
TRACKING TESTING FRAMEWORK

A Major Qualifying Project submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfilment of the requirements for the Degree of Bachelor of Science on March 17, 2014



Submitted By: Michael Burns, Ian Lukens, Christopher McAndrews

Advised By: Professor Elke Rundensteiner

Sponsored By: BAE Systems

Sponsor Liaisons: James Costa and Keith Pray

Abstract

For this project the team created a testing framework for the tracking and fusion domain. This framework allows for automated testing of tracking engines and integrates with the Jenkins continuous integration server. The framework has components that generate truth data, add error to the truth to create modeled data, transform the modeled data into an estimate of the truth, calculate metrics by comparing this estimate to the actual truth, and display the metrics in a human readable format on Jenkins. The team also produced a user guide that provides documentation and instruction for use of the framework.

Executive Summary

Our project was to design a testing framework for tracking and fusion engines. In order to do this, the team researched tracking engines, specifically sensors, different tracking algorithms, the Kalman filter, and generic problems with tracking. The team then established some useful metrics for the tracking domain. Also, the team researched good testing procedures and applied these practices to the testing framework. Additionally, the team did background research on several open source tools including Jenkins and the Simulation of Urban Mobility (SUMO) traffic simulator.

Our design philosophy has sections that correspond to the different components of the testing framework. This includes the data structures our framework uses, the traffic simulator we use to generate truth data, and the error generator we created to manipulate that data. Next, follows sections about the different tracker implementations we wrote to validate our framework, the truth association algorithms involved, and the metrics implemented and displayed on Jenkins.

The paper concludes with a discussion of possible extensions to the framework and different avenues to explore. Some ideas are using more realistic sensor models, a more complex tracking engine utilizing Kalman filters to predict movement, live streaming of data, and more realistic truth association algorithms. Finally, we have included the User Manual for BAE Systems which specifies how to use the framework's features as well as how to extend the framework. Also included is a test plan that explains how we validated our framework.

Acknowledgements

The group would like to offer our thanks to the following people and organizations that assisted and supported us throughout our project, which led to the successful completion of this Major Qualifying Project.

- Elke Rundensteiner for advising our group and providing helpful feedback throughout the project.
- James Costa for educating us on the Tracking and Fusion domain as well as guiding us on our presentations and paper.
- Keith Pray for educating us on the Tracking and Fusion domain as well as guiding us on our presentations and paper.
- BAE Systems for giving us the opportunity to work on this project.

Table of Contents

Abstract.....	1
Executive Summary.....	2
Acknowledgements.....	3
Table of Contents.....	4
Table of Figures.....	6
Table of Equations	7
Introduction	8
Background	9
Tracking.....	9
Sensors.....	9
Tracking Algorithms	11
The Kalman Filter	13
Problems with Tracking.....	14
Metrics	16
Identifying a useful metric	17
Established tracking metrics	18
Testing.....	21
Testing Procedure	21
Testing Framework	22
Traffic Simulator.....	22
Jenkins.....	24
Bamboo	25
Design Philosophy.....	26
Data Structures	27
Traffic Simulator.....	28
Error Generator.....	29
Tracker Interface.....	30
Identity Tracker.....	30
Parameterized Tracker.....	31
Truth Association Interface.....	31

Implementation of Truth Association Interface	32
How the Truth Association Interface is used in the framework	33
Metric Interface	33
List of Included Metrics.....	34
Jenkins Plugin.....	38
Future Extensions	39
Conclusion.....	41
Bibliography	43
Appendix A - User Manual	46
Appendix B – Test Plan.....	69

Table of Figures

Figure 1: Layers of Abstraction from Raw Points to Fused Tracks.....	12
Figure 2: Tracking Engines Predict Next State	13
Figure 3: Kalman Filters Maintain Multiple Hypotheses.....	14
Figure 4: Tracks Crossing and Splitting	16
Figure 5: Starburst Pattern.....	16
Figure 6: Example SUMO Network	23
Figure 7: Components of the Framework.....	26
Figure 8: Position Skew	29
Figure 9: Position Bias.....	29
Figure 10: Extra Readings.....	30
Figure 11: Point Dropping.....	30
Figure 12: Example Output on Jenkins	41

Table of Equations

Equation 1: Accuracy Metric	34
Equation 2: Assignment Accuracy Metric	34
Equation 3: False Discovery Rate	35
Equation 4: False Inclusion Rate	35
Equation 5: Matthew's Correlation Coefficient	36
Equation 6: Negative Predictive Value.....	36
Equation 7: Positive Predictive Value	36
Equation 8: Specificity.....	37
Equation 9: Target Effectiveness	37
Equation 10: Track Purity.....	37
Equation 11: True Inclusion Rate	37

Introduction

The motivation for this project was to develop a simple way to automatically test tracking and fusion engines. Our team developed a framework called the Tracking Testing Framework. Tracking is a complex domain that requires algorithms that match measured data from sensors to accurate approximations of reality. In order to validate tracking engines, it is useful to use testing simulations as this allows users to save time by being able to perform tests without expending resources on acquiring real world data. A desirable feature of the framework was that it interfaces with a continuous integration server, such as Jenkins. The purpose of the framework was to provide feedback based on defined metrics. Metrics are numerical values calculated from simple measurements. The purpose of these metrics is to examine specific behaviors of a tracking engine. In order to validate each metric and the framework, our team developed a JUnit test suite that exercised the metrics and the components of the framework. Through this testing, our team constructed a framework that not only works with simple simulated data, but also can accept input from real world exercises, provided that the data conforms to our data structures. In order to make the system maintainable for future users, a requirement for the project team was documentation. To this end we created a user guide document (Appendix A - User Manual) which outlines basic use cases of the framework as well as how to extend the framework's functionality. The user manual supplements the source code, which contains extensive Javadoc comments and further internal, clarifying comments.

Background

By implementing an automatic and prompt feedback loop for changes to a tracking engine, engineers can more easily make improvements to the software. In order to show how a testing framework for tracking engines could improve their design, this chapter reviews general tracking methodology, the problems that occur in tracking, research on the usefulness of performance metrics, and software testing procedures. The chapter concludes with background information on traffic simulators and continuous integration servers.

Tracking

In this section we describe the basics behind tracking and its principles. We begin with an overview of tracking sensors and the different factors that influence their readings. We then discuss the use of tracking algorithms for interpreting sensor data and the common issues that tracking engines encounter.

Sensors

Tracking builds on sensor systems based on radar, sonar, electro-optical, or infrared readings (Blackman, 2004). The two types of sensor groups are passive and active sensors. Passive sensors measure energy that is naturally available. For example, a microwave radio meter is a passive sensor (Government of Canada, 2014). An active sensor is one which produces the energy that it detects. In general, an active sensor works by propagating a wave outward, and the wave eventually reflects off some target (Skolnik, 2008). The reflection's composition is largely variable on the target, and consequently, using sensors well depends on the analysis of the data they produce. Ideally, a reading returns a perfectly isometric reflection, a radial output equal in all directions clearly indicating the target's exact position. However, in practice, reflections have unbalanced readings, referred to as scatterings, which occur due to differentiating size in objects, multiple reflection sources on a single

object, or inaccurate equipment. Once the scattered wave returns, sensor readers compare its energy strength against the outputted wave to determine characteristics of the target. A tracking system then uses this information to identify targets of interest, and uses the data from the sensor to track these targets over time. Still, sensors have many limiting factors including their response time, accuracy of readings, and interpretation of data.

Consequently, target readings are still not necessarily useful due to signal noise - the number of false or undesired objects picked up by a sensor, or inherent system biases. For example, radar sensor systems must account for noise since any metal object or even other radar waves can cause noise. Noise handling begins with trying to reduce noise at its source by adjusting the gain on a reader (Toomay & Hannen, 2004). This affects sensor sensitivity by reducing or increasing the number of readings a sensor perceives. For example, upping the gain on a sensor will allow it to sense more objects, but also increase the noise or false positives of objects. Conversely, decreasing the gain will reduce noise but can end in no object readings or missed objects. Along with gain manipulation, most sensor algorithms use target thresholds, also known as gates, to filter out noise readings. The specification of these gates considers the specific environment of the sensor and the objects the user wants it to identify.

In addition to noise, sensors can have issues with poor resolution. Resolution refers to how far apart objects must be before they can be identified as separate entities (Skolnik, 2008). With poor resolution, the track of one object can seemingly turn into two separate tracks instantaneously; which is problematic for tracking algorithms. However, a high resolution means more possibly superfluous data input, so consequently, resolution varies depending on a sensor's purpose. For example, weapons tracking radar can have resolution of a few yards, but search radars might have a resolution of several miles (Wolff, 2013).

Once a sensor identifies a desired target, radar systems extrapolate useful information about the target from the sensor reading (Wolff, 2013). In radar systems, the most accurate data a sensor can give is the slant range, meaning the distance from the sensor to a target. Using the speed of light and the time of the wave's round trip, a sensor can yield a precise distance measurement. Sensors also estimate the radar cross section, an estimation of the target size. This size estimate corresponds to the target's components that reflect radio magnetic waves, but does not necessarily correspond to the actual size of the object. Furthermore, a sensor reading gives a radial velocity, the general speed of the object in relation to the radar. Although accurate, the radial velocity is not final in determining the direction of the target; this requires the angle of motion (Skolnik, 2008). A sensor can read the angle, but a slight inaccuracy produces a large area of uncertainty due to the typical distance between the sensor and the target.

As a final note, different sensors are better suited for identifying different attributes of objects (Wolff, 2013). For example, using a small, vertical radar wave typically yields a more precise elevation reading than horizontal radar waves. For the best results, one would want to use multiple sensors to form the most accurate tracks.

Tracking Algorithms

The next step of tracking is to interpret sensor data. Sensors collect large amounts of diverse information including background clutter, hardware errors, thermal noise, and finally, targets of interest (Blackman, 2004). With so much ambiguous data collected by the sensors, proper tracking requires advanced algorithms to interpret it. Tracking algorithms analyze the sensor data making estimations of different tracks for the targets of interest the algorithm identifies (Wolff, 2013). A track is a series of reports corresponding to a single object. The tracking algorithm links tracks, which contain additional information collected by the sensors. Using the last estimated positions and tracks and contrasting

them with any newly generated information, a tracker updates and checks its targets, always fulfilling certain criteria to attempt to build realistic and accurate tracks. During the creation and analysis of tracks, algorithms use data fusion, the process of combining simple data into more advanced, and useful information. This helps to pool either sensor data or multiple tracks into a single aggregate track. See Figure 1 for an illustration of this process.

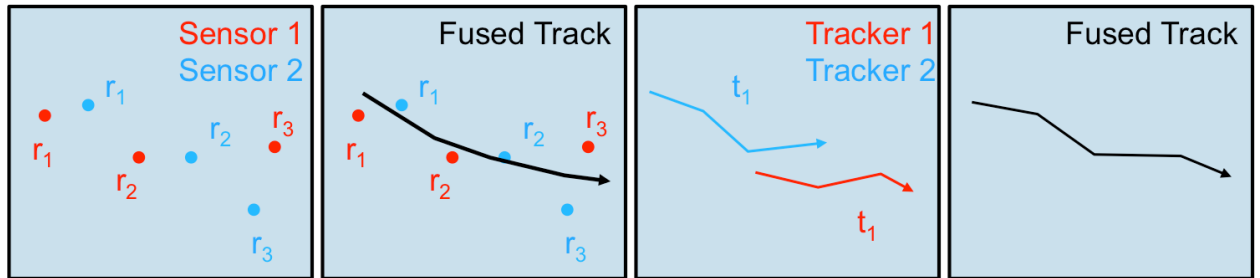


Figure 1: Layers of Abstraction from Raw Points to Fused Tracks

Fusing data depends mostly on the concepts of association and state estimate fusion (Chong, Chang, More, & Barker, 2000). Association is the linking of various reports from sensors and accurately identifying if tracks or readings refer to the same or different objects. The state estimates are the estimations of an object's movement. They refer to the status of assumptions made by previous estimations that then affect future estimations. In other words, they hold the certainty of different states and the assumptions used to make them. These become especially significant when state estimates have correlations between different data states. In particular, a distinction arises when data fusion occurs between tracks rather than reports because the creation of different tracks is likely associated due to using algorithms with similar assumptions. Therefore, any assumptions made by fusing engines need to account for such dependencies and covariance between data and consequently keep certainty correspondingly lower than if there were no such correlation.

