>* City::getNextLinks(Link* otherLink) {
      assert(!otherLink->isFinalLink());
      Road* currentRoad = this->roads->getEntry(otherLink->getNumber());
      Intersection* nextIntersection = getIntersectionFromLink(otherLink, true);
```

```
      GenericMap<long int, Link*>* nextLinks = new GenericMap<long int, Link*>();
      GenericMap<long int, Road*>* connectingRoads = nextIntersection->getRoads();

      long int count = 0;
      connectingRoads->initializeCounter();
      GenericEntry<long int, Road*>* nextRoad = connectingRoads->nextEntry();
      while(nextRoad != NULL)
      {
        if(nextRoad->value->getRoadID() != currentRoad->getRoadID())
        {
             /* OLD
                Link* newLink = otherLink->linkFromRoad(nextRoad->value, nextIntersection);
                nextLinks->addEntry(count, newLink);
             count++;
              */

             // every road is now bi directional
             Link* link0 = new Link(1, nextRoad->value->getRoadID());
             Link* link1 = new Link(0, nextRoad->value->getRoadID());

             nextLinks->addEntry(count, link0);
             count++;

             nextLinks->addEntry(count, link1);
             count++;
        }
        nextRoad = connectingRoads->nextEntry();
      }
    delete(nextRoad);
      nextLinks->addEntry(count, this->link->finalLink());
      return nextLinks;
}

Intersection* City::getIntersectionFromLink(Link* link, bool isIntersection) {
      assert(!link->isFinalLink());
      Road* road = this->roads->getEntry(link->getNumber());

    if(link->getDirection() == 0 != isIntersection)
      {
        return road->getStartIntersection();
      } else {
        return road->getEndIntersection();
      }
}

Intersection* City::getIntersection(long int intersectionNum) {
      assert(getInstersectionMapSize() < intersectionNum || intersectionNum < 1); // || this->intersections[intersectionNum] == NULL
      return this->intersections->getEntry(intersectionNum);
}
```

```
Route* City::getPath(Intersection* start, Intersection* end, std::vector<float>* conditions, int fastest)
{
      GenericMap<long int, float> dist;
      GenericMap<long int, long int> prev;
      GenericMap<long int, float> unvisitedNodes;

      GenericMap<long int, Intersection*>* adjInts = start->getAdjacentIntersections();

      adjInts->initializeCounter();
      GenericEntry<long int, Intersection*>* nextIntersection = adjInts->nextEntry();
      while(nextIntersection != NULL)
      {
        long int intNum = nextIntersection->value->getIntersectionID();
        dist.addEntry(intNum, FLT_MAX);
        prev.addEntry(intNum, -1);
        nextIntersection = adjInts->nextEntry();
      }
    delete(nextIntersection);

    dist.addEntry(start->getIntersectionID(), 0);

      long int closestIntNum = start->getIntersectionID();
      long int distance = 0;
      while(distance != FLT_MAX)
      {
        unvisitedNodes.addEntry(closestIntNum, FLT_MAX);
        GenericMap<long int, Intersection*>* neighbors = getIntersection(closestIntNum)->getAdjacentIntersections();

        neighbors->initializeCounter();
        GenericEntry<long int, Intersection*>* nextNeighbor = neighbors->nextEntry();
        while(nextNeighbor != NULL)
        {
                long int neighborNum = nextNeighbor->value->getIntersectionID();

             Road* connectingRoad = getConnectingRoad(getIntersection(closestIntNum), nextNeighbor->value);

                float alt = dist.getEntry(neighborNum) + connectingRoad->getSplineLength();
                if(dist.getEntry(neighborNum) > alt || !unvisitedNodes.hasEntry(neighborNum))
                {
                        dist.addEntry(neighborNum, alt);
                        prev.addEntry(neighborNum, closestIntNum);
                        unvisitedNodes.addEntry(neighborNum, alt);
                }
                nextNeighbor = neighbors->nextEntry();
        }

        GenericEntry<long int, float>* closestIntPair = unvisitedNodes.getMinEntry();
        distance = closestIntPair->value;
        closestIntNum = closestIntPair->key;
```

```
      }

      int linkCount = 0;
      GenericMap<long int, Link*>* links = new GenericMap<long int, Link*>();
      long int currentIntNum = end->getIntersectionID();
      while(currentIntNum != start->getIntersectionID())
      {
        Intersection* previousInt = getIntersection(prev.getEntry(currentIntNum));
        links->addEntry(linkCount++, this->link->linkFromRoad(getConnectingRoad(previousInt, getIntersection(currentIntNum)), previousInt));
        currentIntNum = previousInt->getIntersectionID();
      }

      links->addEntry(linkCount++, this->link->finalLink());
      Goal* goal = new Goal(end->getIntersectionID(), conditions);
      Route* route = new Route(links, goal);
      return route;
}

std::pair<std::vector<float>, std::vector<float>> City::getData(Road* road, int direction, SpeedPrediction* sp, Eigen::MatrixXd* spdIn, float
distAlongRoad)
{
    // get elevation Data
    std::vector<float> elevData;
    std::vector<float> elevDistData;
    road->getElevData(&elevData, &elevDistData);

    if(direction)
    {
        for(int i = 0; i < elevData.size(); i++)
        {
            elevData.at(i) = elevData.at(elevData.size() - i - 1);
        }
    }
    assert(elevDistData.size() > 2);
    int prevElevMeasIdx = 0;

    // set output values
    std::vector<float> spdOut;
    std::vector<float> elevDataInterp;

    // iterate along speed trace to interpolate elevation trace
    float spdDist = 0.0;
    bool spdDistExceedsRoadDist = false;

    Eigen::MatrixXd spdOut_i(1, sp->get0());
    Eigen::MatrixXd spdIn_i(*spdIn);

    // format speed input
    sp->formatInData(&spdIn_i);
```

```
while(!spdDistExceedsRoadDist)
{
    // predict speed
    sp->predict(&spdIn_i, &spdOut_i);

    // iterate through spd pred output. skip first val since first val is curr spd
    for(int i = 1; i < spdOut_i.cols(); i++)
    {
        spdDist += sp->getDS();

        if(spdDist >= distAlongRoad && spdDist <= road->getSplineLength() + sp->getDS())
        {
            // interpolate elevation data
            for(int j = prevElevMeasIdx; j < elevDistData.size() - 1; j++)
            {
                if(spdDist > elevDistData.at(j))
                {
                    prevElevMeasIdx = j;
                }
                else
                {
                    break;
                }
            }

            // get delta in elevation from previous to next elevation measurement
            float prev2NextElevDelta = elevData.at(prevElevMeasIdx) - elevData.at(prevElevMeasIdx + 1);

            // get delta in distnce from previous to next elevation measurement
            float next2PrevElevDist = elevDistData.at(prevElevMeasIdx + 1) - elevDistData.at(prevElevMeasIdx);

            // get delta in distance from previous elevation measurement to current speed distance
            float prevElev2SpdDist = spdDist - elevDistData.at(prevElevMeasIdx);

            // derive interpolation factor as a function prev 2 next elevation meas and speed distance after prev elev meas
            float interpFactor = prevElev2SpdDist / next2PrevElevDist;

            spdOut.push_back((spdOut_i.coeffRef(0, i) - sp->getSpeedOffset()) * sp->getMaxSpeed());
            elevDataInterp.push_back(elevData.at(prevElevMeasIdx) + interpFactor * prev2NextElevDelta);
            this->routeDataLabels.push_back(road->getRoadID());
        }
        else if(spdDist > road->getSplineLength())
        {
            spdDistExceedsRoadDist = true;
            break;
        }
    }

    if(!spdDistExceedsRoadDist)
    {
```

```
            // place output into input and repeat if needed
            sp->output2Input(&spdIn_i, &spdOut_i);
        }
    }

    return std::pair<std::vector<float>, std::vector<float>>(spdOut, elevDataInterp);
}

Road* City::getConnectingRoad(Intersection* one, Intersection* two) {
      GenericMap<long int, Road*>* roads = one->getRoads();
      roads->initializeCounter();
      GenericEntry<long int, Road*>* nextRoad = roads->nextEntry();
      while(nextRoad != NULL)
      {
        if(one->getNextIntersection(nextRoad->value)->getIntersectionID() == two->getIntersectionID())
        {
                return nextRoad->value;
        }
        nextRoad = roads->nextEntry();
      }
      return NULL;
}

Route* City::randomPath(Intersection* startInt, Route* initialRoute, int totalLength, std::vector<float>* conditions) {

      GenericMap<long int, Link*>* startLinks = startInt->getOutgoingLinks();
    GenericMap<long int, Link*>* links = new GenericMap<long int, Link*>();

    // copy outgoing links to prevent mutilation
    startLinks->initializeCounter();
    GenericEntry<long int, Link*>* nextLinkEntry = startLinks->nextEntry();
    while(nextLinkEntry != NULL)
    {
        Link* outLink = nextLinkEntry->value;
        links->addEntry(nextLinkEntry->key, new Link(outLink->getDirection(), outLink->getNumber()));
        nextLinkEntry = startLinks->nextEntry();
    }
    delete(nextLinkEntry);

      GenericMap<long int, Link*>* path = new GenericMap<long int, Link*>();
      GenericMap<long int, Intersection*> passedInts;
      passedInts.addEntry(startInt->getIntersectionID(), startInt);

      for(int i = 0; i < totalLength; i++)
      {
        Link* nextLink = NULL;
        if(initialRoute->getLinkSize() > i)
        {
                links->initializeCounter();
                GenericEntry<long int, Link*>* nextLinkEntry = links->nextEntry();
```

```
        while(nextLinkEntry != NULL)
        {
                if(nextLinkEntry->value->isEqual(initialRoute->getLinks()->getEntry(i)))
                {
                        nextLink = initialRoute->getLinks()->getEntry(i);
                        break;
                }
                nextLinkEntry = links->nextEntry();
        }
    delete(nextLinkEntry);
}

if(nextLink == NULL)
{
        links->initializeCounter();
        GenericEntry<long int, Link*>* nextLinkEntry = links->nextEntry();
        while(nextLinkEntry != NULL)
        {
                if(!nextLinkEntry->value->isFinalLink())
                {
                        Intersection* intersection = getIntersectionFromLink(nextLinkEntry->value, true);

                        passedInts.initializeCounter();
                        GenericEntry<long int, Intersection*>* nextPassedInt = passedInts.nextEntry();
                        while(nextPassedInt != NULL)
                        {
                                if(intersection->getIntersectionID() == nextPassedInt->value->getIntersectionID())
                                {
                                        links->indexErase(nextLinkEntry->key);
                                }
                                nextPassedInt = passedInts.nextEntry();
                        }
            delete(nextPassedInt);
                }
         nextLinkEntry = links->nextEntry();
    }
    delete(nextLinkEntry);

    // break of no next link
    if(links->getSize() == 1) { break; }

    int randIdx = std::floor((float)std::rand() / RAND_MAX * links->getSize());
    nextLink = links->getEntry(randIdx);
}

while(nextLink->isFinalLink() && links->getSize() > 1 && nextLink != NULL)
{
    int randIdx = std::floor((float)std::rand() / RAND_MAX * links->getSize());
        nextLink = links->getEntry(randIdx);
}
```

```
        Intersection* newPassedInt = getIntersectionFromLink(nextLink, true);
        passedInts.addEntry(newPassedInt->getIntersectionID(), newPassedInt);
        path->addEntry(i, nextLink);

        delete(links);
        links = getNextLinks(nextLink);
      }
      path->addEntry(path->getSize(), this->link->finalLink());
      Goal* goal = new Goal(getIntersectionFromLink(path->getEntry(path->getSize()-2),true)->getIntersectionID(), conditions);
      Route *route = new Route(path, goal);

      return route;
}

bool City::legalRoute(Route* route) {
      Route* routeCopy = route->copy();
      while(routeCopy->getLinkSize() > 1)
      {
        GenericMap<long int, Link*>* legalLinks = getNextLinks(routeCopy->getLinks()->getEntry(1));
        legalLinks->initializeCounter();
        GenericEntry<long int, Link*>* nextLink = legalLinks->nextEntry();
        bool error = true;
        while(nextLink != NULL)
        {
                if(nextLink->value->isEqual(routeCopy->getLinks()->getEntry(1)))
                {
                        error = false;
                        break;
                }
        }
        if(error)
        {
                return false;
        }
        routeCopy->removeFirstLink();
      }
      return true;
}

std::pair<std::vector<float>, std::vector<float>> City::routeToData(Route* route, float dist, SpeedPrediction* sp, Eigen::MatrixXd* spdIn)
{
    // clear route data labels
    this->routeDataLabels.clear();

    // container for speed and elevation
    std::vector<float> elevData;
    std::vector<float> spdData;

    Eigen::MatrixXd spdIn_i(*spdIn);
```

```
    bool isFirstLink = true;
      route->getLinks()->initializeCounter();
      GenericEntry<long int, Link*>* nextLink = route->getLinks()->nextEntry();
      while(nextLink != NULL
          && !nextLink->value->isFinalLink()
          && nextLink->key != route->getLinks()->getSize() - 2)
      {
        // get road from link
        assert(this->roads->hasEntry(nextLink->value->getNumber()));
        Road* roadFromLink = this->roads->getEntry(nextLink->value->getNumber());

        float distAlongLink = 0.0;
        if(isFirstLink)
        {
            distAlongLink = dist;
            isFirstLink = false;
        }

        // get spd and elevation data
        if(nextLink->value->linkHasWeights())
        {
            sp->setVals(nextLink->value->getWeights(nextLink->value->getDirection()));
            std::pair<std::vector<float>,  std::vector<float>>  linkData  =  this->getData(roadFromLink,  nextLink->value->getDirection(),  sp,
&spdIn_i, distAlongLink);
            std::vector<float> spdOut_i = linkData.first;
            std::vector<float> elevData_i = linkData.second;

            // adjust speed input for next link speed prediction
            Eigen::MatrixXd spdOutMat_i(1, spdOut_i.size());
            for(int i = 0; i < spdOut_i.size(); i++) { spdOutMat_i.coeffRef(0, i)  = spdOut_i.at(i); }
            sp->output2Input(&spdIn_i, &spdOutMat_i);

            // concatenate data
            for(int i = 0; i < spdOut_i.size(); i++) { spdData.push_back(spdOut_i.at(i)); }
            for(int i = 0; i < elevData_i.size(); i++) { elevData.push_back(elevData_i.at(i)); }
        }

        nextLink = route->getLinks()->nextEntry();
      }
    delete(nextLink);
    return std::pair<std::vector<float>, std::vector<float>>(spdData, elevData);
}

GenericMap<int, Bounds*>* City::getBoundsMap() {
      return this->boundsMap;
}

GenericMap<long int, Road*>* City::getRoads() {
      return this->roads;
```

```
}

int City::getBoundsMapSize() {
      return this->boundsMap->getSize();
}

GenericMap<long int, Intersection*>* City::getIntersections() {
      return this->intersections;
}

void City::addRoad(Road* road)
{
    this->roads->addEntry(road->getRoadID(), road);
}

void City::addBounds(Bounds* bounds)
{
    this->boundsMap->addEntry(bounds->getID(), bounds);
}

void City::addIntersection(Intersection* intersection)
{
    this->intersections->addEntry(intersection->getIntersectionID(), intersection);
}

Route* City::getRouteFromGPSTrace(GenericMap<long int, std::pair<double, double>*>* trace)
{
    GPS gps;
    Road* prevRoad = NULL;
    Road* currRoad = NULL;
    Intersection* nextIntersection = NULL;

    GenericMap<long int, Link*>* links = new GenericMap<long int, Link*>();

    GenericMap<long int, std::pair<double, double>*>* traceCopy = trace->copy();

    long int linkCount = 0;
    bool isFirstRoad = true;

    // create new csv
    std::string csvName = "/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/TRACE.csv";
    FILE* csv = std::fopen(csvName.c_str(), "w");

    // add header to csv
    fprintf(csv, "name, description, color, latitude, longitude\n");

    trace->initializeCounter();
    GenericEntry<long int, std::pair<double, double>*>* nextMeas = trace->nextEntry();
    double lat = nextMeas->value->first;
    double lon = nextMeas->value->second;
```

```
prevRoad = gps.getCurrentRoad2(this, lat, lon);

nextMeas = trace->nextEntry();

while(nextMeas != NULL)
{
    // to csv
    fprintf(csv, "%ld,", nextMeas->key);
    fprintf(csv, "Lat & Lon: %.12f %.12f,", lat, lon);
    fprintf(csv, "red,");
    fprintf(csv, "%.12f,%.12f\n", lat, lon);

    lat = nextMeas->value->first;
    lon = nextMeas->value->second;

    currRoad = gps.getCurrentRoad2(this, lat, lon);

    if(currRoad == NULL)
    {
        nextMeas = trace->nextEntry();
        continue;
    }
    else if(prevRoad == NULL)
    {
        prevRoad = currRoad;
        nextMeas = trace->nextEntry();
        continue;
    }

    if(currRoad->getRoadID() != prevRoad->getRoadID() && this->roadIsOnTrace(currRoad, traceCopy, .10))
    {
        // ensure previous road was on trace
        bool prevRoadIsOnTrace = this->roadIsOnTrace(prevRoad, traceCopy, .50);

        if(prevRoadIsOnTrace || isFirstRoad)
        {
            std::cout << "********" << prevRoad->getRoadID() << "********" << std::endl;

            Intersection* start = prevRoad->getStartIntersection();
            Intersection* end = prevRoad->getEndIntersection();

            float toStartIntDist = gps.deltaLatLonToXY(lat, lon, start->getLat(), start->getLon());
            float toEndIntDist = gps.deltaLatLonToXY(lat, lon, end->getLat(), end->getLon());

            int direction;

            // find nearest distances to start and end int of first road
            if(isFirstRoad)
            {
```

```
    traceCopy->initializeCounter();
    GenericEntry<long int, std::pair<double, double>*>* nextMeasCopy = traceCopy->nextEntry();

    float nearestStartDist = FLT_MAX;
    float nearestEndDist = FLT_MAX;

    while(nextMeasCopy != NULL)
    {
        // next measurement cop lat / lon
        double nmcLat = nextMeasCopy->value->first;
        double nmcLon = nextMeasCopy->value->second;

        // calc start and end distances to measurement ith
        float startDist_i = gps.deltaLatLonToXY(nmcLat, nmcLon, start->getLat(), start->getLon());
        float endDist_i = gps.deltaLatLonToXY(nmcLat, nmcLon, end->getLat(), end->getLon());

        if(startDist_i < nearestStartDist)
        {
            nearestStartDist = startDist_i;
        }

        if(endDist_i < nearestEndDist)
        {
            nearestEndDist = endDist_i;
        }

        nextMeasCopy = traceCopy->nextEntry();
    }
    delete(nextMeasCopy);

    toStartIntDist = nearestStartDist;
    toEndIntDist = nearestEndDist;

    if(toStartIntDist > toEndIntDist)
    {
        direction = 0;
    }
    else
    {
        direction = 1;
    }

    isFirstRoad = false;
}
else
{
    if(toEndIntDist < toStartIntDist)
    {
        direction = 0;
    }
```

```
        else
        {
            direction = 1;
        }
    }

    printf("curr: %.6f,%.6f\n", lat, lon);
    std::cout << "curr->start dist: " << toStartIntDist << std::endl;
    std::cout << "curr->end dist: " << toEndIntDist << std::endl;
    std::cout << "dir: " << direction << std::endl;
    std::cout << "next road: " << currRoad->getRoadID() << std::endl;

    Link* link = new Link(direction, prevRoad->getRoadID());

    Intersection* currInt;
    if(toStartIntDist <= toEndIntDist)
    {
        currInt = start;
    }
    else
    {
        currInt = end;
    }

    if(currInt->getIntersectionID() == currRoad->getStartIntersection()->getIntersectionID())
    {
        nextIntersection = currRoad->getEndIntersection();
    }
    else
    {
        nextIntersection = currRoad->getStartIntersection();
    }

    // ensure link does not already exist
    bool linkAlreadyExists = false;
    links->initializeCounter();
    GenericEntry<long int, Link*>* nextLink = links->nextEntry();
    while(nextLink !=  NULL)
    {
        // can go down the same road in opossite directions but cannot go in circles
        if(nextLink->value->isEqual(link) || (nextLink->value->getNumber() == link->getNumber() && linkCount == nextLink->key +
1))
        {
            linkAlreadyExists = true;
            break;
        }
        nextLink = links->nextEntry();
    }
    delete(nextLink);
```

```
            if(!linkAlreadyExists)
            {
                links->addEntry(linkCount, link);
                linkCount++;
            }
        }

        prevRoad = currRoad;

    }

    nextMeas = trace->nextEntry();
}
delete(nextMeas);

fclose(csv);

// add last road
if(this->roadIsOnTrace(currRoad, traceCopy, .25))
{
    Intersection* start = currRoad->getStartIntersection();
    Intersection* end = currRoad->getEndIntersection();

    // find nearest measurement between start and end intersections of last link
    traceCopy->initializeCounter();
    GenericEntry<long int, std::pair<double, double>*>* nextMeasCopy = traceCopy->nextEntry();

    float nearestStartDist = FLT_MAX;
    float nearestEndDist = FLT_MAX;

    while(nextMeasCopy != NULL)
    {
        // next measurement cop lat / lon
        double nmcLat = nextMeasCopy->value->first;
        double nmcLon = nextMeasCopy->value->second;

        // calc start and end distances to measurement ith
        float startDist_i = gps.deltaLatLonToXY(nmcLat, nmcLon, start->getLat(), start->getLon());
        float endDist_i = gps.deltaLatLonToXY(nmcLat, nmcLon, end->getLat(), end->getLon());

        if(startDist_i < nearestStartDist)
        {
            nearestStartDist = startDist_i;
        }

        if(endDist_i < nearestEndDist)
        {
            nearestEndDist = endDist_i;
        }
```

```
            nextMeasCopy = traceCopy->nextEntry();
        }
        delete(nextMeasCopy);

        int direction;
        if(nearestEndDist < nearestStartDist)
        {
            direction = 1;
        }
        else
        {
            direction = 0;
        }

        std::cout << "********" << currRoad->getRoadID() << "********" << std::endl;
        printf("curr: %.6f,%.6f\n", lat, lon);
        std::cout << "curr->start dist: " << nearestStartDist << std::endl;
        std::cout << "curr->end dist: " << nearestEndDist << std::endl;
        std::cout << "dir: " << direction << std::endl;

        Link* link = new Link(direction, currRoad->getRoadID());
        links->addEntry(linkCount++, link);

        if(nearestEndDist > nearestStartDist)
        {
            nextIntersection = currRoad->getEndIntersection();
        }
        else
        {
            nextIntersection = currRoad->getStartIntersection();
        }
    }
    delete(traceCopy);

    // add final link
    links->addEntry(linkCount, Link().finalLink());

    // create route
    Route* route = new Route(links, new Goal(nextIntersection->getIntersectionID()));
    return route;
}

bool City::roadIsOnTrace(Road* road, GenericMap<long int, std::pair<double, double>**>* trace, float thresh)
{
    GPS gps;

    int closeNodeCount = 0;
    road->getNodes()->initializeCounter();
    GenericEntry<long int, Node**>* prevNode = road->getNodes()->nextEntry();
    GenericEntry<long int, Node**>* currNode = road->getNodes()->nextEntry();
```

```
while(currNode != NULL)
{
    double currRoadLat = currNode->value->getLat();
    double currRoadLon = currNode->value->getLon();

    trace->initializeCounter();
    GenericEntry<long int, std::pair<double, double>*>* prevMeasCopy = trace->nextEntry();
    GenericEntry<long int, std::pair<double, double>*>* currMeasCopy = trace->nextEntry();
    while(currMeasCopy != NULL)
    {
        double currTraceLat = currMeasCopy->value->first;
        double currTraceLon = currMeasCopy->value->second;

        float dist = gps.deltaLatLonToXY(currRoadLat, currRoadLon, currTraceLat, currTraceLon);

        // check proximity
        if(dist < gps.getDeltaXYTolerance())
        {
            // check heading
            double prevRoadLat = prevNode->value->getLat();
            double prevRoadLon = prevNode->value->getLon();

            double prevTraceLat = prevMeasCopy->value->first;
            double prevTraceLon = prevMeasCopy->value->second;

            double roadAngle = std::atan2(prevRoadLat - currRoadLat, prevRoadLon - currRoadLon);
            double traceAngle = std::atan2(prevTraceLat - currTraceLat, prevTraceLon - currTraceLon);

            roadAngle = gps.boundTheta(roadAngle);
            traceAngle = gps.boundTheta(traceAngle);

            float angleDiff = std::abs(roadAngle - traceAngle);

            //      same heading             opposite heading                same heading
            if(angleDiff < M_PI / 4 || angleDiff - M_PI < M_PI / 4 || angleDiff > 2 * M_PI - M_PI / 4)
            {
                closeNodeCount++;
            }
            break;
        }

        prevMeasCopy = currMeasCopy;
        currMeasCopy = trace->nextEntry();
    }
    delete(prevMeasCopy);
    delete(currMeasCopy);

    prevNode = currNode;
    currNode = road->getNodes()->nextEntry();
}
```

```
    delete(prevNode);
    delete(currNode);

    return closeNodeCount > thresh * (float) road->getNodes()->getSize();
}

void City::printIntersectionsAndRoads()
{
    // csv name
    std::string csvName = "/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/CITY_INTERSECTIONS_AND_ROADS.csv";

    // delete existing csv if found
    std::string rm = "rm " + csvName;
    system(rm.c_str());

    // create new csv
    FILE* csv;
    csv = std::fopen(csvName.c_str(), "w");

    // add header to csv
    fprintf(csv, "name,icon,description,color,latitude,longitude\n");

    std::cout << "**** printing intersection lat/lon ****" << std::endl;
    this->intersections->initializeCounter();
    GenericEntry<long int, Intersection*>* nextInt = this->intersections->nextEntry();
    while(nextInt != NULL)
    {
        // print intersection lat/lon to console
        printf("%.12f,%.12f\n", nextInt->value->getLat(), nextInt->value->getLon());

        // print intersection lat/lon to csv
        fprintf(csv, "%ld,", nextInt->value->getIntersectionID());
        fprintf(csv, "googlemini,");
        fprintf(csv, "Int ID: %ld | ", nextInt->value->getIntersectionID());
        fprintf(csv, "Int Ele: %f | ", nextInt->value->getElevation());
        fprintf(csv, "Lat & Lon: %.12f %.12f | ", nextInt->value->getLat(), nextInt->value->getLon());
        fprintf(csv, "Connecting Roads: ");

        nextInt->value->getRoads()->initializeCounter();
        GenericEntry<long int, Road*>* nextConnectingRoad = nextInt->value->getRoads()->nextEntry();
        while(nextConnectingRoad != NULL)
        {
            fprintf(csv, "%ld ", nextConnectingRoad->key);
            nextConnectingRoad = nextInt->value->getRoads()->nextEntry();
        }
        delete(nextConnectingRoad);

        fprintf(csv, ",");
        fprintf(csv, "red,");
        fprintf(csv, "%.12f,%.12f\n", nextInt->value->getLat(), nextInt->value->getLon());
```

```
        // print road spline to csv
        nextInt->value->getRoads()->initializeCounter();
        nextConnectingRoad = nextInt->value->getRoads()->nextEntry();
        while(nextConnectingRoad != NULL)
        {
            // iterate along connecting road spline control points
            nextConnectingRoad->value->getNodes()->initializeCounter();
            GenericEntry<long int, Node*>* nextNode = nextConnectingRoad->value->getNodes()->nextEntry();

            // burn a node so they are not superimposed on intersection
            nextNode = nextConnectingRoad->value->getNodes()->nextEntry();

            int iterCount = 1;
            while(iterCount < nextConnectingRoad->value->getNodes()->getSize() - 1)
            {
                fprintf(csv, "%ld,", nextConnectingRoad->value->getRoadID());
                fprintf(csv, "cirlce,");
                fprintf(csv, "Road ID: %ld | ", nextConnectingRoad->value->getRoadID());
                fprintf(csv, "Start Int ID: %ld | ", nextConnectingRoad->value->getStartIntersection()->getIntersectionID());
                fprintf(csv, "Node Ele: %f | ", nextNode->value->getEle());
                fprintf(csv, "End Int ID: %ld | ", nextConnectingRoad->value->getEndIntersection()->getIntersectionID());
                fprintf(csv, "Lat & Lon: %.12f %.12f,", nextNode->value->getLat(), nextNode->value->getLon());
                fprintf(csv, "blue,");
                fprintf(csv, "%.12f,%.12f\n", nextNode->value->getLat(), nextNode->value->getLon());

                // print every other or so control point for faster visuals
                for(int i = 0; i < 3; i++)
                {
                    nextNode = nextConnectingRoad->value->getNodes()->nextEntry();
                    iterCount++;
                }
            }
            delete(nextNode);

            nextConnectingRoad = nextInt->value->getRoads()->nextEntry();
        }
        delete(nextConnectingRoad);

        nextInt = this->intersections->nextEntry();
    }
    delete(nextInt);
    fclose(csv);
}

std::vector<long int> City::getRouteDataLabels()
{
    return this->routeDataLabels;
}
```

```
} /* namespace PredictivePowertrain */
```

12.9 City.h

```
/*
 * CityObj.h
 *
 *  Created on: Jan 8, 2016
 *      Author: Amanda
 */

#ifndef CITY_CITY_H_
#define CITY_CITY_H_
#define  max_slope_percent 6

#include <string>
#include <map>

#include "Intersection.h"
#include "Road.h"
#include "../map/GenericMap.h"
#include "../driver_prediction/Link.h"
#include "../route_prediction/Route.h"
#include "../speed_prediction/SpeedPrediction.h"
#include "../data_management/Bounds.h"

#include <assert.h>
#include <algorithm>
#include <stdlib.h>
#include <climits>
#include <math.h>
#include <ctime>
#include <vector>
#include <float.h>

namespace PredictivePowertrain {

class GPS;              // forward declaration
class Route;            // forward declaration
class SpeedPrediction;  // forward declaration

class City {

private:
      Link* link;
      int intervalDistance = 1;
      int dateTimeCreated;
      double maxSlopePercent = 6;
      GenericMap<long int, Road*>* roads;
      GenericMap<long int, Intersection*>* intersections;
      GenericMap<int, Bounds*>* boundsMap;
```

```
      Road* getConnectingRoad(Intersection* one, Intersection* two);

    std::vector<long int> routeDataLabels;

public:
      City();
      City(GenericMap<long int, Intersection*>* intersections, GenericMap<long int, Road*>* roads, GenericMap<int, Bounds*>* boundsMap) ;
    virtual ~City();
      int getRoadMapSize();
      int getInstersectionMapSize();
      int getBoundsMapSize();
      GenericMap<long int, Link*>* getNextLinks(Link* link);
      Intersection* getIntersectionFromLink(Link* link, bool isIntersection);
    Route* randomPath(Intersection* startInt, Route* initialRoute, int totalLength, std::vector<float>* conditions);
      Intersection* getIntersection(long int intersectionNum);
    Route* getPath(Intersection* start, Intersection* end, std::vector<float>* conditions, int fastest);
    std::pair<std::vector<float>, std::vector<float>> getData(Road* road, int direction, SpeedPrediction* sp, Eigen::MatrixXd* spdIn, float
dist);
    std::pair<std::vector<float>, std::vector<float>> routeToData(Route* route, float dist, SpeedPrediction* sp, Eigen::MatrixXd* spdIn);
      GenericMap<int, Bounds*>* getBoundsMap();
      GenericMap<long int, Road*>* getRoads();
      GenericMap<long int, Intersection*>* getIntersections();
    bool legalRoute(Route* route);
    void addRoad(Road* road);
    void addBounds(Bounds* bounds);
    void addIntersection(Intersection* intersection);
    Route* getRouteFromGPSTrace(GenericMap<long int, std::pair<double, double>*>* trace);
    void printIntersectionsAndRoads();
    bool roadIsOnTrace(Road* road, GenericMap<long int, std::pair<double, double>*>* trace, float thresh);
    std::vector<long int> getRouteDataLabels();

};

} /* namespace PredictivePowertrain */

#endif /* CITY_CITY_H_ */
```

12.10 CityUnitTest.cpp

```
/*
 * City.cpp
 *
 *  Created on: Apr 19, 2016
 *      Author: vagrant
 */

#include "UnitTests.h"
#include "../city/City.h"
#include "../data_management/DataManagement.h"
#include "../route_prediction/Route.h"
#include "../route_prediction/RoutePrediction.h"
```

```
#include "../driver_prediction/DriverPrediction.h"

#include <assert.h>
#include <chrono>
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>

#include <utility>
#include <cmath>
#include <math.h>

#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <vector>

using namespace PredictivePowertrain;

void city_ut()
{
    DataManagement dm;

    // print city
    City* city = dm.getCityData();
    city->printIntersectionsAndRoads();

    GenericMap<long int, std::pair<double, double>*>* trace = dm.getMostRecentTripData();
    Route* route = city->getRouteFromGPSTrace(trace);
    Route* route2 = route->copy();

    Intersection* startIntersection = city->getIntersectionFromLink(route->getLinks()->getEntry(0), true);

    std::vector<float>* conditions = new std::vector<float>(1);
    conditions->at(0) = -1;

    // get route prediction
    RoutePrediction* rp = dm.getRoutePredictionData();
    rp->addCity(city);

    std::ifstream input("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/DP_ACTUAL_SPEED_FUEL_FLOW.csv");

    std::string num;
```

```
    std::vector<float> actualSpds;

    // read in speed from csv
    while(1)
    {
        std::getline(input, num, ',');
        std::stringstream fs(num);
        float f = 0.0;
        fs >> f;

        if(f == -1)
        {
            break;
        }

        actualSpds.push_back(f);
    }

    DriverPrediction dp(rp);
    dp.parseRoute(route, &actualSpds, trace);

    dm.addRoutePredictionData(dp.getRP());

    FILE* csvRoute = std::fopen("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/DP_ACTUAL_ROUTE.csv", "w");
    route->saveRoute2CSV(csvRoute, city, true);

    // test rp across generated route
    std::cout << " ---------------------- original rp ---------------------- " << std::endl;

    int predIter = 1;
    Route* predRoute = rp->startPrediction(route->getLinks()->getEntry(0), startIntersection, conditions);
    while(route->getLinkSize() > 2)
    {
        std::cout << "--- route prediction iteration " << predIter << " ---" << std::endl;

        // update actual route as it's 'driven' over
        route->removeFirstLink();

        // print actual route
        route->printLinks();

        // print predicted route
        predRoute->printLinks();

        // predict route
        predRoute = rp->predict(route->getLinks()->getEntry(0));

        // check if predicted route is actual route
//        if(route->isEqual(predRoute))
//        {
```

```
//            std::cout << "predicted routed before reaching end destination!" << std::endl;
//            std::cout << "Links traversed: " << predIter << std::endl;
//            std::cout << "Links in route: " << route->getLinkSize() << std::endl;
//            break;
//        }

        predIter++;

    }

    // test rp across stored route
    std::cout << " ---------------------- stored rp ---------------------- " << std::endl;
    RoutePrediction* rp2 = dm.getRoutePredictionData();
    rp2->addCity(city);

    int predIter2 = 1;
    Route* predRoute2 = rp2->startPrediction(route2->getLinks()->getEntry(0), startIntersection, conditions);
    while(route2->getLinkSize() > 2)
    {
        std::cout << "--- route prediction iteration " << predIter2 << " ---" << std::endl;

        // update actual route as it's 'driven' over
        route2->removeFirstLink();

        // print actual route
        route2->printLinks();

        // print predicted route
        predRoute2->printLinks();

        // predict route
        predRoute2 = rp2->predict(route2->getLinks()->getEntry(0));

        // check if predicted route is actual route
//        if(route->isEqual(predRoute))
//        {
//            std::cout << "predicted routed before reaching end destination!" << std::endl;
//            std::cout << "Links traversed: " << predIter << std::endl;
//            std::cout << "Links in route: " << route->getLinkSize() << std::endl;
//            break;
//        }

        predIter2++;
    }

    int test = 2;


}
```

12.11 BuildCity.cpp

```
/*
 * BuildCity.cpp
 *
 *  Created on: Apr 4, 2016
 *      Author: vagrant
 */

#include "BuildCity.h"

namespace PredictivePowertrain {

BuildCity::BuildCity()
{

}

BuildCity::~BuildCity()
{
    if(this->newBoundsFound)
    {
        delete(this->rawRoads);
    }
}

std::pair<GenericMap<int, Intersection*> *, GenericMap<long int, Road*>*>* BuildCity::parseAdjMat() {
    // TODO yeah uhhh no thanks
    return NULL;
}

void BuildCity::updateGridDataXMLSpline()
{
    if(this->hasNewBounds())
    {
        std::cout << "identifying intersections from XML splines" << std::endl;

        // container of raw intersections
        GenericMap<long int, Intersection*> rawInts;;

        // get map data
        std::pair<DataCollection*, Bounds*>* newMapData = this->setupDataCollection();
        DataCollection* dc = newMapData->first;
        this->newBounds = newMapData->second;

        // parse new map data to gen raw roads
        dc->pullDataXML(this->latCenter, this->lonCenter);
        this->rawRoads = dc->makeRawRoads();

        // gps converter for finding spline ends with close proximity
```

```
GPS converter(this->latCenter, this->lonCenter);

// container of new intersections identified
this->newInts = new GenericMap<long int, Intersection*>();

// copy of raw roads
GenericMap<long int, Road*>* rawRoadsCopy = this->rawRoads->copy();

// ******** FIND SPLINE EVALUATIONS IN CLOSE PROXIMITY ********
this->rawRoads->initializeCounter();
GenericEntry<long int, Road*>* nextRawRoad = this->rawRoads->nextEntry();
while(nextRawRoad != NULL)
{
    std::cout << "***" << nextRawRoad->key << "***" << std::endl;

    // spline of current road
    Eigen::Spline<double,2> currSpline = nextRawRoad->value->getSpline();
    float nextEvalStepSize = this->evalIntervalLength / nextRawRoad->value->getSplineLength();

    // start at a given evaluation point of the next spline and look for nearest other points in the map
    for(double u = 0; u <= 1.0; u += nextEvalStepSize)
    {
        // create a container of close intersections and store values that are within range
        GenericMap<long int, std::pair<double, double>*> closeSplineEvals;

        // get current evaluation point of current spline
        Eigen::Spline<double,2>::PointType currPt = currSpline(u);

        // create a count of interesections to given point
        int intersectNum = 0;

        // loop through all other roads
        rawRoadsCopy->initializeCounter();
        GenericEntry<long int, Road*>* nextOtherRawRoad = rawRoadsCopy->nextEntry();
        while(nextOtherRawRoad != NULL)
        {
            // don't look at the same roads
            if(nextRawRoad->key == nextOtherRawRoad->key)
            {
                nextOtherRawRoad = rawRoadsCopy->nextEntry();
                continue;
            }

            // spline of other road
            Eigen::Spline<double,2> otherSpline = nextOtherRawRoad->value->getSpline();
            double nextOtherEvalStepSize = this->evalIntervalLength / nextOtherRawRoad->value->getSplineLength();

            // check for close proximity evaluation points
            for(double v = 0; v <= 1.0; v += nextOtherEvalStepSize)
            {
```

```
        Eigen::Spline<double,2>::PointType otherPt = otherSpline(v);

        float evalDist = converter.deltaLatLonToXY(currPt(0,0), currPt(1,0), otherPt(0,0), otherPt(1,0));

        if(evalDist < this->evalIntervalLength + 2.0)
        {
            std::cout << nextOtherRawRoad->key << std::endl;
            closeSplineEvals.addEntry(nextOtherRawRoad->key, new std::pair<double, double>(otherPt(0,0), otherPt(1,0)));
        }
    }

    nextOtherRawRoad = rawRoadsCopy->nextEntry();
}
delete(nextOtherRawRoad);

if(closeSplineEvals.getSize() > 0)
{
    double avgLat = currPt(0,0);
    double avgLon = currPt(1,0);

    GenericMap<long int, Road*>* connectingRoads = new GenericMap<long int, Road*>();
    connectingRoads->addEntry(nextRawRoad->key, this->rawRoads->getEntry(nextRawRoad->key));

    long int intID = nextRawRoad->key;

    closeSplineEvals.initializeCounter();
    GenericEntry<long int, std::pair<double, double>*>* nextLatLon = closeSplineEvals.nextEntry();
    while(nextLatLon != NULL)
    {
        avgLat += nextLatLon->value->first;
        avgLon += nextLatLon->value->second;

        if(!connectingRoads->hasEntry(nextLatLon->key))
        {
            connectingRoads->addEntry(nextLatLon->key, this->rawRoads->getEntry(nextLatLon->key));
            intID += nextLatLon->key;
        }

        delete(nextLatLon->value);
        nextLatLon = closeSplineEvals.nextEntry();
    }
    delete(nextLatLon);

    if(!rawInts.hasEntry(intID))
    {
        float cses = closeSplineEvals.getSize() + 1.0;
        rawInts.addEntry(intID, new Intersection(connectingRoads, avgLat/cses, avgLon/cses, -1000, intID));
    }

}
```

```
            }
            nextRawRoad = this->rawRoads->nextEntry();
        }
        delete(nextRawRoad);

        // ******** CLUSTER INTERSECTIONS ********
        std::cout << "clustering raw intersections" << std::endl;
        GenericMap<long int, Intersection*> refinedInts;

        while(rawInts.getSize() != 0)
        {
            Intersection* rawInt = rawInts.getFirstEntry();

            GenericMap<long int, Intersection*> closeInts;
            closeInts.addEntry(rawInt->getIntersectionID(), rawInt);

            // iterate through all raw interestcions and find other intersections in close proximity
            rawInts.initializeCounter();
            GenericEntry<long int, Intersection*>* nextRawInt = rawInts.nextEntry();
            while(nextRawInt != NULL)
            {
                // get other intersection
                Intersection* otherRawInt = nextRawInt->value;

                // don't look at the same raw intersection
                if(closeInts.hasEntry(otherRawInt->getIntersectionID()))
                {
                    nextRawInt = rawInts.nextEntry();
                    continue;
                }

                // iterate through close intersections to see if other raw int is in close proximity to cluster
                closeInts.initializeCounter();
                GenericEntry<long int, Intersection*>* nextCloseInt = closeInts.nextEntry();
                while(nextCloseInt != NULL)
                {
                    float   dist   =   converter.deltaLatLonToXY(nextCloseInt->value->getLat(),   nextCloseInt->value->getLon(),   otherRawInt-
>getLat(), otherRawInt->getLon());

                    // tuned for NE greenlake
                    if(dist < 13.0)
                    {
                        // add close proximity intersection to cluster
                        closeInts.addEntry(otherRawInt->getIntersectionID(), otherRawInt);

                        // re-init iterator to assess cluster against whole set of raw ints
                        rawInts.initializeCounter();

                        // break out since intersection was found to be in cluster
                        break;
```

```
        }

        nextCloseInt = closeInts.nextEntry();
    }
    delete(nextCloseInt);

    nextRawInt = rawInts.nextEntry();
}
delete(nextRawInt);

// iterate through cluster of close ints and average lat / lon and create list of all connecting roads
double avgLat = 0;
double avgLon = 0;

GenericMap<long int, Road*>* connectingRoads = new GenericMap<long int, Road*>();
int intersectionID = 0;

closeInts.initializeCounter();
GenericEntry<long int, Intersection*>* nextCloseInt = closeInts.nextEntry();
while(nextCloseInt != NULL)
{
    // average lat / lon
    avgLat += nextCloseInt->value->getLat();
    avgLon += nextCloseInt->value->getLon();
    intersectionID += nextCloseInt->value->getIntersectionID();

    // add connecting roads to list of all connecting roads
    GenericMap<long int, Road*>* closeIntConnectingRoads = nextCloseInt->value->getRoads();

    closeIntConnectingRoads->initializeCounter();
    GenericEntry<long int, Road*>* nextCloseIntConectingRoad = closeIntConnectingRoads->nextEntry();
    while(nextCloseIntConectingRoad != NULL)
    {
        if(!connectingRoads->hasEntry(nextCloseIntConectingRoad->key))
        {
            connectingRoads->addEntry(nextCloseIntConectingRoad->key, nextCloseIntConectingRoad->value);
        }
        nextCloseIntConectingRoad = closeIntConnectingRoads->nextEntry();
    }
    delete(nextCloseIntConectingRoad);

    // remove close intersections from raw intersections
    rawInts.erase(nextCloseInt->key);

    nextCloseInt = closeInts.nextEntry();
}
delete(nextCloseInt);

// create new intersection with average lat / lon and new connecting road
float cis = closeInts.getSize();
```

```
    refinedInts.addEntry(intersectionID, new Intersection(connectingRoads, avgLat/cis, avgLon/cis, -1000, intersectionID));
}

// ******** TRIM ROAD SECTIONS ********

// pool new raw intersections and existing intersections
GenericMap<long int, Intersection*> allInts;

if(this->city != NULL && this->city->getIntersections() != NULL && this->city->getIntersections()->getSize() > 0)
{
    this->city->getIntersections()->initializeCounter(); // existing intersections
    GenericEntry<long int, Intersection*>* nextCityInt = this->city->getIntersections()->nextEntry();
    while(nextCityInt != NULL)
    {
        allInts.addEntry(nextCityInt->key, nextCityInt->value);
        nextCityInt = this->city->getIntersections()->nextEntry();
    }
    delete(nextCityInt);
}

refinedInts.initializeCounter(); // new intersections
GenericEntry<long int, Intersection*>* nextRefInt = refinedInts.nextEntry();
while(nextRefInt != NULL)
{
    allInts.addEntry(nextRefInt->key, nextRefInt->value);
    nextRefInt = refinedInts.nextEntry();
}
delete(nextRefInt);

// iterate through refined ints to make road connections
refinedInts.initializeCounter();
GenericEntry<long int, Intersection*>* nextInt = refinedInts.nextEntry();
while(nextInt != NULL)
{
    // get connecting roads of intersection
    GenericMap<long int, Road*>* intConnectingRoads = nextInt->value->getRoads();

    // create a container of new atomic roads
    GenericMap<long int, Road*>* newConnectingRoads = new GenericMap<long int, Road*>();

    // iterate through all connecting roads to find atomic road segments between 2 intersections
    intConnectingRoads->initializeCounter();
    GenericEntry<long int, Road*>* nextConnectingRoad = intConnectingRoads->nextEntry();
    while(nextConnectingRoad != NULL)
    {
        // iterate along connecting road spline to find intersections in close proximity
        Eigen::Spline<double,2> spline = nextConnectingRoad->value->getSpline();
        double evalStepSize = this->evalIntervalLength / nextConnectingRoad->value->getSplineLength();

        // create a container of control points for road
```

```
                GenericMap<long int, Node*>* nodes = new GenericMap<long int, Node*>();
                long int splineStartIntID = -1;
                long int splineEndIntID = -1;
                int evalCount = 0;

                // iterate along each connecting road until another intersection is found in close proximity
                for(double u = 0; u <= 1.0; u += evalStepSize)
                {
                    Eigen::Spline<double,2>::PointType pt = spline(u);

                    // find where ends of spline meet intersection <---- mix existing intersection in here!
                    allInts.initializeCounter();
                    GenericEntry<long int, Intersection*>* nextOtherInt = allInts.nextEntry();
                    while(nextOtherInt != NULL)
                    {
                        float   dist   =   converter.deltaLatLonToXY(pt(0,0),   pt(1,0),   nextOtherInt->value->getLat(),   nextOtherInt->value-
>getLon());

                        if(dist < this->evalIntervalLength + 2)
                        {
                            // check to see if intersection already exists on path incase the intersection has 2 close points near spline
evaluation pt
                            bool hasInt = false;
                            nodes->initializeCounter();
                            GenericEntry<long int, Node*>* nextNode = nodes->nextEntry();
                            while(nextNode != NULL)
                            {
                                if(nextNode->value->getID() == nextOtherInt->value->getIntersectionID())
                                {
                                    hasInt = true;
                                    break;
                                }
                                nextNode = nodes->nextEntry();
                            }
                            delete(nextNode);

                            // break if repeated close int if found near same eval pt
                            if(hasInt) { break; }

                            // add intersection
                            if(splineStartIntID == -1)
                            {
                                splineStartIntID = nextOtherInt->value->getIntersectionID();
                            }

                            else
                            {
                                splineEndIntID = nextOtherInt->value->getIntersectionID();
                            }
```

```
        double intLat = nextOtherInt->value->getLat();
        double intLon = nextOtherInt->value->getLon();
        long int intID = nextOtherInt->value->getIntersectionID();

        nodes->addEntry(evalCount, new Node(intLat, intLon, -1000, intID));
        evalCount++;

        break;
    }

    nextOtherInt = allInts.nextEntry();
}
delete(nextOtherInt);

// take measurement of spline evaluation between intersections
if(splineStartIntID != -1 && splineEndIntID == -1)
{
    nodes->addEntry(evalCount, new Node(pt(0,0), pt(1,0), -1000, nextConnectingRoad->key));
    evalCount++;
}

// found road segment between intersection
else if (splineStartIntID != -1 && splineEndIntID != -1)
{
    if(splineStartIntID == splineEndIntID)
    {
        int test = 2;
    }

    // ensure start or end intersection is nextInt
    bool startIntIsNotNextInt = splineStartIntID != nextInt->key;
    bool endIntIsNotNextInt = splineEndIntID != nextInt->key;
    if(startIntIsNotNextInt && endIntIsNotNextInt)
    {
        splineStartIntID = -1;
        splineEndIntID = -1;

        delete(nodes);
        nodes = new GenericMap<long int, Node*>();

        evalCount = 0;
        continue;
    }

    Road* newRoad;

    // check for unique atomic road segmenet
    long int adjacentIntID;
    if(startIntIsNotNextInt)
    {
```

```
                        adjacentIntID = splineStartIntID;
                    }
                    else if(endIntIsNotNextInt)
                    {
                        adjacentIntID = splineEndIntID;
                    }

                    bool foundUniqueRoad = true;
                    GenericMap<long int, Road*>* adjacentIntConnectingRoads = allInts.getEntry(adjacentIntID)->getRoads();

                    adjacentIntConnectingRoads->initializeCounter();
                    GenericEntry<long int, Road*>* nextAdjacentIntConnectingRoad = adjacentIntConnectingRoads->nextEntry();
                    while(nextAdjacentIntConnectingRoad != NULL)
                    {
                        Intersection* connectingRoadStartInt = nextAdjacentIntConnectingRoad->value->getStartIntersection();
                        Intersection* connectingRoadEndInt = nextAdjacentIntConnectingRoad->value->getEndIntersection();

                        if(connectingRoadStartInt == NULL || connectingRoadEndInt == NULL)
                        {
                            nextAdjacentIntConnectingRoad = adjacentIntConnectingRoads->nextEntry();
                            continue;
                        }

                        bool  hasStartInt  =  connectingRoadStartInt->getIntersectionID()  ==  splineStartIntID  ||  connectingRoadEndInt-
>getIntersectionID() == splineStartIntID;
                        bool   hasEndInt   =   connectingRoadStartInt->getIntersectionID()   ==   splineEndIntID   ||   connectingRoadEndInt-
>getIntersectionID() == splineEndIntID;

                        if(hasStartInt && hasEndInt)
                        {
                            newRoad = nextAdjacentIntConnectingRoad->value;
                            foundUniqueRoad = false;
                            break;
                        }

                        nextAdjacentIntConnectingRoad = adjacentIntConnectingRoads->nextEntry();
                    }
                    delete(nextAdjacentIntConnectingRoad);

                    // create new road with shorted spline if road is unique
                    if(foundUniqueRoad)
                    {
                        // create a container of control points to fit spline to
                        Eigen::MatrixXd points(2, nodes->getSize());
                        int latLonCount = 0;

                        // get first segement evaluation to calculate a distance
                        Node* firstNode = nodes->getFirstEntry();
                        double prevLat = firstNode->getLat();
                        double prevLon = firstNode->getLon();
```

```
            double currLat = 0;
            double currLon = 0;
            float dist = 0;

            // iterate along nodes to fit new shortened spline between interesections
            nodes->initializeCounter();
            GenericEntry<long int, Node*>* nextNode = nodes->nextEntry();
            while(nextNode != NULL)
            {
                currLat = nextNode->value->getLat();
                currLon = nextNode->value->getLon();

                // update spline container
                points(0, latLonCount) = currLat;
                points(1, latLonCount) = currLon;
                latLonCount++;

                // update distance
                dist += converter.deltaLatLonToXY(prevLat, prevLon, currLat, currLon);

                prevLat = currLat;
                prevLon = currLon;

                nextNode = nodes->nextEntry();
            }
            delete(nextNode);

            // fit first order spline spline
            typedef Eigen::Spline<double, 2> spline2f;
            spline2f atomicRoadSpline = Eigen::SplineFitting<spline2f>::Interpolate(points, 1);

            // make new road
            newRoad = new Road(nextConnectingRoad->value->getRoadType(), splineStartIntID + splineEndIntID, nodes);
            newRoad->assignSpline(atomicRoadSpline);
            newRoad->assignSplineLength(dist);
            newRoad->setMinMaxLatLon();
            newRoad->setStartIntersection(allInts.getEntry(splineStartIntID));
            newRoad->setEndIntersection(allInts.getEntry(splineEndIntID));
            newRoad->setBoundsID(this->newBounds->getID());
        }

        newConnectingRoads->addEntry(newRoad->getRoadID(), newRoad);

        // attempt to grab another connecting road from the same raw road
        splineStartIntID = -1;
        splineEndIntID = -1;
        nodes = new GenericMap<long int, Node*>();
        evalCount = 0;
    }
}
```

```
                nextConnectingRoad = intConnectingRoads->nextEntry();
            }
            delete(nextConnectingRoad);

            // replace connecting roads
            nextInt->value->replaceRoads(newConnectingRoads);

            // set bounds id of intersection
            nextInt->value->setBoundsID(this->newBounds->getID());

            // add intersection to new bounds
            this->newInts->addEntry(nextInt->key, nextInt->value);
            nextInt = refinedInts.nextEntry();
        }
        delete(nextInt);

        // add elevation
        std::cout << "adding elevations to spline control points" << std::endl;
        this->newInts->initializeCounter();
        GenericEntry<long int, Intersection*>* nextNewInt = this->newInts->nextEntry();
        while(nextNewInt != NULL)
        {
            dc->updateElevationData(nextNewInt->value->getRoads());
            nextNewInt = this->newInts->nextEntry();
        }
        delete(nextNewInt);
    }
}

void BuildCity::printNewIntersectionsAndRoads()
{
    if(this->newBoundsFound && this->newInts->getSize() > 0)
    {
        // csv name
        std::string csvName = "/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/parsedMapData.csv";

        // delete existing csv if found
        std::string rm = "rm " + csvName;
        system(rm.c_str());

        // create new csv
        FILE* csv;
        csv = std::fopen(csvName.c_str(), "w");

        // add header to csv
        fprintf(csv, "name,icon,description,color,latitude,longitude\n");

        std::cout << "**** printing intersection lat/lon ****" << std::endl;
        this->newInts->initializeCounter();
```

```
GenericEntry<long int, Intersection*>* nextInt = this->newInts->nextEntry();
while(nextInt != NULL)
{
    // print intersection lat/lon to console
    printf("%.12f,%.12f\n", nextInt->value->getLat(), nextInt->value->getLon());

    // print intersection lat/lon to csv
    fprintf(csv, "%ld,", nextInt->value->getIntersectionID());
    fprintf(csv, "googlemini,");
    fprintf(csv, "Int ID: %ld | ", nextInt->value->getIntersectionID());
    fprintf(csv, "Int Ele: %f | ", nextInt->value->getElevation());
    fprintf(csv, "Lat & Lon: %.12f %.12f | ", nextInt->value->getLat(), nextInt->value->getLon());
    fprintf(csv, "Connecting Roads: ");

    nextInt->value->getRoads()->initializeCounter();
    GenericEntry<long int, Road*>* nextConnectingRoad = nextInt->value->getRoads()->nextEntry();
    while(nextConnectingRoad != NULL)
    {
        fprintf(csv, "%ld ", nextConnectingRoad->key);
        nextConnectingRoad = nextInt->value->getRoads()->nextEntry();
    }
    delete(nextConnectingRoad);

    fprintf(csv, ",");
    fprintf(csv, "red,");
    fprintf(csv, "%.12f,%.12f\n", nextInt->value->getLat(), nextInt->value->getLon());

    // print road spline to csv
    nextInt->value->getRoads()->initializeCounter();
    nextConnectingRoad = nextInt->value->getRoads()->nextEntry();
    while(nextConnectingRoad != NULL)
    {
        // iterate along connecting road spline control points
        nextConnectingRoad->value->getNodes()->initializeCounter();
        GenericEntry<long int, Node*>* nextNode = nextConnectingRoad->value->getNodes()->nextEntry();

        // burn a node so they are not superimposed on intersection
        nextNode = nextConnectingRoad->value->getNodes()->nextEntry();

        int iterCount = 1;
        while(iterCount < nextConnectingRoad->value->getNodes()->getSize() - 1)
        {
            fprintf(csv, "%ld,", nextConnectingRoad->value->getRoadID());
            fprintf(csv, "cirlce,");
            fprintf(csv, "Road ID: %ld | ", nextConnectingRoad->value->getRoadID());
            fprintf(csv, "Start Int ID: %ld | ", nextConnectingRoad->value->getStartIntersection()->getIntersectionID());
            fprintf(csv, "Node Ele: %f | ", nextNode->value->getEle());
            fprintf(csv, "End Int ID: %ld | ", nextConnectingRoad->value->getEndIntersection()->getIntersectionID());
            fprintf(csv, "Lat & Lon: %.12f %.12f,", nextNode->value->getLat(), nextNode->value->getLon());
            fprintf(csv, "blue,");
```

```
                    fprintf(csv, "%.12f,%.12f\n", nextNode->value->getLat(), nextNode->value->getLon());

                    nextNode = nextConnectingRoad->value->getNodes()->nextEntry();
                    iterCount++;
                }
                delete(nextNode);

                nextConnectingRoad = nextInt->value->getRoads()->nextEntry();
            }
            delete(nextConnectingRoad);

            nextInt = this->newInts->nextEntry();
        }
        delete(nextInt);
        fclose(csv);
    }
}

void BuildCity::updateGridDataXMLAdj() {

    if(this->hasNewBounds())
    {
        std::cout << "identifying intersections from XML Adjacency Matrix" << std::endl;
        std::pair<DataCollection*, Bounds*>* newMapData = this->setupDataCollection();
        DataCollection* dc = newMapData->first;
        this->newBounds = newMapData->second;

        dc->pullDataXML(this->latCenter, this->lonCenter);
        this->rawRoads = dc->makeRawRoads();

        int latRowsSpline = this->latDelta/this->adjMatPrecFromSplines;
        int lonColsSpline = this->lonDelta/this->adjMatPrecFromSplines;
        this->adjMatFromSplines = Eigen::MatrixXd::Zero(latRowsSpline, lonColsSpline);

        this->rawRoads->initializeCounter();
        GenericEntry<long int, Road*>* nextRawRoad = this->rawRoads->nextEntry();
        while(nextRawRoad != NULL)
        {
            // adjacent matrix of splines
            std::cout << "---- new road: " << nextRawRoad->key << " ----" << std::endl;

            bool splineWithinNodes = false;
            int latLonCount = 1;

            GenericMap<long int, Node*>* nodes = nextRawRoad->value->getNodes()->copy();
            GenericMap<int, std::pair<int, int>*>* adjMatIndicies = new GenericMap<int, std::pair<int, int>*>();


            Eigen::Spline<double,2> spline = nextRawRoad->value->getSpline();
            for(double u = 0; u <= 1; u += this->splineStep)
```

```
    {
        Eigen::Spline<double,2>::PointType point = spline(u);

        double newLat = point(0,0);
        double newLon = point(1,0);

        nodes->initializeCounter();
        GenericEntry<long int, Node*>* nextNode = nodes->nextEntry();
        while(nextNode != NULL)
        {
            // make sure spline is within node bounds
            if(       newLat + this->gpsTolerance > nextNode->value->getLat() &&
                newLat - this->gpsTolerance < nextNode->value->getLat() &&
                newLon + this->gpsTolerance > nextNode->value->getLon() &&
                newLon - this->gpsTolerance < nextNode->value->getLon() )
            {
                splineWithinNodes = true;
                nodes->erase(nextNode->key);
                break;
            }
            nextNode = nodes->nextEntry();
        }
        delete(nextNode);

        // hop out of for loop if all nodes have been iterated over
        if(nodes->getSize() == 0) { break; }

        // add spline data to adjacency matrix if spline within node bounds
        if(splineWithinNodes)
        {
            int latRow = this->adjMatFromSplines.rows() - (newLat - this->minLat) / this->latDelta * latRowsSpline;
            int lonCol = (newLon - this->minLon) / this->lonDelta * lonColsSpline;

            if(latRow >= 0 && latRow < this->adjMatFromSplines.rows() && lonCol >= 0 && lonCol < this->adjMatFromSplines.cols())
            {
                std::cout << u << "\tlat: " << (double)point(0,0) << "\tlon: " << (double)point(1,0) << std::endl;
                adjMatIndicies->addEntry(latLonCount++, new std::pair<int, int>(latRow, lonCol));
                this->adjMatFromSplines(latRow, lonCol) = this->scaleID(nextRawRoad->key);
            }
        }
    }
    nextRawRoad->value->assignAdjMatIndicies(adjMatIndicies);
    nextRawRoad = this->rawRoads->nextEntry();
}

// fill in holes in the adjMat
this->connectifyAjdMat();

// parse out nodes
std::pair<GenericMap<int, Intersection*>*, GenericMap<long int, Road*>*>* parsedData = this->parseAdjMat();
```

```
    }
}

void BuildCity::printAdjMats() {
    std::cout << "printing adjacency matrix from build city" << std::endl;
    if(this->hasNewBounds())
    {
        std::string csvName = "/Users/Brian/Desktop/ecocar/Git/predictive_thermo_controller/data/   adjMatSpline.csv";

        // remove old csv
        std::string rm = "rm " + csvName;
        system(rm.c_str());

        // populate new csv
        std::ofstream csv;
        csv.open(csvName.c_str(), std::fstream::in | std::fstream::out | std::fstream::app);
        for(int row = 0; row < this->adjMatFromSplines.rows(); row++)
        {
            for(int col = 0; col < this->adjMatFromSplines.cols(); col++)
            {
                csv << this->unScaleID((double)this->adjMatFromSplines(row, col)) << ",";
            }
            csv << "\n";
        }
        csv.close();
    } else {
        std::cout << "no new bounds" << std::endl;
    }
}

void BuildCity::connectifyAjdMat() {
    this->rawRoads->initializeCounter();
    GenericEntry<long int, Road*>* nextRawRoad = this->rawRoads->nextEntry();
    while(nextRawRoad != NULL)
    {
        GenericMap<int, std::pair<int, int>*>* currIndicies = nextRawRoad->value->getAdjMatIndicies();

        currIndicies->initializeCounter();
        GenericEntry<int, std::pair<int, int>*>* currIdx = currIndicies->nextEntry();
        GenericEntry<int, std::pair<int, int>*>* nextIdx = currIndicies->nextEntry();

        while(nextIdx != NULL)
        {
            if(!this->isAdj(currIdx, nextIdx))
            {
                int fillCount = 0;
                GenericEntry<int, std::pair<int, int>*>* fillIdx = new GenericEntry<int, std::pair<int, int>*>(1, currIdx->value);
                while(!this->isAdj(fillIdx, nextIdx))
                {
                    // adjust x
```

```
                    int currX = fillIdx->value->first;
                    int currY = fillIdx->value->second;
                    int nextX = nextIdx->value->first;
                    int nextY = nextIdx->value->second;

                    if(currX < nextX)
                    {
                        fillIdx->value->first += 1;
                    } else if(currX < nextX) {
                        fillIdx->value->first -= 1;
                    }

                    // adjust y
                    if(currY < nextY)
                    {
                        fillIdx->value->second += 1;
                    } else if(currY > nextY) {
                        fillIdx->value->second -= 1;
                    }
                    this->adjMatFromSplines(fillIdx->value->first, fillIdx->value->second) = this->scaleID(nextRawRoad->key);

                    // incase going to HAM connectifying
                    if(fillCount++ > 100)
                    {
                        break;
                    }
                }
            }
            currIdx = nextIdx;
            nextIdx = currIndicies->nextEntry();
        }
        delete(currIdx);
        delete(nextIdx);

        nextRawRoad = this->rawRoads->nextEntry();
    }
    delete(nextRawRoad);
}

bool BuildCity::isAdj(GenericEntry<int, std::pair<int, int>*>* idx1, GenericEntry<int, std::pair<int, int>*>* idx2) {
    int x1 = idx1->value->first;
    int y1 = idx1->value->second;
    int x2 = idx2->value->first;
    int y2 = idx2->value->second;

    return std::abs(x1-x2) <= 1 && std::abs(y1-y2) <= 1;
}

double BuildCity::scaleID(long int id) {
    return id / this->idScalar;
```

```
}

long int BuildCity::unScaleID(double id) {
    return id * this->idScalar;
}

void BuildCity::updateGridDataPNG() {
    if(this->hasNewBounds())
    {
        std::cout << "identifying intersections from PNG" << std::endl;
        std::pair<DataCollection*, Bounds*>* newMapData = this->setupDataCollection();
        DataCollection* dc = newMapData->first;
        this->newBounds = newMapData->second;

        std::pair<int, cv::Mat>* newGridData = pullAndFormatMapPNG(dc);
        int zoomIdx = newGridData->first;
        cv::Mat map = newGridData->second;

        GenericMap<int, cv::Point*>* rawInts = getIntersectionPointsFromMapPNG(map, zoomIdx);

        std::pair<GenericMap<int, Road*>*, GenericMap<int, Intersection*>*>* roadsAndInts = makeRoadsAndIntersections(rawInts, map);

        // Display the detected hough lines
        cv::imshow("incoming road dots", map);
        cv::imwrite("/Users/Brian/Desktop/misc/kernelDev.png", map);
    }
}

std::pair<GenericMap<int,  Road*>*,  GenericMap<int,  Intersection*>*>*  BuildCity::makeRoadsAndIntersections(GenericMap<int,  cv::Point*>*
rawInts, cv::Mat map) {

    GenericMap<int, Intersection*>* intersections = new GenericMap<int, Intersection*>();
    GenericMap<int, Road*>* roads = new GenericMap<int, Road*>();
    int searchKernelSideDim = 700;

    rawInts->initializeCounter();
    cv::Point* nextCentralIntPoint = rawInts->getFirstEntry();
    while(nextCentralIntPoint != NULL)
    {

        // get find all intersection points in close proximity to current intersection
        int startRow = nextCentralIntPoint->x - .5*searchKernelSideDim;
        int endRow = nextCentralIntPoint->x + .5*searchKernelSideDim;
        int startCol = nextCentralIntPoint->y -.5*searchKernelSideDim;
        int endCol = nextCentralIntPoint->y + .5*searchKernelSideDim;

        GenericMap<int, double>* closeIntPnts = new GenericMap<int, double>();

        for(int row = startRow; row < endRow; row++)
        {
```

```
            for(int col = startCol; col < endCol; col++)
            {
                int key = hashCoords(col, row);
                if(rawInts->hasEntry(key))
                {
                    double distance = sqrt(pow(nextCentralIntPoint->x - col, 2) + pow(nextCentralIntPoint->y - row, 2));
                    closeIntPnts->addEntry(key, distance);
                }
            }
        }

        GenericMap<int, cv::Point*> searchedIntPnts;

        while(true)
        {
            GenericEntry<int, double>* nextClosestIntPntData = closeIntPnts->getMinEntry();
            cv::Point* nextClosestIntPnt = rawInts->getEntry(nextClosestIntPntData->key);

            // trace out to closest point
            std::pair<bool,   GenericMap<int,   cv::Point*>*>*   roadTraceData   =   traceRoad(nextClosestIntPnt,   nextCentralIntPoint,   map,
searchedIntPnts);

        }

        nextCentralIntPoint = rawInts->getFirstEntry();
    }


    return NULL;
}

std::pair<bool, GenericMap<int, cv::Point*>*>* BuildCity::traceRoad(cv::Point* nextClosestPnt, cv::Point* pnt, cv::Mat map, GenericMap<int,
cv::Point*>& searchedIntPnts) {

    bool isPreTraveledRoad = false;
    GenericMap<int, cv::Point*>* roadPntTrace = new GenericMap<int, cv::Point*>();

    for(int deg = 0; deg < 360; deg++)
    {

    }

    return new std::pair<bool, GenericMap<int, cv::Point*>*>;
}

GenericMap<int, std::pair<cv::Point*, cv::Point*>*>* BuildCity::perimeterScanKernelForRoads(cv::Mat kernel) {

    GenericMap<int, cv::Point*> points;

    int lastPixel = kernel.at<uchar>(0, 0);
```

```
    // top edge
    std::cout << "top edge" << std::endl;
    for(int i = 1; i < kernel.cols; i++)
    {
        checkNextPixel(i, 0, points, kernel, lastPixel);
    }

    // right edge
    std::cout << "right edge" << std::endl;
    for(int i = 0; i < kernel.rows; i++)
    {
        checkNextPixel(kernel.cols-1, i, points, kernel, lastPixel);
    }

    // bottom edge
    std::cout << "bottom edge" << std::endl;
    for(int i = kernel.cols-1; i >= 0; i--)
    {
        checkNextPixel(i, kernel.rows-1, points, kernel, lastPixel);
    }

    // left edge
    std::cout << "left edge" << std::endl;
    for(int i = kernel.rows-1; i >= 0; i--)
    {
        checkNextPixel(0, i, points, kernel, lastPixel);
    }

    if(points.getSize() > 0)
    {
        cv::imshow("kernel", kernel);
    }
    return NULL;
}

void BuildCity::checkNextPixel(int x, int y, GenericMap<int, cv::Point*>& points, cv::Mat& kernel, int& lastPixel){
    int nextPixel = kernel.at<uchar>(x, y);

    std::cout << lastPixel << " " << nextPixel << std::endl;

    // found difference
    if(nextPixel != lastPixel)
    {
        cv::Point* newPt = new cv::Point(x, y);
        drawPoint(kernel, *newPt);
        points.addEntry(points.getSize()+1, newPt);
    }
    lastPixel = nextPixel;
}
```

```
// Draw the detected intersection point on an image
void BuildCity::drawPoint(cv::Mat &image, cv::Point point) {
    int rad = 7;
    cv::Scalar color = cv::Scalar(255,255,255); // line color
    cv::circle( image, point, rad, color, -1, 8 );
}

BuildCity::zoomParams BuildCity::getZoomParams(int zoom) {

    zoomParams newZoomParams;

    switch(zoom)
    {
        case 19 :
            newZoomParams.kernelSideDim = 175;
            newZoomParams.coordBinResolution = 6;
            newZoomParams.maxLines = 200;
            newZoomParams.maxClusterPntDistance = 25;
            break;
        case 18 :
            newZoomParams.kernelSideDim = 150;
            newZoomParams.coordBinResolution = 6;
            newZoomParams.maxLines = 200;
            newZoomParams.maxClusterPntDistance = 20;
            break;
        default :
            std::cout << "no zoom params for " << zoom << std::endl;
    }

    newZoomParams.angleThreshold = 30;
    newZoomParams.pntThreshold = .75*newZoomParams.kernelSideDim;
    newZoomParams.imageProcessingResolution = .20*newZoomParams.kernelSideDim;

    return newZoomParams;
}

GenericMap<int, cv::Point*>* BuildCity::getIntersectionPointsFromMapPNG(cv::Mat map, int zoom) {

    GenericMap<int, cv::Point*> rawIntersectionPoints;

    zoomParams zp = getZoomParams(zoom);

    for(int row = 0; row < map.rows; row += zp.imageProcessingResolution)
    {
        for(int col = 0; col < map.cols; col += zp.imageProcessingResolution)
        {
            if((row + zp.kernelSideDim) < map.rows && (col + zp.kernelSideDim) < map.cols)
            {
                cv::Rect roi(col, row, zp.kernelSideDim, zp.kernelSideDim);
```

```
                cv::Mat kernel = map(roi);

                std::vector<cv::Vec2f> lines;
                cv::HoughLines(kernel, lines, 1, CV_PI/180, zp.pntThreshold, 0, 0 );

                // Draw the hough lines
                std::vector<cv::Vec2f>::const_iterator it1 = lines.begin();
                if(lines.size() < zp.maxLines)
                {
                    int xCount[zp.kernelSideDim];
                    int yCount[zp.kernelSideDim];

                    std::fill_n(xCount, zp.kernelSideDim, 0);
                    std::fill_n(yCount, zp.kernelSideDim, 0);

                    while(it1 != lines.end())
                    {
                        std::pair<cv::Point, cv::Point>* linePoints1 = this->polarToCartisian((*it1)[0], (*it1)[1], kernel.rows);

                        std::vector<cv::Vec2f>::const_iterator it2 = lines.begin();
                        while(it2 != lines.end())
                        {
                            std::pair<cv::Point, cv::Point>* linePoints2 = this->polarToCartisian((*it2)[0], (*it2)[1], kernel.rows);

                            double lineAngle = std::abs((*it1)[1] - (*it2)[1])*180/CV_PI;
                            if(lineAngle > zp.angleThreshold && lineAngle < 180 - zp.angleThreshold) {

                                cv::Point intPnt;
                                bool  linesIntersect  =  getIntersectionPoint(linePoints1->first,  linePoints1->second,  linePoints2->first,
linePoints2->second, intPnt);

                                if(linesIntersect && intPnt.x < zp.kernelSideDim && intPnt.y < zp.kernelSideDim && intPnt.x >= 0 && intPnt.y
>= 0)
                                {
                                    xCount[intPnt.x] += 1;
                                    yCount[intPnt.y] += 1;
                                }
                            }
                            ++it2;
                        }
                        ++it1;
                    }

                    cv::Point* intersectPnt = new cv::Point();
                    intersectPnt->x = getCoord(xCount, zp.kernelSideDim, zp.coordBinResolution);
                    intersectPnt->y = getCoord(yCount, zp.kernelSideDim, zp.coordBinResolution);

                    if(intersectPnt->x != -1)
                    {
                        intersectPnt->x += col;
```

```
                        intersectPnt->y += row;

                        rawIntersectionPoints.addEntry(hashCoords(intersectPnt->x, intersectPnt->y), intersectPnt);

                    } else {
                        delete(intersectPnt);
                    }
                }
            }
        }
    }

    // pick best point from cluster
    GenericMap<int, cv::Point*>* intersections = new GenericMap<int, cv::Point*>();

    cv::Point* nextRawIntersectionPoint = rawIntersectionPoints.getFirstEntry();
    while(nextRawIntersectionPoint != NULL)
    {
        GenericMap<int, cv::Point*> intersectionPointCluster;

        int key = hashCoords(nextRawIntersectionPoint->x, nextRawIntersectionPoint->y);
        intersectionPointCluster.addEntry(key, nextRawIntersectionPoint);
        rawIntersectionPoints.erase(key);

        declusterIntersectionPoints(nextRawIntersectionPoint,    rawIntersectionPoints,    intersectionPointCluster,    .25*zp.kernelSideDim,
zp.maxClusterPntDistance);

        // average point locations in cluster and plop new intersection point
        if(intersectionPointCluster.getSize() > 1)
        {
            int sumX = 0;
            int sumY = 0;

            intersectionPointCluster.initializeCounter();
            GenericEntry<int, cv::Point*>* nextIntPnt = intersectionPointCluster.nextEntry();
            while(nextIntPnt != NULL)
            {
                sumX += nextIntPnt->value->x;
                sumY += nextIntPnt->value->y;

                nextIntPnt = intersectionPointCluster.nextEntry();
            }

            int avgX = sumX / intersectionPointCluster.getSize();
            int avgY = sumY / intersectionPointCluster.getSize();

            cv::Point* intersection = new cv::Point(avgX, avgY);
            intersections->addEntry(hashCoords(avgX, avgY), intersection);

            cv::circle(map, *intersection, 3, cv::Scalar(0,0,0));
```

```
        }
        nextRawIntersectionPoint = rawIntersectionPoints.getFirstEntry();
    }

    return intersections;
}

void  BuildCity::declusterIntersectionPoints(cv::Point*  rawIntPnt,  GenericMap<int,  cv::Point*>&  rawIntPnts,  GenericMap<int,  cv::Point*>&
intPnts, int kernelSideDim, double maxDistance)
{
    int rawIntPntX = rawIntPnt->x;
    int rawIntPntY = rawIntPnt->y;

    int rowStart = rawIntPntY - .5*kernelSideDim;
    int rowEnd = rawIntPntY + .5*kernelSideDim;
    int colStart = rawIntPntX - .5*kernelSideDim;
    int colEnd = rawIntPntX + .5*kernelSideDim;

    for(int row = rowStart; row < rowEnd; row++)
    {
        for(int col = colStart; col < colEnd; col++)
        {
            int key = hashCoords(col, row);
            if(rawIntPnts.hasEntry(key))
            {
                double distance = sqrt(pow(rawIntPntX - col, 2) + pow(rawIntPntY - row, 2));
                if(distance < maxDistance)
                {
                    cv::Point* closeRawIntPnt = rawIntPnts.getEntry(key);

                    rawIntPnts.erase(key);
                    intPnts.addEntry(key, closeRawIntPnt);
                    declusterIntersectionPoints(closeRawIntPnt, rawIntPnts, intPnts, kernelSideDim, maxDistance);
                }
            }
        }
    }
}

int BuildCity::hashCoords(int x, int y) {
    return y * 6000 + x; // assign pixel number counting left to right, top to bottom
}

int BuildCity::getCoord(int* dimCount, int dim, int tol) {
    int i, coordCount = 0, max = 0, coord = -1, itr = 0;
    for(i = 0; i < dim; i++) {
        if(itr == tol) {
            if(coordCount > max) {
                max = coordCount;
                coord = i + tol / 2;
```

```
            }
            coordCount = 0;
            itr = 0;
        } else if(dimCount[i] != 0) {
            int c = dimCount[i];
            coordCount += c;
        }
        itr++;
    }
    return coord;
}

// get intersection points
template<typename K>
bool BuildCity::getIntersectionPoint(cv::Point_<K> a1, cv::Point_<K> a2, cv::Point_<K> b1, cv::Point_<K> b2, cv::Point_<K> & intPnt) {
    cv::Point_<K> p = a1;
    cv::Point_<K> q = b1;
    cv::Point_<K> r(a2-a1);
    cv::Point_<K> s(b2-b1);

    // check for crossage
    double crossage = cross(r,s);
    if(crossage < .00000001)
    {
        return false;
    }

    double t = cross(q-p,s)/cross(r,s);

    intPnt = p + t*r;
    return true;
}

template<typename K>
double BuildCity::cross(cv::Point_<K> v1, cv::Point_<K> v2) {
    return v1.x*v2.y - v1.y*v2.x;
}

std::pair<cv::Point, cv::Point>* BuildCity::polarToCartisian(float mag, float angle, int rows) {
    cv::Point pt1(mag / cos(angle), 0);
    cv::Point pt2((mag - rows * sin(angle)) / cos(angle), rows);
    return new std::pair<cv::Point, cv::Point>(pt1, pt2);
}

std::pair<int, cv::Mat>* BuildCity::pullAndFormatMapPNG(DataCollection* dc) {

    // pull in image
    int zoomIdx = dc->pullDataPNG(this->latCenter, this->lonCenter);
    std::string dataFolder = dc->getDataFolder();
    std::string mapPNGName = dc->getMapPNGName();
```

```
    std::string mapPicLoc = dataFolder + "/" + mapPNGName + "-full.png";

    // get raw image
    cv::Mat mapImage = cv::imread(mapPicLoc, CV_LOAD_IMAGE_COLOR);

    // trim raw image
    cv::Rect roi(350, 55, 5650, 5945);
    cv::Mat croppedMapImage = mapImage(roi);

    // red highway color filter
    cv::Mat redMask;
    cv::Scalar redLB = cv::Scalar(80, 65, 190);
    cv::Scalar redUB = cv::Scalar(170, 147, 251);
    cv::inRange(croppedMapImage, redLB, redUB, redMask);

    // yellow access road color filter
    cv::Mat yellowMask;
    cv::Scalar yellowLB = cv::Scalar(107, 220, 210);
    cv::Scalar yellowUB = cv::Scalar(205, 255, 255);
    cv::inRange(croppedMapImage, yellowLB, yellowUB, yellowMask);

    // white residential road color filter
    cv::Mat whiteMask;
    cv::Scalar whiteLB = cv::Scalar(250, 250, 250);
    cv::Scalar whiteUB = cv::Scalar(255, 255, 255);
    cv::inRange(croppedMapImage, whiteLB, whiteUB, whiteMask);

    // orange small highways color filter
    cv::Mat orangeMask;
    cv::Scalar orangeLB = cv::Scalar(120, 180, 220);
    cv::Scalar orangeUB = cv::Scalar(175, 230, 255);
    cv::inRange(croppedMapImage, orangeLB, orangeUB, orangeMask);

    cv::imwrite(dataFolder + "/test.png", redMask + yellowMask + whiteMask + orangeMask);
    return new std::pair<int, cv::Mat>(zoomIdx, redMask + yellowMask + whiteMask + orangeMask);
}

std::pair<DataCollection*, Bounds*>* BuildCity::setupDataCollection() {

    Bounds* newBoundsFromTrip = new Bounds(this->maxLat, this->minLat, this->maxLon, this->minLon);
    newBoundsFromTrip->assignID(this->boundsID+1);

    this->latCenter = (this->maxLat + this->minLat) / 2.0;
    this->lonCenter = (this->maxLon + this->minLon) / 2.0;
    this->latDelta = this->maxLat - this->minLat;
    this->lonDelta = this->maxLon - this->minLon;

    DataCollection* dc;
    if(this->latDelta == 0 && this->lonDelta == 0)
    {
```

```
        dc = new DataCollection();
    } else {
        dc = new DataCollection(this->latDelta, this->lonDelta);
    }

    return new std::pair<DataCollection*, Bounds*>(dc, newBoundsFromTrip);
}

bool BuildCity::hasNewBounds()
{
    DataManagement dm;
    GenericMap<long int, std::pair<double, double>*>* tripLatLon = dm.getMostRecentTripData();
    GenericMap<long int, std::pair<double, double>*>* tripLatLonCopy = tripLatLon->copy();

    this->city = dm.getCityData();

    this->maxLat = -DBL_MAX;
    this->maxLon = -DBL_MAX;
    this->minLat = DBL_MAX;
    this->minLon = DBL_MAX;

    if(this->city != NULL)
    {
        // bounds data already exists
        GenericMap<int, Bounds*>* bounds = this->city->getBoundsMap();

        // erase in-bounds measurements
        tripLatLonCopy->initializeCounter();
        GenericEntry<long int, std::pair<double, double>*>* nextTripLatLonCopy = tripLatLonCopy->nextEntry();
        while(nextTripLatLonCopy != NULL)
        {
            double lat = nextTripLatLonCopy->value->first;
            double lon = nextTripLatLonCopy->value->second;

            bounds->initializeCounter();
            GenericEntry<int, Bounds*>* nextBounds = bounds->nextEntry();
            while(nextBounds != NULL)
            {
                Bounds* bound = nextBounds->value;

                // measurement is in bounds
                if(lat < bound->getMaxLat() && lat > bound->getMinLat() && lon < bound->getMaxLon() && lon > bound->getMinLon())
                {
                    tripLatLon->erase(nextTripLatLonCopy->key);
                    break;
                }
                nextBounds = bounds->nextEntry();
            }
            nextTripLatLonCopy = tripLatLonCopy->nextEntry();
            delete(nextBounds);
```

```
        }
        delete(nextTripLatLonCopy);
    }

    // find min / max of out of bounds measurements
    if(tripLatLon->getSize() > 0)
    {
        this->newBoundsFound = true;

        tripLatLon->initializeCounter();
        GenericEntry<long int, std::pair<double, double>*>* nextTripLatLon = tripLatLon->nextEntry();
        while(nextTripLatLon != NULL)
        {
            double lat = nextTripLatLon->value->first;
            double lon = nextTripLatLon->value->second;

            if(lat > this->maxLat) { this->maxLat = lat; }
            if(lat < this->minLat) { this->minLat = lat; }

            if(lon > this->maxLon) { this->maxLon = lon; }
            if(lon < this->minLon) { this->minLon = lon; }

            nextTripLatLon = tripLatLon->nextEntry();
        }
        delete(nextTripLatLon);
    }

    // expand min / max by a small margin
    double expandMargin = .0001;
    this->maxLat += expandMargin;
    this->minLat -= expandMargin;

    this->maxLon += expandMargin;
    this->minLon -= expandMargin;

    delete(tripLatLon);
    delete(tripLatLonCopy);

    return this->newBoundsFound;
}

GenericMap<long int, Intersection*>* BuildCity::getNewIntersections()
{
    return this->newInts;
}

GenericMap<long int, Road*>* BuildCity::getNewRoads()
{
    GenericMap<long int, Road*>* newRoads = new GenericMap<long int, Road*>();
```

```
    this->newInts->initializeCounter();
    GenericEntry<long int, Intersection*>* nextInt = this->newInts->nextEntry();
    while(nextInt != NULL)
    {
        GenericMap<long int, Road*>* connectingRoads = nextInt->value->getRoads();

        connectingRoads->initializeCounter();
        GenericEntry<long int, Road*>* nextRoad = connectingRoads->nextEntry();
        while(nextRoad != NULL)
        {
            if(!newRoads->hasEntry(nextRoad->value->getRoadID()))
            {
                newRoads->addEntry(nextRoad->value->getRoadID(), nextRoad->value);
            }
            nextRoad = connectingRoads->nextEntry();
        }
        delete(nextRoad);

        nextInt = this->newInts->nextEntry();
    }
    delete(nextInt);

    return newRoads;
}

Bounds* BuildCity::getNewBounds()
{
    return this->newBounds;
}

City* BuildCity::getUpdatedCity()
{
    if(!this->newBoundsFound)
    {
        this->updateGridDataXMLSpline();
        this->printNewIntersectionsAndRoads();
    }

    // incase no city data provided
    if(this->city == NULL)
    {
        this->city = new City();
    }

    if(this->newBoundsFound)
    {
        return this->updateCity();
    }
    else
    {
```

```
        return this->city;
    }
}

City* BuildCity::updateCity()
{
    GenericMap<long int, Road*>* newRoads = this->getNewRoads();

    newRoads->initializeCounter();
    GenericEntry<long int, Road*>* nextRoad = newRoads->nextEntry();
    while(nextRoad != NULL)
    {
        this->city->addRoad(nextRoad->value);
        nextRoad = newRoads->nextEntry();
    }
    delete(nextRoad);

    this->newInts->initializeCounter();
    GenericEntry<long int, Intersection*>* nextInt = this->newInts->nextEntry();
    while(nextInt != NULL)
    {
        this->city->addIntersection(nextInt->value);
        nextInt = this->newInts->nextEntry();
    }
    delete(nextInt);

    this->city->addBounds(this->newBounds);

    return this->city;
}

} // end of predictivepowertrain
```

12.12 BuildCity.h

```
/*
 * BuildCity.h
 *
 *  Created on: Apr 4, 2016
 *      Author: vagrant
 */

#ifndef CITY_BUILDCITY_H_
#define CITY_BUILDCITY_H_

#define EIGEN_NO_DEBUG

#include "City.h"
#include "../data_management/DataManagement.h"
#include "../data_management/DataCollection.h"
#include "../map/GenericMap.h"
```