When a sensor identifies data that the tracking algorithm deems unassociated with a previous track, the algorithm creates a new track (Wolff, 2013). As more readings come in, this new track will

either lengthen or the algorithm will discard the track as noise. This method helps generate a better picture of the actually relevant data. The intake of data also allows the software to generate further predictions about the targets' attributes and future movement (Figure 2). More advanced algorithms will take these predictions into account when taking in other readings.

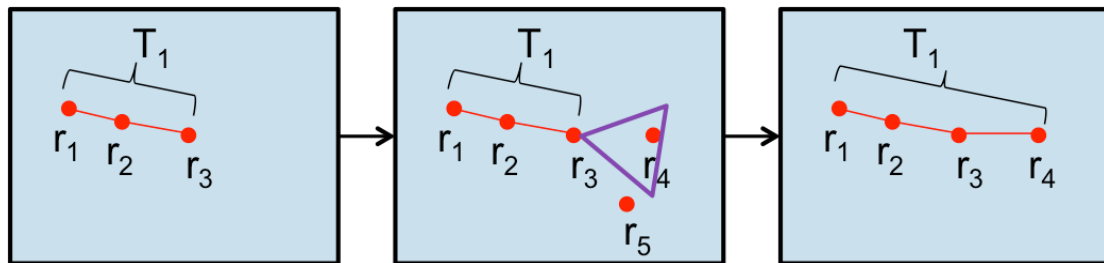


Figure 2: Tracking Engines Predict Next State

The Kalman Filter

Trackers use many different algorithms and equations, but the one that we will focus on in this paper is the Kalman filter. Most modern tracking systems that utilize radar use the Kalman filter (Wolff, 2013). The Kalman filter is a proposed solution to understanding discrete data gathered over time, such as sensor data, and uses a system of mathematical equations to remove extraneous noise or outlier data, such as sensor noise (Bayoumi, 2012). The purpose of the Kalman filter is to establish values closer to the truth from unclean and uncertain data. The Kalman filter also allows estimations of past, present, and future readings. The key basis of the Kalman Filter is that it assumes the relationship between data is linear and Gaussian distributed. By applying a linear operator to each new reading while factoring in its previously predicted state, the Kalman filter measures any new possible associations and then updates its state accordingly while also keeping track of its own uncertainty (Wolff, 2013). Covariance matrices represent this uncertainty.

Covariance is a measure of the correlation between two variables, and in the case of its use for tracking, the Kalman filter derives covariance from the variation that could be present in sensor

readings. A matrix encapsulates the uncertainty of covariance in each dimension. This allows further extrapolation of the given sensor readings to produce an area of uncertainty around a sensor's reported location. This area is important to consider since any changes can greatly affect how the sensor data gets interpreted (Figure 3). Tracking this area is necessary to allow Kalman filters to make accurate predictions due to the unreliability of sensor readings.

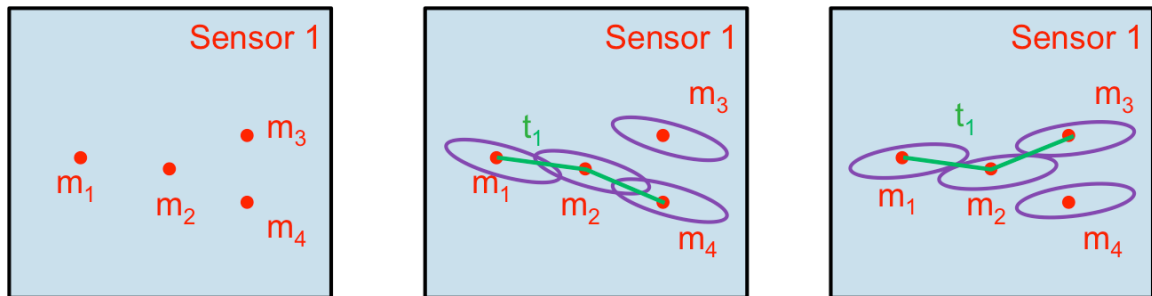


Figure 3: Kalman Filters Maintain Multiple Hypotheses

Uncertainty is particularly important for Kalman filter, because once they run long enough and collect enough data, the math can become over-confident in its own predictions rather than relying on sensor data (Wolff, 2013). Therefore, it is necessary to keep a proper weight between the Kalman filter's data and the sensor data.

Problems with Tracking

Despite advanced algorithms and highly technological sensors, tracking still has failure cases. One major issue is the need for timeliness. Although a tracking algorithm could theoretically take in gigabytes of data at once and painstakingly check every possible association upon each update, limiting factors of time and computing power factor in (Wolff, 2013). Therefore, a need for quicker algorithms develops, and to make it faster, something must be lost while still maintaining accuracy. Subsequently, this creates the dilemma of labeling part of the algorithm superfluous. Should the algorithm throw out more readings as noise? Should it reduce the amount of different possibilities for tracks it keeps in memory or reduce the area of uncertainty to cut down on different possibilities altogether? To help

answer these questions, it is important to know the specific problems trackers encounter. These issues include high false alarm rates with sensors, target fading, sensor uncertainty, unpredictable maneuvers, confusion events, radar jamming, and non-uniformly distributed clutter.

High false alarm rates occur when a sensor detects too many objects, either due to natural interference or imprecise sensor settings. Altering the gain on the radar or identifying interference patterns can help alleviate this issue (Toomay & Hannen, 2004).

Target fading occurs due to either change in a target's detectability or inherently low target detectability. Targets will disappear and reappear between sensor readings, so the tracking algorithm must keep a larger memory of tracks rather than instantly discarding those that end abruptly.

Sensor uncertainty means that tracks will also have a larger covariance, and more track options will exist in a given data set. A tracking algorithm in this scenario must accurately acknowledge the resultant uncertainty of its own assumptions.

Unpredictable target maneuvers hinder a tracking algorithm's ability to estimate future motion. To account for this, a Kalman filter applied to unpredictable targets ought to give more weight to sensor readings.

A confusion event concerns junctures when sensor data creates an extremely ambiguous situation. One example is target crossing. This occurs when two tracks appear to cross from data reports. Due to the incremental updates of sensors, it is unclear whether the two objects on these tracks crossed paths, as in both continuing on their course, or split, meaning that they essentially met at a point and then turned away from each other. Figure 4, below, shows an illustration of the issue.

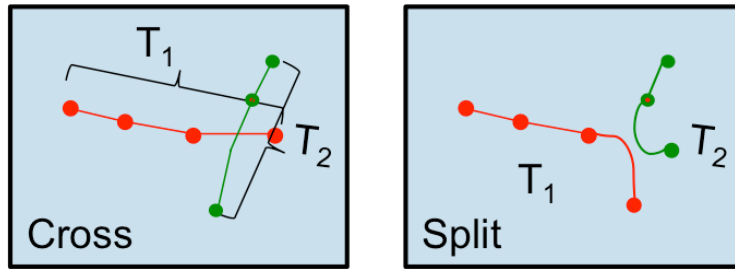


Figure 4: Tracks Crossing and Splitting

Starbursts are similar problems that occur when tracked objects split into multiple, smaller targets (Figure 5). In this situation, a tracking algorithm must create new tracks seemingly out of nowhere.

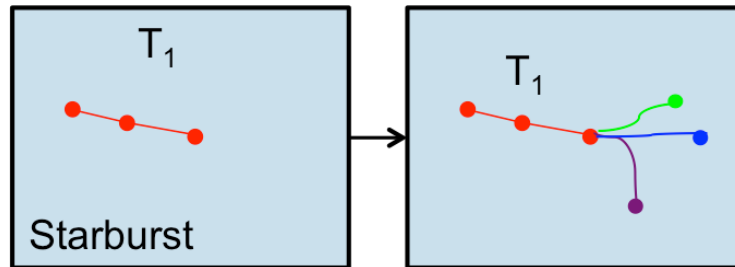


Figure 5: Starburst Pattern

Radar jamming is when some external factor is blocking sensor readings. It has similar repercussions as target fading.

Non-uniformly distributed clutter refers to the noise or unwanted feedback a radar receives. As previously stated, there must be a delicate balance between allowing noise and discarding it for tracking to be useful.

Metrics

In this section we discuss metrics and how they can be used for analyzing a tracking system. The first section describes the characteristics needed for a metric to be useful, and the second section describes metrics used in evaluating the performance of tracker engines.

Identifying a useful metric

The main purpose of analytics is to measure progress towards a goal (Yoskovitz, 2013). Metrics are one of the tools used in analytics to help evaluate progress by creating sub-goals. A metric itself is a formal standard or goal that ought to be met to indicate success, and by using a combination of metrics, people are able to analyze what they did well and what they need to improve in order to meet their goal.

To identify a helpful metric, one must leverage knowledge of what success means and an understanding of how to succeed. The goal of tracking is to accurately track objects. A simple ratio of the right answer over wrong answer, although a useful metric, paints a very general picture of possible problems without a direction on how to improve. If a tracker returns a fifty percent accuracy rate for identifying tracks, the only assumption is that the tracker is working poorly. However, this does not begin to answer why. Is it sensor noise or too much sensor uncertainty? Do the tracks cross too much, or do objects keep fading? Perhaps the algorithm simply makes bad assumptions. By gathering more data and calculating additional metrics, these questions start becoming answerable.

A useful metric is actionable (Yoskovitz, 2013). This means that a metric should create a feedback loop whereupon receiving its value, a person takes some appropriate response. The better a metric is, the clearer it is what this response should be. For instance, after receiving a low grade on a calculus exam, a student might respond with study more, but if the metric were a low test grade on areas that covered integrals, the student would know to specifically study more integrals. A metric is not always perfectly clear in the actions needed to improve it, but it does create a starting point for exploration and problem solving.

A useful metric is also comparable (Yoskovitz, 2013). This means that people can compare the values of a metric over time. This way, people can graph metrics to show progress, decline, or no

change. Often times, this means that most metrics are quantitative, standard number formats that improve either the higher or lower they get. Such comparisons are useful for analyzing progress.

The last important note on metrics is that they do not always mean what people might first think. Like all tools, metrics can have faults. Specifically, correlation does not imply causation. When analyzing metrics, it is always good to find the cause of problems as this then allows someone to fix them. However, metrics are sometimes limited in their analytical power, and people should not follow them blindly. Each metric and metric reading is different and greatly depends on the context behind it.

Established tracking metrics

Previous studies have established useful tracking metrics. The difficulties with these metrics arise due to the complexity of a standard multi-part tracking system. Within a complete system, there are sensor uncertainties, noise, track impurities, and reading interpretations that all affect the output of the tracking engine. In theory, the correct result from a tracking engine is not necessarily one that accurately identifies a track when accounting for these issues (Smith, Register, Blair, & Levedahl, 2010). Consider the situation where a tracker correctly identifies an object despite the presence of a lot of noise. Although this seems like a useful reading, this may lead the algorithm to incorrectly disregard objects that it perceives as extraneous noise in future cases. Situations like this necessitate the use of advanced metrics with multiple simulations and evaluations to provide useful feedback. To this end, there are two subcategories of metrics: measures of effectiveness and measures of performance. A measure of effectiveness analyzes the algorithms and processes used to arrive at a conclusion while the measures of performance focus on the results. From our studies, it appears most established tracking metrics fit the measure of performance category rather than measure of effectiveness.

Typically a tracking metric is derived from using ground truth data and simulating a sensor reading from the truth data (Smith, Register, Blair, & Levedahl, 2010). This simulated sensor reading is

entered into the tracking engine, and its output gets compared to the truth data. Through this comparison, four general categories of data interpretation arise: true inclusion, true exclusion, false inclusion, and false exclusion (Canavan, McCullough, & Farrell, 2009). True inclusion refers to a sensor data point correctly identified within a track. True exclusion refers to a sensor data point correctly excluded from a track such as noise or a separate track. False inclusion refers to any noise or incorrect data points wrongly included in a track, and false exclusion denotes data points left out of a track that should have been included. Although the determination of these four categories still has inherent imprecision to account for simulated area of uncertainty, they allow creation of a decent array of general performance metrics as defined in the paper by Canavan, McCullough, and Farrell and represented in Table 1.

Metric	Description	Calculation
Track Purity	The percentage of correctly included points divided by the total included points plus the wrongly excluded points	$\frac{TI}{TI + FI + FE}$
Target Effectiveness	The number of correct inclusions divided by the size of the fusion track set	$\frac{TI}{\text{Size of data set}}$
Assignment Accuracy	The number of correct inclusions divided by the size of the truth track set	$\frac{TI}{TI + TE + FI + FE}$
Accuracy	The number of correct inclusions and exclusion divided by the total size of the track set	$\frac{TI + TE}{TI + TE + FI + FE}$
Specificity	The number of correct exclusions divided by the number of correct exclusions plus the number of false inclusions	$\frac{TE}{TE + FI}$
Positive Predictive Value	The number of correct inclusions divided by the number of correct inclusions and exclusions	$\frac{TI}{FI + TI}$
Negative Predictive Value	The number of correct exclusions divided by the number of correct inclusions and exclusions	$\frac{TE}{TE + FE}$
False Discovery Rate	The number of incorrect inclusions over the number of inclusions	$\frac{FI}{FI + TI}$
Matthews Correction Coefficient	Machine learning classification of binary inclusion systems (i.e. included vs. excluded)	$\frac{(TI * TE) - (FI * FE)}{\sqrt{((TI + FI) * (TI + FE) * (TE + FI) * (TE + FE))}}$
True Inclusion Rate	The true inclusions divided by the true inclusion plus the false exclusions	$\frac{TI}{TI + FE}$
False Inclusion Rate	The false inclusions divided by the false inclusion plus the true exclusions	$\frac{FI}{FI + TE}$

Table 1: Example Metrics (TI = true inclusion, TX= true exclusion, FI = false inclusion, FX = false exclusion)

Despite the extensiveness of these calculations, like all metrics, they require appropriate understanding.

Testing

In this section, we describe good testing practices required when evaluating a software system.

Testing Procedure

The first part of creating any test is planning (Spillner, Linz, Rossner, & Winter, 2012). During planning, the testers should provide a detailed testing procedure document that explicitly outlines the purposes of tests and their execution. These testing purposes are called testing requirements that the actual tests must fulfill. This phase includes analysis, design, and evaluation of exit criteria and leads to the eventual derivation of test techniques and strategies. Two of the most important aspects of testing are purpose and role. Since exhaustive testing is usually impractical, the tests that the testers run must be high priority tests that serve specific and useful functions within the testing framework. During planning it is also important to note the different process phases within the software and the amount of control over different program aspects.

After the analysis and design phase comes implementation and execution (Spillner, Linz, Rossner, & Winter, 2012). Testers create concrete test cases derived from logical test cases and execute them. This involves using a test oracle of expected results and comparing them to actual results. All test failures must have a clear origin or be traceable.

However, execution does not imply completion in this case. A large part of testing is analyzing results, improving tests, and retesting. This is why it is important for tests to have sufficient reporting on their results for further analysis. The main evaluation of the software starts with asking what the test results mean. After that, testers decide if the results are okay or if the software needs improvement. Besides evaluation of the software, testers ought to evaluate the tests themselves. For example, if test

results show inaccurate specification, testers must improve their techniques to generate better results. Most testing results in software improvement and further testing.

Testing Framework

In this section, we discuss the tools we researched in order to create a suitable testing framework for our metric simulations. In the first section, we discuss our process for finding a suitable traffic simulator, and in the second section, we provide an overview of Jenkins, the integration and testing server we used. Finally, we discuss an alternative to Jenkins known as Bamboo.

Traffic Simulator

In order to test our tracking framework, we need to generate realistic data that the framework can use as input for the different trackers. For the sake of simplicity, we decided to limit our simulated data to two dimensions, removing altitude as a variable. With the goal of our simulation confined to two dimensions, we decided that traffic simulators would be a reasonable solution. Implementing a traffic simulator from scratch is beyond the intended scope of this project, so our group decided to utilize available open-source traffic simulators.

The first traffic simulator that our group investigated was a “Microscopic Traffic Simulator” developed by Martin Trieber. It fit many of our requirements, in that it was open-source, developed in Java, and could simulate different road scenarios. However, the main purpose of this traffic simulator was to demonstrate the “fundamental issues of traffic dynamics rather than simulating specific road networks” (Trieber, 2011). Unfortunately, our goal was to find a traffic simulator that focused on individual vehicles, rather than the traffic system as a whole. Additionally, this simulator turned out to not be very extensible, and it would have required a significant amount of development work in order to get the desired output from the simulator. As such, our team decided to use the SUMO traffic simulator instead.

The Simulation of Urban Mobility (SUMO) is “an open source, highly portable, microscopic and continuous road traffic simulation package” (Sumo Wiki, 2013). Our team ultimately decided to use SUMO because it has many important features. For example, SUMO can be executed from the command line, so our tests can automatically run SUMO simulations without any real-time input required. Additionally, the output of SUMO simulations can be saved to a file, which we can then use as input for multiple tests, making it possible to have predictable test results when we are evaluating our trackers and metrics. The simulations themselves are also highly customizable. It is possible to select the road configuration used in the simulation. For example, the simulation can involve a single road, or two roads that cross each other (Figure 6).

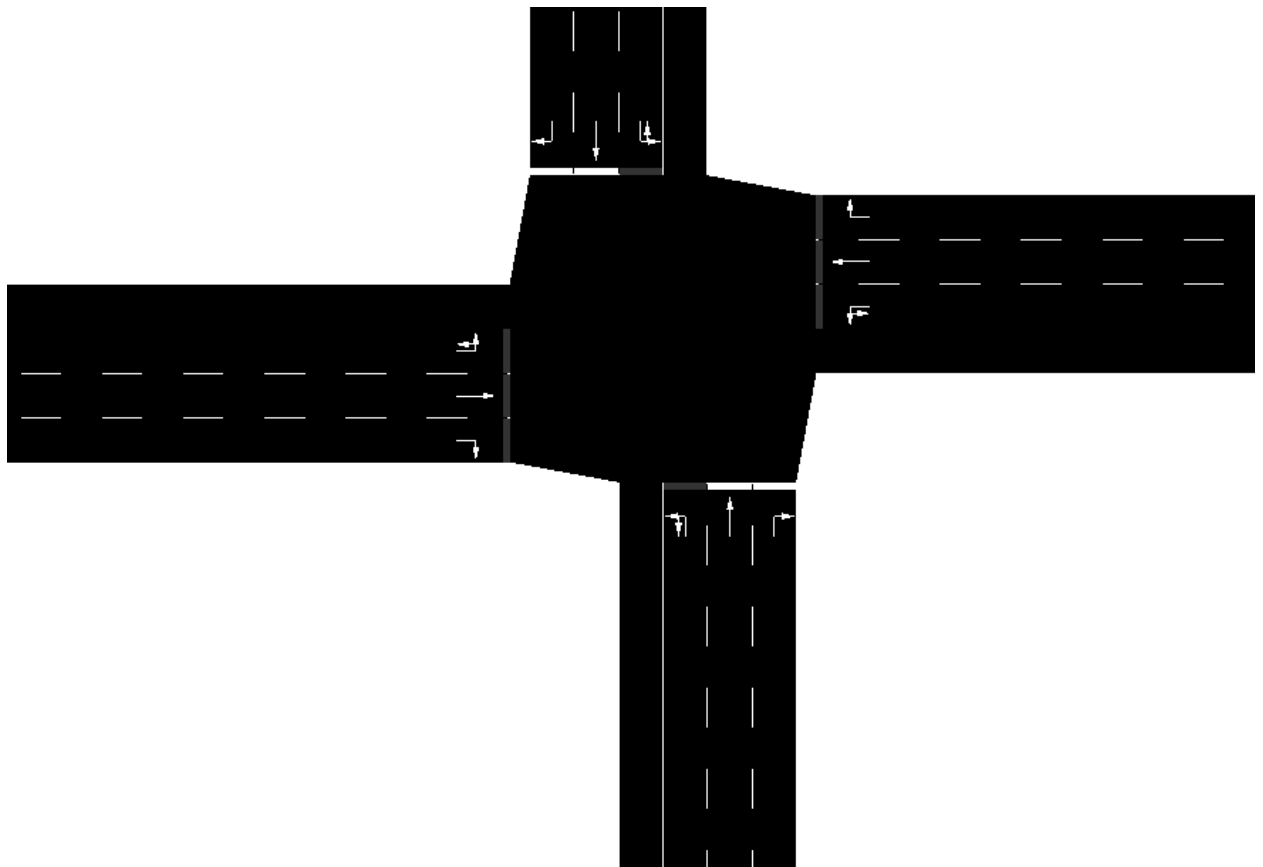


Figure 6: Example SUMO Network

Additionally, the number of vehicles per simulation is variable. These customizations make it very easy to test our trackers and metrics on different scenarios.

Jenkins

Jenkins is an open source continuous integration server that allows teams of software developers to work quickly and efficiently (Kawaguchi, n.d.). A more concrete definition of continuous integration is that it is a process in which members of a team make contributions to a project daily (or even more often) and automated builds and tests validate and verify each contribution (Fowler, 2006). This helps ensure that changes to a build do not cause any new errors and also gives results of tests per build iteration. Jenkins helps to accomplish a continuous integration environment by handling the automated build and test process described above.

Jenkins is also highly customizable (Moser & O'Brien, 2011). Jenkins allows developers a great deal of configuration options in many different aspects of continuous integration. It allows the developer to specify versions of Java Developer Kits (JDKs) to use when building the project. Jenkins also is flexible with its build options and supports automated Ant builds, or integrating with Maven to execute build goals. As both Maven and Ant can set automated testing goals, Jenkins is popular for utilizing test driven development, and it is easy to setup. Jenkins further promotes continuous integration by working with popular source control systems such as Subversion (SVN) and Git to start automated builds by probing the source control for changes. At the conclusion of the build process, Jenkins allows communication to the team via test results by email, IRC, RSS, or a Jenkins web server.

In addition to these basic functions, Jenkins supports extensions via plugins (Kawaguchi, n.d.). Plugins extend the functionality of Jenkins at over one hundred possible extension points allowing for customization of nearly all aspects of the system. Plugins add a great deal of options to Jenkins such as

adding more graphical user interface options or customizable build options. Using plugins, a user can graphically display test output for all their builds and tests.

Bamboo

Another tool that is popular for continuous integration is Bamboo. Bamboo is a continuous integration server made by Atlassian. A common use for Bamboo is to automate the build process and run tests to validate each developer's contribution to a shared code base. Some of the key features of Bamboo are its strong integration with JIRA, an issue tracking tool to aid in code development, stronger integration with Git, such that it can automate merging, and a better method of deploying the finished product to customers. Bamboo also includes a great deal of customization in order to support different project sets and build goals that an end user can specify. Unfortunately, Bamboo is not open source, and licenses for the system have a monetary cost. However, Bamboo has a feature to import quickly from Jenkins, so teams can opt to use it in the future if the monetary cost makes sense (Atlassian: Bamboo, 2013).