```
#include "../data_management/Node.h"
#include "Road.h"
#include "Intersection.h"

#include "limits.h"
#include <eigen3/Eigen/Core>
#include <eigen3/unsupported/Eigen/Splines>
#include <math.h>

#include <opencv2/opencv.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

namespace PredictivePowertrain {

class BuildCity {
private:
    // functional
    City* city;
    GenericMap<long int, Road*>* rawRoads;
    GenericMap<long int, Intersection*>* newInts;
    Bounds* newBounds;
    Eigen::MatrixXd adjMatFromSplines;
    double maxLat;
    double maxLon;
    double minLat;
    double minLon;
    double latCenter;
    double lonCenter;
    double latDelta;
    double lonDelta;

    struct zoomParams
    {
        int kernelSideDim;
        int angleThreshold;
        int pntThreshold;
        int imageProcessingResolution;
        int coordBinResolution;
        int maxLines;
        double maxClusterPntDistance;
    };

    int boundsID = 0;
    double idScalar = 10000.0;
    bool newBoundsFound = false;

    // for spline shit using osm
```

```
    double splineStep = 0.025; // s-value
    double evalIntervalLength = 5.0; // meters
    double adjMatPrecFromSplines = 0.00001;
    double gpsTolerance = 0.0001;

    std::pair<GenericMap<int, Intersection*>*, GenericMap<long int, Road*>*>* parseAdjMat();
    double scaleID(long int id);
    long int unScaleID(double id);
    void connectifyAjdMat();
    bool isAdj(GenericEntry<int, std::pair<int, int>*>* idx1, GenericEntry<int, std::pair<int, int>*>* idx2);
    std::pair<DataCollection*, Bounds*>* setupDataCollection();
    std::pair<int, cv::Mat>* pullAndFormatMapPNG(DataCollection* dc);
    template<typename K> bool getIntersectionPoint(cv::Point_<K> a1, cv::Point_<K> a2, cv::Point_<K> b1, cv::Point_<K> b2, cv::Point_<K> &
intPnt);
    std::pair<cv::Point, cv::Point>* polarToCartisian(float mag, float angle, int rows);
    template<typename K> double cross(cv::Point_<K> v1, cv::Point_<K> v2);
    int getCoord(int* dimCount, int dim, int tol);
    GenericMap<int, cv::Point*>* getIntersectionPointsFromMapPNG(cv::Mat map, int zoom);
    GenericMap<int, std::pair<cv::Point*, cv::Point*>*>* perimeterScanKernelForRoads(cv::Mat kernel);
    void drawPoint(cv::Mat &image, cv::Point point);
    void checkNextPixel(int x, int y, GenericMap<int, cv::Point*>& points, cv::Mat& kernel, int& lastPixel);
    void declusterIntersectionPoints(cv::Point* rawIntPnt, GenericMap<int, cv::Point*>& rawIntPnts, GenericMap<int, cv::Point*>& intPnts, int
kernelSideDim, double maxDistance);
    zoomParams getZoomParams(int zoom);
    int hashCoords(int x, int y);
    std::pair<GenericMap<int, Road*>*, GenericMap<int, Intersection*>*>* makeRoadsAndIntersections(GenericMap<int, cv::Point*>* rawInts,
cv::Mat map);
    std::pair<bool, GenericMap<int, cv::Point*>*>* traceRoad(cv::Point* nextClosestPnt, cv::Point* pnt, cv::Mat map, GenericMap<int,
cv::Point*>& searchedIntPnts);

public:
    BuildCity();
    virtual ~BuildCity();
    void updateGridDataXMLSpline();
    void updateGridDataXMLAdj();
    void updateGridDataPNG();
    void printAdjMats();
    void printNewIntersectionsAndRoads();
    bool hasNewBounds();
    GenericMap<long int, Intersection*>* getNewIntersections();
    GenericMap<long int, Road*>* getNewRoads();
    Bounds* getNewBounds();
    City* getUpdatedCity();
    City* updateCity();
};

}

#endif /* CITY_BUILDCITY_H_ */
```

12.13 BuildCityUnitTest.cpp

```
/*
 * BuildCityUnitTest.cpp
 *
 *  Created on: Apr 30, 2016
 *      Author: vagrant
 */

#include "UnitTests.h"
#include "../map/GenericMap.h"
#include "../data_management/DataManagement.h"
#include "../city/BuildCity.h"

using namespace PredictivePowertrain;

void buildCity_ut() {

    // make trip log
    GenericMap<long int, std::pair<double, double>*>* latLon = new GenericMap<long int, std::pair<double, double>*>();

    // greenlake trip log small
//    latLon->addEntry(1, new std::pair<double, double>(47.634, -117.396));
//    latLon->addEntry(2, new std::pair<double, double>(47.635, -117.397));
//    latLon->addEntry(3, new std::pair<double, double>(47.636, -117.398));
//    latLon->addEntry(4, new std::pair<double, double>(47.637, -117.399));
//    latLon->addEntry(5, new std::pair<double, double>(47.638, -117.400));
//    latLon->addEntry(6, new std::pair<double, double>(47.639, -117.401));
//    latLon->addEntry(7, new std::pair<double, double>(47.640, -117.402));
//    latLon->addEntry(8, new std::pair<double, double>(47.650, -117.410));

    // down town trip log big
//    latLon->addEntry(1, new std::pair<double, double>(47.656784, -122.307302));
//    latLon->addEntry(2, new std::pair<double, double>(47.560249, -122.379874));

    // test expansion
    latLon->addEntry(1, new std::pair<double, double>(47.624523266249, -117.404426673014));
    latLon->addEntry(2, new std::pair<double, double>(47.617847, -117.407564));
    latLon->addEntry(3, new std::pair<double, double>(47.617683, -117.406059));

    // jsonify trip log -> delete existing jsons
    DataManagement dm;
    dm.addTripData(latLon);

    BuildCity bc;
    City* city = bc.getUpdatedCity();

    dm.addCityData(city);
}
```

12.14 Road.cpp

```
/*
 * Road.cpp
 *
 *  Created on: Jan 7, 2016
 *      Author: Amanda
 */

#include "Road.h"

namespace PredictivePowertrain {

Road::Road() {
      this->roadID = 0;
    this->initialize();
}

Road::Road(std::string roadType, long int roadID, GenericMap<long int, Node*>* nodes) {
      this->roadType = roadType;
      this->roadID = roadID;
      this->nodes = nodes;
    this->initialize();
}

void Road::initialize()
{
    this->boundsID = 0;
    this->startNode = NULL;
    this->endNode = NULL;
    this->splineLength = -1;
}

void Road::setStartIntersection(Intersection* startNode) {
      this->startNode = startNode;
}

void Road::setEndIntersection(Intersection* endNode) {
      this->endNode = endNode;
}

Intersection* Road::getStartIntersection() {
      return this->startNode;
}

Intersection* Road::getEndIntersection() {
      return this->endNode;
}
void Road::assignID(long int id)
{
```

```
    this->roadID = id;
}

long int Road::getRoadID() {
      return this->roadID;
}

Road::~Road() {
}

double Road::getMaxLat()
{
    return this->maxLat;
}

double Road::getMinLat()
{
    return this->minLat;
}

double Road::getMaxLon()
{
    return this->maxLon;
}

double Road::getMinLon()
{
    return this->minLon;
}

std::vector<float>* Road::getSpeedData() {
    // TODO
    return NULL;
}

void Road::getElevData(std::vector<float>* elev, std::vector<float>* dist)
{
    GPS converter;

    this->nodes->initializeCounter();
    GenericEntry<long int, Node*>* prevNode = this->nodes->nextEntry();
    GenericEntry<long int, Node*>* nextNode = this->nodes->nextEntry();

    float cumulativeDist = 0.0;

    while(nextNode != NULL)
    {
        cumulativeDist += converter.deltaLatLonToXY(prevNode->value->getLat(), prevNode->value->getLon(), nextNode->value->getLat(), nextNode-
>value->getLon());
```

```
        elev->push_back(nextNode->value->getEle());
        dist->push_back(cumulativeDist);

        prevNode = nextNode;
        nextNode = this->nodes->nextEntry();
    }
    delete(nextNode);
}

int Road::getBoundsID() {
      return this->boundsID;
}

void Road::setBoundsID(int id) {
      this->boundsID = id;
}

float Road::getSplineLength()
{
    return this->splineLength;
}

std::string Road::getRoadType() {
      return this->roadType;
}

void Road::assignSplineLength(float dist)
{
    this->splineLength = dist;
}

GenericMap<long int, Node*>* Road::getNodes() {
      return this->nodes;
}

Eigen::Spline<double, 2> Road::getSpline() {
      return this->spline;
}

void Road::assignSpline(Eigen::Spline<double, 2> spline) {
      this->spline = spline;
}

void Road::assignAdjMatIndicies(GenericMap<int, std::pair<int, int>*>* adjMatIndicies) {
      this->adjMatIndices = adjMatIndicies;
}

GenericMap<int, std::pair<int, int>*>* Road::getAdjMatIndicies() {
      return this->adjMatIndices;
}
```

```
std::pair<double, double>* Road::getMidLatLon()
{
    this->setMinMaxLatLon();
    return new std::pair<double, double>(this->maxLat - this->minLat, this->maxLon - this->minLon);
}

void Road::setMinMaxLatLon()
{
    double minLat = DBL_MAX;
    double maxLat = -DBL_MAX;
    double minLon = DBL_MAX;
    double maxLon = -DBL_MAX;

    for(double u = 0; u <= 1; u += .025)
    {
        Eigen::Spline<double,2>::PointType point = this->spline(u);

        double lat = point(0,0);
        double lon = point(1,0);

        if(lat < minLat)
        {
            this->minLat = lat;
        }

        if(lat > maxLat)
        {
            this->maxLat = lat;
        }

        if(lon < minLon)
        {
            this->minLon = lon;
        }

        if(lon > maxLon)
        {
            this->maxLon = lon;
        }
    }
}

} /* namespace PredictivePowertrain */
```

12.15 Road.h

```
/*
 * Road.h
 *
 *  Created on: Jan 7, 2016
```

```
 *      Author: Amanda
 */

#ifndef ROAD_H_
#define ROAD_H_

#include "../map/GenericMap.h"
#include "../data_management/Node.h"
#include "../gps/GPS.h"

#include <string>
#include <limits.h>
#include <utility>
#include <vector>
#include <eigen3/Eigen/Core>
#include <eigen3/unsupported/Eigen/Splines>

namespace PredictivePowertrain {

class Intersection; // forward declaration

class Road {
private:
      Eigen::Spline<double, 2> spline;
    float splineLength;
      GenericMap<long int, Node*>* nodes;
      GenericMap<int, std::pair<int, int>*>* adjMatIndices;
      std::string roadType;
      Intersection* startNode;
      Intersection* endNode;
      long int roadID;
      int boundsID;
    double maxLat;
    double minLat;
    double maxLon;
    double minLon;
    void initialize();

public:
      Road();
      Road(std::string roadType, long int roadID, GenericMap<long int, Node*>* nodes);
      void copy(Road* other);
      virtual ~Road();
      void assignID(long int id);
      void setStartIntersection(Intersection* startNode);
      void setEndIntersection(Intersection* endNode);
      Intersection* getStartIntersection();
      Intersection* getEndIntersection();
      long int getRoadID();
    void getElevData(std::vector<float>* elev, std::vector<float>* dist);
```

```
    std::vector<float>* getSpeedData();
      int getBoundsID();
      void setBoundsID(int id);
    float getSplineLength();
      std::string getRoadType();
      GenericMap<long int, Node*>* getNodes();
      Eigen::Spline<double, 2> getSpline();
      void assignSpline(Eigen::Spline<double, 2> spline);
    void assignSplineLength(float dist);
      void assignAdjMatIndicies(GenericMap<int, std::pair<int, int>*>* adjMatIndicies);
      GenericMap<int, std::pair<int, int>*>* getAdjMatIndicies();
    std::pair<double, double>* getMidLatLon();
    void setMinMaxLatLon();
    double getMaxLat();
    double getMinLat();
    double getMaxLon();
    double getMinLon();
};

} /* namespace PredictivePowertrain */

#endif /* ROAD_H_ */
```

12.16 RoadUnitTest.cpp

```
/*
 * RoadUnitTest.cpp
 *
 *  Created on: Apr 12, 2016
 *      Author: vagrant
 */

#include "../city/Road.h"
#include "../city/Intersection.h"
#include <assert.h>
#include "UnitTests.h"

using namespace PredictivePowertrain;

void road_ut() {

//    //Testing constructor Road::Road(std::string roadType, int* elevationData, int* speedData, int roadID)
//    std::string x = "default";
//    Road road2(x, 1);
//
//    //Intersection::Intersection(Road * roadInput, double lat, double lon, int elev, int intersectNum) {
//    Intersection sect(&road2, 2.0, 2.0, 1, 1);
//
//    //Testing default constructor
//    Road road1;
//    assert(road1.getRoadID() == 0);
```

```
//
//    //Testing setStartNode
//    road2.setStartNode(&sect);
//
//    //Testing setEndNode
//    road2.setEndNode(&sect);
//
//    //Testing getStartNode and getEndNode
//    assert(road2.getStartNode() == sect.startNode && road2.getEndNode() == sect.endNode);
//
//    //Testing getRoadID
//    assert(road2.getRoadID() == 1);
//    //Testing getSpeedData & get ElevData
//    assert(road2.getSpeedData() == 15 && road2.getElevData() == 10);

}
```

12.17 Intersection.cpp

```
/*
 * Intersection.cpp
 *
 *  Created on: Jan 8, 2016
 *      Author: Amanda
 */

#include "Intersection.h"

#include <algorithm>

namespace PredictivePowertrain {

Intersection::Intersection(GenericMap<long int, Road*>* roads, double lat, double lon, float elev, long int intersectNum) {
      this->roads = roads;
//    this->interSectionType = IntersectionTypes(intersectType);
      this->lat = lat;
      this->lon = lon;
      this->elevation = elev;
      this->ID = intersectNum;
}

Intersection::~Intersection() {
    delete(this->roads);
}

Intersection::Intersection() {
    this->roads = new GenericMap<long int, Road*>();
}

void Intersection::assignID(long int ID)
{
```

```
    this->ID = ID;
}

long int Intersection::getIntersectionID() {
      return this->ID;
}

float Intersection::getElevation() {
      return this->elevation;
}

double Intersection::getLat() {
      return this->lat;
}

double Intersection::getLon() {
      return this->lon;
}

GenericMap<long int, Road*>* Intersection::getRoads() {
      return this->roads;
}

void Intersection::addRoad(Road* road, int roadDir) {
      if(roadDir == 0)
      {
        road->setEndIntersection(this);
      } else {
        road->setStartIntersection(this);
      }

      this->roads->addEntry(road->getRoadID(), road);
}

void Intersection::replaceRoads(GenericMap<long int, Road*>* roads)
{
    delete(this->roads);
    this->roads = roads;
}

GenericMap<long int, Link*>* Intersection::getOutgoingLinks(Link* excluded)
{
    GenericMap<long int, Link*>* outGoingLinks = new GenericMap<long int, Link*>();
    long int linkCount = 0;

    this->roads->initializeCounter();
    GenericEntry<long int, Road*>* nextRoad = this->roads->nextEntry();
    while(nextRoad != NULL)
    {
```

```
        if(nextRoad->value->getRoadID() != excluded->getNumber())
        {
            /* OLD
             Link* newLink = this->link->linkFromRoad(nextRoad->value, this);
             outGoingLinks->addEntry(linkCount, newLink);
             linkCount++;
             */

            // every road is now bi directional
            Link* link0 = new Link(1, nextRoad->value->getRoadID());
            Link* link1 = new Link(0, nextRoad->value->getRoadID());

            outGoingLinks->addEntry(linkCount, link0);
            linkCount++;

            outGoingLinks->addEntry(linkCount, link1);
            linkCount++;
        }

        nextRoad = this->roads->nextEntry();

    }
    delete(nextRoad);

    return outGoingLinks;
}

GenericMap<long int, Link*>* Intersection::getOutgoingLinks()
{
      GenericMap<long int, Link*>* outGoingLinks = new GenericMap<long int, Link*>();
      long int linkCount = 0;

      this->roads->initializeCounter();
      GenericEntry<long int, Road*>* nextRoad = this->roads->nextEntry();
      while(nextRoad != NULL)
      {
        /* OLD
        Link* newLink = this->link->linkFromRoad(nextRoad->value, this);
        outGoingLinks->addEntry(linkCount, newLink);
        linkCount++;
         */

        // every road is now bi directional
        Link* link0 = new Link(1, nextRoad->value->getRoadID());
        Link* link1 = new Link(0, nextRoad->value->getRoadID());

        outGoingLinks->addEntry(linkCount, link0);
        linkCount++;

        outGoingLinks->addEntry(linkCount, link1);
```

```
        linkCount++;

        nextRoad = this->roads->nextEntry();

      }
      delete(nextRoad);

      return outGoingLinks;
}

int Intersection::getRoadCount() {
      return this->roads->getSize();
}

Intersection* Intersection::getNextIntersection(Road* road) {
      this->roads->initializeCounter();
      GenericEntry<long int, Road*>* nextRoad = this->roads->nextEntry();
      while(nextRoad != NULL)
      {
        if(road->getRoadID() == nextRoad->value->getRoadID())
        {
                if(this->ID == nextRoad->value->getStartIntersection()->getIntersectionID())
                {
                        delete(nextRoad);
                        return nextRoad->value->getEndIntersection();
                } else {
                        delete(nextRoad);
                        return nextRoad->value->getStartIntersection();
                }
        }
        nextRoad = this->roads->nextEntry();
      }
      delete(nextRoad);
      return NULL;
}

GenericMap<long int, Intersection*>* Intersection::getAdjacentIntersections() {
      GenericMap<long int, Intersection*>* adjInts = new GenericMap<long int, Intersection*>();
      int adjIntCount = 0;

      this->roads->initializeCounter();
      GenericEntry<long int, Road*>* nextRoad = this->roads->nextEntry();
      while(nextRoad != NULL)
      {
        Intersection* adjInt = getNextIntersection(nextRoad->value);
        bool alreadyCounted = false;

        adjInts->initializeCounter();
        GenericEntry<long int, Intersection*>* nextInt = adjInts->nextEntry();
        while(nextInt != NULL)
```

```
        {
                if(nextInt->value->getIntersectionID() == adjInt->getIntersectionID())
                {
                        alreadyCounted = true;
                        break;
                }
                nextInt = adjInts->nextEntry();
        }
         delete(nextInt);

        if(!alreadyCounted)
        {
                adjInts->addEntry(adjInt->getIntersectionID(), adjInt);
        }
        nextRoad = this->roads->nextEntry();
      }
      return adjInts;
}

void Intersection::setBoundsID(int ID) {
      this->boundsID = ID;
}

int Intersection::getBoudsID() {
      return this->boundsID;
}

void Intersection::setElev(float newElev)
{
    this->elevation = newElev;
}

} /* namespace PredictivePowertrain */
```

12.18 Intersection.h

```
/*
 * Intersection.h
 *
 *  Created on: Jan 8, 2016
 *      Author: Amanda
 */

#ifndef INTERSECTION_H_
#define INTERSECTION_H_

#include "IntersectionTypes.h"
#include "Road.h"
#include "../driver_prediction/Link.h"
#include <string>
```

```
namespace PredictivePowertrain {

class Link; // forward declaration
class Road; // forward declaration

class Intersection {

private:
//    Road* roads;
      GenericMap<long int, Road*>* roads;
//    IntersectionTypes interSectionType;
      long int ID;
      float elevation;
      double lat;
      double lon;
      int boundsID;
      Link* link;

public:
      Intersection();
      Intersection(GenericMap<long int, Road*>* roads, double lat, double lon, float elev, long int intersectNum);
      void setBoundsID(int ID);
      virtual ~Intersection();
      void addRoad(Road* road, int roadDir);
      GenericMap<long int, Link*>* getOutgoingLinks();
    GenericMap<long int, Link*>* getOutgoingLinks(Link* excluded);
      int getRoadCount();
      long int getIntersectionID();
      float getElevation();
      double getLat();
      double getLon();
      GenericMap<long int, Road*>* getRoads();
      GenericMap<long int, Intersection*>* getAdjacentIntersections();
      Intersection* getNextIntersection(Road* road);
      int getBoudsID();
    void assignID(long int ID);
    void replaceRoads(GenericMap<long int, Road*>* roads);
    void setElev(float newElev);
};

} /* namespace PredictivePowertrain */

#endif /* INTERSECTION_H_ */
```

12.19 IntersectionUnitTest.cpp

```
/*
 * IntersectionUnitTest.cpp
 *
 *  Created on: Apr 5, 2016
 *      Author: vagrant
```

```
 */


#include "../city/Intersection.h"
#include "../city/Road.h"
#include "../city/RoadTypes.h"
#include "../driver_prediction/Link.h"
#include <iostream>
#include <assert.h>
#include "UnitTests.h"

using namespace PredictivePowertrain;

void intersection_ut(){
      //Road(std::string roadType, int* elevationData, int* speedData, int roadID, double* lat, double* lon)
      Road road1;
      Road road2;


      //Intersection(Road * roadInput, double lat, double lon, int elev, int intersectNum)
//    Intersection intersection1(&road1, 0, 0, 0, 0);
//    Intersection intersection2(&road1, 1.0, 1.0, 1, 1);
//    Intersection intersection3(&road2, 10.0, 10.0, 4, 5);
      Intersection intersection4;
      Intersection intersection5;

      //Testing getNumber(), getElevation(), getLat(), and getLon()
//    assert(intersection1.getIntersectionID() == 0 && intersection1.getElevation() == 0);
//    assert(intersection1.getLat() == 0 && intersection1.getLon() == 0);

//    assert(intersection2.getIntersectionID() == 1 && intersection2.getElevation() == 1);
//    assert(intersection2.getLat() == 1.0 && intersection2.getLon() == 1.0);


      //Testing getRoads()
//    assert(intersection2.getRoads() == &road1);
//    assert(intersection3.getRoads() == &road2);

      //Testing addRoad() and getRoadCount()
      intersection5.addRoad(&road2, 0);
      assert(intersection5.getRoadCount() == 1);

      //Testing getOutgoingLinks()
      Link link1;
//    assert(intersection4.getOutgoingLinks() == &link1);


      //Testing getAdjacentIntersection()
      //assert(intersection4.getAdjacentIntersection() == NULL);
```

```
      //Testing getNextIntersection()
      assert(intersection4.getNextIntersection(&road1) == NULL);
}
```

12.20 Node.cpp

```
/*
 * Node.cpp
 *
 *  Created on: Apr 8, 2016
 *      Author: vagrant
 */

#include "Node.h"

namespace PredictivePowertrain {

Node::Node() {
}

Node::Node(double lat, double lon, float ele, long int id) {
      this->lat = lat;
      this->lon = lon;
      this->ele = ele;
      this->id = id;
}

double Node::getLat() {
      return this->lat;
}

double Node::getLon() {
      return this->lon;
}

long int Node::getID() {
      return this->id;
}

float Node::getEle() {
      return this->ele;
}

void Node::setEle(float newEle)
{
    this->ele = newEle;
}

}
```

12.21 Node.h

```
/*
```

```
 * Node.h
 *
 *  Created on: Apr 8, 2016
 *      Author: vagrant
 */

#ifndef DATACOLLECTION_NODE_H_
#define DATACOLLECTION_NODE_H_

namespace PredictivePowertrain {

class Node {
private:
      double lat;
      double lon;
      float ele;
      int long id;

public:
      Node();
      Node(double lat, double lon, float ele, long int id);
      double getLat();
      double getLon();
      long int getID();
      float getEle();
    void setEle(float newEle);
};

}

#endif /* DATACOLLECTION_NODE_H_ */
```

12.22 NodeUnitTest.cpp

```
/*
 * NodeUnitTest.cpp
 *
 *  Created on: Apr 19, 2016
 *      Author: vagrant
 */


#include "../data_management/Node.h"
#include <iostream>
#include <assert.h>
#include "UnitTests.h"


using namespace PredictivePowertrain;

void node_ut(){
```

```
	Node node1(0.0, 0.0 , 0, 0);
	Node node2(1.0, 1.0, 1, 1);
	Node node3(15.5, 16.5, 20, 22);

	assert(node1.getLat() == 0);
	assert(node1.getLon() == 0);
	assert(node1.getEle() == 0);
	assert(node1.getID() == 0);


	assert(node2.getLat() == 1.0);
	assert(node2.getLon() == 1.0);
	assert(node2.getEle() == 1);
	assert(node2.getID() == 1);

	assert(node3.getLat() == 15.5);
	assert(node3.getLon() == 16.5);
	assert(node3.getEle() == 20);
	assert(node3.getID() == 22);

}
```

12.23 Way.cpp

```
/*
 * Way.cpp
 *
 *  Created on: Apr 15, 2016
 *      Author: vagrant
 */

#include "Way.h"

namespace PredictivePowertrain {

Way::Way() {
}

Way::Way(GenericMap<int, long int>* nodeIDs, long int id, std::string wayType, int waySpeed) {
	this->nodeIDs = nodeIDs;
	this->ID = id;
	this->wayType = wayType;
	this->waySpeed = waySpeed;
}

GenericMap<int, long int>* Way::getNodeIDs() {
	return this->nodeIDs;
}

long int Way::getID() {
```

```
	return this->ID;
}

std::string Way::getWayType() {
	return this->wayType;
}

int Way::getWaySpeed() {
	return this->waySpeed;
}

}
```

12.24 Way.h

```
/*
 * Way.h
 *
 *  Created on: Apr 15, 2016
 *      Author: vagrant
 */

#ifndef DATA_MANAGEMENT_WAY_H_
#define DATA_MANAGEMENT_WAY_H_

#include "../map/GenericMap.h"
#include <string>
#include "../data_management/Node.h"

namespace PredictivePowertrain {

class Way {
private:
	GenericMap<int, long int>* nodeIDs;
	std::string wayType;
	int waySpeed;
	long int ID;

public:
	Way();
	Way(GenericMap<int, long int>* nodeIDs, long int id, std::string wayType, int waySpeed);
	GenericMap<int, long int>* getNodeIDs();
	long int getID();
	std::string getWayType();
	int getWaySpeed();
};

}

#endif /* DATA_MANAGEMENT_WAY_H_ */
```

12.25 Bounds.cpp

```
/*
 * Bounds.cpp
 *
 *  Created on: Apr 18, 2016
 *      Author: vagrant
 */

#include "Bounds.h"

namespace PredictivePowertrain {

Bounds::Bounds()
{
}

Bounds::Bounds(double maxLat, double minLat, double maxLon, double minLon) {
      this->maxLat = maxLat;
    this->minLat = minLat;
    this->maxLon = maxLon;
      this->minLon = minLon;
      this->id = -1;
}

double Bounds::getMaxLat() {
      return this->maxLat;
}

double Bounds::getMaxLon() {
      return this->maxLon;
}

double Bounds::getMinLat() {
      return this->minLat;
}

double Bounds::getMinLon() {
      return this->minLon;
}

void Bounds::assignID(int id) {
      this->id = id;
}

int Bounds::getID() {
      return this->id;
}

std::pair<double, double>* Bounds::getMidLatLon()
```

```
{
    return new std::pair<double, double>(this->maxLat - this->minLat, this->maxLon - this->minLon);
}

}
```

12.26 Bounds.h

```
/*
 * Bounds.h
 *
 *  Created on: Apr 18, 2016
 *      Author: vagrant
 */

#ifndef DATA_MANAGEMENT_BOUNDS_H_
#define DATA_MANAGEMENT_BOUNDS_H_

#include <utility>

namespace PredictivePowertrain {

class Bounds {
private:
      double maxLat;
      double maxLon;
      double minLat;
      double minLon;
      int id;

public:
      Bounds();
      Bounds(double maxLat, double minLat, double maxLon, double minLon);
      double getMaxLat();
      double getMaxLon();
      double getMinLat();
      double getMinLon();
     std::pair<double, double>* getMidLatLon();
      void assignID(int id);
      int getID();
};

} /* namespace PredictivePowertrain */

#endif /* DATA_MANAGEMENT_BOUNDS_H_ */
```

12.27 DataCollection.cpp

```
/*
 * DataCollection.cpp
 *
 *  Created on: Apr 4, 2016
 *      Author: vagrant
```

```
 */

#include "../data_management/DataCollection.h"

using namespace boost::asio;
using boost::asio::ip::tcp;
using boost::lexical_cast;
using boost::property_tree::ptree;

using namespace boost::property_tree;

namespace PredictivePowertrain {

DataCollection::DataCollection() {
    this->initialize(.0002, .0002);
}

DataCollection::DataCollection(double latDelta, double lonDelta) {
    this->initialize(latDelta, lonDelta);
}

void DataCollection::initialize(double latDelta, double lonDelta) {
    this->latDelta = latDelta;
    this->lonDelta = lonDelta;
    this->zoomMax = 19;
    this->zoomMin = 18;
    this->wayCount = 0;
    this->boundsCountXML = 0;
    this->boundsCountPNG = 0;
    this->setZoomSpreads();
    checkDataFoler();
}

// must call functions in specific order
void DataCollection::pullDataXML(double lat, double lon) {
      this->pullSRTMData(lat, lon);
      this->pullOSMDataXML(lat, lon);
}

int DataCollection::pullDataPNG(double lat, double lon) {
      this->pullSRTMData(lat, lon);
      return this->pullOSMDataPNG(lat, lon);
}

void DataCollection::pullOSMDataXML(double lat, double lon) {
      std::cout << "pulling OSM data ..." << std::endl;

      this->mapFile = "mapFile_";
      this->mapFile += lexical_cast<std::string>(lat) + "_";
      this->mapFile += lexical_cast<std::string>(lon) + ".xml";
```

```
std::string mapFilePath = this->dataFolder + "/" + this->mapFile;

std::ifstream test(mapFilePath);
if(!test)
{
  double lowLat = lat - .5*this->latDelta;
  double lowLon = lon - .5*this->lonDelta;
  double hiLat = lat + .5*this->latDelta;
  double hiLon = lon + .5*this->lonDelta;

  std::string serverName = "overpass-api.de";
  std::string getCommand = "/api/map?bbox=";
  getCommand += lexical_cast<std::string>(lowLon) + ",";
  getCommand += lexical_cast<std::string>(lowLat) + ",";
  getCommand += lexical_cast<std::string>(hiLon) + ",";
  getCommand += lexical_cast<std::string>(hiLat);

  queryFile(serverName, getCommand, this->dataFolder + "/" + this->mapFile);
}
test.close();

ptree tree;
read_xml(mapFilePath, tree);

const ptree& formats = tree.get_child("osm", empty_ptree());
std::cout << "OSM Data Size: " <<  formats.size() << std::endl;

BOOST_FOREACH(const ptree::value_type & f, formats)
{

  std::string tagName = lexical_cast<std::string>(f.first);
  std::cout << "============== " << tagName << " ==============" << std::endl;

  if(!tagName.compare("node"))
  {
        long int nodeID;
        float lat, lon;
        const ptree & attributes = f.second.get_child("<xmlattr>", empty_ptree());
        BOOST_FOREACH(const ptree::value_type &v, attributes)
        {
               std::string attr = lexical_cast<std::string>(v.first.data());

               if(!attr.compare("id"))
               {
                      nodeID = lexical_cast<long int>(v.second.data());
                      std::cout << attr << " = " << nodeID << std::endl;
               } else if(!attr.compare("lat")) {
                      lat = lexical_cast<float>(v.second.data());
                      std::cout << attr << " = " << lat << std::endl;
```

```
                        } else if(!attr.compare("lon")) {
                                lon = lexical_cast<float>(v.second.data());
                                std::cout << attr << " = " << lon << std::endl;
                        }

                }
                int ele = getElevation(lat, lon);
                Node* node = new Node(lat, lon, ele, nodeID);
                this->nodeMap.addEntry(nodeID, node);

        } else if (!tagName.compare("way")) {

                // get children of way specifically looking for node IDs and way types to see if way is desireable
                int waySpeed = -1;
                int refCount = 0;
                std::string wayType;
                bool isDesiredWay = false;
                GenericMap<int, long int>* nodeIDs = new GenericMap<int, long int>();
                BOOST_FOREACH(const ptree::value_type &v, f.second)
                {
                        std::string childTagName = lexical_cast<std::string>(v.first);

                        // look at child attributes to get node IDs
                        if(!childTagName.compare("nd") || !childTagName.compare("tag"))
                        {
                                std::cout << "- " << childTagName << " -" << std::endl;
                                const ptree & attributes = v.second.get_child("<xmlattr>", empty_ptree());
                                BOOST_FOREACH(const ptree::value_type &z, attributes)
                                {
                                        std::string attr = lexical_cast<std::string>(z.first.data());
                                        if(!childTagName.compare("nd") && !attr.compare("ref"))
                                        {
                                                long int ref = lexical_cast<long int>(z.second.data());
                                                std::cout << attr << " = " << z.second.data() << std::endl;
                                                refCount++;
                                                nodeIDs->addEntry(refCount, ref);
                                        } else if(!childTagName.compare("tag") && !attr.compare("v")) {
                                                std::string val = lexical_cast<std::string>(z.second.data());
                                                std::size_t found = val.find("mph");
                                                if(!val.compare("motorway")   ||   !val.compare("primary")   ||   !val.compare("secondary")   ||
!val.compare("tertiary")
                                                                ||   !val.compare("motorway   link")   ||   !val.compare("primary   link")   ||
!val.compare("unclassified")
                                                                || !val.compare("road") || !val.compare("residential") || !val.compare("service"))
                                                {
                                                        wayType = val;
                                                        isDesiredWay = true;
                                                        std::cout << attr << " = " << val << std::endl;
                                                } else if(found!=std::string::npos) {
                                                        std::stringstream ss(val);
```

```
                                        ss >> waySpeed;
                                        std::cout << attr << " = " << val << std::endl;
                                }
                        }
                }
        }

        if(isDesiredWay)
        {
                // grab the way ID and bounce
                long int wayID;
                const ptree & attributes = f.second.get_child("<xmlattr>", empty_ptree());
                BOOST_FOREACH(const ptree::value_type &v, attributes)
                {
                        std::string attr = lexical_cast<std::string>(v.first.data());
                        if(!attr.compare("id"))
                        {
                                wayID = lexical_cast<long int>(v.second.data());
                                std::cout << attr << " = " << wayID << std::endl;
                                break;
                        }
                }

                this->wayCount++;
                Way* way = new Way(nodeIDs, wayID, wayType, waySpeed);
                this->wayMap.addEntry(this->wayCount, way);
        }

} else if(!tagName.compare("bounds")) {

        double maxLat, maxLon, minLat, minLon;
        const ptree & attributes = f.second.get_child("<xmlattr>", empty_ptree());
        BOOST_FOREACH(const ptree::value_type &v, attributes)
        {
                std::string attr = lexical_cast<std::string>(v.first.data());
                if(!attr.compare("minlat"))
                {
                        minLat = lexical_cast<double>(v.second.data());
                        std::cout << attr << " = " << minLat << std::endl;
                } else if(!attr.compare("minlon")) {
                        minLon = lexical_cast<double>(v.second.data());
                        std::cout << attr << " = " << minLon << std::endl;
                } else if(!attr.compare("maxlat")) {
                        maxLat = lexical_cast<double>(v.second.data());
                        std::cout << attr << " = " << maxLat << std::endl;
                } else if(!attr.compare("maxlon")) {
                        maxLon = lexical_cast<double>(v.second.data());
                        std::cout << attr << " = " << maxLon << std::endl;
                }
```

```
                }
                this->boundsCountXML++;
                Bounds* bounds = new Bounds(maxLat, maxLon, minLat, minLon);
                boundsMapXML.addEntry(this->boundsCountXML, bounds);
            }
        }
}

void DataCollection::pullSRTMData(double lat, double lon) {

        std::cout << "Gathering Elevation Data" << std::endl;

        // get bin for given lat and lon
        double latMax = 60.0;
        double latMin = -56.75;
        double lonMax = 176.56;
        double lonMin = -180.0;

        assert(lat < latMax && lat > latMin && lon < lonMax && lon > lonMin);

        std::string latBin = getBin(latMax, latMin, 24, lat, true);
        std::string lonBin = getBin(lonMax, lonMin, 72, lon, false);

        // create srtm file name from bins
        std::string srtmName = "srtm_" + lonBin + "_" + latBin;

        // check if srtm file already exists and pull it if not
        this->eleFile = srtmName + ".asc";
        std::ifstream test(this->dataFolder + "/" + this->eleFile);
        if(!test)
        {
          // queary srtm for ele data
          std::string srtmZip = this->dataFolder + "/" + srtmName + ".zip";
          std::string serverName = "srtm.csi.cgiar.org";
          std::string getCommand = "/SRT-ZIP/SRTM_v41/SRTM_Data_ArcASCII/" + srtmName + ".zip";

          // place download
          std::string wget = "wget -O " + srtmZip + " http://" + serverName + getCommand;
          system(wget.c_str());

          // unpack zip
          std::string unzip = "sudo unzip " + srtmZip + " -d " + this->dataFolder;
          system(unzip.c_str());
        }
        test.close();

        // read file
        std::ifstream ifs;
        std::string eleFilePath = this->dataFolder + "/" + this->eleFile;
        ifs.open(eleFilePath.c_str(), std::fstream::in | std::fstream::out | std::fstream::app);
```

```
	if(ifs.is_open())
	{
	  std::string line;
	  double eleFeatures[6];

	  std::string feature;
	  for(int i = 0; i < 6; i++)
	  {
		getline(ifs, line);
		std::stringstream ss(line);
		ss >> feature;
		ss >> eleFeatures[i];
	  }

	  this->numEleLatsSRTM = eleFeatures[0];
	  this->numEleLonsSRTM = eleFeatures[1];
	  this->eleLowerLeftLon = eleFeatures[2];
	  this->eleLowerLeftLat = eleFeatures[3];
	  this->eleCellSize = eleFeatures[4];
	  this->voidEle = eleFeatures[5];

	  int row = 0;
	  int col = 0;
	  int **eleRay = new int*[this->numEleLonsSRTM];
	  while(getline(ifs, line))
	  {
		std::stringstream ss(line);
		eleRay[row] = new int[this->numEleLatsSRTM];
		while(ss >> eleRay[row][col]) { col++; }
		if(row % 1000 == 0) { std::cout << row << " ..." << std::endl; }
		col = 0;
		row++;
	  }
	  ifs.close();
	  this->eleDataSRTM = eleRay;
	}
}

std::string DataCollection::getBin(double hi, double lo, int bins, double latLon, bool isLat) {
	double inc = (hi - lo) / bins;
	int bin;
	for(bin = 1; bin < bins; bin++)
	{
	  if(latLon < (lo + bin*inc)) { break; }
	}

	if(isLat) { bin = bins - bin + 1; }

	std::string out;
	if(bin < 10)
```

```
      {
        out = "0";
        out += lexical_cast<std::string>(bin);
      } else {
        out = lexical_cast<std::string>(bin);
      }
      return out;
}

void DataCollection::updateElevationData(GenericMap<long int, Road*>* roads)
{
    const char* latPrint = "\"lat\"";
    const char* lonPrint = "\"lon\"";
    const char* shapePrint = "\"shape\"";

    std::string server = "elevation.mapzen.com";
    std::string command = "/height?json=";
    std::string apiKey = "&api_key=mapzen-4mUVq1A";
    std::string elevFile = this->dataFolder + "/newRoadElevation.txt";

    roads->initializeCounter();
    GenericEntry<long int, Road*>* nextRoad = roads->nextEntry();
    while(nextRoad != NULL)
    {
        std::cout << nextRoad->key << std::endl;

        std::stringstream json;
        json << "{" << shapePrint << ":[";

        // iterate through nodes and add them to string to be concatenated yo
        bool alreadyAssignedElevation = false;
        nextRoad->value->getNodes()->initializeCounter();
        GenericEntry<long int, Node*>* nextNode = nextRoad->value->getNodes()->nextEntry();

        json << "{" << latPrint << ":";
        json << boost::lexical_cast<std::string>(nextNode->value->getLat());
        json << "," << lonPrint << ":";
        json << boost::lexical_cast<std::string>(nextNode->value->getLon());
        json << "}";

        while(true)
        {
            nextNode = nextRoad->value->getNodes()->nextEntry();
            if(nextNode == NULL) { break; }

            if(nextNode->value->getEle() > -500)
            {
                alreadyAssignedElevation = true;
                break;
            }
```

```
    json << ",{" << latPrint << ":";
    json << boost::lexical_cast<std::string>(nextNode->value->getLat());
    json << "," << lonPrint << ":";
    json << boost::lexical_cast<std::string>(nextNode->value->getLon());
    json << "}";
}
delete(nextNode);

if(alreadyAssignedElevation)
{
    nextRoad = roads->nextEntry();
    continue;
}

// check start and end ints if they have elevation
bool startIntHasElevation = nextRoad->value->getStartIntersection()->getElevation() > -500;
bool endIntHasElevation = nextRoad->value->getEndIntersection()->getElevation() > -500;

if(!startIntHasElevation)
{
    json << ",{" << latPrint << ":";
    json << boost::lexical_cast<std::string>(nextRoad->value->getStartIntersection()->getLat());
    json << "," << lonPrint << ":";
    json << boost::lexical_cast<std::string>(nextRoad->value->getStartIntersection()->getLon());
    json << "}";
}

if(!endIntHasElevation)
{
    json << ",{" << latPrint << ":";
    json << boost::lexical_cast<std::string>(nextRoad->value->getEndIntersection()->getLat());
    json << "," << lonPrint << ":";
    json << boost::lexical_cast<std::string>(nextRoad->value->getEndIntersection()->getLon());
    json << "}";
}

// make query
json << "]}";
std::string newCommand = command + json.str() + apiKey;
this->queryFile(server, newCommand, elevFile);

auto start = std::chrono::system_clock::now();
while(1)
{
    auto end = std::chrono::system_clock::now();
    std::chrono::duration<double> diff = end - start;

    if(diff.count() > 1.0) { break; }
}
```

```
        // read query output
        ptree elevData;
        read_json(elevFile, elevData);
        BOOST_FOREACH(ptree::value_type &u, elevData)
        {
            std::string elevFeature = u.first.data();
            if(!elevFeature.compare("height"))
            {
                long int elevCount = 0;
                // iterate through heigts and add them to road
                BOOST_FOREACH(ptree::value_type &v, u.second)
                {
                    float elev = lexical_cast<float>(v.second.data());

                    if(elevCount < nextRoad->value->getNodes()->getSize())
                    {
                        nextRoad->value->getNodes()->getEntry(elevCount)->setEle(elev);
                    }
                    else if(elevCount == nextRoad->value->getNodes()->getSize() && !startIntHasElevation)
                    {
                        nextRoad->value->getStartIntersection()->setElev(elev);
                    }
                    else if(elevCount == nextRoad->value->getNodes()->getSize() && startIntHasElevation && !endIntHasElevation)
                    {
                        nextRoad->value->getEndIntersection()->setElev(elev);
                    }
                    else if(elevCount == nextRoad->value->getNodes()->getSize() + 1 && !startIntHasElevation && !endIntHasElevation)
                    {
                        nextRoad->value->getEndIntersection()->setElev(elev);
                    }

                    elevCount++;
                }
            }
        }

        nextRoad = roads->nextEntry();
    }
    delete(nextRoad);
}

void DataCollection::queryFile(std::string serverName, std::string getCommand, std::string fileName) {

      std::ofstream outFile(fileName, std::ofstream::out | std::ofstream::binary);

      boost::asio::io_service io_service;

      // Get a list of endpoints corresponding to the server name.
      tcp::resolver resolver(io_service);
```

```
tcp::resolver::query query(serverName, "http");
tcp::resolver::iterator endpoint_iterator = resolver.resolve(query);
tcp::resolver::iterator end;