Design Philosophy

According to Joshua Bloch, “an interface is generally the best way to define a type that permits multiple implementations” (Bloch, 2008). Our overall goal when designing the framework was to keep it flexible and extensible by abstracting specific implementations with interfaces, and using files to pass data between the different components of the framework. In order to make the framework clear and easily extensible, we divided the framework into several distinct, independent components.

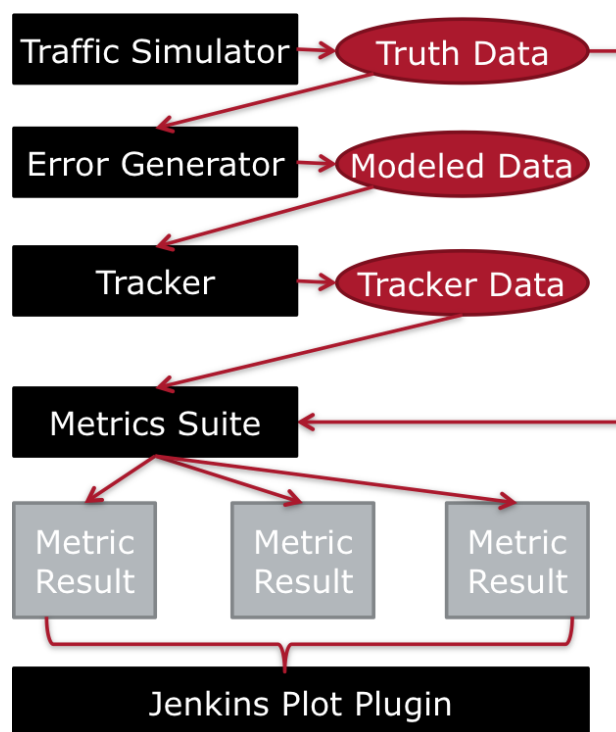


Figure 7: Components of the Framework

As seen in Figure 7, the components of our framework are the traffic simulator, the error generator, the tracker interface, the metric interface, and the Jenkins plugin. Though not pictured here, the framework also includes truth association. The framework implements the traffic simulator using the Simulation of Urban Mobility (SUMO). The error generator is a custom built module that produces more realistic data by introducing one or more pre-defined error types. The tracker interface allows the framework to use any tracking algorithm, as long as the tracker implements the simple functions

defined in the interface. The truth association interface allows the framework to use any truth association algorithm, as long as the algorithm implements the interface. The metric interface allows the framework to calculate any metric, as long as the metric implements the functions defined in the interface. The Jenkins plugin, which we did not create, allows for quick updating and visualization of metric results because of its integration with the source code management of the development team.

Data Structures

To allow information to travel through the testing framework we created several data structures that encapsulate the data of sensor reports and tracker output.

The first data structure, named a `TrackDataPoint`, mirrors a report of an object from a sensor. This contains a time of when a sensor spotted the object, the position of the object, and a covariance allowing for trackers to create an area of uncertainty around the point. The `CoordinatePosition` data structure represents the position, which is stored as x, y, and z coordinates. Additionally, the covariance itself is another data structure specifying a matrix of uncertainty values. The framework also adds an identification number to each `TrackDataPoint` in order to help with truth association.

The next data structure is a `Track`. A `Track` is a list of `TrackDataPoints` and an identifying name. A collection of `Tracks` makes up the main data structure used by the framework – the `TrackStruct`. `TrackStructs` contain any number of `Tracks` and a unique identifying name. We extended `TrackStructs` to make `ErrorTrackStructs`, which the `ErrorGenerator` outputs. `ErrorTrackStructs` have a truth source, which is a field containing the identifying name of the original `TrackStruct` used to create the `ErrorTrackStruct`. This allows us to keep track of where the modeled data came from, information that can then be used in our metrics. Modeled data is data that has had errors introduced to it by the `ErrorGenerator`. We use these `TrackStructs` to represent all of the tracking data moving through the framework including the truth data, modeled data, and tracker data.

Traffic Simulator

In order to test our framework, we needed to be able to generate large amounts of truth data, which is data where we know the position of objects at given points in time. We decided to use a traffic simulator for this purpose because it would be able to handle the positions and times of multiple objects, and it would be easy to extract that information from the simulator.

For this framework we utilized the Simulation of Urban Mobility (SUMO) for a variety of reasons. The primary motivation for using SUMO was its command line functionality. From the command line we were able to run simulations and extract the positions of the vehicles into a file. Our framework then parsed the file and used it as truth data for testing purposes. While the framework does not support automatically running SUMO simulations, it does contain scripts to facilitate converting the output of simulations into the data structure that the framework uses.

In addition to the above features, SUMO was favored because it could handle popular file types already in use by the geographic information community. SUMO contains tool scripts that support the conversion of TIGER database Shapefiles to SUMO simulations. The TIGER (Topologically Integrated Geographic Encoding and Referencing) database contains spatial representations that include “features such as roads, railroads, rivers, as well as legal and statistical geographic areas” (United States Census Bureau, 2014). The process of converting Shapefiles into SUMO simulations involved three different sub-tools. The first sub-tool converted the Shapefile to a SUMO network file. The second tool took the existing network and generated a trips file. The third and final tool took both of these files and generated different routes for the cars to follow from the trips. Finally, a SUMO configuration file would wrap these three files, allowing the simulator to run the simulation. This process was poorly documented and error prone, so our team created a script to handle the process. By combining the

various existing tools our team was able to make it fairly simple and less error prone for any user to come up with fairly complex and somewhat realistic simulations of actual traffic and tracking scenarios.

Error Generator

Another tool we used to test our framework was a custom-built error generator. We designed this error generator to take in truth data. The error generator outputs mutations of the input data based on specified error configurations. These error configurations are a collection of error types and associated probabilities. Our framework supports several error types, which simulate realistic errors that could occur with real-world sensors. The first error type is position skew, in which data points are randomly moved within a maximum displacement in any direction on the x and y axes (Figure 8).

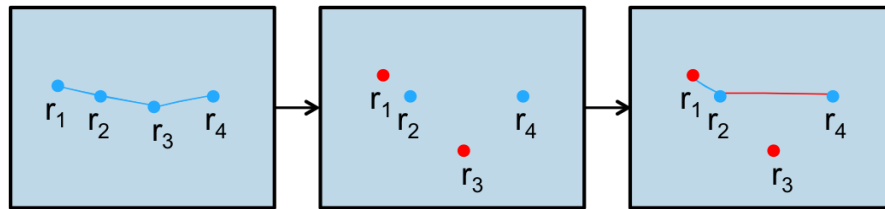


Figure 8: Position Skew

This is to simulate uncertainty in a sensor's measurement. The next two error types are x and y position bias respectively (Figure 9).

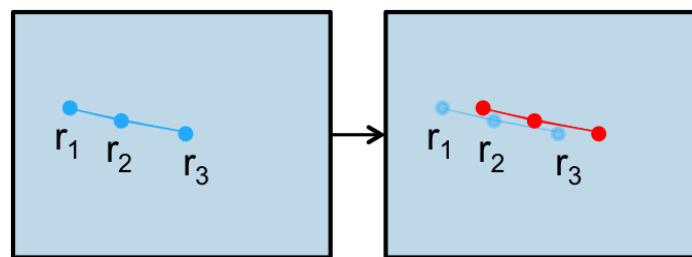


Figure 9: Position Bias

These errors simulate if a sensor was reading all measurements a fixed distance from the truth. For example, a sensor might shift all of the data points five meters to the right of where the object is actually located. Another error type is time skew, which is where the time values of the data points are

increased or decreased within a threshold. Finally, there are extra readings and dropping readings, which either generate new points within the track, or drop old points from the track (Figure 10 and Figure 11).

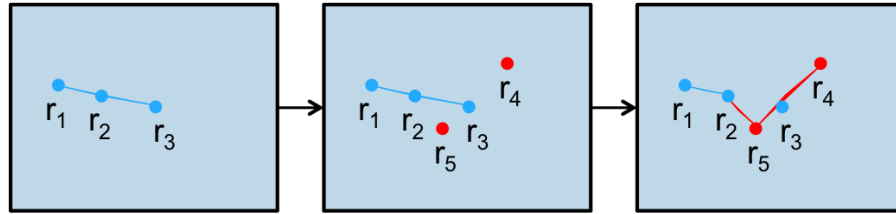


Figure 10: Extra Readings

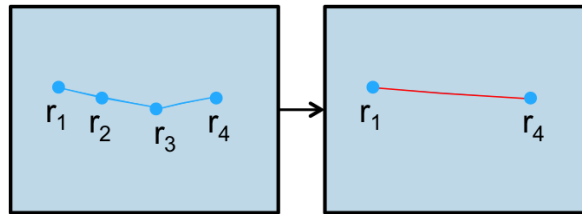


Figure 11: Point Dropping

All of these error types can be included in any combination in an error configuration. This allows for highly customizable error scenarios that can be increasingly complex.

Tracker Interface

The third step of the Testing Tracking Framework involves running various tracking and fusion engines against the modeled data output from the various error configurations. These engines will then output their own set of tracks that metrics will analyze. In the following sections, we describe two tracker implementations that we used in order to validate the testing framework.

Identity Tracker

The first tracking engine that we implemented in the Testing Tracking Framework was an identity tracker. The identity tracker takes in a list of tracks and outputs the same set of tracks without alteration. This tracker served as an important first step to implementing a full run through of the

system. This allowed the team to discover any components that were missing from the initial design of the framework. The primary benefit of this identity tracker was that it allowed for the ability to verify that the metrics scored by the system had some sanity to them. For example, the identity tracker was excellent for validating metrics such as average track length, and number of tracks. However, the identity tracker was insufficient to test more complicated metrics, especially metrics that required truth data, because the tracker did not interact with the input data in a meaningful manner.

Parameterized Tracker

To compensate for the limitations of the identity tracker, we designed a more robust, parameter based tracker. The parameterized tracker is different from the identity tracker in several useful ways. The first is that it takes in the truth data in addition to the modeled data. Using this data it is able to restore the modeled data to a more accurate state. In order to determine how much correction of the modeled data should be performed the parameterized tracker takes in several variable parameters. The parameters are percentages for the number of falsely added points to remove, incorrectly dropped points to re-insert, and incorrectly split tracks to reconnect. The ability to quickly tweak the inputs to the parameterized tracker allowed for faster testing of metric behavior. This allowed us to verify that metrics would produce different values when the tracker had different inputs, and therefore performed differently. Additionally, it demonstrated that our framework could support multiple, because the framework handled multiple instances of the parameterized tracker running concurrently but with different inputs.

Truth Association Interface

The fourth component of our testing framework is truth association. Truth association is the process of associating the measured points specified from the output of the tracker back to their

corresponding truth points. This step is essential to a testing framework as it allows for many more complex metrics that can give meaningful insight into the performance of a tracking engine.

Implementation of Truth Association Interface

We developed a simple Point Association Interface that has a single function call that returns the data points outputted by the tracker mapped to associated truth points. From this data, we associate tracks from the tracker with truth tracks to generate true and false inclusions and exclusions.

Nearest Neighbor

The Nearest Neighbor algorithm associates points outputted from the tracker with those closest to them using the Mahalanobis distance as its measurement (Orlov). In this algorithm, multiple tracker points may associate with the same truth point.