// Try each endpoint until we successfully establish a connection.
tcp::socket socket(io_service);
boost::system::error_code error = boost::asio::error::host_not_found;
while (error && endpoint_iterator != end)
{
  socket.close();
  socket.connect(*endpoint_iterator++, error);
}

boost::asio::streambuf request;
std::ostream request_stream(&request);

request_stream << "GET " << getCommand << " HTTP/1.0\r\n";
request_stream << "Host: " << serverName << "\r\n";
request_stream << "Accept: */*\r\n";
request_stream << "Connection: close\r\n\r\n";

// Send the request.
boost::asio::write(socket, request);

// Read the response status line.
boost::asio::streambuf response;
boost::asio::read_until(socket, response, "\r\n");

// Check that response is OK.
std::istream response_stream(&response);
std::string http_version;
response_stream >> http_version;
unsigned int status_code;
response_stream >> status_code;
std::string status_message;
std::getline(response_stream, status_message);


// Read the response headers, which are terminated by a blank line.
boost::asio::read_until(socket, response, "\r\n\r\n");

// Process the response headers.
std::string header;
while (std::getline(response_stream, header) && header != "\r")
{
}

// Write whatever content we already have to output.
if (response.size() > 0)
{
```

```
        outFile << &response;
      }
      // Read until EOF, writing data to output as we go.
      while (boost::asio::read(socket, response, boost::asio::transfer_at_least(1), error))
      {
        outFile << &response;
      }
}

void DataCollection::checkDataFoler() {
      struct stat sb;
      if(!(stat(this->dataFolder.c_str(), &sb) == 0 && S_ISDIR(sb.st_mode)))
      {
        std::string mkdir = "mkdir " + this->dataFolder;
        system(mkdir.c_str());
      }
}

const ptree& DataCollection::empty_ptree() {
    static ptree t;
    return t;
}

// WARNING! map image is always overwritten
int DataCollection::pullOSMDataPNG(double lat, double lon) {

      std::cout << "Pulling Map Image" << std::endl;

      double latDeltaMin = 10;
      double lonDeltaMin = 10;
    int zoomIdx = this->zoomMin;

    for(int i = this->zoomMin; i <= this->zoomMax; i++)
    {
        std::pair<double, double>* latLonDeltas = this->zoomSpreads.getEntry(i);
        double currLatDelta = latLonDeltas->first;
        double currLonDelta = latLonDeltas->second;

        if(this->latDelta < currLatDelta && this->lonDelta < currLonDelta && this->latDelta < latDeltaMin && this->lonDelta < lonDeltaMin)
        {
            latDeltaMin = currLatDelta;
            lonDeltaMin = currLonDelta;
            zoomIdx = i;
        }
    }

      std::string webkit2png = "/usr/local/bin/webkit2png -W 6000 -H 6000 -o ";
    webkit2png += this->dataFolder + "/" + this->mapPNGName + " ";
    webkit2png += "https://www.openstreetmap.org/export#map=";
      webkit2png += lexical_cast<std::string>(zoomIdx) + "/";
```

```
      webkit2png += lexical_cast<std::string>(lat) + "/";
    webkit2png += lexical_cast<std::string>(lon);

    std::cout << "command used ..." << std::endl;
    std::cout << webkit2png << std::endl;

    system(webkit2png.c_str());

      this->boundsCountPNG++;
      Bounds* newBounds = new Bounds(lat+.5*latDeltaMin, lon+.5*lonDeltaMin, lat-.5*latDeltaMin, lon-.5*lonDeltaMin);
      newBounds->assignID(this->boundsCountPNG);
      this->boundsMapPNG.addEntry(this->boundsCountPNG, newBounds);

      return zoomIdx;
}

void DataCollection::setZoomSpreads() {
      // spreads for zoom index 19
      double latSpread = 0.01075;
      double lonSpread = 0.01515;

      for(int i = this->zoomMax; i >= this->zoomMin; i--)
      {
        this->zoomSpreads.addEntry(i, new std::pair<double, double>(latSpread, lonSpread));
        latSpread *= 2.0;
        lonSpread *= 2.0;
      }
}

std::string DataCollection::getDataFolder() {
      return this->dataFolder;
}

GenericMap<int, Bounds*>* DataCollection::getBoundsMapPNG() {
      return &this->boundsMapPNG;
}

std::string DataCollection::getMapPNGName() {
      return this->mapPNGName;
}

// lat lon increase L->R and B->T
int DataCollection::getElevation(double lat, double lon)
{
      double nextLat, nextLon, latEleDiff, lonEleDiff, latScalar, lonScalar;

      for(int k = this->numEleLatsSRTM; k >= 0; k--)
      {
        nextLat = this->eleLowerLeftLat + (this->numEleLatsSRTM-k+1)*this->eleCellSize;
        if(lat < nextLat)
```

```
        {
                for(int l = 0; l < this->numEleLonsSRTM; l++)
                {
                        nextLon = this->eleLowerLeftLon + (l+1)*this->eleCellSize;
                        if(lon < nextLon)
                        {
                                int elevation = this->eleDataSRTM[l][k];

                                if(elevation == this->voidEle)
                                {
                                        std::cout << "------- void elevation -------" << std::endl;
                                        return elevation;
                                }
                     else if((k < this->numEleLonsSRTM-1) && (l < this->numEleLatsSRTM-1))
                                {
                                        latEleDiff = this->eleDataSRTM[l][k] - this->eleDataSRTM[l+1][k];
                                        lonEleDiff = this->eleDataSRTM[l][k] - this->eleDataSRTM[l][k+1];

                                        latScalar = (nextLat - lat) / this->eleCellSize;
                                        lonScalar = (nextLon - lon) / this->eleCellSize;
                                        assert(latScalar >= 0 && lonScalar >= 0);
                                        assert(latScalar <= 1 && lonScalar <= 1);

                                        latEleDiff *= latScalar;
                                        lonEleDiff *= lonScalar;

                                        elevation += (latEleDiff + lonEleDiff);
                                }
                                return elevation;
                        }
                }
        }
      }
      return -1;
}

GenericMap<long int, Node*>* DataCollection::getNodeMap() {
      return &this->nodeMap;
}

GenericMap<int, Way*>* DataCollection::getWayMap() {
      return &this->wayMap;
}

// for loading into http://www.gpsvisualizer.com/
GenericMap<long int, Road*>* DataCollection::makeRawRoads()
{
    GPS converter;

      std::cout << "making raw roads" << std::endl;
```

```
  std::string csvName = this->dataFolder + "/" + "rawMapData.csv";
  GenericMap<long int, Road*>* rawRoads = new GenericMap<long int, Road*>();

  // delete existing csv if found
  std::string rm = "rm " + csvName;
  system(rm.c_str());

  // create new csv
FILE* csv;
csv = std::fopen(csvName.c_str(), "w");

  // add header to csv
  fprintf(csv, "name, description, color, latitude, longitude\n");

  // begin iterating through ways
  this->wayMap.initializeCounter();
  GenericEntry<int, Way*>* nextWay = this->wayMap.nextEntry();
  while(nextWay != NULL)
  {
    Way* way = nextWay->value;

    // grab nodes from way
    GenericMap<long int, Node*>* nodes = new GenericMap<long int, Node*>();

    // iterate through node IDs of way
     long int node_count = 0;
    way->getNodeIDs()->initializeCounter();
    GenericEntry<int, long int>* nextWayNodeID = way->getNodeIDs()->nextEntry();
    while(nextWayNodeID != NULL)
    {
           Node* node = this->nodeMap.getEntry(nextWayNodeID->value);
           if(node != NULL)
           {
                  nodes->addEntry(node_count, node);
            node_count++;
        }
           nextWayNodeID = way->getNodeIDs()->nextEntry();
    }
    delete(nextWayNodeID);

    // make a raw road if enough nodes exist
    if(nodes->getSize() > 2)
    {
           Road* newRoad = new Road(way->getWayType(), way->getID(), nodes);

           try { // potential source of really weird run-time errors

            // acquire data for curve fitting
            int latLonCount = 0;
```

```
// for splines
       Eigen::MatrixXd points(2, nodes->getSize());

std::cout << "******** node control points ********" << std::endl;

// set control points for spline
       nodes->initializeCounter();
       GenericEntry<long int, Node*>* nextNode = nodes->nextEntry();
       while(nextNode != NULL)
       {
    double lat = nextNode->value->getLat();
    double lon = nextNode->value->getLon();

    // print control points
    printf("%.6f,%.6f\n", lat, lon);

    // for splines
    points(0, latLonCount) = lat;
    points(1, latLonCount) = lon;

              nextNode = nodes->nextEntry();
              latLonCount++;
       }
       delete(nextNode);

       // fit first order spline spline
       typedef Eigen::Spline<double, 2> spline2f;

// fit spline
std::cout << "******** 1st order spline " << "********" << std::endl;
spline2f rawRoadSpline = Eigen::SplineFitting<spline2f>::Interpolate(points, 1);

// evaluate spline and save points to csv for viewing
Eigen::Spline<double,2>::PointType pt = rawRoadSpline(0);
double prev_lat = pt(0,0);
double prev_lon = pt(1,0);
float dist = 0.0;

// get distance of spline to set evaluation distance correctly
for(double u = 0; u <= 1; u += 0.025)
{
    pt = rawRoadSpline(u);
    double curr_lat = pt(0,0);
    double curr_lon = pt(1,0);

    // get distance of eval point along spline
    dist += converter.deltaLatLonToXY(prev_lat, prev_lon, curr_lat, curr_lon);

    prev_lat = curr_lat;
```

```
                prev_lon = curr_lon;
            }

            double evalStepSize = 5.0 / dist;
            pt = rawRoadSpline(0);
            prev_lat = pt(0,0);
            prev_lon = pt(1,0);
            dist = 0.0;

            for(double u = 0; u <= 1; u += evalStepSize)
            {
                pt = rawRoadSpline(u);
                double curr_lat = pt(0,0);
                double curr_lon = pt(1,0);

                // to console
                printf("%.6f,%.6f\n", curr_lat, curr_lon);

                // get distance of eval point along spline
                dist += converter.deltaLatLonToXY(prev_lat, prev_lon, curr_lat, curr_lon);

                // to csv
                fprintf(csv, "%ld,", way->getID());
                fprintf(csv, "Way ID: %ld | ", way->getID());
                fprintf(csv, "Way Type: %s | ", way->getWayType().c_str());
                fprintf(csv, "Way Speed: %d | ", way->getWaySpeed());
                fprintf(csv, "Lat & Lon: %.12f %.12f | ", curr_lat, curr_lon);
                fprintf(csv, "Distance: %.2f,", dist);
                fprintf(csv, "red,");
                fprintf(csv, "%.12f,%.12f\n", curr_lat, curr_lon);

                prev_lat = curr_lat;
                prev_lon = curr_lon;
            }

            newRoad->assignSplineLength(dist);
                   newRoad->assignSpline(rawRoadSpline);
            newRoad->setMinMaxLatLon();

           } catch(const std::exception& e) {
                   std::cout << e.what() << std::endl;
           }

           rawRoads->addEntry(way->getID(), newRoad);
    }
    nextWay = this->wayMap.nextEntry();
  }
  delete(nextWay);

fclose(csv);
```

```
	return rawRoads;
}

int DataCollection::getVoidEle()
{
	return this->voidEle;
}

GenericMap<int, Bounds*>* DataCollection::getBoundsMapXML() {
	return &this->boundsMapXML;
}

}
```

12.28 DataCollection.h

```
/*
 * DataCollection.h
 *
 *  Created on: Apr 4, 2016
 *      Author: vagrant
 */

#ifndef DATACOLLECTION_MAKEOSM_H_
#define DATACOLLECTION_MAKEOSM_H_

#define EIGEN_NO_DEBUG

#include "../city/Road.h"
#include "../city/Intersection.h"
#include "../map/GenericMap.h"
#include "../gps/GPS.h"
#include "boost/foreach.hpp"
#include "boost/property_tree/ptree.hpp"
#include "boost/property_tree/xml_parser.hpp"
#include <boost/property_tree/json_parser.hpp>

// query
#include "boost/lexical_cast.hpp"
#include "boost/asio.hpp"

#include <eigen3/Eigen/Core>
#include <eigen3/unsupported/Eigen/Splines>

#include <chrono>
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <stdio.h>
#include <string>
```

```
#include <stdlib.h>
#include <sys/stat.h>
#include "../data_management/Bounds.h"
#include "../data_management/Node.h"
#include "../data_management/Way.h"

namespace PredictivePowertrain {

class DataCollection {
private:
      double latDelta;
      double lonDelta;
      std::string mapFile;
      std::string eleFile;
      std::string dataFolder = "/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data";
      std::string mapPNGName =  "mapImage.png";
      int** eleDataSRTM;
      int numEleLatsSRTM;
      int numEleLonsSRTM;
      int wayCount;
      int boundsCountXML;
      int boundsCountPNG;
    int zoomMax;
    int zoomMin;
      double eleLowerLeftLat;
      double eleLowerLeftLon;
      double eleCellSize;
      int voidEle;
      int maxVisEntries = 5000;
      GenericMap<long int, Node*> nodeMap;
      GenericMap<int, Way*> wayMap;
      GenericMap<int, Bounds*> boundsMapXML;
      GenericMap<int, Bounds*> boundsMapPNG;
      GenericMap<int, std::pair<double, double>*> zoomSpreads;

      const boost::property_tree::ptree& empty_ptree();
      void queryFile(std::string serverName, std::string getCommand, std::string fileName);
      std::string getBin(double hi, double lo, int bins, double latLon, bool isLat);
      void checkDataFoler();
      void pullSRTMData(double lat, double lon);
      void pullOSMDataXML(double lat, double lon);
      int pullOSMDataPNG(double lat, double lon);
      void setZoomSpreads();
    void initialize(double latDelta, double lonDelta);

public:
      DataCollection();
      DataCollection(double latDelta, double lonDelta);
      void pullDataXML(double lat, double lon);
      int pullDataPNG(double lat, double lon);
```

```
      GenericMap<long int, Node*>* getNodeMap();
      GenericMap<int, Way*>* getWayMap();
      GenericMap<int, Bounds*>* getBoundsMapXML();
      GenericMap<int, Bounds*>* getBoundsMapPNG();
      int getVoidEle();
      GenericMap<long int, Road*>* makeRawRoads();
      std::string getDataFolder();
      std::string getMapPNGName();
      int getElevation(double lat, double lon);
    void updateElevationData(GenericMap<long int, Road*>* roads);

};



} /* namespace PredictivePowertrain */

#endif /* DATACOLLECTION_MAKEOSM_H_ */
```

12.29 DataCollectionUnitTest.cpp

```
/*
 * DataCollectionUnitTest.cpp
 *
 *  Created on: Apr 27, 2016
 *      Author: vagrant
 */

#include "UnitTests.h"
#include "../data_management/DataCollection.h"

using namespace PredictivePowertrain;

void dataCollection_ut() {

      DataCollection testDC;

      std::cout << testDC.pullDataPNG(47.681, -122.328) << std::endl; // greenlake

      testDC.pullDataXML(47.681, -122.328); // greenlake
      std::cout << "node size " << testDC.getNodeMap()->getSize() << std::endl;
      std::cout << "boundCount " << testDC.getBoundsMapXML()->getSize() << std::endl;

      std::cout << "road Count " << testDC.makeRawRoads()->getSize() << std::endl;
      GenericMap<long int, Road*>* roads = testDC.makeRawRoads();

      testDC.pullDataXML(47.618174, -122.330838); // downtown
      std::cout << "node size " << testDC.getNodeMap()->getSize() << std::endl;
      std::cout << "boundCount " << testDC.getBoundsMapXML()->getSize() << std::endl;
      std::cout << "road Count " << testDC.makeRawRoads()->getSize() << std::endl;
}
```

12.30 DataManagement.cpp

```
/*
 * DataManagement.cpp
 *
 *  Created on: Apr 18, 2016
 *      Author: vagrant
 */

#include "DataManagement.h"

using boost::property_tree::ptree;
using boost::lexical_cast;

namespace PredictivePowertrain {

DataManagement::DataManagement() {
      std::string fileRay[3] = {this->routePredictionData, this->cityData, this->tripData};
      for(int i = 0; i < 3; i++)
      {
        std::ifstream dataFile(fileRay[i].c_str());
        if(!dataFile.good())
        {
                std::ofstream newFile(fileRay[i].c_str());
                newFile.close();
        }
        dataFile.close();
      }
}

int DataManagement::countFileLine(std::string fileLoc)
{
    int numLines = 0;
    std::string line;
    std::ifstream file(fileLoc.c_str());

    if(file.is_open()){
        while(!file.eof())
        {
            std::getline(file,line);
            numLines++;
        }
        file.close();
    }

    return numLines;
}

void DataManagement::addRoutePredictionData(RoutePrediction* rp)
{
```

```
std::cout << "updating route predicition data" << std::endl;
// links
ptree links_ptree;

std::ofstream out(this->nnDataLoc.c_str(), std::ios::out | std::ios::binary | std::ios::trunc);

rp->getLinks()->initializeCounter();
GenericEntry<long int, Link*>* nextLink = rp->getLinks()->nextEntry();
while(nextLink != NULL)
{
    ptree link_ptree;

    link_ptree.put("NUMBER", nextLink->value->getNumber());
    link_ptree.put("DIRECTION", nextLink->value->getDirection());

    // print NN weights and activations
    if(nextLink->value->linkHasWeights())
    {
        // get NN data
        std::vector<std::vector<std::vector<Eigen::MatrixXd*>*>*> nnData;
        std::vector<std::string> nnDataType;

        if(nextLink->value->linkHasAWeights())
        {
            nnData.push_back(nextLink->value->getWeights(1));
            nnDataType.push_back("A");
        }

        if(nextLink->value->linkHasBWeights())
        {
            nnData.push_back(nextLink->value->getWeights(0));
            nnDataType.push_back("B");
        }

        // write to json
        if(this->jsonifyLinkNNData)
        {
            for(int h = 0; h < nnData.size(); h++)
            {
                for(int i = 0; i < nnData.at(h)->size(); i++)
                {
                    ptree type_ptree;

                    type_ptree.put("SIZE", nnData.at(h)->at(i)->size());

                    for(int j = 0; j < nnData.at(h)->at(i)->size(); j++)
                    {
                        ptree type_i_ptree;

                        type_i_ptree.put("ROWS", nnData.at(h)->at(i)->at(j)->rows());
```

```
                type_i_ptree.put("COLS", nnData.at(h)->at(i)->at(j)->cols());

                for(int row = 0; row < nnData.at(h)->at(i)->at(j)->rows(); row++)
                {
                    ptree type_i_row_ptree;

                    for(int col = 0; col < nnData.at(h)->at(i)->at(j)->cols(); col++)
                    {

                        type_i_row_ptree.put(lexical_cast<std::string>(col), nnData.at(h)->at(i)->at(j)->coeffRef(row, col));

                    }

                    type_i_ptree.push_back(std::make_pair(lexical_cast<std::string>(row), type_i_row_ptree));

                }

                type_ptree.push_back(std::make_pair(lexical_cast<std::string>(j), type_i_ptree));

            }

            std::string type = "wts";
            if(i == 1)
            {
                type = "yHid";
            }

            else if(i == 2)
            {
                type = "yInHid";
            }

            link_ptree.push_back(std::make_pair(type + nnDataType.at(h), type_ptree));

        }
    }
}

// binary serialization
else
{
    bool addedLink = false;

    // type A or B
    for(int h = 0; h < nnData.size(); h++)
    {
        // wts or yHidAct or yInAct
        for(int i = 0; i < nnData.at(h)->size(); i++)
        {
            // layer
```

```
                    for(int j = 0; j < nnData.at(h)->at(i)->size(); j++)
                    {
                        Eigen::MatrixXd* nnMat = nnData.at(h)->at(i)->at(j);

                        if(nnMat != NULL)
                        {
                            addedLink = true;

                            // write it twice becuase last mat written sometimes doesnt get picked up in read
                            this->writeBinaryNNMat(out, *nnMat, i, j, nextLink->value->getHash());
                            this->writeBinaryNNMat(out, *nnMat, i, j, nextLink->value->getHash());
                        }
                    }
                }
            }

            if(addedLink)
            {
                std::cout << nextLink->value->getNumber() << std::endl;
            }
        }
    }

    links_ptree.push_back(std::make_pair(lexical_cast<std::string>(nextLink->value->getHash()),link_ptree));

    nextLink = rp->getLinks()->nextEntry();
}
delete(nextLink);
out.close();

// goals
ptree goals_ptree;
rp->getGoals()->initializeCounter();
GenericEntry<long int, Goal*>* nextGoal = rp->getGoals()->nextEntry();
while(nextGoal != NULL)
{
    ptree goal_ptree, bins_ptree, numSeen_ptree;

    // add conditions
    for(int i = 0; i < nextGoal->value->getBins()->size(); i++)
    {
        ptree bin_ptree;
        bin_ptree.put("", nextGoal->value->getBins()->at(i));
        bins_ptree.push_back(std::make_pair(boost::lexical_cast<std::string>(i), bin_ptree));
    }

    // add num seen
    numSeen_ptree.put("", nextGoal->value->getNumSeen());

    // add conditions
```

```
        goal_ptree.push_back(std::make_pair("conditions", bins_ptree));

        // add number of times goal is seen
        goal_ptree.push_back(std::make_pair("numSeen", numSeen_ptree));

        // add goal to tree of goal
        goals_ptree.push_back(std::make_pair(boost::lexical_cast<std::string>(nextGoal->value->getDestination()), goal_ptree));

        nextGoal = rp->getGoals()->nextEntry();
    }
    delete(nextGoal);

    // link to state map
    ptree linkToStateMap_ptree;

    rp->getLinkToState()->getGoalMap()->initializeCounter();
    GenericEntry<long  int,  GoalMapEntry<long  int,  LinkToStateMapEntry*>*>*  nextGoalMapEntryL2S  =  rp->getLinkToState()->getGoalMap()-
>nextEntry();
    while(nextGoalMapEntryL2S != NULL)
    {
        ptree nextGoalMapEntryL2S_ptree;

        nextGoalMapEntryL2S->value->getMap()->initializeCounter();
        GenericEntry<long int, LinkToStateMapEntry*>* nextLinkToStateMap = nextGoalMapEntryL2S->value->getMap()->nextEntry();
        while(nextLinkToStateMap != NULL)      {
            ptree nextLinkToStateMap_ptree;

            nextLinkToStateMap->value->getEntries()->initializeCounter();
            GenericEntry<long int, int>* nextLinkToStateEntry = nextLinkToStateMap->value->getEntries()->nextEntry();
            while(nextLinkToStateEntry != NULL)
            {
                ptree nextLinkToStateMapEntry_ptree;

                nextLinkToStateMapEntry_ptree.put("", nextLinkToStateEntry->value);

                // add entry to linkToStateMap Entry
                nextLinkToStateMap_ptree.push_back(std::make_pair(lexical_cast<std::string>(nextLinkToStateEntry->key),
nextLinkToStateMapEntry_ptree));

                nextLinkToStateEntry = nextLinkToStateMap->value->getEntries()->nextEntry();
            }
            delete(nextLinkToStateEntry);

            // add linkToStateMap to goalMapEntry
            nextGoalMapEntryL2S_ptree.push_back(std::make_pair(lexical_cast<std::string>(nextLinkToStateMap->key), nextLinkToStateMap_ptree));

            nextLinkToStateMap = nextGoalMapEntryL2S->value->getMap()->nextEntry();
        }
        delete(nextLinkToStateMap);
```

```
        // add next goal map entry to link to state map
        linkToStateMap_ptree.push_back(std::make_pair(boost::lexical_cast<std::string>(nextGoalMapEntryL2S->key), nextGoalMapEntryL2S_ptree));

        nextGoalMapEntryL2S = rp->getLinkToState()->getGoalMap()->nextEntry();
    }
    delete(nextGoalMapEntryL2S);

    // goal to link map
    ptree goalToLinkMap_ptree;

    rp->getGoalToLink()->getGoalMap()->initializeCounter();
    GenericEntry<long int, GoalMapEntry<long int, int>*>* nextGoalMapEntryG2L = rp->getGoalToLink()->getGoalMap()->nextEntry();
    while(nextGoalMapEntryG2L != NULL)
    {
        ptree nextGoalMapEntryG2L_ptree;

        nextGoalMapEntryG2L->value->getMap()->initializeCounter();
        GenericEntry<long int, int>* nextGoalToLinkEntry = nextGoalMapEntryG2L->value->getMap()->nextEntry();
        while(nextGoalToLinkEntry != NULL)
        {
            ptree nextGoalToLinkEntry_ptree;

            nextGoalToLinkEntry_ptree.put("", nextGoalToLinkEntry->value);

            // add goal to link association to goal to link map
            nextGoalMapEntryG2L_ptree.push_back(std::make_pair(lexical_cast<std::string>(nextGoalToLinkEntry->key),
nextGoalToLinkEntry_ptree));

            nextGoalToLinkEntry = nextGoalMapEntryG2L->value->getMap()->nextEntry();
        }
        delete(nextGoalToLinkEntry);

        // add goal to link map entry to tree of all map entries
        goalToLinkMap_ptree.push_back(std::make_pair(boost::lexical_cast<std::string>(nextGoalMapEntryG2L->key), nextGoalMapEntryG2L_ptree));

        nextGoalMapEntryG2L = rp->getGoalToLink()->getGoalMap()->nextEntry();
    }
    delete(nextGoalMapEntryG2L);

    // add all tree to route prediction json tree
    ptree rp_ptree;
    rp_ptree.push_back(std::make_pair("LINKS", links_ptree));
    rp_ptree.push_back(std::make_pair("GOALS", goals_ptree));
    rp_ptree.push_back(std::make_pair("LINK2STATE", linkToStateMap_ptree));
    rp_ptree.push_back(std::make_pair("GOAL2LINK", goalToLinkMap_ptree));

    write_json(this->routePredictionData, rp_ptree);

}
```

```
void DataManagement::addCityData(City* city)
{
    std::cout << "updating city data" << std::endl;
      GenericMap<long int, Road*>* roadMap = city->getRoads();
      GenericMap<long int, Intersection*>* intersectionMap = city->getIntersections();
      GenericMap<int, Bounds*>* boundsMap = city->getBoundsMap();
      GenericMap<int, Bounds*>* newBoundsMap = new GenericMap<int, Bounds*>();

    ptree cityLogs;
      bool newBounds = false;
    if(this->countFileLine(this->cityData) == 0)
    {
        newBounds = true;
        delete(newBoundsMap);
        newBoundsMap = boundsMap;
    }
    else
    {
        try {
            read_json(this->cityData, cityLogs);

            boundsMap->initializeCounter();
            GenericEntry<int, Bounds*>* nextBounds = boundsMap->nextEntry();
            while(nextBounds != NULL)
            {
                bool boundsLogged = false;
                BOOST_FOREACH(ptree::value_type& v, cityLogs)
                {
                    int boundsID = lexical_cast<int>(v.first.data());
                    if(boundsID == nextBounds->key)
                    {
                        boundsLogged = true;
                        break;
                    }
                }
                if(!boundsLogged) {
                    newBounds = true;
                    newBoundsMap->addEntry(nextBounds->key, NULL);
                }
                nextBounds = boundsMap->nextEntry();
            }
        } catch(std::exception& e) {
            std::cout << e.what() << std::endl;
            newBoundsMap = boundsMap;
            newBounds = true;
        }
    }

      if(newBounds)
      {
```

```
// add road data
newBoundsMap->initializeCounter();
GenericEntry<int, Bounds*>* nextBounds = newBoundsMap->nextEntry();
while(nextBounds != NULL)
{
        ptree boundsData, roads_ptree, intersections_ptree;

     std::ofstream out(this->nodeDataLoc.c_str(), std::ios::out | std::ios::binary | std::ios::trunc);

        roadMap->initializeCounter();
        GenericEntry<long int, Road*>* nextRoad = roadMap->nextEntry();
        while(nextRoad != NULL)
        {
                if(nextRoad->value->getBoundsID() == nextBounds->value->getID())
                {
                        ptree road, startNode, endNode, roadType, splineLength;
                        startNode.put("", nextRoad->value->getStartIntersection()->getIntersectionID());
                        endNode.put("", nextRoad->value->getEndIntersection()->getIntersectionID());
                        roadType.put("", nextRoad->value->getRoadType());
            splineLength.put("", nextRoad->value->getSplineLength());

            if(this->jsonifyRoadNodes)
            {
                ptree lats, lons, nodeElevations, nodeIDs, nodes;
                nextRoad->value->getNodes()->initializeCounter();
                GenericEntry<long int, Node*>* nextNode = nextRoad->value->getNodes()->nextEntry();
                while(nextNode != NULL)
                {
                    Node* node = nextNode->value;
                    ptree lat, lon, ele, id;
                    lat.put("", node->getLat());
                    lon.put("", node->getLon());
                    ele.put("", node->getEle());
                    id.put("", node->getID());

                    lats.push_back(std::make_pair("", lat));
                    lons.push_back(std::make_pair("", lon));
                    nodeElevations.push_back(std::make_pair("", ele));
                    nodeIDs.push_back(std::make_pair("", id));

                    nextNode = nextRoad->value->getNodes()->nextEntry();
                }
                delete(nextNode);

                nodes.push_back(std::make_pair("latitude", lats));
                nodes.push_back(std::make_pair("longitude", lons));
                nodes.push_back(std::make_pair("elevation", nodeElevations));
                nodes.push_back(std::make_pair("nodeIDs", nodeIDs));
                road.push_back(std::make_pair("nodes", nodes));
            }
```

```
        else
        {
            Eigen::MatrixXd nodesMat(nextRoad->value->getNodes()->getSize(), 4);
            int nodeRow = 0;

            nextRoad->value->getNodes()->initializeCounter();
            GenericEntry<long int, Node*>* nextNode = nextRoad->value->getNodes()->nextEntry();
            while(nextNode != NULL)
            {
                Node* node = nextNode->value;

                nodesMat.coeffRef(nodeRow, 0) = node->getLat();
                nodesMat.coeffRef(nodeRow, 1) = node->getLon();
                nodesMat.coeffRef(nodeRow, 2) = node->getEle();
                nodesMat.coeffRef(nodeRow, 3) = node->getID();
                nodeRow++;

                nextNode = nextRoad->value->getNodes()->nextEntry();
            }
            delete(nextNode);

            this->writeBinaryNodeMat(out, nodesMat, nextRoad->value->getRoadID());
        }

                  road.push_back(std::make_pair("startNodeID", startNode));
                  road.push_back(std::make_pair("endNodeID", endNode));
                  road.push_back(std::make_pair("roadType", roadType));
       road.push_back(std::make_pair("splineLength", splineLength));

                  roads_ptree.add_child(lexical_cast<std::string>(nextRoad->value->getRoadID()), road);
          }
          nextRoad = roadMap->nextEntry();
   }
delete(nextRoad);

out.close();

// add intersecion data
   intersectionMap->initializeCounter();
   GenericEntry<long int, Intersection*>* nextIntersection = intersectionMap->nextEntry();
   while(nextIntersection != NULL)
   {
          if(nextIntersection->value->getBoudsID() == nextBounds->value->getID())
          {
                 ptree intersection, roadCount, elevation, lat, lon;
                 roadCount.put("", nextIntersection->value->getRoadCount());
                 elevation.put("", nextIntersection->value->getElevation());
                 lat.put("", nextIntersection->value->getLat());
                 lon.put("", nextIntersection->value->getLon());
```

```
                                ptree roadIDs;
                                GenericMap<long int, Road*>* connectingRoads = nextIntersection->value->getRoads();
                                connectingRoads->initializeCounter();
                                GenericEntry<long int, Road*>* nextRoad = connectingRoads->nextEntry();
                                while(nextRoad != NULL)
                                {
                                        ptree roadID;
                                        roadID.put("", nextRoad->value->getRoadID());
                                        roadIDs.push_back(std::make_pair("", roadID));
                                        nextRoad = connectingRoads->nextEntry();
                                }

                                intersection.push_back(std::make_pair("roadCount", roadCount));
                                intersection.push_back(std::make_pair("elevation", elevation));
                                intersection.push_back(std::make_pair("latitude", lat));
                                intersection.push_back(std::make_pair("longitude", lon));
                                intersection.push_back(std::make_pair("roadIDs", roadIDs));

                                intersections_ptree.add_child(lexical_cast<std::string>(nextIntersection->value->getIntersectionID()),
intersection);
                        }
                        nextIntersection = intersectionMap->nextEntry();
                }
            delete(nextIntersection);

                ptree bounds, maxLat, maxLon, minLat, minLon;
                maxLat.put("", boundsMap->getEntry(nextBounds->key)->getMaxLat());
                maxLon.put("", boundsMap->getEntry(nextBounds->key)->getMaxLon());
                minLat.put("", boundsMap->getEntry(nextBounds->key)->getMinLat());
                minLon.put("", boundsMap->getEntry(nextBounds->key)->getMinLon());

                bounds.push_back(std::make_pair("maxLat", maxLat));
                bounds.push_back(std::make_pair("maxLon", maxLon));
                bounds.push_back(std::make_pair("minLat", minLat));
                bounds.push_back(std::make_pair("minLon", minLon));

                boundsData.push_back(std::make_pair("roads", roads_ptree));
                boundsData.push_back(std::make_pair("intersections", intersections_ptree));
                boundsData.push_back(std::make_pair("bounds", bounds));

                cityLogs.add_child(lexical_cast<std::string>(nextBounds->key), boundsData);
                nextBounds = newBoundsMap->nextEntry();
        }
        write_json(this->cityData, cityLogs);
      }
}

void DataManagement::addTripData(GenericMap<long int, std::pair<double, double>*>* latLon)
{
    std::cout << "updating trip log" << std::endl;
```

```
      int tripID = 0;
      ptree tripLogs;

    int numLines = this->countFileLine(this->tripData);
    if(numLines > 1)
    {
        // check for existing trips
        try
        {
            read_json(this->tripData, tripLogs);
            BOOST_FOREACH(ptree::value_type &v, tripLogs)
            {
                tripID = lexical_cast<int>(v.first.data());
            }
        }
        catch(const std::exception& e)
        {
            std::cout << e.what() << std::endl;
        }
    }

      // add trip
      ptree trip, allLat, allLon;
      latLon->initializeCounter();
      GenericEntry<long int, std::pair<double, double>*>* nextLatLon = latLon->nextEntry();
      while(nextLatLon != NULL)
      {
        ptree lat, lon;
        lat.put("", nextLatLon->value->first);
        lon.put("", nextLatLon->value->second);

        allLat.push_back(std::make_pair("", lat));
        allLon.push_back(std::make_pair("", lon));
        nextLatLon = latLon->nextEntry();
      }

      trip.push_back(std::make_pair("latitude", allLat));
      trip.push_back(std::make_pair("longitude", allLon));

      tripLogs.add_child(lexical_cast<std::string>(tripID+1), trip);

      write_json(this->tripData, tripLogs);
}


RoutePrediction* DataManagement::getRoutePredictionData()
{
    std::cout << "getting route prediciton data" << std::endl;
      try
```

```
{

    ptree rpLogs;
    GenericMap<long int, Link*>* links = new GenericMap<long int, Link*>();
    GenericMap<long int, Goal*>* goals = new GenericMap<long int, Goal*>();
    LinkToStateMap* linkToState = new LinkToStateMap();
    GoalToLinkMap* goalToLink = new GoalToLinkMap();

   read_json(this->routePredictionData, rpLogs);
   BOOST_FOREACH(ptree::value_type &v, rpLogs)
   {
        std::string rpData = v.first.data();

        // LINKS
        if(!rpData.compare("LINKS"))
        {
            bool hasANNVals = false;
            bool hasBNNVals = false;

            std::cout << "in get rp, links" << std::endl;
            BOOST_FOREACH(ptree::value_type &u, v.second)
            {
                int linkDirection;
                long int linkNumber;
                long int linkHash = lexical_cast<long int>(u.first.data());

                std::vector<Eigen::MatrixXd*>* wtsA;
                std::vector<Eigen::MatrixXd*>* wtsB;
                std::vector<Eigen::MatrixXd*>* yHidA;
                std::vector<Eigen::MatrixXd*>* yHidB;
                std::vector<Eigen::MatrixXd*>* yInHidA;
                std::vector<Eigen::MatrixXd*>* yInHidB;

                BOOST_FOREACH(ptree::value_type &s, u.second)
                {
                    std::string linkDataType = s.first.data();

                    // read direction
                    if(!linkDataType.compare("DIRECTION"))
                    {
                        linkDirection = lexical_cast<int>(s.second.data());
                    }
                    // read link number
                    else if(!linkDataType.compare("NUMBER"))
                    {
                        linkNumber = lexical_cast<long int>(s.second.data());
                    }
                    // read one of the NN matrices
                    else if(this->jsonifyLinkNNData)
                    {
```

```
// get matrix array size
int matRaySize;
BOOST_FOREACH(ptree::value_type &t, s.second)
{
    std::string matFeatureRayType = t.first.data();

    if(!matFeatureRayType.compare("SIZE"))
    {
        matRaySize = lexical_cast<int>(t.second.data());
        break;
    }
}

// iterate through matrix array to
std::vector<Eigen::MatrixXd*>* newMatRay = new std::vector<Eigen::MatrixXd*>(matRaySize);
BOOST_FOREACH(ptree::value_type &t, s.second)
{
    std::string matFeatureRayType = t.first.data();

    if(matFeatureRayType.compare("SIZE"))
    {
        int matRayIndex = lexical_cast<int>(matFeatureRayType);
        int rows = -1;
        int cols = -1;

        // iterate through matrix data to get size
        BOOST_FOREACH(ptree::value_type &x, t.second)
        {
            std::string matFeatureType = x.first.data();
            if(!matFeatureType.compare("ROWS"))
            {
                rows = lexical_cast<int>(x.second.data());
            }

            else if(!matFeatureType.compare("COLS"))
            {
                cols = lexical_cast<int>(x.second.data());
            }

            if(rows != -1 && cols != -1)
            {
                break;
            }
        }

        // iterate thorugh matrix to populate data
        newMatRay->at(matRayIndex) = new Eigen::MatrixXd(rows, cols);
        BOOST_FOREACH(ptree::value_type &x, t.second)
        {
            std::string matFeatureType = x.first.data();
```

```
            if(matFeatureType.compare("ROWS") && matFeatureType.compare("COLS"))
            {
                int row = lexical_cast<int>(matFeatureType);

                // iterate along matrix row
                BOOST_FOREACH(ptree::value_type &y, x.second)
                {
                    int col = lexical_cast<int>(y.first.data());
                    newMatRay->at(matRayIndex)->coeffRef(row, col) = lexical_cast<double>(y.second.data());
                }
            }
        }
    }
}

if(!linkDataType.compare("wtsA"))
{
    hasANNVals = true;
    wtsA = newMatRay;
}

else if(!linkDataType.compare("wtsB"))
{
    hasBNNVals = true;
    wtsB = newMatRay;
}

else if(!linkDataType.compare("yHidA"))
{
    hasANNVals = true;
    yHidA = newMatRay;
}

else if(!linkDataType.compare("yHidB"))
{
    hasBNNVals = true;
    yHidB = newMatRay;
}

else if(!linkDataType.compare("yInHidA"))
{
    hasANNVals = true;
    yInHidA = newMatRay;
}

else if(!linkDataType.compare("yInHidB"))
{
    hasBNNVals = true;
    yInHidB = newMatRay;
}
```

```
        }
    }

    Link* newLink = new Link(linkDirection, linkNumber);

    if(hasANNVals)
    {
        newLink->setWeights(wtsA, yHidA, yInHidA, 1);
    }

    if(hasBNNVals)
    {
        newLink->setWeights(wtsB, yHidB, yInHidB, 0);
    }

    links->addEntry(linkHash, newLink);
}

// add final link because link to state include final link
Link* finalLink = Link().finalLink();
links->addEntry(finalLink->getHash(), finalLink);

// read binary data if available
if(!this->jsonifyLinkNNData && this->countFileLine(this->nnDataLoc) > 2)
{
    // need copy of speed prediction to get the number of layers used
    SpeedPrediction sp;

    // map nn data to link hash
    GenericMap<long int, std::vector<std::vector<Eigen::MatrixXd*>*>*> nnDataMap;

    // open binary containing link nn data
    std::ifstream in(this->nnDataLoc.c_str(), std::ios::in | std::ios::binary);

    // matrix and matrix identifiers
    Eigen::MatrixXd nnMat;
    int matrixTypeNum;
    int layerNum;
    long int matLinkHash;

    // begin reading binary
    while(!in.eof())
    {
        // read binary
        this->readBinaryNNMat(in, nnMat, matrixTypeNum, layerNum, matLinkHash);

        // create new mat pointer and pointer to nn data container
        Eigen::MatrixXd* newNNMat = new Eigen::MatrixXd(nnMat);
        std::vector<std::vector<Eigen::MatrixXd*>*>* nnData;
```

```
            // grab nn data container if available
            if(nnDataMap.hasEntry(matLinkHash))
            {
                nnData = nnDataMap.getEntry(matLinkHash);
            }

            // create new container if not
            else
            {
                nnData = new std::vector<std::vector<Eigen::MatrixXd*>*>(3);

                for(int i = 0; i < nnData->size(); i++)
                {
                    nnData->at(i) = new std::vector<Eigen::MatrixXd*>(sp.getNumLayers());
                }

                // add new nn data container to map of nn data to link hashes
                nnDataMap.addEntry(matLinkHash, nnData);
            }

            // update the map
            if(nnData->at(matrixTypeNum)->at(layerNum) == NULL)
            {
                nnData->at(matrixTypeNum)->at(layerNum) = newNNMat;
            }
        }
        in.close();

        // iterate through map of link nn data and update map of links
        nnDataMap.initializeCounter();
        GenericEntry<long int, std::vector<std::vector<Eigen::MatrixXd*>*>*>* nextNNData = nnDataMap.nextEntry();
        while(nextNNData != NULL)
        {
            long int nextLinkHash = nextNNData->key;
            std::vector<std::vector<Eigen::MatrixXd*>*>* nnData = nextNNData->value;

            if(links->hasEntry(nextLinkHash))
            {
                Link* storedLink = links->getEntry(nextLinkHash);
                std::cout << storedLink->getNumber() << std::endl;
                storedLink->setWeights(nnData->at(0), nnData->at(1), nnData->at(2), storedLink->getDirection());
            }

            nextNNData = nnDataMap.nextEntry();
        }
        delete(nextNNData);
    }
}

// GOALS
```

```
else if(!rpData.compare("GOALS"))
{
    std::cout << "in get rp, goals" << std::endl;
    BOOST_FOREACH(ptree::value_type &u, v.second)
    {
        long int destinationNum = lexical_cast<long int>(u.first.data());
        std::vector<float>* conditionsVec;
        int numSeen;

        // get conditions and numseen
        BOOST_FOREACH(ptree::value_type &t, u.second)
        {
            std::string goalFeature = t.first.data();

            if(!goalFeature.compare("conditions"))
            {
                // get conditions from json
                GenericMap<int, float> conditionsMap;
                BOOST_FOREACH(ptree::value_type &s, t.second)
                {
                    int index = lexical_cast<int>(s.first.data());
                    float condition = lexical_cast<float>(s.second.data());

                    conditionsMap.addEntry(index, condition);
                }

                // add conditions to vector
                conditionsVec = new std::vector<float>(conditionsMap.getSize());

                conditionsMap.initializeCounter();
                GenericEntry<int, float>* nextCondition = conditionsMap.nextEntry();
                while(nextCondition != NULL)
                {
                    conditionsVec->at(nextCondition->key) = nextCondition->value;
                    nextCondition = conditionsMap.nextEntry();
                }
                delete(nextCondition);
            } else if(!goalFeature.compare("numSeen"))
            {
                numSeen = lexical_cast<int>(t.second.data());
            }
        }