Global Nearest Neighbor

The Global Nearest Neighbor algorithm is similar to the Nearest Neighbor algorithm except that it matches data points on a one to one basis. This algorithm works in three stages. In the first stage, it goes through a gating process that discards points that are not close enough in time and distance for the algorithm to consider them for association. These time and distance gates are parameters of the truth association function due to the varying nature of sensor reliability and different expectations of test cases. In the next step, the algorithm calculates the Mahalanobis distance between tracker points and each of the truth points. Then, using an A* style searching method with Mahalanobis distance as its scoring system, the algorithm attempts to associate each point with a truth point (Russell & Norvig, 1994). The searching method continues until each point has an association or there are no associations left to make. The algorithm scores unassociated points as the maximum distance allowed due to gating. This allows the search to advance and still find the best possible fit of truth points to tracker points.

Known Truth

Due to the complexity of the Nearest Neighbor and Global Nearest Neighbor algorithms, we determined that we ought to use a more transparent association technique to demonstrate the effectiveness of metrics when data had a higher guarantee of accuracy. To that end, we implemented the Known Truth algorithm that uses data point identification numbers to associate points. We set up this method by adding point identification numbers to the truth data and maintaining them when outputted from the tracker. This allows easy association by ID with guaranteed correctness even after the error generator distorts points.

How the Truth Association Interface is used in the framework

The truth association occurs after the tracker outputs its results from truth data. Then, if the framework calculates a metric that uses truth data on this data set, the truth association occurs and generates true and false inclusions and exclusions. To do this, the association technique matches tracks from the tracker with truth tracks using the best fit possible as determined by a computation of the sum of true inclusions and exclusions minus the sum of false inclusions and exclusions to score tracks. The framework uses the highest scoring track and its associated inclusion and exclusion values in the overall association of the TrackStruct. This technique of simple association helps create more advanced metrics.

Metric Interface

To help evaluation of a tracking system, the fifth part of our framework involves outputting metrics of different values depending on tracker performance. These metrics create unique files for all interpreted data put through the testing framework. The metric interface includes a calculate function that takes in the tracker's outputted TrackStruct. Metrics that use truth data extend the truth metric

class that gives access to true and false inclusion and exclusion data generated from the truth association.

The TestingController class calculates the metrics using their calculate function, before outputting their return values to files. Rather than running every metric in the framework, only those included as xml files in the MetricsToRun folder get calculated. This allows for picking and choosing those metrics that developers wish to use while still being easily extensible for adding more metrics.

List of Included Metrics

The following metrics correspond to the example metrics given in Table 1. For metrics that use truth data, their calculations will be shown as TI = true inclusions, TE = true exclusions, FI = false inclusions, FE = false exclusions.

Accuracy Metric

This metric calculates the accuracy of track inclusions based on truth data. It provides a good indicator of overall tracker performance.

$$\frac{TI + TE}{TI + TE + FI + FE}$$

Equation 1: Accuracy Metric

Assignment Accuracy Metric

This metric calculates the percent of true inclusions over the size of the dataset. It provides an indication of how well the tracking engine is identifying objects correctly.

$$\frac{TI}{TI + TE + FI + FE}$$

Equation 2: Assignment Accuracy Metric

Average Track Lifespan Metric

This metric calculates the average time length of a reported track. It helps provide insight on how quickly tracks might end or continue according to the tracking engine.

Average Track Update Rate Metric

This metric returns the average time between updates in the reported tracks. This can show how well a tracker does depending on the rate of the data it receives.

False Discovery Rate Metric

This metric calculates the rate of false inclusions in the data against all inclusions. This value helps show whether or not the tracker includes too much noise.

$$\frac{FI}{FI + TI}$$

Equation 3: False Discovery Rate

False Inclusion Rate Metric

This metric calculates the false inclusions compared to the true exclusions. High values in this metric mean the tracker is too lenient in adding points to its tracks.

$$\frac{FI}{FI + TE}$$

Equation 4: False Inclusion Rate

Lifespan Similarity Metric

This metric calculates the average lifespan of the reported tracks divided by the average lifespan of the truth data tracks. This helps generate a better idea of how well the tracker handles track length.

Matthews Correlation Coefficient Metric

This metric calculates the Matthew's Correlation Coefficient using true and false inclusions and exclusions. It indicates overall performance based on distance from pure chance with a ceiling being that everything is predicted correctly or incorrectly.

$$\frac{(TI * TE) - (FI * FE)}{\sqrt{((TI + FI) * (TI + FE) * (TE + FI) * (TE + FE))}}$$

Equation 5: Matthew's Correlation Coefficient

Negative Predictive Value Metric

This metric calculates the percentage of true exclusions versus false exclusions. A high value here indicates that the tracker effectively identifies noise.

$$\frac{TE}{TE + FE}$$

Equation 6: Negative Predictive Value

Number of Tracks Metric

This metric calculates the number of tracks. This can be useful in tracking performance due to memory overhead with a higher number of tracks or incorrect splitting of longer tracks into multiple smaller ones.

Positive Predictive Value Metric

This metric calculates the percentage of true inclusions versus false inclusions. This metric shows how accurately the tracker includes points within its tracks.

$$\frac{TI}{FI + TI}$$

Equation 7: Positive Predictive Value

Specificity Metric

This metric calculates the percentage of true exclusions versus false inclusions. A high value shows that the tracker excludes points very well.

$$\frac{TE}{TE + FI}$$

Equation 8: Specificity

Target Effectiveness Metric

This metric calculates the percentage of true inclusions versus the size of the track.

$$\frac{TI}{\text{Size of data set}}$$

Equation 9: Target Effectiveness

Track Count Purity Metric

This metric calculates the percentage of the size of the outputted tracks versus the size of the truth data. This helps indicate the data discrepancy between the sets.

Track Purity Metric

This metric calculates the number of true inclusions divided by the true and false inclusions and the false exclusions. This value helps give a general idea of how well the tracker is performing.

$$\frac{TI}{TI + FI + FE}$$

Equation 10: Track Purity

True Inclusion Rate Metric

This metric calculates the true inclusions compared to the false exclusions. Low values in this metric mean the tracker is too strict in adding points to its tracks.

$$\frac{TI}{TI + FE}$$

Equation 11: True Inclusion Rate

Jenkins Plugin

The final component of the Tracking Testing Framework is the integration with Jenkins. Our use of Jenkins was twofold; Jenkins organized a set of jobs which worked together to generate output data from a JUnit test suite of metrics, and then displayed the results on graphs. To plot the test output in a human readable format, we utilized an existing plugin, the Plot Plugin (Neilson, n.d.). Our framework integrates with the plugin by creating a variety of properties files that the plugin parses in order to display. By configuring the plugin with Jenkins it is possible to display a wide variety of graphs and other metrics that a user wants to examine. This plugin simplifies tracking the performance of a tracker, as it allows users to quickly see if there are any major changes in performance that would require further investigation.

Future Extensions

We acknowledge that some of our components are only basic implementations, so below we discuss some possible future extensions of our framework.

An extension to this project could be more realistic simulations for data generation. In such a method, simulations would not only be the generation of data points, but would also simulate sensor behavior. The framework does not currently have a sensor interface that has associated error generation. Realistically, a sensor would observe only a certain area, and it would have unique imperfections that would cause errors in the data reports. The framework could simulate these imperfections using our current implementation of error generation and covariance. A possible implementation of sensors would involve a sensor interface where each sensor is associated with a specific error configuration, covariance, and data capture area. This would allow the simulations to generate data points from multiple sensors for the same object, making data fusion a more important feature of tracking engines, which our current simulations do not address.

For another possible continuation of the project, a future group could implement a more realistic tracking engine, specifically a tracking engine that utilizes a Kalman filter to predict each track's next data point. Using a more robust tracking engine would enable the framework to calculate more interesting data for the metric suite, which allows for more in depth analysis of the use of the metrics. Due to the limitations of the parameterized tracker, it is unclear which metrics would be the most useful in the development of a realistic tracking engine.

The current truth association algorithms implemented in the framework are a naïve method of analyzing data due to the complexity of tracking and fusion engines. Our implementation would have little value to actual tracking systems due to the incorrect assumption that points associate on a one to one or near one to one basis. Real world tracker engines would have multiple sensor inputs that could

duplicate points that would correspond to a single tracked object. A better truth association algorithm would associate outputted tracks with truth tracks using their position and shape, and then determine if its points ought to have been included rather than our reverse method of associating points and then associating tracks.

Future teams could also extend the project in a variety of non-component focused areas. For example, the framework is currently a file-based system. However, it could also support a stream-based system, which takes in data reports in real time. This would allow for more dynamic testing of a tracker actually receiving sensor reports asynchronously while constantly updating its tracks with its current state. Another extension would be improvements for ease of use through the development of a GUI, possibly implemented either as a desktop application or a Jenkins plugin. The framework could also allow for users to have more control over their tests.

Conclusion

For this project, we successfully implemented a proof of concept framework for a tracking engine. Using Jenkins in conjunction with the Plot plugin, the framework provides a prompt feedback loop by outputting metrics based on tracker performance.

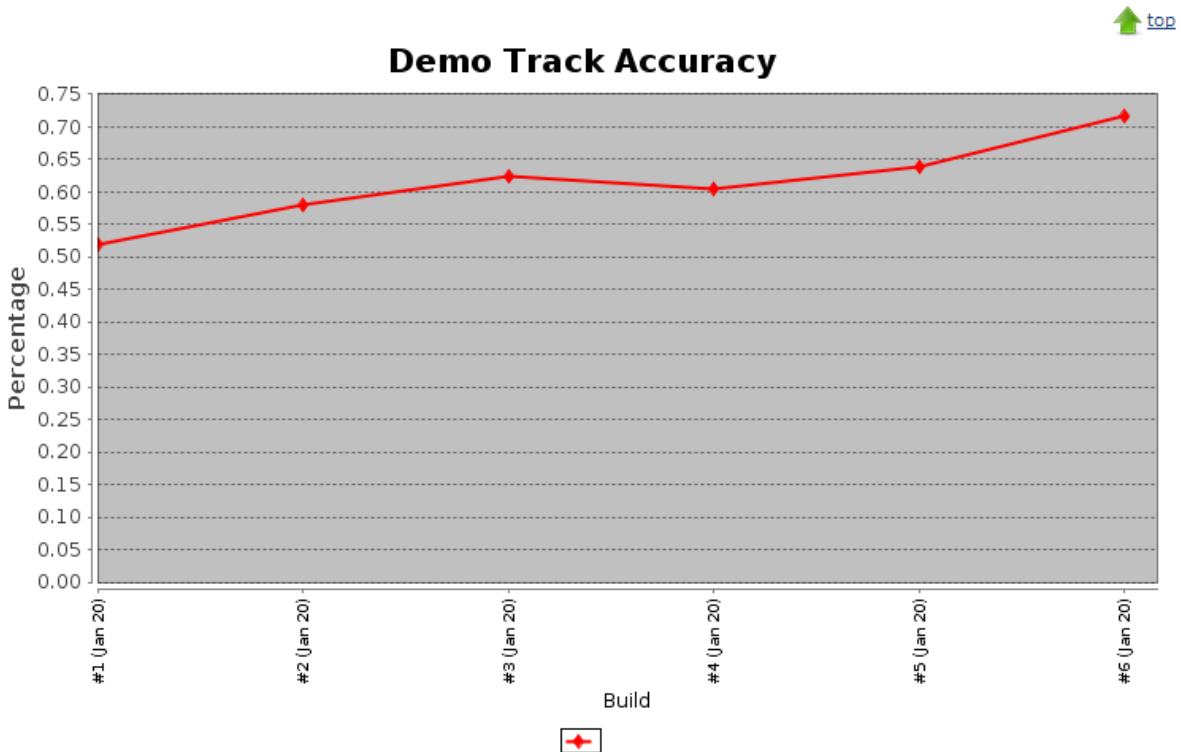


Figure 12: Example Output on Jenkins

The framework benefitted greatly from using various open source software tools, such as SUMO and Jenkins, because they saved us a great deal of time. Particularly, SUMO allowed us to generate realistic scenarios by using simulations based on Shapefiles (Figure 13).