        Goal* newGoal = new Goal(destinationNum, conditionsVec);
        newGoal->setNumSeen(numSeen);

        goals->addEntry(newGoal->getHash(), newGoal);
    }
}
```

```
// LINK2STATE
else if(!rpData.compare("LINK2STATE"))
{
    std::cout << "in get rp, link2state" << std::endl;
    BOOST_FOREACH(ptree::value_type &u, v.second)
    {
        // iterate through each goal map entry
        long int goalHash = lexical_cast<long int>(u.first.data());
        assert(goals->hasEntry(goalHash));

        BOOST_FOREACH(ptree::value_type &s, u.second)
        {
            // grab all link assocations for a particular goal
            long int fromLinkHash = lexical_cast<long int>(s.first.data());
            assert(links->hasEntry(fromLinkHash));

            GenericMap<long int, int> toLinkAssociations;
            BOOST_FOREACH(ptree::value_type &t, s.second)
            {
                long int toLinkHash = lexical_cast<long int>(t.first.data());
                int numAssociated = lexical_cast<int>(t.second.data());
                toLinkAssociations.addEntry(toLinkHash, numAssociated);
            }

            // populate link to state map using known associations
            toLinkAssociations.initializeCounter();
            GenericEntry<long int, int>* nextToLinkAssociation = toLinkAssociations.nextEntry();
            while(nextToLinkAssociation != NULL)
            {
                assert(links->hasEntry(nextToLinkAssociation->key));

                Link* fromLink = links->getEntry(fromLinkHash);
                Link* toLink = links->getEntry(nextToLinkAssociation->key);
                Goal* toGoal = goals->getEntry(goalHash);

                // make number of associations specified
                for(int i = 0; i < nextToLinkAssociation->value; i++)
                {
                    linkToState->incrementTransition(fromLink, toGoal, toLink);
                }

                nextToLinkAssociation = toLinkAssociations.nextEntry();
            }
            delete(nextToLinkAssociation);
        }
    }
}

// GOAL2LINK
else if(!rpData.compare("GOAL2LINK"))
```

```
            {
                std::cout << "in get rp, goal2link" << std::endl;
                BOOST_FOREACH(ptree::value_type &u, v.second)
                {
                    long int goalHash = lexical_cast<long int>(u.first.data());
                    assert(goals->hasEntry(goalHash));

                    // grab all link assocations to given goal
                    BOOST_FOREACH(ptree::value_type &t, u.second)
                    {
                        long int linkHash = lexical_cast<long int>(t.first.data());
                        assert(links->hasEntry(linkHash));

                        // impress assocations on goal2link map
                        int numAssociations = lexical_cast<int>(t.second.data());
                        for(int i = 0; i < numAssociations; i++)
                        {
                            Link* linki = links->getEntry(linkHash);
                            Goal* goali = goals->getEntry(goalHash);

                            goalToLink->linkTraversed(linki, goali);
                        }
                    }
                }
            }
        }

        RoutePrediction* rp = new RoutePrediction();
        rp->addPredictionElements(links, goals, goalToLink, linkToState);

        return rp;
      } catch(const std::exception& e) {
        std::cout << e.what() << std::endl;
      }

    return new RoutePrediction();
}

City* DataManagement::getCityData()
{
    std::cout << "getting city data" << std::endl;
    if(this->countFileLine(this->cityData) == 0)
    {
        return NULL;
    }

      ptree cityLogs;
      try
    {
        read_json(this->cityData, cityLogs);
```

```
        GenericMap<long int, Road*>* roads = new GenericMap<long int, Road*>();
        GenericMap<long int, Intersection*>* intersections = new GenericMap<long int, Intersection*>();
        GenericMap<int, Bounds*>* bounds = new GenericMap<int, Bounds*>();
        GenericMap<long  int,  std::pair<int,  int>*>*  roadIntersections  =  new  GenericMap<long  int,  std::pair<int,  int>*>();  //  <roadID,
<startID, endID>>

        // get roads first
        BOOST_FOREACH(ptree::value_type& u, cityLogs)
        {
                int boundsID = lexical_cast<int>(u.first.data());
                BOOST_FOREACH(ptree::value_type& v, u.second)
                {
                        std::string child = v.first.data();
                        if(!child.compare("roads"))
                 {

                    GenericMap<long int, GenericMap<long int, Node*>*> roadsNodes;

                    if(!this->jsonifyRoadNodes)
                    {
                        std::ifstream in(this->nodeDataLoc.c_str(), std::ios::in | std::ios::binary);

                        Eigen::MatrixXd nodeMat;
                        long int roadID;

                        while(!in.eof())
                        {
                            this->readBinaryNodeMat(in, nodeMat, roadID);

                            GenericMap<long int, Node*>* roadNodes = new GenericMap<long int, Node*>();
                            for(int i = 0; i < nodeMat.rows(); i++)
                            {
                                double lat = nodeMat.coeffRef(i, 0);
                                double lon = nodeMat.coeffRef(i, 1);
                                float ele = nodeMat.coeffRef(i, 2);
                                long int id = nodeMat.coeffRef(i, 3);

                                Node* newNode = new Node(lat, lon, ele, id);

                                roadNodes->addEntry(i, newNode);
                            }

                            roadsNodes.addEntry(roadID, roadNodes);
                        }

                        in.close();
                    }

                    std::cout << "in get city, roads" << std::endl;
                              BOOST_FOREACH(ptree::value_type& z, v.second)
```

```
      {
GenericMap<int, double> nodeLats;
GenericMap<int, double> nodeLons;
GenericMap<int, int> nodeEles;
GenericMap<int, long int> nodeIDs;

              long int roadID = lexical_cast<long int>(z.first.data());
              long int startNodeID, endNodeID;
float splineLength;
              std::string roadType;

              BOOST_FOREACH(ptree::value_type& a, z.second)
              {
                     std::string roadFeature = a.first.data();
                     if(!roadFeature.compare("startNodeID")) {
                            startNodeID = lexical_cast<int>(a.second.data());
                     } else if(!roadFeature.compare("endNodeID")) {
                            endNodeID = lexical_cast<int>(a.second.data());
                     } else if(!roadFeature.compare("roadType")) {
                            roadType = a.second.data();
    }else if(!roadFeature.compare("splineLength")) {
        splineLength = lexical_cast<float>(a.second.data());
                     } else if(!roadFeature.compare("nodes") && this->jsonifyRoadNodes) {
                            int latCount = 0; int lonCount = 0; int eleCount = 0; int idCount = 0;
                            BOOST_FOREACH(ptree::value_type& b, a.second)
                            {
                                   std::string nodeFeature = b.first.data();
                                   BOOST_FOREACH(ptree::value_type& c, b.second)
                                   {
                                          if(!nodeFeature.compare("latitude")) {
                                                 nodeLats.addEntry(latCount, lexical_cast<double>(c.second.data()));
                     latCount++;
                                          } else if(!nodeFeature.compare("longitude")) {
                                                 nodeLons.addEntry(lonCount, lexical_cast<double>(c.second.data()));
                     lonCount++;
                                          } else if(!nodeFeature.compare("elevation")) {
                                                 nodeEles.addEntry(eleCount, lexical_cast<int>(c.second.data()));
                     eleCount++;
                                          } else if(!nodeFeature.compare("nodeIDs")) {
                                                 nodeIDs.addEntry(idCount, lexical_cast<long int>(c.second.data()));
                     idCount++;
                                          }
                                   }
                            }
                     }
              }

GenericMap<long int, Node*>* nodes;

//                    // fit spline
```

```
//                        typedef Eigen::Spline<double, 2> spline2f;
//                        spline2f roadSpline;

                        if(this->jsonifyRoadNodes)
                        {
                            assert(nodeLats.getSize() == nodeLons.getSize());
                            assert(nodeLons.getSize() == nodeEles.getSize());
                            assert(nodeEles.getSize() == nodeIDs.getSize());

//                              // for splines
//                              Eigen::MatrixXd points(2, nodeLats.getSize());

                            nodes = new GenericMap<long int, Node*>();
                            for(int i = 0; i < nodeLats.getSize(); i++)
                            {
//                                  points(0, i) = nodeLats.getEntry(i);
//                                  points(1, i) = nodeLons.getEntry(i);

                                nodes->addEntry(i,     new     Node(nodeLats.getEntry(i),     nodeLons.getEntry(i),     nodeEles.getEntry(i),
nodeIDs.getEntry(i)));
                            }

//                              roadSpline = Eigen::SplineFitting<spline2f>::Interpolate(points, 1);
                        }
                        else
                        {
                            assert(roadsNodes.hasEntry(roadID));
                            nodes = roadsNodes.getEntry(roadID);

//                              // for splines
//                              Eigen::MatrixXd points(2, nodes->getSize());
//
//                              for(int i = 0; i < nodes->getSize(); i++)
//                              {
//                                  points(0, i) = nodes->getEntry(i)->getLat();
//                                  points(1, i) = nodes->getEntry(i)->getLon();
//                              }
//
//                              roadSpline = Eigen::SplineFitting<spline2f>::Interpolate(points, 1);
                        }

                                    roadIntersections->addEntry(roadID, new std::pair<int, int>(startNodeID, endNodeID));

                        Road* road = new Road(roadType, roadID, nodes);

                        road->assignSplineLength(splineLength);
//                        road->assignSpline(roadSpline);
                        road->setMinMaxLatLon();
                        road->setBoundsID(boundsID);
                                    roads->addEntry(roadID, road);
```

```
                        }
                }
        }
}

// get intersections and bounds next adding roads to the intersections
BOOST_FOREACH(ptree::value_type& u, cityLogs)
{
        int boundsID = lexical_cast<int>(u.first.data());
        BOOST_FOREACH(ptree::value_type& v, u.second)
        {
                std::string child = v.first.data();
                if(!child.compare("intersections"))
         {
             std::cout << "in get city, intersections" << std::endl;
                        BOOST_FOREACH(ptree::value_type& z, v.second)
                        {
                                int intID = lexical_cast<int>(z.first.data());
                                GenericMap<long int, Road*>* intRoads = new GenericMap<long int, Road*>();
                                int ele; double lat, lon;
                                BOOST_FOREACH(ptree::value_type& a, z.second)
                                {
                                        std::string intFeature = a.first.data();
                                        if(!intFeature.compare("elevation")) {
                                                ele = lexical_cast<int>(a.second.data());
                                        } else if(!intFeature.compare("latitude")) {
                                                lat = lexical_cast<double>(a.second.data());
                                        } else if(!intFeature.compare("longitude")) {
                                                lon = lexical_cast<double>(a.second.data());
                                        } else if(!intFeature.compare("roadIDs")) {

                                                BOOST_FOREACH(ptree::value_type& b, a.second)
                                                {
                                                        long int roadID = lexical_cast<long int>(b.second.data());
                                                        intRoads->addEntry(roadID, roads->getEntry(roadID));
                                                }
                                        }
                                }
                                Intersection* newInt = new Intersection(intRoads, lat, lon, ele, intID);
                                newInt->setBoundsID(boundsID);
                                intersections->addEntry(intID, newInt);
                        }
                } else if(!child.compare("bounds")) {
             std::cout << "in get city, bounds" << std::endl;
                        double maxLat, maxLon, minLat, minLon;
                        BOOST_FOREACH(ptree::value_type& z, v.second)
                        {
                                std::string boundsFeature = z.first.data();
                                if(!boundsFeature.compare("maxLat")) {
                                        maxLat = lexical_cast<double>(z.second.data());
```

```
                                        } else if(!boundsFeature.compare("maxLon")) {
                                                maxLon = lexical_cast<double>(z.second.data());
                                        } else if(!boundsFeature.compare("minLat")) {
                                                minLat = lexical_cast<double>(z.second.data());
                                        } else if(!boundsFeature.compare("minLon")) {
                                                minLon = lexical_cast<double>(z.second.data());
                                        }
                                }
                                Bounds* newBounds = new Bounds(maxLat, minLat, maxLon, minLon);
                                newBounds->assignID(boundsID);
                                bounds->addEntry(boundsID, newBounds);
                        }
                }
        }

        roadIntersections->initializeCounter();
        GenericEntry<long int, std::pair<int, int>*>* nextRoadInts = roadIntersections->nextEntry();
        while(nextRoadInts != NULL)
        {
                roads->getEntry(nextRoadInts->key)->setStartIntersection(intersections->getEntry(nextRoadInts->value->first));
                roads->getEntry(nextRoadInts->key)->setEndIntersection(intersections->getEntry(nextRoadInts->value->second));
                nextRoadInts = roadIntersections->nextEntry();
        }

        delete(roadIntersections);
        return new City(intersections, roads, bounds);
      } catch (std::exception& e) {
        std::cout << e.what() << std::endl;
        return NULL;
      }
}

GenericMap<long int, std::pair<double, double>*>* DataManagement::getMostRecentTripData() {

    std::cout << "getting most recent trip log data" << std::endl;

    if(this->countFileLine(this->tripData) == 0)
    {
        return NULL;
    }

      ptree tripLog;
      int dayID = 0;
      try {
        read_json(this->tripData, tripLog);
        BOOST_FOREACH(ptree::value_type& v, tripLog)
        {
                dayID = lexical_cast<int>(v.first.data());
        }
```

```
	GenericMap<GenericMap<int,  double>*,  GenericMap<int,  double>*>*  rawRecentTripData  =  new  GenericMap<GenericMap<int,  double>*,
GenericMap<int, double>*>();
	int latCount = 0, lonCount = 0;
	BOOST_FOREACH(ptree::value_type& v, tripLog)
	{
		if(lexical_cast<int>(v.first.data()) == dayID)
		{
			GenericMap<int, double>* lats = new GenericMap<int, double>();
			GenericMap<int, double>* lons = new GenericMap<int, double>();
			BOOST_FOREACH(ptree::value_type& z, v.second)
			{
				std::string subTree = z.first.data();

				BOOST_FOREACH(ptree::value_type& b, z.second)
				{
					if(!subTree.compare("latitude"))
					{
						double lat = lexical_cast<double>(b.second.data());
						lats->addEntry(latCount++, lat);
					} else if(!subTree.compare("longitude")) {
						double lon = lexical_cast<double>(b.second.data());
						lons->addEntry(lonCount++, lon);

					}
				}

			}
			assert(lats->getSize() == lons->getSize());
			rawRecentTripData->addEntry(lats, lons);
		}
	}

	GenericMap<long int, std::pair<double, double>*>* recentTripData = new GenericMap<long int, std::pair<double, double>*>();
	long int tripCount = 1;

	rawRecentTripData->initializeCounter();
	GenericEntry<GenericMap<int, double>*, GenericMap<int, double>*>* nextLatLonSet = rawRecentTripData->nextEntry();
	while(nextLatLonSet != NULL)
	{
		nextLatLonSet->key->initializeCounter();
		nextLatLonSet->value->initializeCounter();
		GenericEntry<int, double>* nextLat = nextLatLonSet->key->nextEntry();
		GenericEntry<int, double>* nextLon = nextLatLonSet->value->nextEntry();
		while(nextLat != NULL && nextLon != NULL)
		{
			recentTripData->addEntry(tripCount++, new std::pair<double, double>(nextLat->value, nextLon->value));
			nextLat = nextLatLonSet->key->nextEntry();
			nextLon = nextLatLonSet->value->nextEntry();
		}
		delete(nextLat);
```

```
                delete(nextLon);
                nextLatLonSet = rawRecentTripData->nextEntry();
        }
        delete(nextLatLonSet);
        delete(rawRecentTripData);

        return recentTripData;
      } catch(const std::exception& e) {
        std::cout << e.what() << std::endl;
      }
      return NULL;
}

template<class Matrix>
void DataManagement::writeBinaryNNMat(std::ofstream& out, const Matrix& matrix, int matrixTypeNum, int layerNum, long int linkHash)
{
    // matrix id
    out.write((char*) (&matrixTypeNum), sizeof(matrixTypeNum));
    out.write((char*) (&layerNum), sizeof(layerNum));
    out.write((char*) (&linkHash), sizeof(linkHash));

    // save matrix
    typename Matrix::Index rows=matrix.rows(), cols=matrix.cols();
    out.write((char*) (&rows), sizeof(typename Matrix::Index));
    out.write((char*) (&cols), sizeof(typename Matrix::Index));
    out.write((char*) matrix.data(), rows*cols*sizeof(typename Matrix::Scalar) );
}

template<class Matrix>
void DataManagement::readBinaryNNMat(std::ifstream& in, Matrix& matrix, int& matrixTypeNum, int& layerNum, long int& linkHash)
{
    // get matrix id
    in.read((char*) (&matrixTypeNum), sizeof(matrixTypeNum));
    in.read((char*) (&layerNum), sizeof(layerNum));
    in.read((char*) (&linkHash), sizeof(linkHash));

    // get matrix
    typename Matrix::Index rows=0, cols=0;
    in.read((char*) (&rows),sizeof(typename Matrix::Index));
    in.read((char*) (&cols),sizeof(typename Matrix::Index));
    matrix.resize(rows, cols);
    in.read( (char *) matrix.data() , rows*cols*sizeof(typename Matrix::Scalar) );
}

template<class Matrix>
void DataManagement::writeBinaryNodeMat(std::ofstream& out, const Matrix& matrix, long int roadID)
{
    // matrix id
    out.write((char*) (&roadID), sizeof(roadID));
```

```
    // save matrix
    typename Matrix::Index rows=matrix.rows(), cols=matrix.cols();
    out.write((char*) (&rows), sizeof(typename Matrix::Index));
    out.write((char*) (&cols), sizeof(typename Matrix::Index));
    out.write((char*) matrix.data(), rows*cols*sizeof(typename Matrix::Scalar) );
}

template<class Matrix>
void DataManagement::readBinaryNodeMat(std::ifstream& in, Matrix& matrix, long int& roadID)
{
    // get matrix id
    in.read((char*) (&roadID), sizeof(roadID));

    // get matrix
    typename Matrix::Index rows=0, cols=0;
    in.read((char*) (&rows),sizeof(typename Matrix::Index));
    in.read((char*) (&cols),sizeof(typename Matrix::Index));
    matrix.resize(rows, cols);
    in.read( (char *) matrix.data() , rows*cols*sizeof(typename Matrix::Scalar) );
}

}
```

12.31 DataManagement.h

```
/*
 * DataManagement.h
 *
 *  Created on: Apr 18, 2016
 *      Author: vagrant
 */

#ifndef DATA_MANAGEMENT_DATAMANAGEMENT_H_
#define DATA_MANAGEMENT_DATAMANAGEMENT_H_

#include "../route_prediction/RoutePrediction.h"
#include "../city/Intersection.h"
#include "../city/Road.h"
#include "../city/City.h"
#include "../map/GenericMap.h"

#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>
#include <boost/foreach.hpp>
#include "boost/lexical_cast.hpp"

#include <eigen3/Eigen/Dense>

#include <tuple>
#include <string>
#include <sys/stat.h>
```

```
#include <unistd.h>
#include <iostream>
#include <fstream>
#include <list>

namespace PredictivePowertrain {

class DataManagement {
private:
    std::string saveFolder = "/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/";
      std::string routePredictionData = saveFolder + "routePrediction.json";
      std::string cityData = saveFolder + "cities.json";
      std::string tripData = saveFolder + "tripLogs.json";
    std::string nnDataLoc = saveFolder + "nnData.bin";
    std::string nodeDataLoc = saveFolder + "nodeData.bin";

    bool jsonifyLinkNNData = false;
    bool jsonifyRoadNodes = false;

    int countFileLine(std::string fileLoc);

    // for serializing NN matrices
    template<class Matrix> void writeBinaryNNMat(std::ofstream& out, const Matrix& matrix, int matrixType, int layerNum, long int linkHash);
    template<class Matrix> void readBinaryNNMat(std::ifstream& in, Matrix& matrix, int& matrixTypeNum, int& layerNum, long int& linkHash);

    // for serializing node matrices
    template<class Matrix> void writeBinaryNodeMat(std::ofstream& out, const Matrix& matrix, long int roadID);
    template<class Matrix> void readBinaryNodeMat(std::ifstream& in, Matrix& matrix, long int& roadID);


public:
     DataManagement();
     void addRoutePredictionData(RoutePrediction* rp);
     void addCityData(City* city);
     void addTripData(GenericMap<long int, std::pair<double, double>*>* latLon);
     RoutePrediction* getRoutePredictionData();
     City* getCityData();
     GenericMap<long int, std::pair<double, double>*>* getMostRecentTripData();

};

} /* namespace PredictivePowertrain */


#endif /* DATA_MANAGEMENT_DATAMANAGEMENT_H_ */
```

12.32 DataManagementUnitTest.cpp

```
/*
 * DataManagement_ut.cpp
 *
```

```
 *  Created on: Apr 27, 2016
 *      Author: vagrant
 */


#include "UnitTests.h"
#include "../data_management/DataManagement.h"
#include "../driver_prediction/DriverPrediction.h"
#include "../route_prediction/RoutePrediction.h"

using namespace PredictivePowertrain;

void dataManagement_ut()
{
    float version = 2.0;

    DataManagement testDM;

    if(version == 2.0)
    {
        // print city
        City* cityStore = testDM.getCityData();
        cityStore->printIntersectionsAndRoads();
        testDM.addCityData(cityStore);

        // get route prediction
        RoutePrediction* rpStore = testDM.getRoutePredictionData();
        rpStore->addCity(cityStore);

        GenericMap<long int, std::pair<double, double>*>* trace = testDM.getMostRecentTripData();
        Route* route = cityStore->getRouteFromGPSTrace(trace);
        Route* route2 = route->copy();

        std::ifstream input("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/DP_ACTUAL_SPEED_FUEL_FLOW.csv");

        std::string num;
        std::vector<float> actualSpds;

        // read in speed from csv
        while(1)
        {
            std::getline(input, num, ',');
            std::stringstream fs(num);
            float f = 0.0;
            fs >> f;

            if(f == -1)
            {
                break;
            }
```

```
        actualSpds.push_back(f);
    }

    DriverPrediction dp(rpStore);
    dp.parseRoute(route, &actualSpds, trace);

    testDM.addRoutePredictionData(dp.getRP());
}

if(version == 1.0)
{
    // test trip log add and get
    GenericMap<long int, std::pair<double, double>*>* latLon = new GenericMap<long int, std::pair<double, double>*>();
    latLon->addEntry(1, new std::pair<double, double>(47.654, -122.345));
    latLon->addEntry(2, new std::pair<double, double>(47.654, -122.345));
    latLon->addEntry(3, new std::pair<double, double>(47.654, -122.345));
    testDM.addTripData(latLon);
    testDM.addTripData(latLon);
    GenericMap<long int, std::pair<double, double>*>* storedLatLon = testDM.getMostRecentTripData();
    assert(storedLatLon->getSize() == latLon->getSize());

    storedLatLon->initializeCounter();
    bool latLonsMatch = true;
    GenericEntry<long int, std::pair<double, double>*>* nextStoredLatLon = storedLatLon->nextEntry();
    while(nextStoredLatLon != NULL)
    {
        if(!latLon->hasEntry(nextStoredLatLon->key) || nextStoredLatLon->value->first != latLon->getEntry(nextStoredLatLon->key)->first
                || nextStoredLatLon->value->second != latLon->getEntry(nextStoredLatLon->key)->second)
        {
            latLonsMatch = false;
        }
        nextStoredLatLon = storedLatLon->nextEntry();
    }
    assert(latLonsMatch);

    // test city logging add old test
    GenericMap<long int, Node*>* nodes = new GenericMap<long int, Node*>();
    nodes->addEntry(1, new Node(42.3, -122.4, 44, 1234566));
    nodes->addEntry(2, new Node(42.3, -122.4, 44, 1234566));

    GenericMap<long int, Road*>* blankRoads;
    GenericMap<int, Bounds*>* boundsMap = new GenericMap<int, Bounds*>();
    boundsMap->addEntry(0, new Bounds(2, 3, 4, 5));
    boundsMap->addEntry(1, new Bounds(2, 3, 4, 5));
    boundsMap->addEntry(2, new Bounds(2, 3, 4, 5));

    Intersection* startInt = new Intersection(blankRoads, 42.3, -122.4, 44, 345);
    Intersection* endInt = new Intersection(blankRoads, 42.3, -122.4, 44, 456);
```

```
Road* road1 = new Road("hilly", 987654, nodes);
road1->setStartIntersection(startInt);
road1->setEndIntersection(endInt);
road1->setBoundsID(0);

Road* road2 = new Road("flat", 8765, nodes);
road2->setStartIntersection(startInt);
road2->setEndIntersection(endInt);
road2->setBoundsID(1);

Road* road3 = new Road("medium", 9854, nodes);
road3->setStartIntersection(startInt);
road3->setEndIntersection(endInt);
road3->setBoundsID(2);

GenericMap<long int, Road*>* roads = new GenericMap<long int, Road*>();
roads->addEntry(road1->getRoadID(), road1);
roads->addEntry(road2->getRoadID(), road2);
roads->addEntry(road3->getRoadID(), road3);

Intersection* intersection1 = new Intersection(roads, 42.3, -122.4, 44, 123);
intersection1->setBoundsID(0);
Intersection* intersection2 = new Intersection(roads, 42.3, -122.4, 44, 234);
intersection2->setBoundsID(0);
Intersection* intersection3 = new Intersection(roads, 42.3, -122.4, 44, 345);
intersection3->setBoundsID(1);
Intersection* intersection4 = new Intersection(roads, 42.3, -122.4, 44, 456);
intersection4->setBoundsID(2);

GenericMap<long int, Intersection*>* intersections = new GenericMap<long int, Intersection*>();
intersections->addEntry(intersection1->getIntersectionID(), intersection1);
intersections->addEntry(intersection2->getIntersectionID(), intersection2);
intersections->addEntry(intersection3->getIntersectionID(), intersection3);
intersections->addEntry(intersection4->getIntersectionID(), intersection4);

// store city
City* city = new City(intersections, roads, boundsMap);
testDM.addCityData(city);

// add prediction data
SpeedPrediction* sp = speedPrediction_ut();
RoutePrediction* rp = routePrediction_ut();

rp->getLinks()->initializeCounter();
GenericEntry<long int, Link*>* nextLink = rp->getLinks()->nextEntry();
while(nextLink != NULL)
{
    std::vector<std::vector<Eigen::MatrixXd*>*>* nnData = sp->getVals();
    std::vector<Eigen::MatrixXd*>* wts = nnData->at(0);
    std::vector<Eigen::MatrixXd*>* yHid = nnData->at(1);
```

```
    std::vector<Eigen::MatrixXd*>* yInHid = nnData->at(2);

    nextLink->value->setWeights(wts, yHid, yInHid, 1);
    nextLink->value->setWeights(wts, yHid, yInHid, 0);
    nextLink->value->setNumNNLayers(sp->getNumLayers());

    nextLink = rp->getLinks()->nextEntry();
}

testDM.addRoutePredictionData(rp);

// retrieve stored city data
City* storedCity = testDM.getCityData();

// check sizes
assert(city->getInstersectionMapSize() == storedCity->getInstersectionMapSize());
assert(city->getRoadMapSize() == storedCity->getRoadMapSize());
assert(city->getBoundsMapSize() == storedCity->getBoundsMapSize());

// check roads
GenericMap<long int, Road*>* storedRoads = storedCity->getRoads();
bool hasAllRoads = true;
storedRoads->initializeCounter();
GenericEntry<long int, Road*>* nextStoredRoad = storedRoads->nextEntry();
while(nextStoredRoad != NULL)
{
    if(!roads->hasEntry(nextStoredRoad->key))
    {
        hasAllRoads = false;
    }
    assert(intersections->hasEntry(nextStoredRoad->value->getStartIntersection()->getIntersectionID()));
    assert(intersections->hasEntry(nextStoredRoad->value->getEndIntersection()->getIntersectionID()));
    nextStoredRoad = storedRoads->nextEntry();
}
assert(hasAllRoads);

// check ints
GenericMap<long int, Intersection*>* storedInts = storedCity->getIntersections();
bool hasAllInts = true;
storedInts->initializeCounter();
GenericEntry<long int, Intersection*>* nextStoredInt = storedInts->nextEntry();
while(nextStoredInt != NULL)
{
    if(!intersections->hasEntry(nextStoredInt->key))
    {
        hasAllInts = false;
    }
    GenericMap<long int, Road*>* connectingRoads = nextStoredInt->value->getRoads();
    bool hasAllConnectingRoads = true;
    connectingRoads->initializeCounter();
```

```
            GenericEntry<long int, Road*>* nextStoredConnectingRoad = connectingRoads->nextEntry();
            while(nextStoredConnectingRoad != NULL)
            {
                if(!roads->hasEntry(nextStoredConnectingRoad->key))
                {
                    hasAllConnectingRoads = false;
                }
                nextStoredConnectingRoad = connectingRoads->nextEntry();
            }
            nextStoredInt = storedInts->nextEntry();
            assert(hasAllConnectingRoads);
        }
        assert(hasAllInts);

        // check bounds
        GenericMap<int, Bounds*>* storedBounds = storedCity->getBoundsMap();
        bool hasAllBounds = true;
        storedBounds->initializeCounter();
        GenericEntry<int, Bounds*>* nextStoredBounds = storedBounds->nextEntry();
        while(nextStoredBounds != NULL)
        {
            if(!boundsMap->hasEntry(nextStoredBounds->key))
            {
                hasAllBounds = false;
            }
            nextStoredBounds = storedBounds->nextEntry();
        }
        assert(hasAllBounds);

        // retrieve stored route prediction data
        RoutePrediction* storedrp = testDM.getRoutePredictionData();
    }

}
```

12.33 DriverPrediction.cpp

```
/*
 * DriverPrediction.cpp
 *
 *  Created on: Jan 5, 2016
 *      Author: Brian
 */

#include "DriverPrediction.h"

namespace PredictivePowertrain {


DriverPrediction::DriverPrediction(RoutePrediction* newRP)
{
```

```
    this->city = newRP->getCity();
    this->rp = newRP;

    this->intitalize();
}


DriverPrediction::DriverPrediction()
{
    this->intitalize();
}

void DriverPrediction::intitalize()
{
    this->sp = new SpeedPrediction();
}

DriverPrediction::~DriverPrediction()
{
    delete(this->sp);
}

void DriverPrediction::setCurrentLink(Link* currentLink)
{
this->currLink = currentLink;
}

// assumes vehicle speed is zero
DriverPrediction::PredData DriverPrediction::startPrediction(Link* currentLink,
                                                             float spd,
                                                             std::vector<float>* currentConditions,
                                                             float distAlongLink)
{
    // update current link
    this->currLink = currentLink;

    // get start intersection
    Intersection* startIntersection = this->city->getIntersectionFromLink(currentLink, true);
    std::cout << "start intersection: " << startIntersection->getIntersectionID() << std::endl;

    // get pred route
    Route* predRoute = this->copyRoute(this->rp->startPrediction(currentLink, startIntersection, currentConditions));

    // update current route
    this->predRoute = predRoute;

    PredData predData;
    if(!predRoute->isEqual(this->rp->getUnknownRoute())
       && !predRoute->isEqual(this->rp->getOverRoute())
       /*&& this->allLinksHaveWeigths(predRoute)*/)
```

```
    {
        predRoute->addLinkToFront(currentLink);

        // add NN values to route since route prediction always returns old links
        this->addWeightedLinksToRoute(predRoute);

        // get route speed and elevation data
        Eigen::MatrixXd spdIn = this->getSpeedPredInpunt(spd);
        predData = this->city->routeToData(this->predRoute, distAlongLink, this->sp, &spdIn);
    }

    return predData;
}

DriverPrediction::PredData DriverPrediction::nextPrediction(Link* currentLink,
                                                             float spd,
                                                             float distAlongLink)
{
    // current link
    std::cout << "current link: " << this->currLink->getNumber() << std::endl;

    // print current predicted route
    std::cout << "predicted route: " << std::endl;
    this->predRoute->printLinks();

    // --- AT END OF PREDICTION ---
    if(currentLink->isFinalLink())
    {
        std::cout << "current link is final link" << std::endl;
    }

    // --- REPREDICT ROUTE AND TRAIN SPEED PREDICTION ---
    else if(!this->currLink->isEqual(currentLink))
    {
        if(!this->predRoute->isEqual(this->rp->getUnknownRoute())
           && !this->predRoute->isEqual(this->rp->getOverRoute()))
        {
            // quick train of speed prediction over last link
            this->trainSpeedPredictionOverLastLink();
        }
        // perform route prediction
        Route* newPredRoute = this->copyRoute(this->rp->predict(currentLink));

        // add speed prediction vals to route and attached current link to front
        if(!newPredRoute->isEqual(this->rp->getUnknownRoute())
           && !newPredRoute->isEqual(this->rp->getOverRoute()))
        {
            newPredRoute->addLinkToFront(currentLink);
            this->addWeightedLinksToRoute(newPredRoute);
        }
```

```
        // update current link and predicted route
        this->currLink = currentLink;
        this->predRoute = newPredRoute;
    }
    else if(this->predRoute->isEqual(this->rp->getUnknownRoute()))
    {
        // hack
        std::vector<float> currConditions(1);
        currConditions.at(0) = -1;

        return this->startPrediction(currentLink, spd, &currConditions, distAlongLink);
    }

    PredData predData;
    if(!this->predRoute->isEqual(this->rp->getUnknownRoute())
       && !this->predRoute->isEqual(this->rp->getOverRoute())
       /*&& this->allLinksHaveWeigths(this->predRoute)*/)
    {
        Eigen::MatrixXd spdIn = this->getSpeedPredInpunt(spd);
        predData = this->city->routeToData(this->predRoute, distAlongLink, this->sp, &spdIn);
    }

    return predData;
}

void DriverPrediction::parseRoute(Route* currRoute, std::vector<float>* spds, GenericMap<long int, std::pair<double, double>*>* trace)
{
    std::cout << "in Driver Prediction: parsing route" << std::endl;

    if(spds->size() > 0)
    {
        // make sure all links have their weights
        this->addWeightedLinksToRoute(currRoute);

        // train speed prediction
        GPS gps;

        // clear out dp buffers since no longer training NN on the fly
        this->beforeLinkSpds.clear();
        this->linkSpds.clear();
        std::queue<float> empty;
        std::swap(this->lastSpds, empty);

        // make copy of actual speed values since this will be mutilated
        std::vector<float> spdsCopy(*spds);

        // trim off actual speed measurements taken over first link
        Link* firstRouteLink = currRoute->getLinks()->getEntry(0);
```

```
// erase from front of speed trace
Road* road_i = this->city->getRoads()->getEntry(firstRouteLink->getNumber());

road_i->getNodes()->initializeCounter();
GenericEntry<long int, Node*>* nextNode = road_i->getNodes()->nextEntry();
while(nextNode != NULL)
{
    trace->initializeCounter();
    GenericEntry<long int, std::pair<double, double>*>* nextMeas = trace->nextEntry();
    while(nextMeas != NULL)
    {
        double nodeLat = nextNode->value->getLat();
        double nodeLon = nextNode->value->getLon();

        double traceLat = nextMeas->value->first;
        double traceLon = nextMeas->value->second;

        float dist = gps.deltaLatLonToXY(nodeLat, nodeLon, traceLat, traceLon);

        if(dist < gps.getDeltaXYTolerance())
        {
            spdsCopy.erase(spdsCopy.begin());
            break;
        }

        nextMeas = trace->nextEntry();
    }
    delete(nextMeas);

    nextNode = road_i->getNodes()->nextEntry();
}
delete(nextNode);

// get distance covered from trimmed actual speed trace
float spdDist = spdsCopy.size() * this->sp->getDS();

// begin training loop over link that have actual speed
// measurements from start to end
currRoute->getLinks()->initializeCounter();
GenericEntry<long int, Link*>* nextLink = currRoute->getLinks()->nextEntry();
while(nextLink != NULL && nextLink->value != NULL && !nextLink->value->isFinalLink())
{
    // exclude first and last link since we only want links with
    // actual speed measurements over they're entirety
    if(nextLink->key == 0 || nextLink->key == currRoute->getLinks()->getSize() - 2)
    {
        nextLink = currRoute->getLinks()->nextEntry();
        continue;
    }
```

```
            std::cout << nextLink->value->getNumber() << std::endl;

            // get necessary road data
            Road* road_i = this->city->getRoads()->getEntry(nextLink->value->getNumber());
            float roadDist_i = road_i->getSplineLength();
            float roadSpdIndices_i = (float) roadDist_i / spdDist * spds->size();

            // vector containing actual speed values for given link
            std::vector<float> roadSpds_i;

            // get road speeds from actual trace of speed values.
            for(int i = 0; i < roadSpdIndices_i; i++)
            {
                roadSpds_i.push_back(spdsCopy.front());
                spdsCopy.erase(spdsCopy.begin());
            }

            // train for speed at beginnging of next link
            roadSpds_i.push_back(spdsCopy.at(0));

            // set current link
            this->setCurrentLink(nextLink->value);

            // populate link speed buffers
            while(roadSpds_i.size() > 0)
            {
                this->updateSpeedsbyVal(roadSpds_i.front());
                roadSpds_i.erase(roadSpds_i.begin());
            }

            // train
            this->trainSpeedPredictionOverLastLink();

            nextLink = currRoute->getLinks()->nextEntry();
        }
        delete(nextLink);
    }

    // train routpe prediction
    this->rp->parseRoute(currRoute);
}

void DriverPrediction::trainSpeedPredictionOverLastLink()
{
    int trainIters = 1;

    // matrices for speed prediction
    Eigen::MatrixXd spdIn(1,this->sp->getI()+1);
    Eigen::MatrixXd spdOut(1,this->sp->getO());
    Eigen::MatrixXd spdAct(1,this->sp->getO());
```

```
// get spd input from before line and reset before speed input
for(int i = 0; i < spdIn.cols(); i++)
{
    spdIn.coeffRef(0,i) = this->beforeLinkSpds.at(i);

    float spd_i = this->lastSpds.front();
    this->beforeLinkSpds.at(i) = spd_i;

    this->lastSpds.pop();
    this->lastSpds.push(spd_i);
}

// prep speed prediction for training
if(this->currLink->linkHasWeights())
{
    std::vector<std::vector<Eigen::MatrixXd*>*>* nnVals = this->currLink->getWeights(this->currLink->getDirection());
    this->sp->setVals(nnVals);
}

// consume link speed values to train NN
for(int i = 0; i < spdAct.cols(); i++)
{
    // use link speed logs to set actual speed values while log is large enough
    if(this->linkSpds.size() > 0)
    {
        spdAct.coeffRef(0,i) = this->linkSpds.at(0);
        this->linkSpds.erase(this->linkSpds.begin());
    }

    // otherwise just use the NN output as act and do not penalize NN for wrong predictions
    else
    {
        spdAct.coeffRef(0,i) = 0.0;
    }
}

// scale training data
this->sp->formatInData(&spdIn);
this->sp->scaleTrainingSpeed(&spdAct);

// train
for(int i = 0; i < trainIters; i++)
{
    this->sp->predict(&spdIn, &spdOut);
    this->sp->train(&spdOut, &spdAct, &spdIn);
}

// perform rolling window trainging
while(this->linkSpds.size() > 0)
```

```
    {
        // left shift speed input
        for(int i = 0; i < spdIn.cols(); i++)
        {
            spdIn.coeffRef(0,i) = spdIn.coeffRef(0,i+1);
        }
        spdIn.coeffRef(0, this->sp->getI()) = spdAct.coeffRef(0, 0);

        // left shift speed act values
        for(int i = 0; i < spdAct.cols() - 1; i++)
        {
            spdAct.coeffRef(0, i) = spdAct.coeffRef(0,i+1);
        }
        spdAct.coeffRef(0, spdAct.cols() - 1) = this->linkSpds.at(0) / sp->getMaxSpeed() + sp->getSpeedOffset();

        // remove first value from link speed
        this->linkSpds.erase(this->linkSpds.begin());

        // train
        for(int i = 0; i < trainIters; i++)
        {
            this->sp->predict(&spdIn, &spdOut);
            this->sp->train(&spdOut, &spdAct, &spdIn);
        }
    }

    // update link weights
    std::vector<std::vector<Eigen::MatrixXd*>*>* spVals = this->sp->getVals();
    this->currLink->setWeights(spVals->at(0), spVals->at(1), spVals->at(2), this->currLink->getDirection());
}

void DriverPrediction::updateSpeedsByVec(std::vector<float>* spds)
{
    while(spds->size() > 0)
    {
        this->updateSpeedsbyVal(spds->front());
        spds->erase(spds->begin());
    }
}

void DriverPrediction::updateSpeedsbyVal(float spd)
{
    // because before links speeds doesn't exist on startup, use repeated first measurement.
    if(this->beforeLinkSpds.size() == 0)
    {
        for(int i = 0; i < this->sp->getI() + 1; i++)
        {
            this->beforeLinkSpds.push_back(spd);
        }
    }
```

```
    // update link spd trace for training later
    this->linkSpds.push_back(spd);

    // update speed FIFO queue
    this->lastSpds.push(spd);
    if(this->lastSpds.size() > this->sp->getI() + 1)
    {
        this->lastSpds.pop();
    }
}

Eigen::MatrixXd DriverPrediction::getSpeedPredInpunt(float spd)
{
    // update speeds
    this->updateSpeedsbyVal(spd);

    // create a matrix of zero input
    Eigen::MatrixXd spdIn = Eigen::MatrixXd::Zero(1, this->sp->getI() + 1);

    // take most recent speed values from link speed and populate end of speed pred input
    for(size_t i = 0; i < spdIn.cols(); i++)
    {
        float spd_i;

        if(i < this->lastSpds.size())
        {
            spd_i = this->lastSpds.front();
            this->lastSpds.pop();
            this->lastSpds.push(spd_i);
        }
        else
        {
            spd_i = this->lastSpds.back();
        }

        spdIn.coeffRef(0, i) = spd_i;
    }

    return spdIn;
}

void DriverPrediction::addWeightedLinksToRoute(Route* unweightedRoute)
{
    GenericMap<long int, Link*>* weightedRouteLinks = new GenericMap<long int, Link*>();
    GenericMap<long int, Link*>* unweightedLinks = unweightedRoute->getLinks();

    unweightedLinks->initializeCounter();
    GenericEntry<long int, Link*>* nextUnweightedLink = unweightedLinks->nextEntry();
    while(nextUnweightedLink != NULL)
```

```
    {
        long int key = nextUnweightedLink->key;
        Link* nextUnweightedLink_i = nextUnweightedLink->value;

        long int linkHash = nextUnweightedLink_i->getHash();

        if(this->rp->getLinks()->hasEntry(linkHash))
        {
            weightedRouteLinks->addEntry(key, this->rp->getLinks()->getEntry(linkHash));
        }
        else
        {
            weightedRouteLinks->addEntry(key, nextUnweightedLink_i);
        }

        nextUnweightedLink = unweightedLinks->nextEntry();
    }
    delete(nextUnweightedLink);

    unweightedRoute->replaceLinks(weightedRouteLinks);
}

RoutePrediction* DriverPrediction::getRP()
{
    return this->rp;
}

SpeedPrediction* DriverPrediction::getSP()
{
    return this->sp;
}

bool DriverPrediction::allLinksHaveWeigths(Route* route)
{
    route->getLinks()->initializeCounter();
    GenericEntry<long int, Link*>* nextLink = route->getLinks()->nextEntry();
    while(nextLink != NULL)
    {
        if(nextLink->key == route->getLinks()->getSize() - 2)
        {
            break;
        }

        if(!nextLink->value->linkHasWeights() &&
           !nextLink->value->isFinalLink())
        {
            return false;
        }
        nextLink = route->getLinks()->nextEntry();
    }
```

```
    delete(nextLink);

    return true;
}

std::vector<long int> DriverPrediction::getRouteDataLabels()
{
    return this->city->getRouteDataLabels();
}

Route* DriverPrediction::getPredRoute()
{
    return this->predRoute;
}

Route* DriverPrediction::copyRoute(Route* route)
{
    GenericMap<long int, Link*>* copyLinks = new GenericMap<long int, Link*>();
    GenericMap<long int, Link*>* links = route->getLinks();

    links->initializeCounter();
    GenericEntry<long int, Link*>* nextLink = links->nextEntry();
    while(nextLink != NULL)
    {
        copyLinks->addEntry(nextLink->key, nextLink->value);
        nextLink = links->nextEntry();
    }

    return new Route(copyLinks, route->getGoal());
}

} /* namespace PredictivePowertrain */
```

12.34 DriverPrediction.h

```
/*
 * DriverPrediction.h
 *
 *  Created on: Jan 5, 2016
 *      Author: Brian
 */

#ifndef DRIVER_PREDICTION_DRIVERPREDICTION_H_
#define DRIVER_PREDICTION_DRIVERPREDICTION_H_

#include "../route_prediction/RoutePrediction.h"
#include "../speed_prediction/SpeedPrediction.h"
#include "../city/City.h"
#include "Link.h"
#include "../gps/GPS.h"
```

```
#include <queue>
#include <vector>

namespace PredictivePowertrain {

    class DriverPrediction {
    private:
        std::vector<float> beforeLinkSpds;
        std::vector<float> linkSpds;
        std::queue<float> lastSpds;
        RoutePrediction* rp;
        SpeedPrediction* sp;
        Route* predRoute;
        Link* currLink;
        City* city;

        void addWeightedLinksToRoute(Route* unweightedRoute);
        bool allLinksHaveWeigths(Route* route);
        Route* copyRoute(Route* route);
        void intitalize();

    public:
        typedef std::pair<std::vector<float>, std::vector<float>> PredData;

        virtual ~DriverPrediction();
        DriverPrediction(RoutePrediction* newRP);
        DriverPrediction();
        void setCurrentLink(Link* currentLink);
        PredData startPrediction(Link* currentLink, float spd, std::vector<float>* currentConditions, float distAlongLink);
        PredData nextPrediction(Link* currentLink, float spd, float distAlongLink);
        void parseRoute(Route* currRoute, std::vector<float>* speeds, GenericMap<long int, std::pair<double, double>*>* trace);
        RoutePrediction* getRP();
        SpeedPrediction* getSP();
        Eigen::MatrixXd getSpeedPredInpunt(float spd);
        void trainSpeedPredictionOverLastLink();
        void updateSpeedsByVec(std::vector<float>* spds);
        void updateSpeedsbyVal(float spd);
        std::vector<long int> getRouteDataLabels();
        Route* getPredRoute();

    };

} /* namespace PredictivePowertrain */

#endif /* DRIVER_PREDICTION_DRIVERPREDICTION_H_ */
```

12.35 DriverPredictionUnitTest.cpp

```
/*
 * DriverPrediction.cpp
 *
```

```
 *  Created on: Apr 14, 2016
 *      Author: vagrant
 */

#include "../route_prediction/Goal.h"
#include "../route_prediction/Route.h"
#include "../driver_prediction/DriverPrediction.h"
#include "../route_prediction/RoutePrediction.h"
#include "../speed_prediction/SpeedPrediction.h"
#include "../data_management/DataManagement.h"
#include "../city/BuildCity.h"
#include "../city/City.h"
#include "UnitTests.h"

#include <assert.h>
#include <fstream>
#include <iostream>
#include <vector>
#include <ctime>

using namespace PredictivePowertrain;

void setRouteNeuralNetworkVals(Route* route, std::vector<float> spds, float spdDist, City* city)
{
    std::cout << "Driver Prediction Speed Training" << std::endl;
    DriverPrediction dp;

    std::vector<float> spdsTemp(spds);

    // add neural net values to route prediction links
    GenericMap<long int, Link*>* links = route->getLinks();

    links->initializeCounter();
    GenericEntry<long int, Link*>* nextLink = links->nextEntry();
    while(nextLink != NULL && !nextLink->value->isFinalLink())
    {
        // get necessary road data
        Road* road_i = city->getRoads()->getEntry(nextLink->value->getNumber());
        float roadDist_i = road_i->getSplineLength();
        float roadSpdIndices_i = (float) roadDist_i / spdDist * spds.size();

        std::vector<float> roadSpds_i;

        // get road speeds from arbitrary trace of continuous speed values.
        for(int i = 0; i < roadSpdIndices_i; i++)
        {
            if(spdsTemp.size() == 0)
            {
                for(int j = 0; j < spds.size(); j++)
                {
```

```
                    spdsTemp.push_back(spds.at(j));
                }
            }

            roadSpds_i.push_back(spdsTemp.at(0));
            spdsTemp.erase(spdsTemp.begin());
        }

        // train for speed at beginnging of next link
        roadSpds_i.push_back(spdsTemp.at(0));

        std::cout << roadSpds_i.size() << std::endl;

        dp.setCurrentLink(nextLink->value);

        while(roadSpds_i.size() > 0)
        {
            dp.getSpeedPredInpunt(roadSpds_i.front());
            roadSpds_i.erase(roadSpds_i.begin());
        }

        dp.trainSpeedPredictionOverLastLink();

        nextLink = links->nextEntry();
    }
    delete(nextLink);
}

void driverPrediction_ut()
{
    // ------------------------------------
    // BUILD SMALL CITY SECTION IN SPOKANE
    // ------------------------------------

    DataManagement dm;
    bool getCityFromDM = true;

    City* rpCity;

    if(!getCityFromDM)
    {
        // make trip log
        GenericMap<long int, std::pair<double, double>*>* latLon = new GenericMap<long int, std::pair<double, double>*>();

        // spokane trip log small
        latLon->addEntry(1, new std::pair<double, double>(47.634, -117.396));
        latLon->addEntry(2, new std::pair<double, double>(47.635, -117.397));
        latLon->addEntry(3, new std::pair<double, double>(47.636, -117.398));
        latLon->addEntry(4, new std::pair<double, double>(47.637, -117.399));
        latLon->addEntry(5, new std::pair<double, double>(47.638, -117.400));
```

```
    latLon->addEntry(6, new std::pair<double, double>(47.639, -117.401));
    latLon->addEntry(7, new std::pair<double, double>(47.640, -117.402));
    latLon->addEntry(8, new std::pair<double, double>(47.645, -117.406));

    // jsonify trip log -> delete existing jsons
    dm.addTripData(latLon);

    BuildCity bc;
    bc.updateGridDataXMLSpline();
    bc.printNewIntersectionsAndRoads();

    GenericMap<long int, Intersection*>* intersections = bc.getNewIntersections();
    GenericMap<long int, Road*>* roads = bc.getNewRoads();
    GenericMap<int, Bounds*> bounds;
    bounds.addEntry(0, bc.getNewBounds());

    rpCity = new City(intersections, roads, &bounds);
    dm.addCityData(rpCity);
}
else
{
    rpCity = dm.getCityData();
}

// ----------------------
// SETUP SPEED PREDICTION
// ----------------------

SpeedPrediction sp;

std::ifstream input("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/spd.csv");
std::string num;

// speeds for routes, both actual and random
std::vector<float> routeSpds;

// distance covered by speed predictions
float routeSpdsDist = 0.0;

// fill actual speed with EPA drive cycle speed traces
for (int i = 0; i<17000; i++)
{
    std::getline(input, num, ',');
    std::stringstream fs(num);
    float f = 0.0;
    fs >> f;
    routeSpds.push_back(f);
    routeSpdsDist += sp.getDS();
}
```

```
    // ----------------------
    // SETUP ROUTE PREDICTION
    // ----------------------

    // set route length
    int routeLength = 10;

    // set conditions
    std::vector<float>* conditions = new std::vector<float>(2);
    conditions->at(0) = 1;
    conditions->at(1) = 2;

    Intersection* startIntersection = rpCity->getIntersections()->getFirstEntry();
    Intersection* endIntersection = rpCity->getIntersections()->getLastEntry();
    Link* firstLink = new Link(1, startIntersection->getRoads()->getFirstEntry()->getRoadID());

    RoutePrediction rp(rpCity);

    // generate random actual route
    Route startRoute;
    startRoute.addLink(firstLink);
    startRoute.assignGoal(new Goal(endIntersection->getIntersectionID()));
    Route* actualRoute = rpCity->randomPath(startIntersection, &startRoute, routeLength, conditions);

    // add NN vals
    setRouteNeuralNetworkVals(actualRoute, routeSpds, routeSpdsDist, rpCity);

    // add trainging iterations here (simulates driving over the route multiple times)
    rp.parseRoute(actualRoute);
    rp.parseRoute(actualRoute);
    rp.parseRoute(actualRoute);
    rp.parseRoute(actualRoute);
    rp.parseRoute(actualRoute);

    // create number of random routes to include in test set
    int num_rand_routes = 4;

    for(int i = 1; i <= num_rand_routes; i++)
    {
        // create random route
        Route* randomRoute = rpCity->randomPath(startIntersection, &startRoute, std::ceil((float)std::rand() / RAND_MAX * routeLength),
conditions);
        while(randomRoute->isEqual(actualRoute))
        {
            randomRoute = rpCity->randomPath(startIntersection, &startRoute, std::ceil((float)std::rand() / RAND_MAX * routeLength),
conditions);
        }

        // add NN vals
        setRouteNeuralNetworkVals(randomRoute, routeSpds, routeSpdsDist, rpCity);
```

```
    // add random route to test set
    rp.parseRoute(randomRoute);
}

// ------------------------
// BEGIN DRIVER PREDICTION
// ------------------------

DriverPrediction dp(&rp);
DriverPrediction::PredData predData;

std::vector<float> routeSpdsTmp(routeSpds);
std::ofstream predFile("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/driverPredictionResults.csv");

bool isFirstLink = true;
int predictionCount = 0;
float distanceAlongRoute = 0.0;

// begin driving actual route
while(actualRoute->getLinks()->getSize() > 0)
{
    Link* nextLink = actualRoute->getLinks()->getEntry(0);
    actualRoute->removeFirstLink();

    // break if final link is found
    if(nextLink->isFinalLink())
    {
        std::cout << "Driver Prediction Unit Test Complete" << std::endl;
        break;
    }

    // get road data from actual route
    Road* road_i = rpCity->getRoads()->getEntry(nextLink->getNumber());
    float roadDist_i = road_i->getSplineLength();
    float distAlongLink = 0.0;

    // test handling of starting route partially up first link
    if(isFirstLink)
    {
        distAlongLink = roadDist_i / 2;
        distanceAlongRoute += distAlongLink;

        float removeBeforeIndex = (float) distAlongLink / routeSpdsDist * routeSpds.size();
        for(int i = 0; i < removeBeforeIndex; i++)
        {
            routeSpdsTmp.erase(routeSpdsTmp.begin());
        }
        std::cout << "Starting Driver Prediciton" << std::endl;
        predData = dp.startPrediction(nextLink, routeSpdsTmp.front(), conditions, distAlongLink);
```

```
    // udate distance along link
    distAlongLink += sp.getDS();
    distanceAlongRoute += sp.getDS();

    // actual route speeds
    predFile << "Actual Route Speeds\n";
    for(int i = 0; i < routeSpdsTmp.size(); i++)
    {
        predFile << routeSpdsTmp.at(i);
        predFile << ",";
    }
    predFile << "\n";

    routeSpdsTmp.erase(routeSpdsTmp.begin());
    isFirstLink = false;
}

while(distAlongLink < roadDist_i)
{
    predFile << "Prediction: " << predictionCount << "\n";
    predictionCount++;

    // predicted speed
    for(int i = 0; i < predData.first.size(); i++)
    {
        predFile << predData.first.at(i);
        predFile << ",";
    }
    predFile << "\n";

    // predicted elevation
    for(int i = 0; i < predData.second.size(); i++)
    {
        predFile << predData.second.at(i);
        predFile << ",";
    }
    predFile << "\n";

    if(routeSpdsTmp.size() == 0)
    {
        for(int j = 0; j < routeSpds.size(); j++)
        {
            routeSpdsTmp.push_back(routeSpds.at(j));
        }
    }

    // make next prediction
    predData = dp.nextPrediction(nextLink, routeSpdsTmp.front(), distAlongLink);
```

```
            // udate distance along link
            distAlongLink += sp.getDS();
            distanceAlongRoute += sp.getDS();

            std::cout << distanceAlongRoute << std::endl;

            // update actual speed values along actual route
            routeSpdsTmp.erase(routeSpdsTmp.begin());
        }

        // print routes
        std::cout << "---- route prediciton ----" << std::endl;
        actualRoute->printLinks();
        dp.getRP()->getPredictedRoute()->printLinks();


    }
}
```

12.36 Link.cpp

```
/*
 * Link.cpp
 *
 *  Created on: Jan 5, 2016
 *      Author: Amanda
 */

#include "Link.h"

namespace PredictivePowertrain {

Link::Link() {
      this->link_number = 0;
      this->link_direction = 0;
    this->initialize();
}

Link::Link(int direction, long int linkNumber) {
    this->link_number = linkNumber;
      this->link_direction = direction;
    this->initialize();
}

void Link::initialize()
{
    this->WtsA = NULL;
    this->yHidA = NULL;
    this->yInHidA = NULL;
    this->hasAWeights = false;
```

```
    this->WtsB = NULL;
    this->yHidB = NULL;
    this->yInHidB = NULL;
    this->hasBWeights = false;

    this->numNNLayers = 0;
    this->hasWeights = false;
}

Link::~Link() {
      // TODO Auto-generated destructor stub
}

Link* Link::copy(int direction, long int linkNumber) {
      Link* link = new Link(direction, linkNumber);
      return link;
}

bool Link::isEqual(Link* other) {
    return other->link_direction == this->link_direction && other->link_number == this->link_number;
}

long int Link::getNumber() {
      return this->link_number;
}

int Link::getDirection() {
      return this->link_direction;
}

Link* Link::newLinkFromHash(long int hash) {
      Link* link = new Link(hash % 2, hash / 2);
      return link;
}

long int Link::getHash() {
      return 2 * this->link_direction + this->link_number;
}

Link* Link::finalLink() {
      Link* link = new Link(-1, -1);
      return link;
}

bool Link::isFinalLink()
{
    return this->link_direction == -1 && this->link_number == -1;
}

Link* Link::linkFromRoad(Road* road, Intersection* intersection) {
```

```
      long int linkNum = road->getRoadID();
      int linkDir = road->getEndIntersection()->getIntersectionID() == intersection->getIntersectionID();
      Link* link = new Link(linkDir, linkNum);
      return link;
}

void  Link::setWeights(std::vector<Eigen::MatrixXd*>*  wts,  std::vector<Eigen::MatrixXd*>*  yHid,  std::vector<Eigen::MatrixXd*>*  yInHid,  int
direction)
{
    if(direction == 1)
    {
        this->WtsA = wts;
        this->yHidA = yHid;
        this->yInHidA = yInHid;
        this->hasAWeights = true;
    }
    else if(direction == 0)
    {
        this->WtsB = wts;
        this->yHidB = yHid;
        this->yInHidB = yInHid;
        this->hasBWeights = true;
    }
    this->hasWeights = true;
}

bool Link::linkHasWeights()
{
    return this->hasWeights;
}

bool Link::linkHasAWeights()
{
    return this->hasAWeights;
}

bool Link::linkHasBWeights()
{
    return this->hasBWeights;
}

std::vector<std::vector<Eigen::MatrixXd*>*>* Link::getWeights(int direction)
{
    std::vector<std::vector<Eigen::MatrixXd*>*>* returnList = new std::vector<std::vector<Eigen::MatrixXd*>*>(3);
    if(direction == 1 && this->hasAWeights)
    {
        returnList->at(0) = this->WtsA;
        returnList->at(1) = this->yHidA;
        returnList->at(2) = this->yInHidA;
    }
```

```
    else if(direction == 0 && this->hasBWeights)
    {
        returnList->at(0) = this->WtsB;
        returnList->at(1) = this->yHidB;
        returnList->at(2) = this->yInHidB;
    }
    return returnList;
}

void Link::setNumNNLayers(int num)
{
    this->numNNLayers = num;
}

int Link::getNumNNLayers()
{
    return this->numNNLayers;
}

} /* namespace PredictivePowertrain */
```

12.37 Link.h

```
/*
 * Link.h
 *
 *  Created on: Jan 5, 2016
 *      Author: Amanda
 */

#ifndef LINK_H_
#define LINK_H_

#include "../city/Road.h"
#include "../city/Intersection.h"

#include <eigen3/Eigen/Dense>
#include <list>
#include <vector>

namespace PredictivePowertrain {

class Intersection; // forward declaration
class Road;         // forward declaration

class Link {
private:
    long int link_number;
    int link_direction;

    // input weights on A-end of link
```

```
    std::vector<Eigen::MatrixXd*>* WtsA;            // matrix of weights
    std::vector<Eigen::MatrixXd*>* yHidA;    // hidden layer outputs
    std::vector<Eigen::MatrixXd*>* yInHidA;  // hidden layer inputs
    bool hasAWeights;

    // input weights on B-end of link
    std::vector<Eigen::MatrixXd*>* WtsB;            // matrix of weights
    std::vector<Eigen::MatrixXd*>* yHidB;    // hidden layer outputs
    std::vector<Eigen::MatrixXd*>* yInHidB;  // hidden layer inputs
    bool hasBWeights;

    int numNNLayers;
    bool hasWeights;
    void initialize();

public:
     Link();
     Link(int, long int);
     virtual ~Link();
     Link* copy(int, long int);
     long int getHash();
     bool isEqual(Link* other);
    bool isFinalLink();
     long int getNumber();
     int getDirection();
     static Link* newLinkFromHash(long int);
     static Link* finalLink();
     Link* linkFromRoad(Road* road, Intersection* intersection);
    void  setWeights(std::vector<Eigen::MatrixXd*>*  wts,  std::vector<Eigen::MatrixXd*>*  yHid,  std::vector<Eigen::MatrixXd*>*  yInHid,  int
direction);
    std::vector<std::vector<Eigen::MatrixXd*>*> getWeights(int direction);
    void setNumNNLayers(int num);
    int getNumNNLayers();
    bool linkHasWeights();
    bool linkHasAWeights();
    bool linkHasBWeights();
};

} /* namespace PredictivePowertrain */

#endif /* LINK_H_ */
```

12.38 LinkUnitTest.cpp

```
//  main.cpp
//  Link Unit Test
//
//  Created by Silin Zeng on 3/4/16.
//  Copyright © 2016 Silin Zeng. All rights reserved.
//
```

```
#include "../driver_prediction/Link.h"
#include <iostream>
#include <assert.h>
#include "UnitTests.h"

using namespace std;
using namespace PredictivePowertrain;


void link_ut() {

    //Testing link() constructor, getDirection(), getNumber()
    Link firstLink;
    assert(firstLink.getDirection() == 0 && firstLink.getNumber() == 0);
    //assert(firstLink.getDirection() == 0 && firstLink.getNumber() == 1);

    //Testing link(x, x) constructor
    Link secondLink(10, 20);
    assert(secondLink.getDirection() == 10);
    assert(secondLink.getNumber() == 20);
    //assert(secondLink.getDirection() == 0 && secondLink.getNumber() == 0);

    //Testing get_hash(Link) function
    assert(secondLink.getHash() == 2 * secondLink.getDirection() + secondLink.getNumber());

    Link thirdLink(10, 20);
    Link fourthLink(10, 30);

    assert(secondLink.isEqual(&thirdLink));

    Link final = *(thirdLink.finalLink());
    assert(final.getNumber() == 0);
    assert(final.getDirection() == 0);

}
```

12.39 GPS.cpp

```
/*
 * GPS.cpp
 *
 *  Created on: Apr 17, 2016
 *      Author: vagrant
 */

#include "GPS.h"

namespace PredictivePowertrain {

GPS::GPS() {
      this->tripCount = 0;
```

```
    this->fd = -1;
    this->deltaXYTolerance = 10.0;
}

GPS::GPS(double refLat, double refLon)
{
    this->refLat = refLat;
    this->refLon = refLon;
    this->tripCount = 0;
    this->fd = -1;
    this->deltaXYTolerance = 10.0;
}

GPS::~GPS()
{
    close(this->fd);
}

std::pair<double, double>* GPS::updateTripLog()
{
      std::pair<double, double> latLonRef = this->readGPS();
    std::cout << "GPS: " << latLonRef.first << "," << latLonRef.second << std::endl;

    std::pair<double, double>* latLonPtr = new std::pair<double, double>(latLonRef.first, latLonRef.second);
      this->tripLog.addEntry(this->tripCount, latLonPtr);
    this->tripCount++;

    return latLonPtr;
}

GenericMap<long int, std::pair<double, double>*>* GPS::getTripLog(bool interpolated)
{
    if(interpolated)
    {
        GenericMap<long int, std::pair<double, double>*> beforeInterpTripLog;
        GenericMap<long int, std::pair<double, double>*> afterInterpTripLog;
        this->interpolateTripLog(&this->tripLog, &beforeInterpTripLog);

        for(int i = 0; i < 1; i++)
        {
            this->interpolateTripLog(&beforeInterpTripLog, &afterInterpTripLog);
            beforeInterpTripLog = afterInterpTripLog;
        }

        return afterInterpTripLog.copy();
    }

    return this->tripLog.copy();
}
```

```
void GPS::interpolateTripLog(GenericMap<long int, std::pair<double, double>*>* before, GenericMap<long int, std::pair<double, double>*>*
after)
{
    long int latLonCount = 0;

    before->initializeCounter();
    GenericEntry<long int, std::pair<double, double>*>* prevMeas = before->nextEntry();
    GenericEntry<long int, std::pair<double, double>*>* currMeas = before->nextEntry();
    while(currMeas != NULL)
    {
        double midLat = (prevMeas->value->first + currMeas->value->first) / 2;
        double midLon = (prevMeas->value->second + currMeas->value->second) / 2;

        after->addEntry(latLonCount++, new std::pair<double, double>(prevMeas->value->first, prevMeas->value->second));
        after->addEntry(latLonCount++, new std::pair<double, double>(midLat, midLon));

        prevMeas = currMeas;
        currMeas = before->nextEntry();
    }
    after->addEntry(latLonCount++, new std::pair<double, double>(prevMeas->value->first, prevMeas->value->second));

    delete(prevMeas);
    delete(currMeas);

}

// http://www.movable-type.co.uk/scripts/latlong.html
float GPS::deltaLatLonToXY(double lat1, double lon1, double lat2, double lon2)
{
    double dLat = toRadians(lat2-lat1);
    double dLon = toRadians(lon2-lon1);

    double a = std::sin(dLat/2) * std::sin(dLat/2) +
    std::cos(toRadians(lat1)) * std::cos(toRadians(lat2)) *
    std::sin(dLon/2) * std::sin(dLon/2);

    double c = 2 * std::atan2(std::sqrt(a), std::sqrt(1-a));
    float dist = (float) (EARTH_RADIUS * c);

    return dist;
}

std::pair<double, double>* GPS::convertLatLonToXY(double lat, double lon)
{
    float dist = deltaLatLonToXY(lat, lon, this->refLat, this->refLon);
    double angle = std::atan2(lat - this->refLat, lon - this->refLon);
    return new std::pair<double, double>(dist * std::sin(angle), dist * std::cos(angle));
}

std::pair<double, double>* GPS::convertXYToLatLon(double x, double y)
```

```
{
    double latDelta = y/EARTH_RADIUS;
    double lonDelta = x/(EARTH_RADIUS*std::cos(this->refLat));

    return new std::pair<double, double>(this->refLat+toDegrees(latDelta), this->refLon+toDegrees(lonDelta));
}

double GPS::toDegrees(double radians)
{
    return radians / (2 * M_PI) * 360.0;
}

double GPS::toRadians(double degrees)
{
    return degrees / 180.0 * M_PI;
}

void GPS::initializeGPSReader()
{
    std::cout << "initializing gps" << std::endl;
    this->fd = open("/dev/tty.usbmodemFA131", O_RDONLY | O_NONBLOCK);
    if(this->fd < 0)
    {
        std::cout << "Unable to open /dev/tty." << std::endl;
    }

    struct termios theTermios;

    memset(&theTermios, 0, sizeof(struct termios));
    cfmakeraw(&theTermios);
    cfsetspeed(&theTermios, 115200);

    theTermios.c_cflag = CREAD | CLOCAL;     // turn on READ
    theTermios.c_cflag |= CS8;
    theTermios.c_cc[VMIN] = 0;
    theTermios.c_cc[VTIME] = 10;     // 1 sec timeout
    ioctl(this->fd, TIOCSETA, &theTermios);
}

std::pair<double, double> GPS::readGPS()
{
    if(this->fd < 0)
    {
        this->initializeGPSReader();
    }

    // define vars
    char buf[255];
    size_t res;
```

```
while(true) // wait for read
{
    res = read(this->fd,buf,255);

    if(res > 0)
    {
        buf[res]=0;

        std::string nmeaMsg(buf);
        std::stringstream ss(nmeaMsg);

        std::string token;
        while(std::getline(ss, token, ','))
        {
            if(!token.compare("$GNGLL"))
            {
                std::string latString;
                std::string lonString;
                std::getline(ss, latString, ',');
                std::getline(ss, token, ',');
                std::getline(ss, lonString, ',');

                // break if bad read
                if(latString.length() == 0 || lonString.length() == 0)
                {
                    break;
                }
                // get raw deg / min values
                double latRaw = std::stod(latString);
                double lonRaw = std::stod(lonString);

                // get minutes w/o minute fractions
                double latMin = (int)latRaw % 100;
                double lonMin = (int)lonRaw % 100;

                // get degrees x100
                double latDeg = ((int)latRaw - latMin);
                double lonDeg = ((int)lonRaw - lonMin);

                // add minute fractions to minutes
                latMin += latRaw - latDeg - latMin;
                lonMin += lonRaw - lonDeg - lonMin;

                // divide lat / lon
                latDeg /= 100;
                lonDeg /= 100;

                // add minutes converted to degrees to lat / lon degrees
                latDeg += latMin / 60;
                lonDeg += lonMin / 60;
```

```
                    //printf("%.6f,%.6f\n", lat, lon);
                    std::pair<double, double> latLon(latDeg, -lonDeg);
                    return latLon;
                    break;
                }
                else
                {
                    break;
                }
            }
        }
    }
}

bool GPS::isOnRoad(Road* road)
{
    if(road == NULL)
    {
        return false;
    }

    std::pair<double, double> latLon = this->readGPS();

    road->getNodes()->initializeCounter();
    GenericEntry<long int, Node*>* nextNode = road->getNodes()->nextEntry();
    while(nextNode != NULL)
    {
        if(this->deltaLatLonToXY(latLon.first, latLon.second, nextNode->value->getLat(), nextNode->value->getLon()) < this->deltaXYTolerance)
        {
            return true;
        }
        nextNode = road->getNodes()->nextEntry();
    }
    delete(nextNode);

    return false;

}

bool GPS::isAtIntersection(Intersection* intersection)
{
    std::pair<double, double> latLon = this->readGPS();

    if(this->deltaLatLonToXY(latLon.first, latLon.second, intersection->getLat(), intersection->getLon()) < this->deltaXYTolerance)
    {
        return true;
    }
    return false;
}
```

```
Road* GPS::getCurrentRoad2(City* city, double lat, double lon)
{
    GenericMap<long int, Road*>* roads = city->getRoads();
    Road* closestRoad = NULL;
    float closestDist = MAXFLOAT;

    roads->initializeCounter();
    GenericEntry<long int, Road*>* nextRoad = roads->nextEntry();
    while(nextRoad != NULL)
    {
        GenericMap<long int, Node*>* roadNodes = nextRoad->value->getNodes();

        roadNodes->initializeCounter();
        GenericEntry<long int, Node*>* nextNode = roadNodes->nextEntry();
        while(nextNode != NULL)
        {
            float currDist = this->deltaLatLonToXY(lat, lon, nextNode->value->getLat(), nextNode->value->getLon());
            if(currDist < closestDist)
            {
//                bool headingIsStart2End = this->isHeadingStart2EndOfCurrentRoad(nextRoad->value);
//                float distAlongRoad = this->getDistAlongRoad(nextRoad->value, false, headingIsStart2End);
//
//                if(city->roadIsOnTrace(nextRoad->value, &this->tripLog, distAlongRoad / nextRoad->value->getSplineLength() - .1))
//                {
//                    closestDist = currDist;
//                    closestRoad = nextRoad->value;
//                }

                closestDist = currDist;
                closestRoad = nextRoad->value;
            }
            nextNode = roadNodes->nextEntry();
        }
        delete(nextNode);

        nextRoad = roads->nextEntry();
    }
    delete(nextRoad);

    return closestRoad;
}

Road* GPS::getCurrentRoad1(City* city)
{
    std::pair<double, double> latLon = this->readGPS();
    return this->getCurrentRoad2(city, latLon.first, latLon.second);
}

float GPS::getDistAlongRoad(Road* road, bool updateTripLog, bool headingIsStart2End)
```

```
{
    double lat;
    double lon;
    if(updateTripLog)
    {
        std::pair<double, double>* latLon = this->updateTripLog();
        lat = latLon->first;
        lon = latLon->second;
    }
    else
    {
        std::pair<double, double> latLon = this->readGPS();
        lat = latLon.first;
        lon = latLon.second;
    }

    GenericMap<long int, Node*>* nodes = road->getNodes();
    float distAlongRoad = 0.0;

    nodes->initializeCounter();
    GenericEntry<long int, Node*>* nextNode = nodes->nextEntry();

    double prevLat = nextNode->value->getLat();
    double prevLon = nextNode->value->getLon();

    nextNode = nodes->nextEntry();

    while(nextNode != NULL)
    {
        double currLat = nextNode->value->getLat();
        double currLon = nextNode->value->getLon();

        distAlongRoad += this->deltaLatLonToXY(prevLat, prevLon, currLat, currLon);

        // stop tracking distance once in proximity to current lat / lon
        if(this->deltaLatLonToXY(lat, lon, currLat, currLon) < this->deltaXYTolerance)
        {
            break;
        }

        prevLat = currLat;
        prevLon = currLon;

        nextNode = nodes->nextEntry();
    }
    delete(nextNode);

    Node* startNode = road->getNodes()->getEntry(0);

    Intersection* startInt = road->getStartIntersection();
```

```
    Intersection* endInt = road->getEndIntersection();

    float start2StartDist = this->deltaLatLonToXY(startNode->getLat(), startNode->getLon(), startInt->getLat(), startInt->getLon());
    float start2endDist = this->deltaLatLonToXY(startNode->getLat(), startNode->getLon(), endInt->getLat(), endInt->getLon());

    bool nodeIsStart2End = start2StartDist < start2endDist;

    if(headingIsStart2End != nodeIsStart2End)
    {
        distAlongRoad = road->getSplineLength() - distAlongRoad;
    }

    return distAlongRoad;
}

bool GPS::isHeadingStart2EndOfCurrentRoad(Road* road)
{
    // get lat lon
    std::pair<double, double> latLon = this->readGPS();
    double lat = latLon.first;
    double lon = latLon.second;

    // get road nodes
    GenericMap<long int, Node*>* nodes = road->getNodes();

    // iterate over road nodes
    nodes->initializeCounter();
    GenericEntry<long int, Node*>* nextNode = nodes->nextEntry();

    double prevLat = nextNode->value->getLat();
    double prevLon = nextNode->value->getLon();

    nextNode = nodes->nextEntry();

    double currLat = prevLat;
    double currLon = prevLon;

    while(nextNode != NULL)
    {
        currLat = nextNode->value->getLat();
        currLon = nextNode->value->getLon();

        // stop tracking distance once in proximity to current lat / lon
        if(this->deltaLatLonToXY(lat, lon, currLat, currLon) < this->deltaXYTolerance)
        {
            break;
        }

        // update prev
        prevLat = currLat;
```

```
        prevLon = currLon;

        nextNode = nodes->nextEntry();
    }
    delete(nextNode);

    double dLat = prevLat - currLat;
    double dLon = prevLon - currLon;

    // evaluate spline heading
    double angleSpline = std::atan2(dLat, dLon);
    angleSpline = this->boundTheta(angleSpline);

    // get gps heading
    double angleHeading = this->getHeadingAngle();

    // discern direction of travel relative to eval orientation of spline
    bool evalUp = false;
    float angleDiff = std::abs(angleHeading - angleSpline);
    if(angleDiff < M_PI / 4 || angleDiff > 2 * M_PI - M_PI / 4)
    {
        evalUp = true;
    }

    // determine where start and end intersections are located
    Node* endNodeWRTHeading;
    if(evalUp)
    {
        int nodeSize = road->getNodes()->getSize() - 1;
        endNodeWRTHeading = road->getNodes()->getEntry(nodeSize);
    }
    else
    {
        endNodeWRTHeading = road->getNodes()->getEntry(0);
    }

    double endLat = endNodeWRTHeading->getLat();
    double endLon = endNodeWRTHeading->getLon();

    Intersection* startInt = road->getStartIntersection();
    Intersection* endInt = road->getEndIntersection();

    float startDist = this->deltaLatLonToXY(endLat, endLon, startInt->getLat(), startInt->getLon());
    float endDist = this->deltaLatLonToXY(endLat, endLon, endInt->getLat(), endInt->getLon());

    return endDist < startDist;
}

double GPS::getHeadingAngle()
{
```

```
    std::pair<double, double> latLon1 = this->readGPS();
    std::pair<double, double> latLon2 = this->readGPS();

    double dLat = latLon1.first - latLon2.first;
    double dLon = latLon1.second - latLon2.second;

    double angle = std::atan2(dLat, dLon);
    angle = this->boundTheta(angle);

    return angle;
}

double GPS::boundTheta(double theta)
{
    while(theta < 0)
    {
        theta += 2 * M_PI;
    }

    while(theta > 2 * M_PI)
    {
        theta -= 2 * M_PI;
    }

    return theta;
}

double GPS::getDeltaXYTolerance()
{
    return this->deltaXYTolerance;
}

}
```

12.40 GPS.h

```
/*
 * GPS.h
 *
 *  Created on: Apr 17, 2016
 *      Author: vagrant
 */

#ifndef GPS_GPS_H_
#define GPS_GPS_H_

#include <utility>
#include <cmath>
#include <math.h>

#include <algorithm>
```

```
#include <iostream>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>

#include "../city/City.h"
#include "../city/Road.h"
#include "../city/Intersection.h"

#define EARTH_RADIUS 6371000.0

namespace PredictivePowertrain {

class Road;             // forward declaration
class Intersection;     // forward declaration
class City;             // forward declaration

class GPS {
private:

      GenericMap<long int, std::pair<double, double>*> tripLog;

      long int tripCount;
    double deltaXYTolerance;
    double toRadians(double degrees);
    double toDegrees(double radians);
    double refLat;
    double refLon;
    int fd;

    void initializeGPSReader();
    void  interpolateTripLog(GenericMap<long  int,  std::pair<double,  double>*>*  before,  GenericMap<long  int,  std::pair<double,  double>*>*
after);

public:
      GPS();
    GPS(double refLat, double refLon);
    ~GPS();

    std::pair<double, double>* updateTripLog();
    Road* getCurrentRoad1(City* city);
    Road* getCurrentRoad2(City* city, double lat, double lon);
      GenericMap<long int, std::pair<double, double>*>* getTripLog(bool interpolated);
    float deltaLatLonToXY(double lat1, double lon1, double lat2, double lon2);
```

```
    std::pair<double, double>* convertLatLonToXY(double lat, double lon);
    std::pair<double, double>* convertXYToLatLon(double x, double y);
    std::pair<double, double> readGPS();
    bool isOnRoad(Road* road);
    bool isAtIntersection(Intersection* intersection);
    float getDistAlongRoad(Road* road, bool updateTripLog, bool headingIsStart2End);
    bool isHeadingStart2EndOfCurrentRoad(Road* road);
    double getHeadingAngle();
    double getDeltaXYTolerance();
    double boundTheta(double theta);
};

}

#endif /* GPS_GPS_H_ */
```

12.41 GPSUnitTest.cpp

```
/*
 * GPSUnitTest.cpp
 *
 *  Created on: Apr 30, 2016
 *      Author: vagrant
 */

#include "UnitTests.h"
#include "../gps/GPS.h"
#include "../data_management/DataManagement.h"
#include "../city/City.h"

using namespace PredictivePowertrain;

void GPS_ut() {

    DataManagement dm;
    City* city = dm.getCityData();

    GPS test;

    // heading
    int j = 0;
    while(j < 100)
    {
        Road* currRoad = test.getCurrentRoad1(city);

        std::cout << currRoad->getRoadID() << ": ";
        std::cout << test.isHeadingStart2EndOfCurrentRoad(currRoad);
        std::cout << std::endl;

        j++;
    }
```

```
    // ability to record gps measurements
    int i = 0;
    while(i < 10)
    {
        std::pair<double, double> latLon = test.readGPS();
        printf("%.6f,%.6f\n", latLon.first, latLon.second);
        i++;
    }

}
```

12.42 GenericMap.h

```
/*
 * GenericMap.h
 *
 *  Created on: Mar 12, 2016
 *      Author: vagrant
 */

#ifndef ROUTE_PREDICTION_GENERICMAP_H_
#define ROUTE_PREDICTION_GENERICMAP_H_
#include "stddef.h"
#include <functional>
#include <iostream>
#include <map>
#include "GenericEntry.h"

namespace PredictivePowertrain {

template<typename K, typename V>
class GenericMap {
private:
      std::map<K,V> map;
      typename std::map<K,V>::iterator iter;
      bool hasNextEntry;
public:
      GenericMap();
      GenericMap(GenericMap& other);
      GenericMap* copy();
      void initializeCounter();
      GenericEntry<K,V>* nextEntry();
      GenericEntry<K,V>* getMinEntry();
      bool hasEntry(K key);
      V getEntry(K key);
    V getFirstEntry();
    V getLastEntry();
      typename std::map<K,V>::iterator iterator();
      typename std::map<K,V>::iterator begin();
```

```
      typename std::map<K,V>::iterator end();
      int getSize();
      int addEntry(K key, V value);
      void updateEntry(K key, V value);
      virtual ~GenericMap();
      bool erase(K key);
    bool indexErase(K key);
};

template<class K, class V>
GenericMap<K, V>::GenericMap()
{
      this->map = std::map<K,V>();
      this->iter = map.end();
      this->hasNextEntry = false;
}

template<class K, class V>
GenericMap<K, V>::GenericMap(GenericMap& other)
{
      this->map = std::map<K,V>();
      this->iter = map.end();
//    this->hashCounter = 0;
//    this->arrayCounter = 0;
      this->hasNextEntry = false;
}

template<class K, class V>
GenericMap<K, V>* GenericMap<K, V>::copy()
{
      GenericMap<K, V>* newMap = new GenericMap<K, V>();

      this->initializeCounter();
      GenericEntry<K,V>* nextEntry = this->nextEntry();
      while(nextEntry != NULL)
      {
        newMap->addEntry(nextEntry->key, nextEntry->value);
        nextEntry = this->nextEntry();
      }
      delete(nextEntry);
      return newMap;
}

template<class K, class V>
void GenericMap<K, V>::initializeCounter()
{
      if(this->getSize() > 0)
      {
        this->iter = this->map.begin();
        this->hasNextEntry = true;
```

```
	}
}

template<class K, class V>
GenericEntry<K,V>* GenericMap<K, V>::nextEntry()
{
	GenericEntry<K,V>* entry = NULL;
	if(this->hasNextEntry) {
	  entry = new GenericEntry<K,V>(this->iter->first,this->iter->second);
	  this->iter++;
	}

	if(this->iter == this->map.end()) {
	  this->hasNextEntry = false;
	}
	return entry;
}

template<class K, class V>
GenericEntry<K, V>* GenericMap<K, V>::getMinEntry()
{
	typename std::map<K,V>::iterator iter = this->map.begin();
	V minVal = NULL;
	K key = NULL;
	if (iter != this->map.end()) {
	  minVal = iter->second;
	  key = iter->first;
	  iter++;
	  while (iter != this->map.end()) {
		if (minVal > iter->second) {
			minVal = iter->second;
			key = iter->first;
		}
		iter++;
	  }
	}
	GenericEntry<K,V>* entry = new GenericEntry<K,V>(key, minVal);
	return entry;
}

template<class K, class V>
bool GenericMap<K, V>::hasEntry(K key)
{
	typename std::map<K,V>::iterator iter = this->map.find(key);
	if (iter != this->map.end()) {
	  return true;
	} else {
	  return false;
	}
}
```

```
template<class K, class V>
bool GenericMap<K, V>::erase(K key)
{
      typename std::map<K,V>::iterator iter = this->map.find(key);
      if (iter != this->map.end()) {
        this->map.erase(iter);

        return true;
      } else {
        return false;
      }
}

template<class K, class V>
bool GenericMap<K, V>::indexErase(K key)
{
    typename std::map<K,V>::iterator iter = this->map.find(key);
    if (iter != this->map.end()) {

        iter = this->map.erase(iter);

        while(iter != this->map.end())
        {
            V entry = this->getEntry(iter->first);
            this->addEntry(iter->first - 1, entry);
            iter = this->map.erase(iter);
        }

        if(this->hasNextEntry)
        {
            K searchKey = key;
            if(searchKey != 0)
            {
                searchKey -= 1;
            }

            this->iter = this->map.find(searchKey);
        }

        return true;
    } else {
        return false;
    }

}

template<class K, class V>
V GenericMap<K, V>::getEntry(K key)
{
```

```
      typename std::map<K,V>::iterator iter = this->map.find(key);
      if (iter != this->map.end()) {
        return iter->second;
      } else {
        return NULL;
      }
}

template<class K, class V>
V GenericMap<K, V>::getFirstEntry()
{

    if (this->getSize() > 0) {
        return this->begin()->second;
    } else {
        return NULL;
    }
}

template<class K, class V>
V GenericMap<K, V>::getLastEntry()
{

    if (this->getSize() > 0) {
        return this->map.rbegin()->second;
    } else {
        return NULL;
    }
}

template<class K, class V>
typename std::map<K, V>::iterator GenericMap<K, V>::begin()
{
      return this->map.begin();
}

template<class K, class V>
typename std::map<K, V>::iterator GenericMap<K, V>::end()
{
      return this->map.end();
}

template<class K, class V>
int GenericMap<K, V>::getSize()
{
      return this->map.size();
}

template<class K, class V>
int GenericMap<K, V>::addEntry(K key, V value)
```

```
{
      if(this->getEntry(key) != NULL) {
        return 0;
      } else if(this->map.insert(std::pair<K,V>(key, value)).second == true) {
        return 1;
      } else {
        return 0;
      }
}

template<class K, class V>
void GenericMap<K, V>::updateEntry(K key, V value)
{
      typename std::map<K,V>::iterator iter = this->map.find(key);
        if (iter != this->map.end())
        {
                this->map.erase(iter);
        }
        this->addEntry(key, value);
}

template<class K, class V>
GenericMap<K,V>::~GenericMap()
{
    typename std::map<K,V>::iterator iter = this->map.begin();

    while(iter != this->map.end())
    {
        iter = this->map.erase(iter);
    }
}


} /* namespace PredictivePowertrain */

#endif /* ROUTE_PREDICTION_GENERICMAP_H_ */
```

12.43 GenericEntry.h

```
/*
 * GenericEntry.h
 *
 *  Created on: Mar 12, 2016
 *      Author: vagrant
 */

#ifndef ROUTE_PREDICTION_GENERICENTRY_H_
#define ROUTE_PREDICTION_GENERICENTRY_H_

namespace PredictivePowertrain {
```

```
template<class K, class V>
class GenericEntry {
public:
      K key;
      V value;
      GenericEntry(K, V);
      virtual ~GenericEntry();
};


template<class K, class V>
GenericEntry<K,V>::GenericEntry(K key, V value) {
      // TODO Auto-generated constructor stub
      this->key = key;
      this->value = value;

}

template<class K, class V>
GenericEntry<K,V>::~GenericEntry() {
      // TODO Auto-generated destructor stub
}