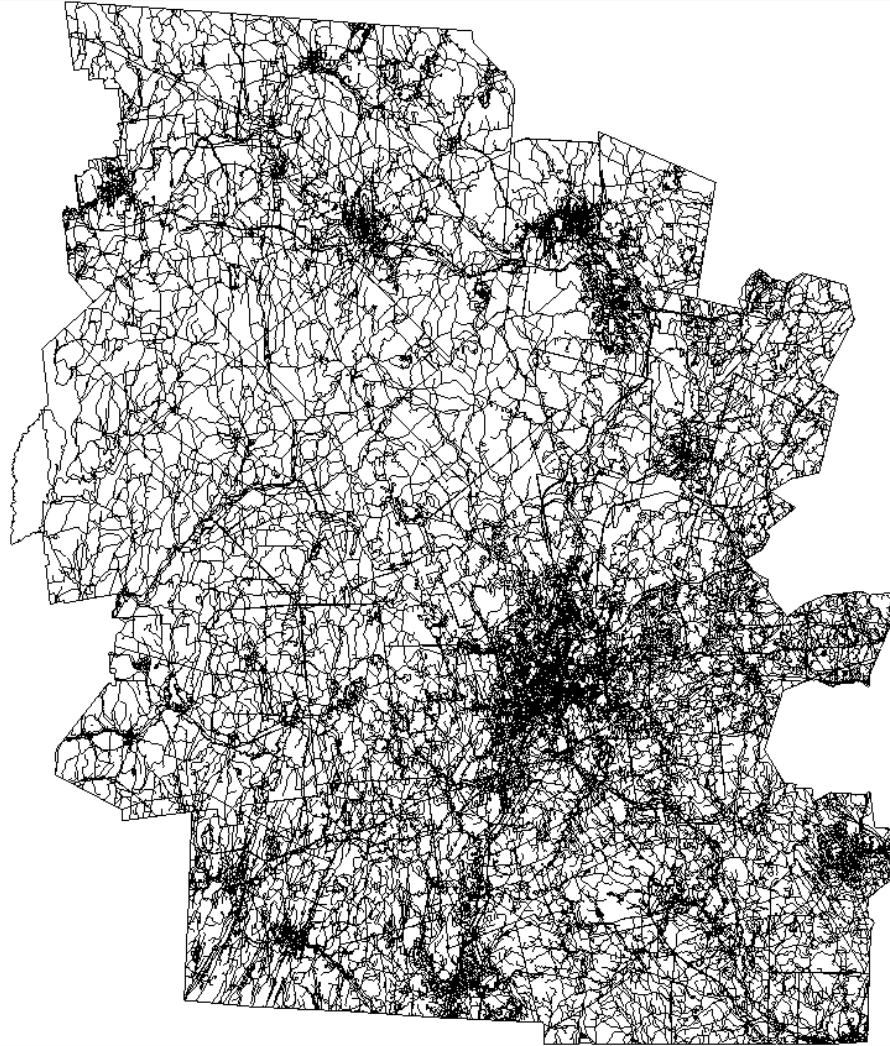


Figure 13: Example Shapefile of Worcester County in SUMO

Additionally, we designed the framework to be easily extensible by making all of the components implement basic interfaces. The framework allows users to automatically run test cases of standardized scenarios, which developers can create. Users can easily compare metrics against established benchmarks on a per build basis. This allows for comparing performance of tracking engines over time, enabling developers to measure improvement over time using actionable feedback. Furthermore, the framework uses Jenkins, providing a continuous integration environment for developers. Overall, the framework removes the burden of manually testing tracking engines from developers, and maintains flexibility for future extensions.

Bibliography

- Atlassian: Bamboo. (2013). Retrieved from Atlassian: <https://www.atlassian.com/software/bamboo/got-jenkins>
- Bayoumi, M. (2012). Kalman Filter. *Resource-Aware Data Fusion Algorithms for Wireless Sensor Networks*, 59.
- Blackman, S. S. (2004, January). Multiple Hypothesis Tracking For Multiple Target Tracking. *Aerorspace and Electronic Systems Magazine, IEEE*, pp. 5-18.
- Bloch, J. (2008). *Effective Java* (2 ed.). Addison-Wesley.
- Canavan, R., McCullough, C., & Farrell, W. (2009, July). Track-centric metrics for track fusion systems. *Information Fusion*, pp. 1147-1154.
- Chong, C.-Y., Chang, K.-C., More, S., & Barker, W. H. (2000, January). Architectures and algorithms for Track Association and Fusion. *Aerorspace and Electronic Systems Magazine, IEEE*, pp. 5-13.
- Fowler, M. (2006, May 1). *Continuous Integration*. Retrieved from <http://www.martinfowler.com/articles/continuousIntegration.html>
- Government of Canada. (2014, January 29). *Natural Resources Canada: Passive vs. Active Sensing*. Retrieved from Natural Resources Canada: <http://www.nrcan.gc.ca/earth-sciences/geomatics/satellite-imagery-air-photos/satellite-imagery-products/educational-resources/14639>
- Kawaguchi, K. (n.d.). *Jenkins*. Retrieved from <http://jenkins-ci.org>
- Moser, M., & O'Brien, T. (2011). *The Hudson Book*. Retrieved from <http://www.eclipse.org/hudson/the-hudson-book/book-hudson.pdf>

Neilson, E. (n.d.). *Plot Plugin*. Retrieved from <https://wiki.jenkins-ci.org/display/JENKINS/Plot+Plugin>

Orlov, A. I. (n.d.). Encyclopedia of Mathematics: Mahalanobis Distance. Retrieved from http://www.encyclopediaofmath.org/index.php?title=Mahalanobis_distance&oldid=17720

Russell, S. J., & Norvig, P. (1994). *Artificial Intelligence: A Modern Approach*.

Skolnik, M. (2008). *Radar Handbook* (3 ed.).

Smith, D., Register, A., Blair, W. D., & Levedahl, M. (2010). A Track Purity Approach for Trackin Metrcs. *Aerospace Conference, 2010 IEEE*, (pp. 1-11). Big Sky, MT.

Spillner, A., Linz, T., Rossner, T., & Winter, M. (2012). *Software Testing Practice: Test Management*. Rocky Nook.

Sumo Wiki. (2013, August 28). Retrieved October 14, 2013, from http://sumo-sim.org/wiki/Main_Page

Toomay, J. C., & Hannen, P. J. (2004). *Radar Principles for the Non-Specialist* (3 ed.). Retrieved from <http://common.books24x/.com.ezproxy.wpi.edu/toc.aspx?bookid=22971>

Trieber, M. (2011, June 1). *Microsimulation of Road Traffic Flow*. Retrieved October 14, 2013, from <http://www.traffic-simulation.de/>

United States Census Bureau. (2014, February 6). *TIGER Products*. Retrieved from <http://www.census.gov/geo/maps-data/data/tiger.html>

Wolff, C. (2013). Radar Tutorial.

Yoskovitz, B. (2013, March 9). Measuring What Matters: How to Pick a Good Metric. Retrieved from <http://onstartups.com/tabid/3339/bid/96738/Measuring-What-Matters-How-To-Pick-A-Good-Metric.aspx>

Appendix A - User Manual

TRACKING TESTING FRAMEWORK USER GUIDE

By Michael Burns, Ian Lukens, and Christopher McAndrews

March 5, 2014

Table of Contents

Table of Contents.....	1
Table of Figures.....	2
Installation	3
Importing Our Framework into Eclipse	3
Installing Jenkins.....	3
Setting up an Initial Job	3
Test Configuration.....	6
Plot Plugin.....	6
Framework Features	8
Truth Generation	8
Using SUMO.....	9
Importing Truth.....	10
Using the SUMO Converter jar file	10
Example Truth Data	10
Error Generator	13
Error Types.....	13
Error Configurations	14
How to Extend the Error Generator	14
How to Import Data with Error.....	15
Tracker Interface.....	15
How to Run Trackers.....	15
How to Add New Trackers to the Framework	15
Metric Interface	16
Example Metrics	16
How to Run Metrics.....	17
How to Add New Metrics to the Framework.....	17
Appendix A – Sample Truth File	18
Appendix B – Sample sumocfg File	21
Appendix C – Sample Error Configuration File.....	22

Table of Figures

Figure 1: Making a Jenkins Job Step 1.....	3
Figure 2: Making a Jenkins Job Step 2.....	4
Figure 3: Making a Jenkins Job Step 3.....	4
Figure 4: Making a Jenkins Job Step 4.....	5
Figure 5: Making a Jenkins Job Step 5.....	5
Figure 6: Making a Jenkins Job Step 6.....	6
Figure 7: Directory Hierarchy	6
Figure 8: Jenkins Configuration Step 1.....	7
Figure 9: Jenkins Configuration Step 2.....	7
Figure 10: Jenkins Configuration Step 3.....	8
Figure 11: TrackStruct format	8
Figure 12: Image of Cross Struct Network	11
Figure 13: Image of Worcester Network	12
Figure 14: Error Configuration Format	14
Figure 15: Existing Error Data	15

Installation

Importing Our Framework into Eclipse

To run the framework in Eclipse, import the .zip file as a new Maven Project. If it is not already installed, install the m2e plugin or an equivalent maven extension through the Eclipse Marketplace. This may require setting up Maven on the host computer. Documentation and download links for Maven can be found at <http://maven.apache.org/download.cgi>.