} /* namespace PredictivePowertrain */

#endif /* ROUTE_PREDICTION_GENERICENTRY_H_ */
```

12.44 Probability.cpp

```
//
#include "Probability.h"

namespace PredictivePowertrain {

Probability::Probability()
{
      this->numerator = 0;
      this->denominator = 0;
}

void Probability::addNumerator(float addition)
{
      this->numerator = this->numerator + addition;
}

void Probability::addDenominator(float addition)
{
      this->denominator = this->denominator + addition;
}

float Probability::getProbability()
```

```
{
      if(this->denominator == 0)
    {
       return 0.0;
    }

      return this->numerator/this->denominator;
}

}
```

12.45 Probability.h

```
#ifndef PROBABILITY_H_
#define PROBABILITY_H_

namespace PredictivePowertrain {

class Probability{
private:
      float numerator;
      float denominator;

public:
      Probability();
      void addNumerator(float addition);
      void addDenominator(float addition);
      float getProbability();
};

}

#endif /* PROBABILITY_H_ */
```

12.46 ProbabilityUnitTest.cpp

```
/*
 * ProbabilityUnitTest.cpp
 *
 *  Created on: Apr 5, 2016
 *      Author: vagrant
 */
#include "../route_prediction/Probability.h"
#include <iostream>
#include <assert.h>
#include "UnitTests.h"

using namespace PredictivePowertrain;

void probability_ut() {
      Probability firstProb;
      // testing constructor(numerator = 0 & denom = 0)
      // testing getProbability method if denom = 0
```

```
      assert(firstProb.getProbability() == 0.0);

      // testing addDenominator & getProbability, (double)0/1 == 0.0
      firstProb.addDenominator(1);
      assert(firstProb.getProbability() == 0.0);

      // testing addDenominator & getProbability, (double)1/1 == 1.0
      firstProb.addNumerator(1);
      assert(firstProb.getProbability() == 1.0);
      assert(firstProb.getProbability() != 0.0);

}
```

12.47 Route.cpp

```
/*
 * Route.cpp
 *
 *  Created on: Mar 6, 2016
 *      Author: vagrant
 */

#include "Route.h"

namespace PredictivePowertrain {

Route::~Route()
{
    delete(this->links);
    delete(this->intersection);
}

Route::Route()
{
      this->linkCount = 0;
    this->links = new GenericMap<long int, Link*>();
    this->intersection = new Intersection();
}

Route::Route(GenericMap<long int, Link*>* links, Goal* goal) {
      this->links = links;
      this->goal = goal;
      this->linkCount = links->getSize();
      Link error(NULL, -1);
      this->error = &error;
    this->intersection = new Intersection();
}

// adds new link to end of route
void Route::addLink(Link* link) {
      this->links->addEntry(this->linkCount, link);
```

```
    this->linkCount++;
}

// checks if the route is equal to the route passed in
bool Route::isEqual(Route * other)
{
      if(this->getLinkSize() == other->getLinkSize()) {
        GenericMap<long int, Link*>* otherLinks = other->getLinks();

         for(int i = 0; i < this->getLinkSize(); i++)
         {
             Link* thisLink = this->links->getEntry(i);
             Link* otherLink = otherLinks->getEntry(i);

             if(!thisLink->isEqual(otherLink))
             {
                 return false;
             }
         }

        return true;
      } else {
        return false;
      }
}

Route* Route::copy() {
      Goal* newGoal = new Goal(this->goal);
      GenericMap<long int, Link*>* newLinks = this->links->copy();
      Route* route = new Route(newLinks, newGoal);
      return route;
}

long int Route::getGoalHash() {
      return this->goal->getHash();
}

Goal* Route::getGoal() {
      return this->goal;
}

void Route::assignGoal(Goal* goal)
{
    this->goal = goal;
}

int Route::getLinkSize() {
      return this->links->getSize();
}
```

```
GenericMap<long int, Link*>* Route::getLinks() {
      return this->links;
}

void Route::setToIntersection(Intersection* other) {
      this->intersection = other;
      this->routeIsIntersection = true;
}

void Route::setToRoute() {
    this->intersection = NULL;
    this->routeIsIntersection = false;
}

bool Route::isIntersection() {
      return this->routeIsIntersection;
}

Intersection* Route::getIntersection() {
      return this->intersection;
}

Link* Route::getLastLink() {
      return this->links->getEntry(this->linkCount - 1);
}

bool Route::isEmpty() {
      return this->links->getSize() == 0;
}

Link* Route::getEntry(int index) {
      if(index > this->linkCount - 1)
      {
        return this->error;
      }
      return this->links->getEntry(index);
}

void Route::removeFirstLink() {
      if(this->links->getSize() > 0)
      {
        this->links->indexErase(0);
      }
    this->linkCount--;
}

void Route::saveRoute2CSV(FILE* file, City* city, bool includeheaderAndCloseFile)
{
    if(includeheaderAndCloseFile)
    {
```

```
    fprintf(file, "name, description, color, latitude, longitude\n");
}

this->links->initializeCounter();
GenericEntry<long int, Link*>* nextLink = this->links->nextEntry();
while(nextLink != NULL && !nextLink->value->isFinalLink())
{

    Road* currRoad = city->getRoads()->getEntry(nextLink->value->getNumber());

    // intersections
    for(int i = 0; i < 2; i++)
    {
        Intersection* intersection = currRoad->getStartIntersection();
        if(i == 1)
        {
            intersection = currRoad->getEndIntersection();
        }

        // intersections
        fprintf(file, "%ld,", intersection->getIntersectionID());
        fprintf(file, "Lat & Lon: %.12f %.12f,", intersection->getLat(), intersection->getLon());
        fprintf(file, "blue,");
        fprintf(file, "%.12f,%.12f\n", intersection->getLat(), intersection->getLon());
    }

    currRoad->getNodes()->initializeCounter();
    GenericEntry<long int, Node*>* nextNode = currRoad->getNodes()->nextEntry();

    // skip first node
    nextNode = currRoad->getNodes()->nextEntry();
    int nodeCount = 1;

    while(nodeCount < currRoad->getNodes()->getSize() - 1)
    {
        // road nodes
        fprintf(file, "%ld,", currRoad->getRoadID());
        fprintf(file, "Lat & Lon: %.12f %.12f | ", nextNode->value->getLat(), nextNode->value->getLon());
        fprintf(file, "Start Intersection: %ld | ", currRoad->getStartIntersection()->getIntersectionID());
        fprintf(file, "End Intersection: %ld,", currRoad->getEndIntersection()->getIntersectionID());
        fprintf(file, "red,");
        fprintf(file, "%.12f,%.12f\n", nextNode->value->getLat(), nextNode->value->getLon());

        nodeCount++;

        nextNode = currRoad->getNodes()->nextEntry();
    }
    delete(nextNode);

    nextLink = this->links->nextEntry();
```

```
    }
    delete(nextLink);

    if(includeheaderAndCloseFile)
    {
        fclose(file);
    }
}

void Route::printLinks()
{
    this->links->initializeCounter();
    GenericEntry<long int, Link*>* nextLink = this->links->nextEntry();
    while(nextLink != NULL)
    {
        std::cout << nextLink->value->getNumber() << " | ";
        nextLink = this->links->nextEntry();
    }
    std::cout << std::endl;

    delete(nextLink);
}

void Route::replaceLinks(GenericMap<long int, Link*>* newLinks)
{
    assert(newLinks->getSize() == this->links->getSize());
    delete(this->links);
    this->links = newLinks;
}

void Route::addLinkToFront(Link* frontLink)
{
    GenericMap<long int, Link*>* newLinks = new GenericMap<long int, Link*>();
    newLinks->addEntry(0, frontLink);

    this->links->initializeCounter();
    GenericEntry<long int, Link*>* nextLink = this->links->nextEntry();
    while(nextLink != NULL)
    {
        newLinks->addEntry(nextLink->key + 1, nextLink->value);
        nextLink = this->links->nextEntry();
    }
    delete(nextLink);

    delete(this->links);
    this->links = newLinks;

    this->linkCount++;
}
```

```
}
```

12.48 Route.h

```
/*
 * Route.h
 *
 *  Created on: Mar 6, 2016
 *      Author: vagrant
 */

#ifndef ROUTE_H_
#define ROUTE_H_

#include <iostream>
#include <fstream>

#include "Goal.h"
#include "../driver_prediction/Link.h"
#include "../city/Intersection.h"
#include "../city/City.h"

namespace PredictivePowertrain {

class City;         // forward delcaration

class Route {
private:
      GenericMap<long int, Link*>* links;
      Goal* goal;
      Link* error;
      Intersection* intersection;
      bool routeIsIntersection;
      int linkCount;
public:
      Route();
      Route(GenericMap<long int, Link*>* links, Goal* goal);
    void printLinks();
    virtual ~Route();
    void addLink(Link* link);
      bool isEqual(Route* other);
      long int getGoalHash();
      int getLinkSize();
    void assignGoal(Goal* goal);
      Link* getEntry(int index);
      void setToIntersection(Intersection* other);
    void setToRoute();
      bool isIntersection();
      bool isEmpty();
      Goal* getGoal();
      GenericMap<long int, Link*>* getLinks();
```

```
      Intersection* getIntersection();
      Link* getLastLink();;
      Route* copy();
      void removeFirstLink();
    void addLinkToFront(Link* frontLink);
    void replaceLinks(GenericMap<long int, Link*>* newLinks);
    void saveRoute2CSV(FILE* file, City* city, bool includeheaderAndCloseFile);
};

}

#endif
```

12.49 RouteUnitTest.cpp

```
/*
 * RouteUnitTest.cpp
 *
 *  Created on: Mar 17, 2016
 *      Author: vagrant
 */

#include "../driver_prediction/Link.h"
#include "../route_prediction/Goal.h"
#include "../route_prediction/Route.h"
#include <iostream>
#include <assert.h>
#include <vector>
#include "UnitTests.h"

using namespace std;
using namespace PredictivePowertrain;

void route_ut() {

      Link link1(1, 0);
      Link link2(2, 1);
      Link link3(3, 0);

      Link links1[3];
      links1[0] = link1;
      links1[1] = link2;
      links1[2] = link3;
      Link links2[3];
      links2[0] = link3;
      links2[1] = link1;
      links2[2] = link2;

    std::vector<float> bins = {1};

      Goal goal1(1, &bins);
```

```
      Goal goal2(2, &bins);

//    Route route1(links1, &goal1);
//    Route route2(links1, &goal2);
//    Route route3(links1, &goal1);
//    Route route4(links2, &goal1);
//    Route route5(links2, &goal2);


      // Test 1: addlink
      Link * emptyLinks;
//    Route route(emptyLinks, &goal1);
//    route.addlink(&link1);
//    assert((route.getLastLinkPtr())->isEqual(&link1) && route.getLinkSize() == 1);
//    route.addlink(&link2);
//    assert((route.getLastLinkPtr())->isEqual(&link2) && route.getLinkSize() == 2);
//    route.addlink(&link3);
//    assert((route.getLastLinkPtr())->isEqual(&link3) && route.getLinkSize() == 3);


      // Test 2: iterator
//    Route tempRoute(links1, &goal1);
//    int linksLength = sizeof(*links1)/sizeof(Link);
//    for(int i = 0; i < linksLength; i++) {
//      Link* tempLink = tempRoute.nextlink();
//      assert(links1[i].isEqual(tempLink));
//    }
//    Link* lastLink = route.nextlink();
//    assert(links1[linksLength - 1].isEqual(lastLink));


      // Test 3: isequal to oneself
//    assert(route1.isequal(&route1));


      // Test 4: isequal is reversible
//    assert(route1.isequal(&route3));
//    assert(route3.isequal(&route1));
//    assert(!route1.isequal(&route2));
//    assert(!route2.isequal(&route1));


      // Test 5: isequal with same links / goals (but still unequal)
//    assert(!route1.isequal(&route4));
//    assert(!route1.isequal(&route5));


      // Test 6: copy
//    Route* route1Copy = route1.copy();
//    assert(route1.isequal(route1Copy));
```

```
//    Link * linksPtr = route2.getLinksPtr();
//    linksPtr[0] = link2;
//    assert(!route1.isequal(route1Copy));
}
```

12.50 LinkToStateMap.cpp

```
/*
 * LinkToStateMap.cpp
 *
 *  Created on: Mar 13, 2016
 *      Author: vagrant
 */

#include "LinkToStateMap.h"

namespace PredictivePowertrain {

int LinkToStateMap::incrementTransition(Link* lj, Goal* gj, Link* li)
{
      long int goalHash = gj->getHash();
      GoalMapEntry<long int, LinkToStateMapEntry*>* goalEntry;
      if(!(this->goalMap->hasEntry(goalHash))) {
        goalEntry = new GoalMapEntry<long int, LinkToStateMapEntry*>(gj);
        this->goalMap->addEntry(goalHash, goalEntry);
      } else {
        goalEntry = this->goalMap->getEntry(goalHash);
      }
    goalEntry->incrementCount();

      long int linkHash = lj->getHash();
      if(!goalEntry->getMap()->hasEntry(linkHash)) {
        goalEntry->addMapEntry(linkHash, new LinkToStateMapEntry);
      }
      return goalEntry->getMapEntry(linkHash)->addEntry(li);
}

float LinkToStateMap::getProbability(Link* li, Link* lj, Goal* gj, bool isSimilar)
{
      Probability pg;
      Probability pllg;
      this->goalMap->initializeCounter();
      GenericEntry<long int, GoalMapEntry<long int, LinkToStateMapEntry*>*>* nextGoalMapEntry = this->goalMap->nextEntry();
      while(nextGoalMapEntry != NULL) { // next->key != 1
        Goal* g = nextGoalMapEntry->value->getGoal();
        if(gj->isEqual(g) || isSimilar) {
                LinkToStateMapEntry* l2Entry = nextGoalMapEntry->value->getMapEntry(lj->getHash());
                if(l2Entry != NULL)
                {
                        pllg.addDenominator(l2Entry->getTotalM());
                        pllg.addNumerator(l2Entry->getM(li));
```

```
                }

        }
        pg.addDenominator(nextGoalMapEntry->value->getM());
        if(g->isEqual(gj)) {
                pg.addNumerator(nextGoalMapEntry->value->getM());
        }
        nextGoalMapEntry = this->goalMap->nextEntry();
      }
      float pl = pllg.getProbability();
      //float pllg = pl * pg.getProbability();
      return pl;
}

LinkToStateMap::LinkToStateMap()
{
    this->goalMap = new GenericMap<long int, GoalMapEntry<long int, LinkToStateMapEntry*>*>();
}

LinkToStateMap::LinkToStateMap(LinkToStateMap& other)
{
    this->goalMap = other.getGoalMap()->copy();
}

LinkToStateMap::~LinkToStateMap()
{
    delete(this->goalMap);
}

GenericMap<long int, GoalMapEntry<long int, LinkToStateMapEntry*>*>* LinkToStateMap::getGoalMap()
{
    return this->goalMap;
}

} /* namespace PredictivePowertrain */
```

12.51 LinkToStateMap.h

```
/*
 * LinkToStateMap.h
 *
 *  Created on: Mar 13, 2016
 *      Author: vagrant
 */

#ifndef ROUTE_PREDICTION_LINKTOSTATEMAP_H_
#define ROUTE_PREDICTION_LINKTOSTATEMAP_H_


#include "../map/GenericMap.h"
```

```
#include "../map/GenericEntry.h"

#include "../driver_prediction/Link.h"

#include "GoalMapEntry.h"
#include "Goal.h"
#include "LinkToStateMapEntry.h"
#include "Probability.h"

namespace PredictivePowertrain {

class LinkToStateMap {
private:
      GenericMap<long int, GoalMapEntry<long int, LinkToStateMapEntry*>*>* goalMap;

public:
    GenericMap<long int, GoalMapEntry<long int, LinkToStateMapEntry*>*>* getGoalMap();
      int incrementTransition(Link* lj, Goal* gj, Link* li);
      float getProbability(Link* li, Link* lj, Goal* gj, bool isSimilar);
      LinkToStateMap();
      LinkToStateMap(LinkToStateMap& other);
      virtual ~LinkToStateMap();
};

} /* namespace PredictivePowertrain */

#endif /* ROUTE_PREDICTION_LINKTOSTATEMAP_H_ */
```

12.52 LinkToStateMapUnitTest.cpp

```
#include "../driver_prediction/Link.h"
#include "../route_prediction/Goal.h"
#include <stdlib.h>
#include <assert.h>
#include "../route_prediction/LinkToStateMap.h"
#include "UnitTests.h"
#include <iostream>
#include <vector>

using namespace PredictivePowertrain;
using namespace std;

void linkToStateMap_ut() {
      int linksSize = 5;
      Link* links[linksSize];
      srand(time(NULL));
      for(int i = 0; i < linksSize; i++) {
        int random = rand() % 5;
        Link newLink(i, random);
        links[i] = &newLink;
      }
```

```
std::vector<float> bins2 = {2};
std::vector<float> bins4 = {4};

  Goal goal1(1, &bins2);
  Goal goal2(2, &bins4);
  Goal goal3(2, &bins2);
  assert(goal1.isSimilar(&goal3)); // goal 1 and goal3 should be similar

  // Test 1: increment transition
  LinkToStateMap map;
  assert(map.incrementTransition(links[0], &goal1, links[1]) == 1);

  // Test 2: increment transition twice
  LinkToStateMap map2;
  int count = 0;
  for(int i = 0; i < 2; i++) {
    count += map2.incrementTransition(links[1], &goal2, links[3]);
  }
  assert(count == 1);

  // Test 3: increment two different transitions
  LinkToStateMap map3;
  int count1 = map3.incrementTransition(links[3], &goal1, links[0]);
  int count2 = map3.incrementTransition(links[4], &goal2, links[1]);
  assert(count1 == 1 && count1 == count2);

  // Test 4: get probability with nothing in map
  LinkToStateMap map4;
  assert(map4.getProbability(links[0], links[1], &goal1, true) == 0);
  map4.incrementTransition(links[2], &goal1, links[1]);

  assert(map4.getProbability(links[0], links[1], &goal1, true) == 0);
  assert(map4.getProbability(links[2], links[0], &goal1, true) == 0);
  assert(map4.getProbability(links[0], links[0], &goal1, true) == 0);

  // Test 5: get probability when only one goal in map
  LinkToStateMap map5;
  map5.incrementTransition(links[0], &goal1, links[1]);
  map5.incrementTransition(links[0], &goal1, links[2]);
  assert(map.getProbability(links[1], links[0], &goal1, false) == 0);

  // Test 6: get probability when other goals in map
  LinkToStateMap map6;
  map6.incrementTransition(links[0], &goal1, links[1]);
  map6.incrementTransition(links[0], &goal1, links[2]);
  map6.incrementTransition(links[0], &goal2, links[1]);
  assert(map6.getProbability(links[1], links[0], &goal1, false) == 0.5);

  // Test 7: test issimilar for getprobability
```

```
    LinkToStateMap map7;
    map7.incrementTransition(links[0], &goal1, links[1]);
    map7.incrementTransition(links[0], &goal1, links[2]);
    map7.incrementTransition(links[0], &goal3, links[1]);

    assert(map7.getProbability(links[1], links[0], &goal1, false) == 0.5);
    double pl = map7.getProbability(links[1], links[0], &goal1, true);
    assert(pl == (2.0/3.0));  // says it is == 1

    // Test 8: test issimilar for getprobability with non similar goals
    LinkToStateMap map8;
    map8.incrementTransition(links[4], &goal1, links[1]);
    map8.incrementTransition(links[4], &goal1, links[2]);
    map8.incrementTransition(links[4], &goal3, links[1]);
    map8.incrementTransition(links[4], &goal2, links[1]);

    assert(map8.getProbability(links[1], links[4], &goal1, false) == 0.5);
    double pl2 = map8.getProbability(links[1], links[4], &goal1, true);
    assert(pl2 == (2.0/3.0));

    // Test 9: unseen transition returns 0
    LinkToStateMap map9;
    assert(map9.getProbability(links[1], links[4], &goal1, false) == 0);

    // No copy function in LTSM
/*  // Test 10: copy
    LinkToStateMap map10;
    map10.incrementTransition(links[4], goal1, links[1]);
    map10.incrementTransition(links[4], goal1, links[2]);
    map10.incrementTransition(links[4], goal3, links[1]);
    map10.incrementTransition(links[4], goal2, links[1]);

    LinkToStateMap map10Copy = map10.copy();
    assert(map10.getProbability(links[1], links[4], goal1, false) ==
            map10Copy.getProbability(links[1], links[4], goal1, false));
    map10.incrementTransition(links[4], goal1, links[1]);
    assert(map10.getProbability(links[1], links[4], goal1, false) !=
                    map10Copy.getProbability(links[1], links[4], goal1, false));*/
}
```

12.53 GoalToLinkMap.cpp

```
/*
 * GoalToLinkMap.cpp
 *
 *  Created on: Mar 13, 2016
 *      Author: vagrant
 */

#include "GoalToLinkMap.h"
```

```
namespace PredictivePowertrain {

GoalToLinkMap::GoalToLinkMap()
{
    this->goalMap = new GenericMap<long int, GoalMapEntry<long int, int>*>();
}

GoalToLinkMap::GoalToLinkMap(GoalToLinkMap& other)
{
    this->goalMap = other.getGoalMap()->copy();
}

int GoalToLinkMap::linkTraversed(Link* link, Goal* goal)
{
      long int goalHash = goal->getHash();
      GoalMapEntry<long int, int>* goalEntry;
      if(!this->goalMap->hasEntry(goalHash)) {
        goalEntry = new GoalMapEntry<long int, int>(goal);
        this->goalMap->addEntry(goalHash, goalEntry);
      } else {
        goalEntry = this->goalMap->getEntry(goalHash);
      }
      goalEntry->incrementCount();

      long int linkHash = link->getHash();
    int count = 1;
    if(goalEntry->getMap()->hasEntry(linkHash))
    {
        count += goalEntry->getMapEntry(linkHash);
        goalEntry->getMap()->updateEntry(linkHash, count);
    }
    else
    {
        goalEntry->addMapEntry(linkHash, count);
    }
      return count;
}

std::vector<std::vector<float>*>* GoalToLinkMap::probabilityOfGoalsGivenLink(Link* link, Goal* goal, bool isSimilar)
{
      this->goalMap->initializeCounter();
      GenericEntry<long int, GoalMapEntry<long int, int>*>* goalEntry = this->goalMap->nextEntry();

    int probLength = this->goalMap->getSize();
    std::vector<std::vector<float>*>* prob = new std::vector<std::vector<float>*>(probLength);

      int probCount = 0;
      int totalLinkCount = 0;

      while(goalEntry != NULL) {
```

```
        prob->at(probCount) = new std::vector<float>(2);
        int linkCount = 0;

        if(goal->isSimilar(goalEntry->value->getGoal()) && goalEntry->value->getMap()->hasEntry(link->getHash()))
        {
            linkCount = goalEntry->value->getMapEntry(link->getHash());
            totalLinkCount += linkCount;
        }
        long int goalHash = goalEntry->value->getGoal()->getHash();

        prob->at(probCount)->at(0) = goalHash;
        prob->at(probCount)->at(1) = linkCount;

        goalEntry = this->goalMap->nextEntry();
        probCount++;
      }

      for (int i = 0; i < probLength; i++) {
        float prob_i = prob->at(i)->at(1) / ((totalLinkCount > 0 ) * totalLinkCount + (totalLinkCount <= 0));
        prob->at(i)->at(1) = prob_i;
      }
      return prob;
}

GenericMap<float, float>* GoalToLinkMap::probabilityOfGoalsGivenLinkMap(Link * link, Goal * goal, bool isSimilar)
{
    std::vector<std::vector<float>*>* matrix = this->probabilityOfGoalsGivenLink(link, goal, isSimilar);
    GenericMap<float, float>* result = new GenericMap<float, float>();
    size_t matrixLength = matrix->size();
    for(int i = 0; i < matrixLength; i++)
    {
        result->addEntry(matrix->at(i)->at(0), matrix->at(i)->at(1));
    }
    return result;
}

float GoalToLinkMap::probabilityOfGoalGivenLink(Link * link, Goal * goal, bool isSimilar)
{
    std::vector<std::vector<float>*>* matrix = this->probabilityOfGoalsGivenLink(link, goal, isSimilar);
      size_t matrixLength = matrix->size();
      double goalHash = (double) goal->getHash();
      for(int i = 0; i < matrixLength; i++) {
        if(matrix->at(i)->at(0) == goalHash) {
            float probability = matrix->at(i)->at(1);
            delete(matrix);
               return probability;
        }
      }
      return 0;
}
```

```
GenericMap<long int, GoalMapEntry<long int, int>*>* GoalToLinkMap::getGoalMap()
{
    return this->goalMap;
}

GoalToLinkMap::~GoalToLinkMap()
{
    delete(this->goalMap);
}

} /* namespace PredictivePowertrain */
```

12.54 GoalToLinkMap.h

```
/*
 * GoalToLinkMap.h
 *
 *  Created on: Mar 13, 2016
 *      Author: vagrant
 */

#ifndef ROUTE_PREDICTION_GOALTOLINKMAP_H_
#define ROUTE_PREDICTION_GOALTOLINKMAP_H_
#include "../map/GenericMap.h"
#include "GoalMapEntry.h"
#include "../driver_prediction/Link.h"
#include "Goal.h"
#include <vector>



namespace PredictivePowertrain {

class GoalToLinkMap {
private:
      GenericMap<long int, GoalMapEntry<long int, int>*>* goalMap;

public:

      GoalToLinkMap();
      GoalToLinkMap(GoalToLinkMap &other);
      int linkTraversed(Link* link, Goal* goal);
     GenericMap<long int, GoalMapEntry<long int, int>*>* getGoalMap();
      std::vector<std::vector<float>*>* probabilityOfGoalsGivenLink(Link* link, Goal* goal, bool isSimilar);
      float probabilityOfGoalGivenLink(Link* link, Goal* goal, bool isSimilar);
      GenericMap<float, float> * probabilityOfGoalsGivenLinkMap(Link * link, Goal * goal, bool isSimilar);
      virtual ~GoalToLinkMap();
};

} /* namespace PredictivePowertrain */
```

```
#endif /* ROUTE_PREDICTION_GOALTOLINKMAP_H_ */
```

12.55 GoalToLinkMapUnitTest.cpp

```
/*
 * GoalToLinkMapUnitTest.cpp
 *
 *  Created on: Apr 8, 2016
 *      Author: vagrant
 */

#include "../route_prediction/GoalToLinkMap.h"
#include "UnitTests.h"
#include <iostream>
#include <vector>
#include <assert.h>

using namespace PredictivePowertrain;

void goalToLinkMap_ut() {
      int linksSize = 5;
      Link* links[linksSize];
      srand(time(NULL));
      for(int i = 0; i < linksSize; i++) {
        int random = rand() % 5;
        Link newLink(i, random);
        links[i] = &newLink;
      }

    std::vector<float> bins1 = {2};
    std::vector<float> bins2 = {3, 4};
      Goal goal1(1, &bins1);
      Goal goal2(2, &bins2);
      Goal goal3(3, &bins1);

      // Test 1: adding links to a single goal
      GoalToLinkMap map1;
      assert(map1.linkTraversed(links[0], &goal1) == 1);
      assert(map1.linkTraversed(links[1], &goal1) == 2);
      assert(map1.linkTraversed(links[0], &goal1) == 2);

      // Test 2: adding links to multiple goals
      GoalToLinkMap map2;
      assert(map2.linkTraversed(links[0], &goal1) == 1);
      assert(map2.linkTraversed(links[0], &goal2) == 2);
      assert(map2.linkTraversed(links[1], &goal1) == 2);
      assert(map2.linkTraversed(links[1], &goal2) == 2);

      // Test 2.5: single goal, multiple links
      GoalToLinkMap map3;
```

```
    assert(map3.linkTraversed(links[0], &goal1) == 1);
    assert(map3.linkTraversed(links[1], &goal1) == 2);
    assert(map3.probabilityOfGoalGivenLink(links[0], &goal1, false) == 1);
    assert(map3.probabilityOfGoalGivenLink(links[1], &goal1, false) == 1);

    // Test 3: getting probabilities with dissimilar goals
    GoalToLinkMap map4;
    assert(map4.linkTraversed(links[0], &goal1) == 1);
    assert(map4.linkTraversed(links[0], &goal2) == 2);
    assert(map4.linkTraversed(links[1], &goal1) == 2);
    assert(map4.linkTraversed(links[0], &goal2) == 2);
    assert(map4.probabilityOfGoalGivenLink(links[0], &goal1, false) == (1/2));
    assert(map4.probabilityOfGoalGivenLink(links[0], &goal1, true) == 1);

    // Test 4: getting probabilities with similar goals
    GoalToLinkMap map5;
    assert(map5.linkTraversed(links[0], &goal1) == 1);
    assert(map5.linkTraversed(links[0], &goal3) == 1);
    assert(map5.linkTraversed(links[1], &goal1) == 1);
    assert(map5.linkTraversed(links[1], &goal3) == 1);
    assert(map5.probabilityOfGoalGivenLink(links[0], &goal1, false) == (1/2));
    assert(map5.probabilityOfGoalGivenLink(links[0], &goal1, true) == (1/2));

    // Test 5: getting probabilities with some similar and some not so similar goals
    GoalToLinkMap map6;
    assert(map6.linkTraversed(links[0], &goal1) == 1);
    assert(map6.linkTraversed(links[0], &goal3) == 1);
    assert(map6.linkTraversed(links[1], &goal1) == 1);
    assert(map6.linkTraversed(links[2], &goal3) == 1);
    assert(map6.linkTraversed(links[0], &goal1) == 2);
    assert(map6.linkTraversed(links[0], &goal2) == 1);
    assert(map6.linkTraversed(links[1], &goal1) == 2);
    assert(map6.linkTraversed(links[2], &goal2) == 1);
    assert(map6.probabilityOfGoalGivenLink(links[0], &goal1, false) == (1/2));
    assert(map6.probabilityOfGoalGivenLink(links[0], &goal1, false) == (2/3));

    // Test 6: return value for unseen transition is zero, with nothing in the map
    GoalToLinkMap map7;
    assert(map7.probabilityOfGoalGivenLink(links[0], &goal1, false) == 0);

    // Test 7: return value for unseen transition is zero with things in the map
    GoalToLinkMap map8;
    map8.linkTraversed(links[0], &goal1);
    assert(map8.probabilityOfGoalGivenLink(links[1], &goal1, false) == 0);
    assert(map8.probabilityOfGoalGivenLink(links[0], &goal2, false) == 0);
    assert(map8.probabilityOfGoalGivenLink(links[1], &goal2, false) == 0);
}
```

12.56 LinkToStateMapEntry.cpp

```
/*
```

```
 * LinkStateMapEntry.cpp
 *
 *  Created on: Mar 13, 2016
 *      Author: vagrant
 */

#include "LinkToStateMapEntry.h"

namespace PredictivePowertrain {

LinkToStateMapEntry::LinkToStateMapEntry() {
    this->entries = new GenericMap<long int, int>();
}

int LinkToStateMapEntry::addEntry(Link* li) {
      long int linkHash = li->getHash();
      if (this->entries->hasEntry(linkHash)){
        int incrementedM = this->entries->getEntry(linkHash) + 1;
        this->entries->updateEntry(linkHash, incrementedM);
         return incrementedM;
      } else {
        this->entries->addEntry(linkHash, 1);
         return 1;
      }
}

LinkToStateMapEntry::~LinkToStateMapEntry() {
    delete(this->entries);
}

LinkToStateMapEntry::LinkToStateMapEntry(LinkToStateMapEntry& other) {
      this->entries = other.entries;
}

int LinkToStateMapEntry::getM(Link* li) {
      long int linkHash = li->getHash();
      if (this->entries->hasEntry(linkHash)){
        return this->entries->getEntry(linkHash);
      } else {
        return 0;
      }
}

int LinkToStateMapEntry::getTotalM() {
      if (this->entries->getSize() == 0)
    {
        return 0;
      }
    else
    {
```

```
        int sum = 0;
        for (auto it = this->entries->begin(); it != this->entries->end(); ++it)
        {
                sum += it->second;
        }
        return sum;
      }
}

GenericMap<long int, int>* LinkToStateMapEntry::getEntries()
{
    return this->entries;
}

int LinkToStateMapEntry::totalM(LinkToStateMapEntry* linkToStateMapEntry)
{
    return linkToStateMapEntry->getTotalM();
}

} /* namespace PredictivePowertrain */
```

12.57 LinkToStateMapEntry.h

```
/*
 * LinkStateMapEntry.h
 *
 *  Created on: Mar 13, 2016
 *      Author: vagrant
 */

#ifndef ROUTE_PREDICTION_LINKTOSTATEMAPENTRY_H_
#define ROUTE_PREDICTION_LINKTOSTATEMAPENTRY_H_
#include "../map/GenericMap.h"
#include "../driver_prediction/Link.h"



namespace PredictivePowertrain {

class LinkToStateMapEntry {
private:
      GenericMap<long int,int>* entries;
public:
      LinkToStateMapEntry();
      LinkToStateMapEntry(LinkToStateMapEntry &other);
      int addEntry(Link* li);
      int getM(Link* li);
      int getTotalM();
      virtual ~LinkToStateMapEntry();
      static int totalM(LinkToStateMapEntry* linkToStateMapEntry);
    GenericMap<long int, int>* getEntries();
```

```
};

} /* namespace PredictivePowertrain */

#endif /* ROUTE_PREDICTION_LINKTOSTATEMAPENTRY_H_ */
```

12.58 LinkToStateMapEntryUnitTest.cpp

```
/*
 * LinkToStateMapEntryUnitTest.cpp
 *
 *  Created on: Apr 13, 2016
 *      Author: vagrant
 */

#include "../route_prediction/LinkToStateMapEntry.h"
#include <iostream>
#include <assert.h>
#include "UnitTests.h"

using namespace std;
using namespace PredictivePowertrain;


void linkToStateMapEntry_ut() {

	//Testing constructors, getTotalM(), totalM()
	LinkToStateMapEntry first;
	LinkToStateMapEntry second(first);

	assert(first.getTotalM()==0 && second.getTotalM() == 0);
	assert(first.totalM(&second) == 0);

	//Testing addEntry()
	Link link1;
	Link link2(10,20);
	first.addEntry(&link1);
	assert(first.getM(&link1) == 1);
	assert(first.getTotalM() == 1);
	assert(second.getM(&link1) == 0);

	first.addEntry(&link2);
	assert(first.getTotalM() == 2);
	assert(second.getTotalM() == 0);
	assert(second.totalM(&first) == 2);

	//updating an entry, increasing m
	first.addEntry(&link1);
	assert(first.getM(&link1) == 2);
	assert(first.getTotalM() == 3);
	assert(second.totalM(&first) == 3);
```

```
}
```

12.59 RoutePrediction.cpp

```
/*
 * RoutePrediction.cpp
 *
 *  Created on: Mar 12, 2016
 *      Author: vagrant
 */

#include "RoutePrediction.h"

namespace PredictivePowertrain {

RoutePrediction::RoutePrediction()
{
    this->city = new City();
    initialize();
}

RoutePrediction::RoutePrediction(City* city)
{
     this->city = city;
    initialize();
}

void RoutePrediction::initialize()
{
    Link* unknownLink = new Link(-2, -2);
    Link* overLink = new Link(-3,-3);
    Goal* unknownGoal = new Goal(-1);
    Goal* overGoal = new Goal(1);

    this->unknownRoute = new Route(new GenericMap<long int, Link*>(), unknownGoal);
    this->unknownRoute->addLink(unknownLink);

    this->overRoute = new Route(new GenericMap<long int, Link*>(), overGoal);
    this->overRoute->addLink(overLink);

    this->linkToState = new LinkToStateMap();
    this->goalToLink = new GoalToLinkMap();
    this->links = new GenericMap<long int, Link*>();
    this->goals = new GenericMap<long int, Goal*>();
    this->states = new GenericMap<int, std::pair<Link*, Goal*>*>();
    this->predictedRoute = new Route();
    this->currentRoute = new Route();
    this->predictedGoal = unknownGoal;

    if(this->city->getInstersectionMapSize() != 0)
```

```
    {
        this->minInitialProbability = 0.1 / this->city->getInstersectionMapSize();
    } else {
        this->minInitialProbability = 0;
    }
}

RoutePrediction::~RoutePrediction()
{
    delete(this->unknownRoute);
    delete(this->overRoute);
    delete(this->linkToState);
    delete(this->goalToLink);
    delete(this->links);
    delete(this->goals);
    delete(this->states);
    delete(this->predictedRoute);
    delete(this->currentRoute);
    delete(this->predictedGoal);
}

GenericMap<long int, Link*>* RoutePrediction::getLinks()
{
    return this->links;
}

GenericMap<long int, Goal*>* RoutePrediction::getGoals()
{
    return this->goals;
}

LinkToStateMap* RoutePrediction::getLinkToState()
{
    return this->linkToState;
}

GoalToLinkMap* RoutePrediction::getGoalToLink()
{
    return this->goalToLink;
}

Route* RoutePrediction::startPrediction(Link* linkTaken, Intersection* currentIntersection, std::vector<float>* currentCondition)
{
    delete(this->predictedGoal);
      this->predictedGoal = new Goal(1, currentCondition);

      GenericMap<long int, Link*>* nextLinks = currentIntersection->getOutgoingLinks(linkTaken);
    this->probabilities = new std::vector<float>(nextLinks->getSize() * this->goals->getSize());

      // creating the probability of each goal based on its relation to the conditions
```

```
  int counter = 0;
  this->goals->initializeCounter();
  GenericEntry<long int, Goal*>* nextGoal = this->goals->nextEntry();
  while(nextGoal != NULL)
  {
    nextLinks->initializeCounter();
    GenericEntry<long int, Link*>* nextLink = nextLinks->nextEntry();
    while(nextLink != NULL)
    {
         // get goal probability
            float goalProbability = this->goalToLink->probabilityOfGoalGivenLink(nextLink->value, nextGoal->value, false);

         // to prevent link jitter
         Intersection* nextLinkInt = this->city->getIntersectionFromLink(nextLink->value, true);
         bool linkJitter = nextLinkInt->getIntersectionID() == currentIntersection->getIntersectionID();

            if(this->predictedGoal->isSimilar(nextGoal->value) && !linkJitter)
            {
                    // high probability since condition is right
                    goalProbability *= (float)nextGoal->value->getNumSeen();
            }
         else
         {
                    // lower probability since condition is wrong
                    goalProbability *= .1 * (float)nextGoal->value->getNumSeen();
            }

            std::pair<Link*, Goal*>* newState = new std::pair<Link*, Goal*>(nextLink->value, nextGoal->value);
            this->states->addEntry(counter, newState);
         this->probabilities->at(counter) = std::max(this->minInitialProbability / ((float)this->goals->getSize()), goalProbability);
            counter++;

            nextLink = nextLinks->nextEntry();
    }
    delete(nextLink);
    nextGoal = this->goals->nextEntry();
  }
  delete(nextGoal);

  // normalize probabilities
  float sum = 0;
  for(int i = 0; i < this->probabilities->size(); i++) { sum += this->probabilities->at(i); }
if(sum != 0)
{
    for(int i = 0; i < this->probabilities->size(); i++) { this->probabilities->at(i) /= sum; }
}

  this->currentRoute->setToIntersection(currentIntersection);

std::vector<float>* probabilitiesCopy = new std::vector<float>(this->probabilities->size());
```

```
    for(int i = 0; i < this->probabilities->size(); i++) { probabilitiesCopy->at(i) = this->probabilities->at(i); }

    this->predictedRoute = predictPrivate(NULL, this->states->copy(), probabilitiesCopy);
    if(this->predictedRoute->isEqual(this->unknownRoute))
    {
        return this->unknownRoute;
    }
    else if(this->predictedRoute->isEqual(this->overRoute))
    {
        return this->overRoute;
    }
      return createRouteConditions(currentCondition);
}

Route* RoutePrediction::predict(Link* linkTaken)
{
      GenericMap<long int, Link*>* legalLinks;
      if(this->currentRoute->isIntersection())
      {
        legalLinks = this->currentRoute->getIntersection()->getOutgoingLinks(linkTaken);
        this->currentRoute = new Route();
      }
    else
    {
       assert(!linkTaken->isFinalLink());
       legalLinks = this->city->getNextLinks(linkTaken);
      }

      // make sure that the link given is legal
      bool error = true;
      legalLinks->initializeCounter();
      GenericEntry<long int, Link*>* nextLegalLink = legalLinks->nextEntry();
      while(nextLegalLink != NULL)
      {
        if(nextLegalLink->value->isEqual(linkTaken))
        {
                error = false;
                break;
        }
         nextLegalLink = legalLinks->nextEntry();
      }
    delete(nextLegalLink);

      // there is an error and the current route is known (ie, wasnt an error before)
    // if not not legal, stop and return an error value
      if(error && this->predictedRoute->isEmpty())
      {
        this->predictedRoute = this->unknownRoute;
        return this->unknownRoute;
      }
```

```
    else if(linkTaken->isFinalLink())
    {
       this->predictedRoute = this->overRoute;
       return this->overRoute;
      }

      std::pair<GenericMap<int,  std::pair<Link*,Goal*>*>*,  std::vector<float>*>*  update  =  updateStates(linkTaken,  this->states,  this-
>probabilities);
    this->states = update->first;
    this->probabilities = update->second;

      this->currentRoute->addLink(linkTaken);

      if(linkTaken->isEqual(this->predictedRoute->getEntry(0)))
      {
        this->predictedRoute->removeFirstLink();
      }
    else if(this->predictedRoute->getLinks()->getSize() >= 2 && linkTaken->isEqual(this->predictedRoute->getEntry(1)))
    {
        this->predictedRoute->removeFirstLink();
        this->predictedRoute->removeFirstLink();
    }
    else
    {
        std::vector<float>* probabilitiesCopy = new std::vector<float>(this->probabilities->size());
        for(int i = 0; i < this->probabilities->size(); i++) { probabilitiesCopy->at(i) = this->probabilities->at(i); }

       this->predictedRoute = predictPrivate(NULL, this->states->copy(), probabilitiesCopy);
      }

      if(!this->predictedRoute->isEmpty())
      {
        return this->predictedRoute;
    }
    else if(this->predictedRoute->isEqual(this->unknownRoute))
    {
        return this->unknownRoute;
    }
    else if(this->predictedRoute->isEqual(this->overRoute))
    {
        return this->overRoute;
    }
    else if (this->predictedRoute->getLinkSize() == 1)
    {
       return createRouteIntersection(this->city->getIntersectionFromLink(linkTaken, true), this->predictedGoal->getBins());
      }
    else
    {
       return createRoute();
      }
```

```
}

std::pair<GenericMap<int, std::pair<Link*,Goal*>*>*, std::vector<float>*>*
RoutePrediction::updateStates(Link* chosenLink, GenericMap<int, std::pair<Link*,Goal*>*>* oldStates, std::vector<float>* oldProbabilites)
{
      // get next links
      GenericMap<long int, Link*>* nextLinks = this->city->getNextLinks(chosenLink);

      // generate new data structures
    GenericMap<int, std::pair<Link*, Goal*>*>* newStates = new GenericMap<int, std::pair<Link*, Goal*>*>();
    std::vector<float>* newProbabilities = new std::vector<float>(nextLinks->getSize() * this->goals->getSize());
    int counter = 0;

      // generate reused fields
      Link* li;
      Goal* gi;
      Goal* gj;
      float pSi, pGl, pLs, minProbability;
    std::pair<Link*, Goal*>* sj;

      // calculate new states and probabilities
      nextLinks->initializeCounter();
      GenericEntry<long int, Link*>* nextLink = nextLinks->nextEntry();
      while(nextLink != NULL)
      {
        this->goals->initializeCounter();
        GenericEntry<long int, Goal*>* nextGoal = this->goals->nextEntry();
         li = nextLink->value;
        while(nextGoal != NULL)
        {
             gi = nextGoal->value;
             std::pair<Link*, Goal*>* si = new std::pair<Link*, Goal*>(li, gi);
                pSi = 0;