Installing Jenkins

To install Jenkins, follow the instructions on the Jenkins wiki at <https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Ubuntu>. We found the nginx proxy instructions easier to use than the apache instructions. After installing Jenkins, you need to install Java in order to run the JUnit tests. To install Java, run the following commands:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java7-installer
sudo apt-get install oracle-java7-set-default
```

Setting up an Initial Job

To set up an initial Jenkins job, use the following instructions:

1. Select the New Item option on the main Jenkins page.



Figure 1: Making a Jenkins Job Step 1

2. Select Free Style Project.

Item name

- ☒ **Build a free-style software project**
This is the central feature of Jenkins. Jenkins will build your
- ☐ **Build a maven2/3 project**
Build a maven 2/3 project. Jenkins takes advantage of you
- ☐ **Build multi-configuration project**
Suitable for projects that need a large number of different
- ☐ **Monitor an external job**
This type of job allows you to record the execution of a pro
[details.](#)
- ☐ **Copy existing Item**
Copy from




Figure 2: Making a Jenkins Job Step 2

3. Add the source control that includes the framework source code.

Item name

Description

[\[Raw HTML\]](#) [Preview](#)

☐ Discard Old Builds

☐ This build is parameterized

☐ Disable Build (No new builds will be executed until the project is re-enabled.)

☐ Execute concurrent builds if necessary

Advanced Project Options

Source Code Management

☐ CVS

☐ CVS Projectset

☒ None

☐ Subversion

Build Triggers

Figure 3: Making a Jenkins Job Step 3

4. Configure the source control with the proper credentials.

Source Code Management

☐ CVS
☐ CVS Pserver
☐ None
☒ Subversion

Repository URL:
 Credentials:
 Initial module directory:
 Repository depth:
 Ignore externals: ☐

Add module...
 Additional Credentials:
 Check-out Strategy:
 Use 'svn update' whenever possible, making the build faster, but this causes the artifacts from the >
 Repository browser:

Build Triggers

☐ Build after other projects are built
☐ Trigger builds remotely (e.g., from scripts)
☐ Build periodically
☒ Poll SCM

Schedule:
 Ignore post-commit hooks: ☐

Figure 4: Making a Jenkins Job Step 4

- Configure the build step to execute the test option.

Build Triggers

☐ Build after other projects are built
☐ Trigger builds remotely (e.g., from scripts)
☐ Build periodically
☒ Poll SCM

Schedule:
 Ignore post-commit hooks: ☐

Build

Add build step ▼
 Execute Windows batch command
 Execute shell
 Invoke Ant
 Invoke top-level Maven targets

Figure 5: Making a Jenkins Job Step 5

- Save your configurations.

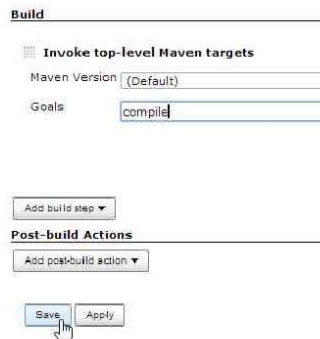


Figure 6: Making a Jenkins Job Step 6

Test Configuration

To execute the testing framework, a test runner should use our `TestingExecutable.jar`. To configure different customized tests, put files in the `lib` folder following the directory hierarchy as shown in Figure 7. The jar assumes that these directories already exists, but it will create them if they do not exist. This is important to note, because if the jar is in the incorrect location, it will generate a directory structure that the user may not want. More specifics about which files go where are in their respective sections.

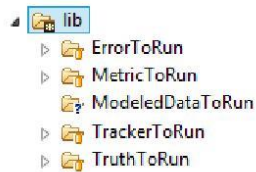


Figure 7: Directory Hierarchy

As an important note, if new features are added to the framework, the `TestingExecutable.jar` will need to be repackaged in order to function properly. To run the jar on Jenkins, it will be necessary to wrap the jar in a simple JUnit test, Ant file, or some other method supported by Jenkins which will execute the jar.

Plot Plugin

To install the plot plugin, go to the Manage Jenkins screen in Jenkins. Then select the Manage Plugins option. Search for Plot on the available tab, and select Install and Restart Jenkins.

To output metrics to a visual plot display, you must configure the plots using the Plots plugin as shown below.

0. Have a Jenkins job configured with the framework.
1. Select the configure option on the Jenkins Job.



Figure 8: Jenkins Configuration Step 1

2. Select "Plot build data" from the "Add post-build action" drop down.

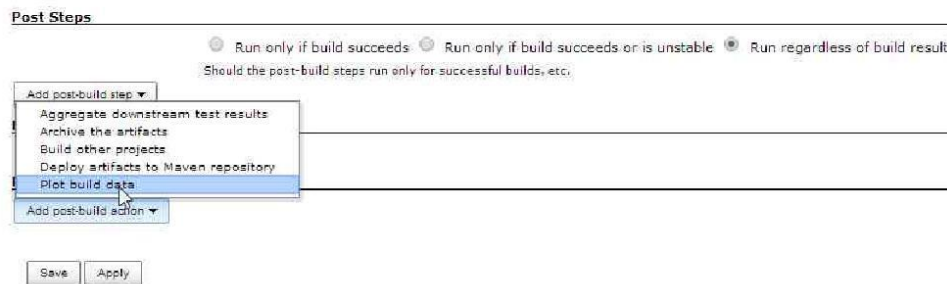


Figure 9: Jenkins Configuration Step 2

3. Fill out the fields with the pertinent data. It is important to note that if no test output exists when the plot plugin is being configured, it will warn that the specified output file does not exist, but this does not impact functionality.

Post-build Actions

☒ **Plot build data**

Plot group:

Plot title:

Number of build data points:

Plot y-axis label:

Plot style:

Auto Descriptions as labels: ☐

Track format file:

☒ Load data from properties file

Data values legend label:

☐ Load data from csv file

☐ Load data from xml file using xpath

Figure 10: Jenkins Configuration Step 3

Framework Features

Truth Generation

Truth generation is required in order to test tracking engines. In order to put truth data into this framework, the data must conform to the following data structure.

```
<track.TrackStruct>
  <name>Unique data set name</name>
  <tracks>
    <track.Track>
      <id>Unique Track ID number</id>
      <dataPoints>
        <track.TrackDataPoint>
          <time>
            <milliseconds>Time value</milliseconds>
          </time>
          <position>
            <x>X value</x>
            <y>Y value</y>
            <z>Z value</z>
          </position>
          <pointID>Unique Point ID number</pointID>
        </track.TrackDataPoint>
      </dataPoints>
    </track.Track>
  </tracks>
</track.TrackStruct>
```

Figure 11: TrackStruct format

An example data file can be found in Appendix A – Sample Truth File. Truth data generation through SUMO is supported by our framework, but other methods can be used if desired.

Using SUMO

SUMO (Simulation of Urban Mobility) is an open-source, external tool that can be found at <http://sumo-sim.org/>. To convert Shapefiles into the format supported by SUMO, you can use a script that we have provided.

Generating SUMO Configuration Files

These instructions are for use with Shapefiles from <http://www.census.gov/geo/maps-data/data/tiger-line.html>. Specifically, this works with Shapefiles from 2012. We selected this year because it was the most recent year that had downloads readily available online.

Using Our Script for Shapefiles

In order to run our script (ShapefileConverter.py) to convert Shapefiles into a SUMO supported file format, you must have sumo and its associated executables on your computer. The script takes in several inputs from the user, but also has some default values for simplification. The inputs are listed in Table 1.

Table 1: ShapefileConverter.py Arguments

Input	Description	Required	Default Value
-d	The directory on your computer where SUMO is installed. At this level you should see the bin and tools directories.	Yes	None
-p	Path to and prefix for the Shapefiles to convert.	Yes	None
-n	Name of the net file.	No	net.net.xml
-t	Name of the trip file.	No	trips.trips.xml
-s	Name of the sumo configuration file.	No	sumocfg.sumocfg
-l	Length of the simulation in SUMO steps.	No	500

An example input to the script is as follows:

```
ShapefileConverter.py -d lib\sumo-0.18.0 -p Shapefiles\tl_2013_25027_edges  
-n example.net.xml -t example.trips.xml -s example.sumocfg -l 123
```

An example sumocfg file produced by this script is included as Appendix B – Sample sumocfg File.

Running SUMO

To run an actual SUMO simulation, use the following command:

```
sumo.exe -c SUMOCFG_FILE --fcd-output OUTPUT_FILE --fcd-output.geo
```

It is important to note that the final option, “--fcd-output.geo”, is only required if you want the data to be output in latitude and longitude, rather than just on an arbitrary x-y grid.

Importing Truth

To import truth data into the framework, convert the data into the TrackStruct format as shown in Figure 11, then place the file in the “lib\TruthToRun” directory. If your data is in SUMO’s output format, it can easily be converted to the TrackStruct format by using the SumoConverter.jar which is included with the framework.

Using the SUMO Converter jar file

The SUMO Converter jar is a tool we have provided that takes an output file from a SUMO simulation and converts it into the TrackStruct data structure used in our framework. It takes in two command line arguments, the first being the path to the SUMO output file, and the second being the desired name of the new TrackStruct. This name is also used as the name of the TrackStruct file, which will be created in the local directory of the jar. This output file should then be moved to the TruthToRun folder if the tester wants it to be used in future tests.

Example Truth Data

The following are example data sets that were used for testing. They were generated with SUMO and are included as examples and for continued unit testing of the framework.

“Cross Struct”

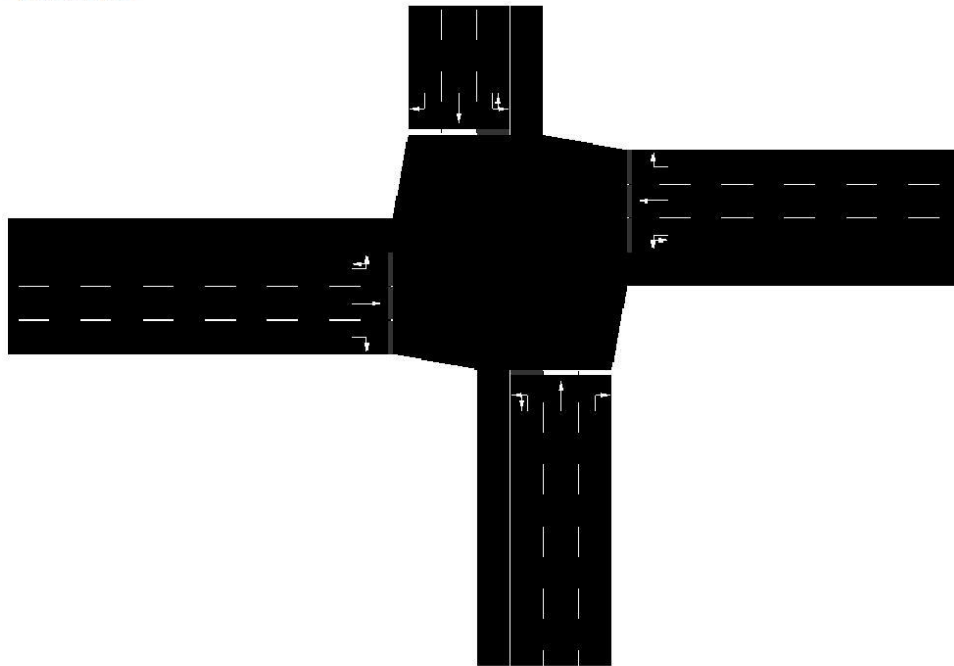


Figure 12: Image of Cross Struct Network

“Worcester”

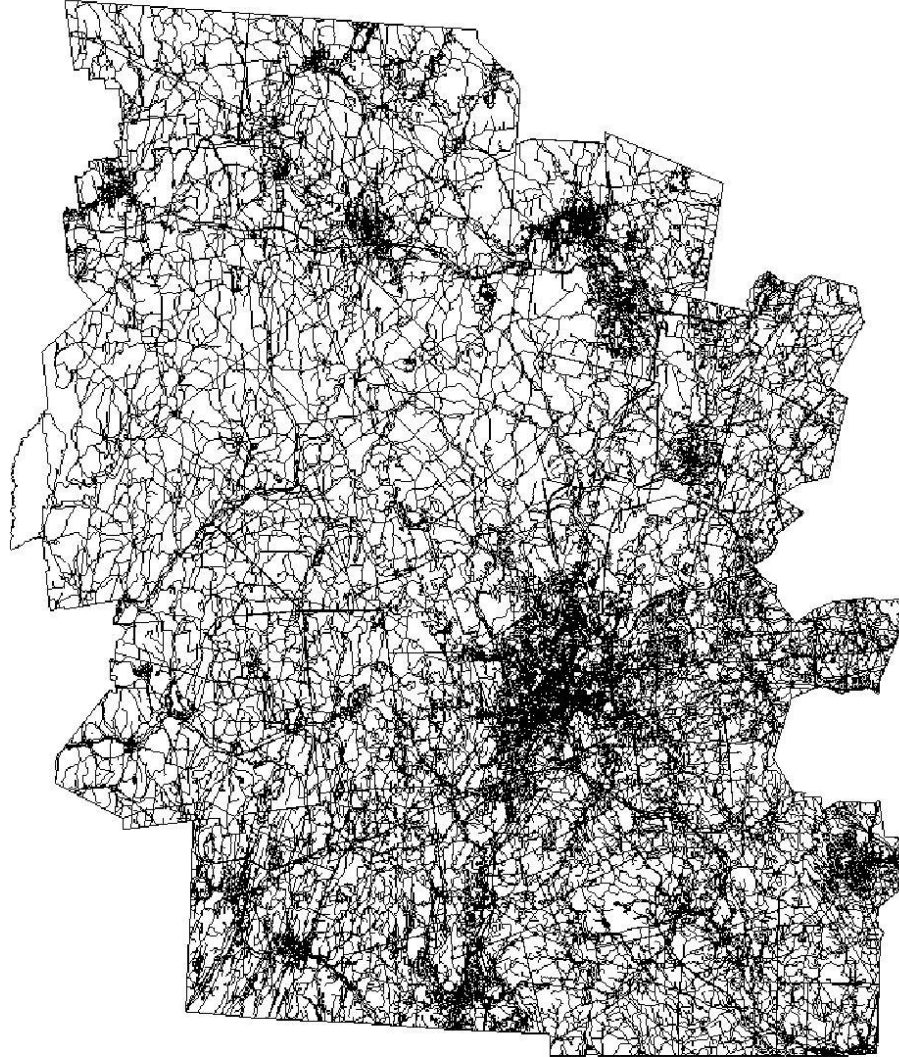


Figure 13: Image of Worcester Network

Error Generator

The framework allows for users to input truth data and to generate error on top of the truth data. This modeled data is used to test the effectiveness of different trackers. Our error generator currently supports six error types.

Error Types

Position Skew

Position Skew is an error that alters the position of individual points based on input parameters. This error takes in a probability and a strength. The probability determines whether or not the point is skewed. If it is selected, the second step of skewing makes use of the strength input. The point gets moved in the X, Y, and Z dimensions according to a random number between -0.5 and 0.5 multiplied by the strength. This number is recalculated on a point by point basis resulting in each point moving unique distances. Thus by upping the strength a user can add a greater window of variance to the position.

X_Position_Bias

X_Position_Bias is an error that alters the position of individual points based on input parameters. This error takes in a probability and a strength. The probability determines whether or not the point is biased. If it is selected, the second step of skewing makes use of the strength input. The point gets moved in the X dimension according to the input strength. Thus by upping the strength a user can add greater distance biasing.

Y_Position_Bias

Y_Position_Bias is an error that alters the position of individual points based on input parameters. This error takes in a probability and a strength. The probability determines whether or not the point is biased. If it is selected, the second step of skewing makes use of the strength input. The point gets moved in the Y dimension according to the input strength. Thus by upping the strength a user can add greater distance biasing.

Time_Skew

Time Skew is an error that alters the recorded time of observation of individual points based on input parameters. This error takes in a probability and a strength. The probability determines whether or not the point is skewed. If it is selected, the second step of skewing makes use of the strength input. The point's time is moved according to a random number between -0.5 and 0.5 multiplied by the strength. This number is recalculated on a point by point basis resulting in each point moving unique times. Thus by upping the strength a user can add a greater window of variance to the time.

Extra_Readings

Extra_Readings is an error that adds in new points based on input parameters. This error takes in a probability and a strength, but the strength is unused. The probability determines if the pair of two points will generate a new point between them. Currently, if the randomly generated number is within the probability, a new point will be generated with the values being the average between the two existing points.

Drop_Readings

Drop_Readings is an error that removes points based on input parameters. This error takes in a probability and a strength, but the strength is unused. The probability determines if a given point is dropped. Currently, if the randomly generated number is within the probability, then the point is removed from the set of tracks.

Error Configurations

In order to support customization our system supports the use of error configurations. Error configurations are made up of the errors mentioned in the section above, and the associated probabilities and strengths. A user may have as many error types as they would like in the configuration. The configuration will run the errors in sequential order, and at the end the modeled data will be returned.

Using Previously Created Error Configurations

Using previously generated error configurations is simple. The system will run every error configuration in ErrorsToRun each time it runs its main method. This means that if a user doesn't remove an error configuration it will run every time without the user doing anything.

Creating New Error Configurations

To create a new error configuration the user will have to make a new ErrorConfig file and place it in the ErrorToRun folder. The error configuration files are XML documents. A basic template XML is given below, if the user wants to add more errors then they simply added more <error.ErrorSpecification> and fill out the fields with the type of error they would like to generate.