             // for link jitter
             bool linkJitter = false;
             if(!chosenLink->isFinalLink() && !li->isFinalLink())
             {
                 Intersection* chosenLinkInt = this->city->getIntersectionFromLink(chosenLink, true);
                 Intersection* nextLinkInt = this->city->getIntersectionFromLink(li, true);

                 linkJitter = chosenLinkInt->getIntersectionID() == nextLinkInt->getIntersectionID();
             }

             if(!linkJitter)
             {
                 for(int j = 0; j < oldStates->getSize(); j++)
                 {
                     sj = oldStates->getEntry(j);
                     if(sj->first->isEqual(chosenLink))
```

```
                    {
                        gj = sj->second;
                        minProbability = this->minInitialProbability / ((float)this->goals->getSize());
                        pGl = this->goalToLink->probabilityOfGoalGivenLink(li, gi, 0);
                        pLs = this->linkToState->getProbability(li, chosenLink, gj, false);
                        pSi  += std::max(minProbability, oldProbabilites->at(j)) * std::max(minProbability, pLs) * std::max(minProbability,
pGl);
                    }
                }
            }

            // add new state to new states
               newStates->addEntry(counter, si);
               newProbabilities->at(counter) = pSi;
            counter++;

               nextGoal = this->goals->nextEntry();
        }
        nextLink = nextLinks->nextEntry();
      }

      // normalize probabilities
      float sum = 0;
      for(int i = 0; i < newProbabilities->size(); i++) { sum += newProbabilities->at(i); }
    if(sum != 0)
    {
        for(int i = 0; i < newProbabilities->size(); i++) { newProbabilities->at(i) /= sum; }
    }

      // update probabilities and states
    std::pair<GenericMap<int, std::pair<Link*,Goal*>*>*, std::vector<float>*>* update =
        new std::pair<GenericMap<int, std::pair<Link*,Goal*>*>*, std::vector<float>*>(newStates, newProbabilities);

    return update;
}

Route* RoutePrediction::predictPrivate(Route* currentRoute, GenericMap<int, std::pair<Link*,Goal*>*>* currentStates, std::vector<float>*
currentProbabilities)
{
      // find max prob
    float maxProbability = -1;
    int nextStateIndex = 0;

      for(int i = 0; i < currentProbabilities->size(); i++)
    {
        if(currentProbabilities->at(i) > maxProbability)
        {
            maxProbability = currentProbabilities->at(i);
            nextStateIndex = i;
        }
```

```
      }

    // return unknown route if no probability associated with next goal in route
    if(maxProbability <= 0 || currentStates->getSize() == 0)
    {
        return this->unknownRoute;
    }

      // get state with highest probability and add it to route
    std::pair<Link*, Goal*>* nextState = currentStates->getEntry(nextStateIndex);
    if(currentRoute == NULL)
    {
        currentRoute = new Route();
    }
    currentRoute->addLink(nextState->first);

      if(!nextState->first->isFinalLink())
      {
        std::pair<GenericMap<int,  std::pair<Link*,Goal*>*>*,  std::vector<float>*>*  update  =  updateStates(nextState->first,  currentStates,
currentProbabilities);

        delete(currentStates);
        delete(currentProbabilities);

        return predictPrivate(currentRoute, update->first, update->second);
      }
      return currentRoute;
}

Route* RoutePrediction::createRoute()
{
      return createRouteConditions(this->predictedGoal->getBins());
}

Route* RoutePrediction::createRouteConditions(std::vector<float>* currentConditions)
{
    int lastLinkIndex = std::max(this->predictedRoute->getLinkSize() - 2, 0);
      return         createRouteIntersection(this->city->getIntersectionFromLink(this->predictedRoute->getEntry(lastLinkIndex),         true),
currentConditions);
}

Route* RoutePrediction::createRouteIntersection(Intersection* intersection, std::vector<float>* currentConditions)
{
    // delete old goal
      delete(this->predictedGoal);

    // add new goal
      this->predictedGoal = new Goal(intersection->getIntersectionID(), currentConditions);

    // assign goal
```

```
    this->predictedRoute->assignGoal(this->predictedGoal);

    // return new route
     Route* route = new Route(this->predictedRoute->getLinks(), this->predictedGoal);
     return route;
}

void RoutePrediction::parseRoute(Route* route)
{
      // get hash of route goal and add it to goals if nonexistent
      long int goalHash = route->getGoalHash();
      if(this->goals->hasEntry(goalHash))
      {
        this->goals->getEntry(goalHash)->incrementNumSeen();
      } else {
        route->getGoal()->setNumSeen(1);
        this->goals->addEntry(goalHash, route->getGoal());
      }

      // add links to link map
      for(int i = 0; i < route->getLinkSize(); i++)
      {
        Link* li; Link* lj;
        lj = route->getLinks()->getEntry(i);

        if(i == route->getLinkSize() - 1)
        {
               li = this->link.finalLink();
        } else {
               li = route->getLinks()->getEntry(i+1);
        }

        this->linkToState->incrementTransition(lj, route->getGoal(), li);
        this->goalToLink->linkTraversed(lj, route->getGoal());
        if(this->links->hasEntry(lj->getHash()))
        {
            this->links->updateEntry(lj->getHash(), lj);
        }
        else
        {
            this->links->addEntry(lj->getHash(), lj);
        }
      }
}

void RoutePrediction::addPredictionElements(GenericMap<long int, Link*>* newLinks,
                                            GenericMap<long int, Goal*>* newGoals,
                                            GoalToLinkMap* newGoalToLink,
                                            LinkToStateMap* newLinkToState)
{
```

```
    delete(this->links);
    this->links = newLinks;

    delete(this->goals);
    this->goals = newGoals;

    delete(this->goalToLink);
    this->goalToLink = newGoalToLink;

    delete(this->linkToState);
    this->linkToState = newLinkToState;
}
void RoutePrediction::addCity(City* newCity)
{
    if(this->city != NULL)
    {
        delete(this->city);
    }
    this->city = newCity;
}

Route* RoutePrediction::getPredictedRoute()
{
    return this->predictedRoute;
}

City* RoutePrediction::getCity()
{
    return this->city;
}

Route* RoutePrediction::getUnknownRoute()
{
    return this->unknownRoute;
}

Route* RoutePrediction::getOverRoute()
{
    return this->overRoute;
}

Route* RoutePrediction::getCurrentRoute()
{
    return this->currentRoute;
}

} /* namespace PredictivePowertrain */
```

12.60 RoutePrediction.h

```
/*
```

```
 * RoutePrediction.h
 *
 *  Created on: Mar 12, 2016
 *      Author: vagrant
 */

#ifndef ROUTE_PREDICTION_ROUTEPREDICTION_H_
#define ROUTE_PREDICTION_ROUTEPREDICTION_H_

#include "LinkToStateMap.h"
#include "GoalToLinkMap.h"
#include "Goal.h"
#include "Route.h"
#include "Probability.h"

#include "../map/GenericMap.h"
#include "../map/GenericEntry.h"

#include "../driver_prediction/Link.h"

#include "../city/City.h"
#include "../city/Intersection.h"

#include <algorithm>
#include <assert.h>
#include "stdlib.h"
#include <vector>

namespace PredictivePowertrain {

class RoutePrediction {
private:
      LinkToStateMap* linkToState;
      GoalToLinkMap* goalToLink;
      GenericMap<long int, Link*>* links;
      GenericMap<long int, Goal*>* goals;
     GenericMap<int, std::pair<Link*,Goal*>*>* states;
      Route* predictedRoute;
      Route* currentRoute;
     std::vector<float>* probabilities;
      City *city;
      Goal *predictedGoal;
      float minInitialProbability;
      Route* unknownRoute;
      Route* overRoute;
      Link link;

      std::pair<GenericMap<int,    std::pair<Link*,Goal*>*>*,    std::vector<float>*>*    updateStates(Link*    chosenLink,    GenericMap<int,
std::pair<Link*,Goal*>*>* oldStates, std::vector<float>* oldProbabilites);
      Route*    predictPrivate(Route*    currentRoute,    GenericMap<int,    std::pair<Link*,Goal*>*>*    currentStates,    std::vector<float>*
```

```
currentProbabilities);
      Route* createRoute();
    Route* createRouteConditions(std::vector<float>* currentCondition);
    Route* createRouteIntersection(Intersection* intersection, std::vector<float>* currentCondition);
    void initialize();

public:
      RoutePrediction();
      RoutePrediction(City* city);
    virtual ~RoutePrediction();
    Route* startPrediction(Link* linkTaken, Intersection* currentIntersection, std::vector<float>* currentCondition);
      Route* predict(Link* linkTaken);
      void parseRoute(Route* route);
    GenericMap<long int, Link*>* getLinks();
    GenericMap<long int, Goal*>* getGoals();
    LinkToStateMap* getLinkToState();
    GoalToLinkMap* getGoalToLink();
    void addPredictionElements(GenericMap<long int, Link*>* newLinks,
                               GenericMap<long int, Goal*>* newGoals,
                               GoalToLinkMap* newGoalToLink,
                               LinkToStateMap* newLinkToState);
    void addCity(City* newCity);
    Route* getPredictedRoute();
    City* getCity();
    Route* getUnknownRoute();
    Route* getOverRoute();
    Route* getCurrentRoute();
};


} /* namespace PredictivePowertrain */

#endif /* ROUTE_PREDICTION_ROUTEPREDICTION_H_ */
```

12.61 RoutePredictionUnitTest.cpp

```
/*
 * RoutePredictionUnitTest.cpp
 *
 *  Created on: Apr 4, 2016
 *      Author: vagrant
 */

#include "../driver_prediction/Link.h"
#include "../city/City.h"
#include "../city/Intersection.h"
#include "../route_prediction/Goal.h"
#include "../data_management/DataManagement.h"
#include "../route_prediction/RoutePrediction.h"
#include "../route_prediction/Route.h"
```

```
#include <math.h>
#include <iostream>
#include <assert.h>
#include <climits>
#include <algorithm>
#include <ctime>
#include <vector>

using namespace PredictivePowertrain;

Link* makeLinks(int numberOfLinks)
{
      Link* links = new Link[numberOfLinks];
      for(int i = 0; i < numberOfLinks; i++)
      {
        Link newLink(rand() > .5, i + 1);
        links[i] = newLink;
      }
    return links;
}

RoutePrediction* routePrediction_ut()
{
    // seed random generator
    std::srand(std::time(0));

    // create a square city grid to predict over
    const int gridSideDim = 4;
    const int numInters = gridSideDim * gridSideDim;
    Intersection inters[numInters];

    GenericMap<long int, Road*>* roads = new GenericMap<long int, Road*>();
    GenericMap<long int, Intersection*>* intersections = new GenericMap<long int, Intersection*>();

    // fence post addage of the first intersection
    inters[0].assignID(1);
    intersections->addEntry(inters[0].getIntersectionID(), &inters[0]);

    // make all grid connections
    long int horzCount = 0;
    long int vertCount = (gridSideDim - 1) * gridSideDim;
    for(int i = 1; i < numInters; i++)
    {
        int interID = i + 1;

        inters[i].assignID(interID);

        // make horizontal connections
        if(i % gridSideDim != 0)
        {
```

```
        horzCount++;

        Road* newHorzRoad = new Road();
        newHorzRoad->assignID(horzCount);

        inters[i].addRoad(newHorzRoad, 1);
        inters[i-1].addRoad(newHorzRoad, 0);

        roads->addEntry(horzCount, newHorzRoad);
    }

    // make vertical conections
    if(interID > gridSideDim)
    {
        vertCount++;

        Road* newVertRoad = new Road();
        newVertRoad->assignID(vertCount);

        inters[i].addRoad(newVertRoad, 1);
        inters[i - gridSideDim].addRoad(newVertRoad, 0);

        roads->addEntry(vertCount, newVertRoad);
    }

    intersections->addEntry(inters[i].getIntersectionID(), &inters[i]);
}

// build city
City* city = new City(intersections, roads, new GenericMap<int, Bounds*>());
Link* links = makeLinks(5);
std::vector<float>* conditions = new std::vector<float>(2);
conditions->at(0) = 1;
conditions->at(1) = 2;
Goal goal(1, conditions);

// add routes to route prediction
RoutePrediction* rp = new RoutePrediction(city);
Link link = links[0];
int routeLength = 10;
Intersection* startIntersection = city->getIntersectionFromLink(&link, true);

// generate random actual route
Route startRoute;
startRoute.addLink(&link);
startRoute.assignGoal(&goal);
Route* actualRoute = city->randomPath(startIntersection, &startRoute, routeLength, conditions);

// add trainging iterations here (simulates driving over the route multiple times)
rp->parseRoute(actualRoute);
```

```
//    rp->parseRoute(actualRoute);
//    rp->parseRoute(actualRoute);
//    rp->parseRoute(actualRoute);
//    rp->parseRoute(actualRoute);
//    rp->parseRoute(actualRoute);


    // create number of random routes to include in test set
    int num_rand_routes = 4;

    for(int i = 1; i <= num_rand_routes; i++)
    {
        // create random route
        Route* randomRoute = city->randomPath(startIntersection, &startRoute, std::ceil((float)std::rand() / RAND_MAX * routeLength),
conditions);
        while(randomRoute->isEqual(actualRoute))
        {
            randomRoute = city->randomPath(startIntersection, &startRoute, std::ceil((float)std::rand() / RAND_MAX * routeLength),
conditions);
        }

        // add random route to test set
        rp->parseRoute(randomRoute);
    }

    // predict actual route as it is 'driven' over

    int traversalNum = 14;
    int totPredIters = 0;

    for(int i = 1; i <= traversalNum; i++)
    {
        std::cout << "++++++++++++++++++++++++++++++++++++++ Route Traversal: " << i << " ++++++++++++++++++++++++++++++++++++++" <<
std::endl;

        int predIter = 1;
        Route* actualRouteCopy = actualRoute->copy();
        Route* predRoute = rp->startPrediction(actualRouteCopy->getLinks()->getEntry(0), startIntersection, conditions);
        while(actualRouteCopy->getLinkSize() > 2)
        {
            std::cout << "--- route prediction iteration " << predIter << " ---" << std::endl;

            // update actual route as it's 'driven' over
            actualRouteCopy->removeFirstLink();

            // print actual route
            actualRouteCopy->printLinks();

            // print predicted route
            predRoute->printLinks();
```

```
            // check if predicted route is actual route
            if(actualRouteCopy->isEqual(predRoute))
            {
                std::cout << "predicted routed before reaching end destination!" << std::endl;
                std::cout << "Links traversed: " << predIter << std::endl;
                std::cout << "Links in route: " << actualRouteCopy->getLinkSize() << std::endl;
                break;
            }
            else
            {
                totPredIters++;
            }

            // predict route
            predRoute = rp->predict(actualRouteCopy->getLinks()->getEntry(0));

            predIter++;

        }

        rp->parseRoute(actualRoute);
    }

    std::cout << "prediction accuracy: " << 1.0 - (float)totPredIters / ((float)traversalNum * (actualRoute->getLinks()->getSize() - 1)) <<
std::endl;

    return rp;

}
```

12.62 Goal.cpp

```
/*
 * Goal.cpp
 *
 *  Created on: Feb 23, 2016
 *      Author: vagrant
 */

#include "Goal.h"

namespace PredictivePowertrain {

Goal::Goal(long int destination)
{
      this->destination = destination;
    this->numSeen = 1;
    this->bins = new std::vector<float>(1);
    this->bins->at(0) = -1;
}
```

```
Goal::Goal(long int destination, std::vector<float>* bins)
{
      this->destination = destination;
      this->bins = bins;
      this->numSeen = 1;
}

Goal::Goal(Goal * other)
{
    if(other != NULL)
    {
        this->destination = other->destination;
        if(other->getBins() != NULL && other->getBins()->size() > 0)
        {
            this->bins = new std::vector<float>(other->getBins()->size());
            for (int i = 0; i < this->bins->size(); i++)
            {
                this->bins->at(i) = other->bins->at(i);
            }
        }
        else
        {
            this->bins = new std::vector<float>();
            this->bins = new std::vector<float>(1);
            this->bins->at(0) = -1;
        }

        this->numSeen = other->numSeen;
    }
}

bool Goal::isSimilar(Goal * other)
{
      if (this->bins->size() == other->getBins()->size())
    {
        for (int i = 0; i < this->bins->size(); i++)
        {
               if (this->bins->at(i) != other->bins->at(i))
            {
                       return false;
                }
        }
        return true;
      } else {
        return false;
      }
}

bool Goal::isEqual(Goal * other)
```

```
{
      return isSimilar(other) && this->destination == other->destination;
}

long int Goal::getHash() const
{
      long int hash = this->destination;
      int bits;
      for (int i = 0; i < this->bins->size(); i++) {
        bits = (int) std::log2(this->bins->at(i));
        hash = hash << bits;
        hash += this->bins->at(i);
      }
      return hash;
}

Goal::Goal() {
}

void Goal::incrementNumSeen()
{
      this->numSeen++;
}

long int Goal::getDestination()
{
    return this->destination;
}

void Goal::setNumSeen(int numSeen)
{
      this->numSeen = numSeen;
}

std::vector<float>* Goal::getBins()
{
      return this->bins;
}

Goal::~Goal() {
}

int Goal::getNumSeen()
{
      return this->numSeen;
}

} /* namespace PredictivePowertrain */
```

12.63 Goal.h

```
/*
 * Goal.h
 *
 *  Created on: Feb 23, 2016
 *      Author: vagrant
 */

#ifndef DRIVER_PREDICTION_GOAL_H_
#define DRIVER_PREDICTION_GOAL_H_

#include <functional>
#include <cmath>
#include <vector>

namespace PredictivePowertrain {

class Goal {
private:
      long int destination;
    std::vector<float>* bins;
      int numSeen;
public:
      Goal();
      Goal(long int destination);
    Goal(long int destination, std::vector<float>* bins);
      Goal(Goal * other);
      bool isSimilar(Goal * other);
      bool isEqual(Goal * other);
      long int getHash() const;
      int getNumSeen();
      void setNumSeen(int numSeen);
      void incrementNumSeen();
    std::vector<float>* getBins();
      virtual ~Goal();
    long int getDestination();
};

} /* namespace PredictivePowertrain */

#endif /* DRIVER_PREDICTION_GOAL_H_ */
```

12.64 GoalUnitTest.cpp

```
/*
 * GoalUnitTest.cpp
 *
 *  Created on: Apr 14, 2016
 *      Author: vagrant
 */
#include "../route_prediction/Goal.h"
```

```
#include <iostream>
#include <assert.h>
#include <vector>
#include "UnitTests.h"

using namespace PredictivePowertrain;

void goal_ut(){
      // testing constructors (int destination, int [] bin, int size)
      Goal goalOne;
      Goal goalTwo(1);
     std::vector<float> arr = {0};
      Goal goalThree(1, &arr);
      Goal goalFour(2, &arr);
      Goal goalFive(goalFour);
      Goal goalSix(3, &arr);

      // test isSimilar (doesn't compare destination)
      assert(goalFour.isSimilar(&goalSix));  // true
      assert(!goalFour.isSimilar(&goalOne)); // !false
      // test isEquals (compares all attributes)
      assert(goalFour.isEqual(&goalFive));           // true
      assert(!goalFour.isEqual(&goalSix));           // !false


      // get,set & increment num seen
      goalOne.setNumSeen(1);
      int oldNum = goalOne.getNumSeen();
      goalOne.incrementNumSeen();
      assert(oldNum + 1 == goalOne.getNumSeen());

      // getBins()
     std::vector<float>* bins = goalThree.getBins();
      assert(arr.at(1) == bins->at(1));
}
```

12.65 GoalMapEntry.h

```
/*
 * GoalMapEntry.h
 *
 *  Created on: Mar 13, 2016
 *      Author: vagrant
 */

#ifndef ROUTE_PREDICTION_GOALMAPENTRY_H_
#define ROUTE_PREDICTION_GOALMAPENTRY_H_
#include "../map/GenericMap.h"
#include "LinkToStateMapEntry.h"
#include "Goal.h"
```

```
namespace PredictivePowertrain {

template<typename K, typename V>
class GoalMapEntry {
private:
      GenericMap<K, V>* map;
      Goal* goal;
      int m;
    void initialize();

public:
      GoalMapEntry();
      GoalMapEntry(Goal* goal);
      void incrementCount();
      virtual ~GoalMapEntry();
      Goal* getGoal();
      V getMapEntry(K key);
    void addMapEntry(K key, V val);
      int getM();
    GenericMap<K,V>* getMap();
};

template<class K, class V>
GoalMapEntry<K, V>::GoalMapEntry()
{
    this->goal = new Goal();
    this->initialize();
}

template<class K, class V>
GoalMapEntry<K, V>::GoalMapEntry(Goal* goal)
{
    this->goal = goal;
    this->initialize();
}

template<class K, class V>
void GoalMapEntry<K, V>::initialize()
{
    this->m = 0;
    this->map = new GenericMap<K, V>();
}

template<class K, class V>
GoalMapEntry<K, V>::~GoalMapEntry()
{
    delete(this->goal);
    delete(this->map);
}
```

```
template<class K, class V>
void GoalMapEntry<K, V>::addMapEntry(K key, V value)
{
    this->map->addEntry(key, value);
}

template<class K, class V>
void GoalMapEntry<K, V>::incrementCount()
{
    this->m++;
}

template<class K, class V>
Goal* GoalMapEntry<K, V>::getGoal()
{
    return this->goal;
}

template<class K, class V>
V GoalMapEntry<K, V>::getMapEntry(K key)
{
    return this->map->getEntry(key);
}

template<class K, class V>
int GoalMapEntry<K, V>::getM()
{
    return this->m;
}

template<class K, class V>
GenericMap<K, V>* GoalMapEntry<K, V>::getMap()
{
    return this->map;
}


} /* namespace PredictivePowertrain */

#endif /* ROUTE_PREDICTION_GOALMAPENTRY_H_ */
```

12.66 GoalMapEntryUnitTest.cpp

```
/*
 * GoalMapEntryUnitTest.cpp
 *
 *  Created on: Apr 11, 2016
 *      Author: vagrant
 */
#include "../route_prediction/GoalMapEntry.h"
#include <iostream>
```

```
#include <assert.h>
#include "UnitTests.h"

//using namespace std;
using namespace PredictivePowertrain;

void goalmapentry_ut(){
      GoalMapEntry<int, int> testGoal;
      // test increment method
      testGoal.incrementCount();
      assert(testGoal.getM() == 1);

      //test goal constructor
      Goal test(1);
      GoalMapEntry<int, int> testGoal2(&test);
      assert(testGoal.getGoal() == &test);
}
```

12.67 SpeedPrediction.cpp

```
/*
 * MakeOSM.cpp
 *
 *  Created on: Apr 4, 2016
 *      Author: vagrant
 */

#include "SpeedPrediction.h"

namespace PredictivePowertrain {

SpeedPrediction::SpeedPrediction()
{
      initParams();

      // initialize weights of NN
    this->Wts = new std::vector<Eigen::MatrixXd*>(this->lastLayer + 1);
      for(int i = 0; i < this->lastLayer + 1; i++)
      {
        int L1 = this->totalLayers[i];
        int L2 = this->totalLayers[i+1];

        this->Wts->at(i) = new Eigen::MatrixXd(L2,L1+1);

        for(int j = 0; j <= L1; j++)
        {
                for(int k = 0; k < L2; k++)
                {
                 this->Wts->at(i)->coeffRef(k, j) = -.01 + .02 * (double) std::rand() / RAND_MAX;
                }
        }
```

```
      }

      // initialize hidden layer outputs
    this->yHid = new std::vector<Eigen::MatrixXd*>(this->lastLayer);
      for(int i = 0; i < this->lastLayer; i++)
      {
        this->yHid->at(i) = new Eigen::MatrixXd(1, this->HN[i] + 1);
      }

      // initialize hidden layer inputs
    this->yInHid = new std::vector<Eigen::MatrixXd*>(this->lastLayer + 1);
      for(int i = 0; i < this->lastLayer+1; i++)
      {
        this->yInHid->at(i) = new Eigen::MatrixXd(1,this->totalLayers[i+1]);
      }
}

// create NN using inputted weights and activations
SpeedPrediction::SpeedPrediction(std::vector<Eigen::MatrixXd*>*  Wts,  std::vector<Eigen::MatrixXd*>*  yHid,  std::vector<Eigen::MatrixXd*>*
yInHid)
{

    // must assert NN geometry matches geometr of input
      initParams();

      this->Wts = Wts;
      this->yHid = yHid;
      this->yInHid = yInHid;
}

void SpeedPrediction::initParams()
{
      // NN architectural params
      this->I = 20;                                         // input neurons
      this->O = 60;                                         // output neurons
      int HN[] = {80, 65, 50, 35, 20};       // hidden layer neurons

      // scaling parameters
      this->alpha = 10.0;                                   // learning rate
      this->maxSpeed = 200;                          // max vehicle speed
      this->lb_offset = .3;                          // lower bound offset
    this->ds = 5.0;

      // index vars
      this->HL = sizeof(HN)/4-1;                            // last hidden layer
      this->lastLayer = this->HL+1;          // last of all layers

      // init hidden layers
      this->HN = new int[this->HL+1];
      for(int i = 0; i < this->HL+1; i++)
```

```
      {
        this->HN[i] = HN[i];
      }

      // init total layers
      this->totalLayers = new int[this->HL+3];
      this->totalLayers[0] = I;
      for(int i = 1; i <= this->HL+1; i++)
      {
        this->totalLayers[i] = this->HN[i-1];
      }
      this->totalLayers[this->lastLayer+1] = this->O;

    // seed random gen
    std::srand(std::time(0));
}

int SpeedPrediction::getNumLayers()
{
    return this->lastLayer+2;
}

// feed-forward prediction
// assumes spd_in is historical data
void SpeedPrediction::predict(Eigen::MatrixXd * spd_in, Eigen::MatrixXd * spd_out)
{
      // fill input buffer with speed values
      Eigen::MatrixXd x = *spd_in;

      // **input to hidden layers
      for(int i = 0; i < this->HN[0]; i++)
      {
         Eigen::MatrixXd tempWts = this->Wts->at(0)->row(i);
        double act = (x*tempWts.transpose())(0);
         this->yInHid->at(0)->coeffRef(0,i) = act;
         this->yHid->at(0)->coeffRef(0,i) = 1 / (1 + std::exp(-act));
      }
    this->yHid->at(0)->coeffRef(0,this->HN[0]) = 1;

      // **through hidden layers
      for(int i = 1; i <= this->HL; i++)
      {
        for(int j = 0; j < this->HN[i]; j++)
        {
            Eigen::MatrixXd tempWts = this->Wts->at(i)->row(j);
               double act = ((*this->yHid->at(i-1))*tempWts.transpose())(0);
            this->yInHid->at(i)->coeffRef(0,j) = act;
            this->yHid->at(i)->coeffRef(0,j) = 1 / (1 + std::exp(-act));
        }
         this->yHid->at(i)->coeffRef(0,this->HN[i]) = 1;
```

```
	}

	// **hidden to output layers
	for(int i = 0; i < this->O; i++)
	{
	  Eigen::MatrixXd tempWts = this->Wts->at(this->lastLayer)->row(i);
	  double act = ((*this->yHid->at(this->lastLayer-1))*tempWts.transpose())(0);
	  this->yInHid->at(this->lastLayer)->coeffRef(0,i) = act;
	  (*spd_out)(0,i) = 1 / (1 + std::exp(-act));
	}
}

// back-propagation
void SpeedPrediction::train(Eigen::MatrixXd * spd_pred, Eigen::MatrixXd  * spd_act, Eigen::MatrixXd * spd_in)
{
	// delta matrix back propagated though all layers
	Eigen::MatrixXd abc = Eigen::MatrixXd::Zero(this->O,this->totalLayers[this->lastLayer]+1);
	Eigen::MatrixXd x = *spd_in;

	// **propagate output to last hidden layer
	for(int j = 0; j < this->O; j++)
	{
	  double outError = (*spd_act)(0,j) - (*spd_pred)(0,j);
	  double outDelta = outError*(*spd_pred)(0,j)*(1-(*spd_pred)(0,j));
	  Eigen::MatrixXd tempWts = this->Wts->at(this->lastLayer)->row(j);

	  abc.row(j) = tempWts*outDelta;
	  this->Wts->at(this->lastLayer)->row(j) = tempWts+this->alpha*(*this->yHid->at(this->lastLayer-1))*outDelta;
	}

	// **propagate output through all hidden layers
	int index = 0;
	int index2 = 1;
	for(int i = this->lastLayer-1; i > 0; i--)
	{
	  // create some dimension dependent locals
	  Eigen::MatrixXd yHidTemp = (*this->yHid->at(i));
	  Eigen::MatrixXd hidDelta = abc.transpose();
	  Eigen::MatrixXd ones = Eigen::MatrixXd::Ones(1,yHidTemp.cols());

	  // compute the hidden layer delta for each neuron
	  for(int j = 0; j < abc.rows(); j++)
	  {
		hidDelta.col(j) = abc.row(j).transpose()*(ones-yHidTemp)*yHidTemp.transpose();
	  }

	  // update weights on the hidden layer neurons and include input bias
	  Eigen::MatrixXd yHtemp = (*this->yHid->at(i-1));
	  Eigen::MatrixXd tempWts = (*this->Wts->at(i));
```

```
    // before updating weights, calculate the delta abc without bias term
    abc = Eigen::MatrixXd::Zero(this->O,this->totalLayers[i]+1);
    for(int j = 0; j < hidDelta.cols(); j++)
    {
            Eigen::MatrixXd tempHidDelta = hidDelta.col(j).head(this->HN[this->HL-index]);
            Eigen::MatrixXd abc_col = tempWts.transpose()*tempHidDelta;
            abc.row(j) = abc_col.transpose();
    }

    // continue with updation of weights
    Eigen::MatrixXd intermediateCalc = Eigen::MatrixXd::Zero(this->HN[this->HL-index2]+1,this->HN[this->HL-index]+1);
    for(int j = 0; j < hidDelta.cols(); j++)
    {
            intermediateCalc = intermediateCalc + this->alpha*yHtemp.transpose()*hidDelta.col(j).transpose();
    }

    Eigen::MatrixXd tempWts_T = tempWts.transpose();
    tempWts = tempWts_T + intermediateCalc.topLeftCorner(intermediateCalc.rows(),intermediateCalc.cols()-1);

    for(int j = 0; j < this->HN[this->HL-index]; j++)
    {
            this->Wts->at(i)->row(j) = tempWts.col(j);
    }

    index++;
    index2++;
  }

  // **propagate from hidden to input layer
Eigen::MatrixXd yHidTemp = (*this->yHid->at(this->HL-index));
  Eigen::MatrixXd hidDelta = abc.transpose();
  Eigen::MatrixXd ones = Eigen::MatrixXd::Ones(1,yHidTemp.cols());

  // compute hidden deltas
  for(int j = 0; j < abc.rows(); j++)
  {
    hidDelta.col(j) = abc.row(j).transpose()*(ones-yHidTemp)*yHidTemp.transpose();
  }

  // update weights on the hidden layer neurons
Eigen::MatrixXd tempWts = (*this->Wts->at(0));
  Eigen::MatrixXd yHtemp = Eigen::MatrixXd::Zero(1,this->I+1);
  yHtemp = x;
  yHtemp(0,this->I) = 1;

  // continue with the updation of weights
  Eigen::MatrixXd intermediateCalc = Eigen::MatrixXd::Zero(this->I+1,this->HN[0]+1);
  for(int j = 0; j < hidDelta.cols(); j++)
  {
    intermediateCalc = intermediateCalc + this->alpha*yHtemp.transpose()*hidDelta.col(j).transpose();
```

```
      }

      Eigen::MatrixXd tempWts_T = tempWts.transpose();
      tempWts = tempWts_T + intermediateCalc.topLeftCorner(intermediateCalc.rows(),intermediateCalc.cols()-1);

      for(int j = 0; j < this->HN[0]; j++)
      {
        this->Wts->at(0)->row(j) = tempWts.col(j);
      }
}

// send address of weights
std::vector<std::vector<Eigen::MatrixXd*>*>* SpeedPrediction::getVals()
{
    std::vector<std::vector<Eigen::MatrixXd*>*>* returnList = new std::vector<std::vector<Eigen::MatrixXd*>*>(3);

    // Wts
    std::vector<Eigen::MatrixXd*>* newWtsVec = new std::vector<Eigen::MatrixXd*>(this->Wts->size());
    for(int i = 0; i < this->Wts->size(); i++)
    {
        if(this->Wts->at(i) != NULL)
        {
            newWtsVec->at(i) = new Eigen::MatrixXd(*this->Wts->at(i));
        }
    }
      returnList->at(0) = newWtsVec;

    // yHid
    std::vector<Eigen::MatrixXd*>* newYHidVec = new std::vector<Eigen::MatrixXd*>(this->yHid->size());
    for(int i = 0; i < this->yHid->size(); i++)
    {
        if(this->yHid->at(i) != NULL)
        {
            newYHidVec->at(i) = new Eigen::MatrixXd(*this->yHid->at(i));
        }
    }
      returnList->at(1) = newYHidVec;

    // yInHid
    std::vector<Eigen::MatrixXd*>* newYInHidVec = new std::vector<Eigen::MatrixXd*>(this->yInHid->size());
    for(int i = 0; i < this->yInHid->size(); i++)
    {
        if(this->yInHid->at(i) != NULL)
        {
            newYInHidVec->at(i) = new Eigen::MatrixXd(*this->yInHid->at(i));
        }
    }
      returnList->at(2) = newYInHidVec;
      return returnList;
}
```

```
int SpeedPrediction::getI()
{
      return this->I;
}

int SpeedPrediction::getO()
{
      return this->O;
}

// scale input data and concatenate bias term
void SpeedPrediction::formatInData(Eigen::MatrixXd * input)
{
      assert((*input).cols() == this->I+1);
      Eigen::MatrixXd offset = Eigen::MatrixXd::Ones(1,this->I+1) * this->lb_offset;
      (*input) = (*input) / this->maxSpeed + offset;
      (*input)(0,this->I) = 1;
}

void SpeedPrediction::scaleTrainingSpeed(Eigen::MatrixXd * input)
{
    // only accept row vector
    assert((*input).rows() == 1);
    Eigen::MatrixXd offset = Eigen::MatrixXd::Ones(1,(*input).cols()) * this->lb_offset;
    (*input) = (*input) / this->maxSpeed + offset;
}

void SpeedPrediction::unscaleTrainingSpeed(Eigen::MatrixXd * output)
{
    // accept only row vector
    assert((*output).rows() == 1);
    Eigen::MatrixXd offset = Eigen::MatrixXd::Ones(1,(*output).cols()) * this->lb_offset;
    (*output) = ((*output) - offset) * this->maxSpeed;
}

// scale output data
void SpeedPrediction::formatOutData(Eigen::MatrixXd * output)
{
      assert((*output).cols() == this->O);
      Eigen::MatrixXd offset = Eigen::MatrixXd::Ones(1,this->O) * this->lb_offset;
      (*output) = ((*output) - offset) * this->maxSpeed;
}

void SpeedPrediction::printAll()
{
      // Weights
      std::cout << "Wts" << std::endl;
      for(int i = 0; i < this->lastLayer + 1; i++)
      {
```

```
      std::cout << (*this->Wts->at(i)) << std::endl << std::endl;
    }

    // yHid
    std::cout << "yHid" << std::endl;
    for(int i = 0; i < this->lastLayer; i++)
    {
      std::cout << (*this->yHid->at(i)) << std::endl << std::endl;
    }

    // yInHid
    std::cout << "yInHid" << std::endl;
    for(int i = 0; i < this->lastLayer+1; i++)
    {
      std::cout << (*this->yInHid->at(i)) << std::endl << std::endl;
    }

    // last layer
    std::cout << "lastLayer" << std::endl;
    std::cout << this->lastLayer << std::endl << std::endl;

    // last hidden layer
    std::cout << "last hidden layer" << std::endl;
    std::cout << this->HL << std::endl << std::endl;
}

void SpeedPrediction::setVals(std::vector<std::vector<Eigen::MatrixXd*>*>* vals)
{
    this->Wts = vals->at(0);
    this->yHid = vals->at(1);
    this->yInHid = vals->at(2);
}

float SpeedPrediction::getDS()
{
    return this->ds;
}

void SpeedPrediction::output2Input(Eigen::MatrixXd* spdIn, Eigen::MatrixXd* spdOut)
{
    if(spdIn->cols() > spdOut->cols())
    {
        // left shift spd In columns
        int deltaCols = spdIn->cols() - spdOut->cols();
        for(int i = 0; i < deltaCols; i++)
        {
            spdIn->coeffRef(0, i) = spdIn->coeffRef(0, i+deltaCols);
        }

        // add in output data
```

```
        for(int i = 0; i < spdOut->cols(); i++)
        {
            spdIn->coeffRef(0, i+deltaCols) = spdOut->coeffRef(0, i);
        }
    }

    else
    {
        for(int i = 0; i < spdIn->cols(); i++)
        {
            spdIn->coeffRef(0, spdIn->cols()-1-i) = spdOut->coeffRef(0, spdOut->cols()-1-i);
        }
    }
}

int SpeedPrediction::getMaxSpeed()
{
    return this->maxSpeed;
}

float SpeedPrediction::getSpeedOffset()
{
    return this->lb_offset;
}

}
```

12.68 SpeedPrediction.h

```
/*
 * speed_prediction.h
 *
 *  Created on: Mar 6, 2016
 *      Author: vagrant
 */

#ifndef SPEED_PREDICTION_SPEEDPREDICTION_H_
#define SPEED_PREDICTION_SPEEDPREDICTION_H_

#include <eigen3/Eigen/Dense>
#include <iostream>
#include <assert.h>
#include <list>
#include <ctime>
#include <vector>
#include "../route_prediction/Route.h"

namespace PredictivePowertrain {

class SpeedPrediction {
private:
```

```
      int I;                                        // number of input units
      int HL;                                       // number of hidden layers
      int O;                                        // number of output units
      int lastLayer;                            // number of the last layer
      int maxSpeed;                             // max vehicle speed
      int *HN;                                  // number of neurons in each hidden layer
      int *totalLayers;                         // total number of layers
      float lb_offset;                 // lower bound offset
      float alpha;                              // learning rate
    float ds;                    // time delta between predicted values
    std::vector<Eigen::MatrixXd*>* Wts;         // matrix of weights
    std::vector<Eigen::MatrixXd*>* yHid;            // hidden layer outputs
    std::vector<Eigen::MatrixXd*>* yInHid;      // hidden layer inputs

      void initParams();
      void printAll();

public:
      SpeedPrediction();
      SpeedPrediction(std::vector<Eigen::MatrixXd*>* Wts, std::vector<Eigen::MatrixXd*>* yHid, std::vector<Eigen::MatrixXd*>* yInHid);
      void predict(Eigen::MatrixXd * spd_in, Eigen::MatrixXd * spd_out);
      void train(Eigen::MatrixXd * spd_pred, Eigen::MatrixXd * spd_act, Eigen::MatrixXd * spd_in);
      std::vector<std::vector<Eigen::MatrixXd*>*>* getVals();
      void setVals(std::vector<std::vector<Eigen::MatrixXd*>*>* vals);
      int getI();
      int getO();
      void formatInData(Eigen::MatrixXd * input);
      void formatOutData(Eigen::MatrixXd * output);
    void scaleTrainingSpeed(Eigen::MatrixXd * input);
    void unscaleTrainingSpeed(Eigen::MatrixXd * output);
    int getNumLayers();
    float getDS();
    int getMaxSpeed();
    float getSpeedOffset();
    void output2Input(Eigen::MatrixXd* spdIn, Eigen::MatrixXd* spdOut);
};

}

#endif /* SPEED_PREDICTION_SPEEDPREDICTION_H_ */
```

12.69 SpeedPredictionUnitTest.cpp

```
//
//  SpeedPredictionUnitTest.cpp
//  PTC_xcode
//
//  Created by Brian on 8/7/16.
//  Copyright © 2016 Brian. All rights reserved.
//
```

```
/*
 * SpeedPredictionUnitTest.cpp
 *
 *  Created on: Apr 14, 2016
 *      Author: vagrant
 */

#include "../speed_prediction/SpeedPrediction.h"
#include "../map/GenericMap.h"
#include "UnitTests.h"

#include <iostream>
#include <assert.h>
#include <fstream>
#include <math.h>


#define PI 3.14159265

using namespace PredictivePowertrain;

// unit test for the SpeedPrediction class
SpeedPrediction* speedPrediction_ut(){

    int testEpochs = 300; // number of calls
    int refreshRate = 1; // call every one unit of input

    SpeedPrediction* sp = new SpeedPrediction();

    std::ofstream myfile("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/speedPredictionResults.csv");
    std::ifstream input("/Users/Brian/Desktop/the_goods/git/predictive_thermo_controller/data/spd.csv");
    int I = sp->getI();
    int O = sp->getO();

    Eigen::MatrixXd spd_in = Eigen::MatrixXd::Zero(1,10000);
    Eigen::MatrixXd spd_temp = Eigen::MatrixXd::Random(1,10000)*150;
    Eigen::MatrixXd spd_act = Eigen::MatrixXd::Zero(1, 9986);
    Eigen::MatrixXd spd_pred = Eigen::MatrixXd::Zero(1, 9986);

    std::string num;

    // read in speed from csv
    for (int i = 0; i<10000; i++){
        std::getline(input, num, ',');
        std::stringstream fs(num);
        double f = 0.0;
        fs >> f;
        spd_in(0, i) = f;
    }
```

```
    int k = I;

    // k < 9985 (original)
    while ( k < I + testEpochs){
        Eigen::MatrixXd temp_in = Eigen::MatrixXd::Ones(1, I+1);
        memcpy(&temp_in(0,0), &spd_in(0, k-I), I*sizeof(double));

        Eigen::MatrixXd temp_act = Eigen::MatrixXd::Zero(1,O);
        memcpy(&temp_act(0,0), &spd_in(0,k), O*sizeof(double));

        Eigen::MatrixXd temp_pred = Eigen::MatrixXd::Zero(1,O);

        // Format input data
        sp->formatInData(&temp_in);
        sp->scaleTrainingSpeed(&temp_act);

        // predict with historical data
        sp->predict(&temp_in, &temp_pred);

        // train the model
        sp->train(&temp_pred, &temp_act, &temp_in);

        // format output data
        sp->formatOutData(&temp_pred);
        sp->unscaleTrainingSpeed(&temp_act);

        // concatenate predicted and actual data
        int index = k-(I+1);
        for (int j =0; j < O; j++){
            spd_pred(0, index+j) = temp_pred(0, j);
            spd_act(0, index+j) = temp_act(0, j);
        }

        // update k
        k = k + refreshRate;

        std::cout<< k << std::endl;
    }

    // write the actual and predicted speed
    for (int i = 0; i < testEpochs; i++){
        myfile << spd_act(0,i);
        myfile << ", ";
        myfile << spd_pred(0,i);
        myfile << "\n";
    }

    return sp;
}
```

12.70 UnitTests.h

```
/*
 * unit_tests.h
 *
 *  Created on: Mar 16, 2016
 *      Author: vagrant
 */

#ifndef UNIT_TESTS_UNITTESTS_H_
#define UNIT_TESTS_UNITTESTS_H_

#include "../route_prediction/RoutePrediction.h"
#include "../speed_prediction/SpeedPrediction.h"

void buildCity_ut();
void citySection_ut();
void dataCollection_ut();
void dataManagement_ut();
void elevationtypes_ut();
void goalmapentry_ut();
void goalToLinkMap_ut();
void link_ut();
void intersection_ut();
void intersectiontypes_ut();
void linkToStateMapEntry_ut();
void linkToStateMap_ut();
void node_ut();
void route_ut();
PredictivePowertrain::RoutePrediction* routePrediction_ut();
PredictivePowertrain::SpeedPrediction* speedPrediction_ut();
void roadTypes_ut();
void kinematics_ut();
void GPS_ut();
void driverPrediction_ut();
void vehicleDiagnostics_ut();
void city_ut();

#endif /* UNIT_TESTS_UNITTESTS_H_ */
```