```
<error.ErrorConfig>
  <name>Insert Name Here</name>
  <errorList>
    <error.ErrorSpecification>
      <type>Type of Error Listed in Enum</type>
      <strength>Insert Strength</strength>
      <probability>Insert Probability</probability>
    </error.ErrorSpecification>
  </errorList>
</error.ErrorConfig>
```

Figure 14: Error Configuration Format

An example error configuration file can be found in Appendix C – Sample Error Configuration File.

How to Extend the Error Generator

To extend the ErrorGenerator to support new types of errors, there are two steps:

1. The new error must be added to the ErrorType enum.
2. A new case statement must be added to the manipulateTrackHelper function in the ErrorGenerator class.

How to Import Data with Error

To import data with error into the framework, convert the data into the ErrorTrackStruct format as shown in Figure 15, then place the file in the “lib\ModeledDataToRun” directory.

```
<track.ErrorTrackStruct>
<name>Unique data set name</name>
<truthSource>Name of truth data set</truthSource>
<tracks>
  <track.Track>
    <id>Unique Track ID number</id>
    <dataPoints>
      <track.TrackDataPoint>
        <time>
          <milliseconds>Time value</milliseconds>
        </time>
        <position>
          <x>X value</x>
          <y>Y value</y>
          <z>Z value</z>
        </position>
        <pointID>Unique Point ID number</pointID>
      </track.TrackDataPoint>
    </dataPoints>
  </track.Track>
</tracks>
</track.ErrorTrackStruct>
```

Figure 15: Existing Error Data

Tracker Interface

All trackers in the framework must implement the Track Interface. The interface requires that trackers implement a calculateTrackStruct function that takes in a TrackStruct and an ErrorTrackStruct, and outputs a TrackStruct produced by the tracker.

How to Run Trackers

To run a tracker with the testing data sets, serialize the tracker and put the file in the TrackerToRun directory. An easy way to serialize the tracker is to use the writeFile method in our GenericFileHandler class.

How to Add New Trackers to the Framework

To create a new tracker that can then be added to the framework go to the tracker package and create a new class that implements the Tracker Interface. Then, implement the tracker’s desired output to return in the calculateTrackStruct function. While the calculateTrackStruct function takes in truth data and modeled data as inputs, a realistic tracker will simply disregard the truth data field.

Metric Interface

All metrics in the framework must implement the Metric Interface. The only function that metrics need to implement is calculate, which takes in the TrackStruct and calculates the metric value as a double.

Example Metrics

The framework includes several example metrics, which are listed below.

Accuracy Metric

This metric calculates the accuracy of track inclusions based on truth data

Assignment Accuracy Metric

This metric calculates the percent of true inclusions over the size of the dataset

Average Track Lifespan Metric

This metric calculates the average time length of a reported track

Average Track Update Rate Metric

This metric returns the average time between updates in the reported tracks

False Discovery Rate Metric

This metric calculates the rate of false inclusions in the data against all inclusions

False Inclusion Rate Metric

This metric calculates the false inclusions compared to the true exclusions

Lifespan Similarity Metric

This metric calculates the average lifespan of the reported tracks divided by the average lifespan of the truth data tracks

Matthews Correction Coefficient Metric

This metric calculates the Matthew's Correction Coefficient using true and false inclusions and exclusions

Negative Predictive Value Metric

This metric calculates the percentage of true exclusions versus false exclusions

Number of Tracks Metric

This metric calculates the number of tracks

Positive Predictive Value Metric

This metric calculates the percentage of true inclusions versus false inclusions

Specificity Metric

This metric calculates the percentage of true exclusions versus false inclusions

Target Effectiveness Metric

This metric calculates the percentage of true inclusions versus the size of the track

Track Count Purity Metric

This metric calculates the percentage of the size of the outputted tracks versus the size of the truth data

Track Purity Metric

This metric calculates the number of true inclusions divided by the true and false inclusions and the false exclusions

True Inclusion Rate Metric

This metric calculates the true inclusions compared to the false exclusions

How to Run Metrics

To run a metric with the testing data sets, serialize the metric and put the file in the MetricToRun directory. An easy way to serialize the metric is to use the writeFile method in our GenericFileHandler class.

How to Add New Metrics to the Framework

To create a new metric that can then be added to the framework go to the metrics package and create a new class that implements the Metric Interface or if the metric uses truth data, extend the new class from the TruthMetric class. Then, implement the metric's desired output to return in the calculate function.

For metrics that extend the TruthMetrics abstract class, you can access the true and false inclusions and exclusions by instantiating an InclusionExclusionFactory instance and using its calls to get the desired data.

Appendix A – Sample Truth File

```
<track.TrackStruct>
  <name>CrossStruct</name>
  <tracks>
    <track.Track>
      <id>1000</id>
      <dataPoints>
        <track.TrackDataPoint>
          <time>
            <milliseconds>0</milliseconds>
          </time>
          <position>
            <x>501.6499938964844</x>
            <y>5.0999999904632568</y>
            <z>0.0</z>
          </position>
          <pointID>0</pointID>
        </track.TrackDataPoint>
        <track.TrackDataPoint>
          <time>
            <milliseconds>1000</milliseconds>
          </time>
          <position>
            <x>501.6499938964844</x>
            <y>5.880000114440918</y>
            <z>0.0</z>
          </position>
          <pointID>1</pointID>
        </track.TrackDataPoint>
        <track.TrackDataPoint>
          <time>
            <milliseconds>2000</milliseconds>
          </time>
          <position>
            <x>501.6499938964844</x>
            <y>7.449999809265137</y>
            <z>0.0</z>
          </position>
          <pointID>2</pointID>
        </track.TrackDataPoint>
        <track.TrackDataPoint>
          <time>
            <milliseconds>3000</milliseconds>
          </time>
          <position>
            <x>501.6499938964844</x>
            <y>9.59000015258789</y>
            <z>0.0</z>
          </position>
          <pointID>3</pointID>
        </track.TrackDataPoint>
      </dataPoints>
    </track.Track>
  </tracks>
</track.TrackStruct>
```

```

        <milliseconds>4000</milliseconds>
      </time>
      <position>
        <x>501.6499938964844</x>
        <y>12.270000457763672</y>
        <z>0.0</z>
      </position>
      <pointID>4</pointID>
    </track.TrackDataPoint>
    <track.TrackDataPoint>
      <time>
        <milliseconds>5000</milliseconds>
      </time>
      <position>
        <x>501.6499938964844</x>
        <y>15.640000343322754</y>
        <z>0.0</z>
      </position>
      <pointID>5</pointID>
    </track.TrackDataPoint>
    <track.TrackDataPoint>
      <time>
        <milliseconds>6000</milliseconds>
      </time>
      <position>
        <x>501.6499938964844</x>
        <y>19.540000915527344</y>
        <z>0.0</z>
      </position>
      <pointID>6</pointID>
    </track.TrackDataPoint>
  </dataPoints>
</track.Track>
<track.Track>
  <id>2000</id>
  <dataPoints>
    <track.TrackDataPoint>
      <time>
        <milliseconds>0</milliseconds>
      </time>
      <position>
        <x>5.099999904632568</x>
        <y>498.3500061035156</y>
        <z>0.0</z>
      </position>
      <pointID>50</pointID>
    </track.TrackDataPoint>
    <track.TrackDataPoint>
      <time>
        <milliseconds>1000</milliseconds>
      </time>
      <position>
        <x>5.539999961853027</x>
        <y>498.3500061035156</y>
        <z>0.0</z>

```

```
</position>
<pointID>51</pointID>
</track.TrackDataPoint>
<track.TrackDataPoint>
  <time>
    <milliseconds>2000</milliseconds>
  </time>
  <position>
    <x>6.590000152587891</x>
    <y>498.3500061035156</y>
    <z>0.0</z>
  </position>
  <pointID>52</pointID>
</track.TrackDataPoint>
</dataPoints>
</track.Track>
</tracks>
</track.TrackStruct>
```

Appendix B – Sample sumocfg File

```
<?xml version="1.0" encoding="UTF-8"?>
  <configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://sumo.sf.net/xsd/sumoConfiguration.xsd">
    <input>
      <net-file value="burlington.net.xml"/>
      <route-files value="routes.routes.xml"/>
    </input>
    <time>
      <begin value="0"/>
      <end value="500"/>
    </time>
  </configuration>
```

Appendix C – Sample Error Configuration File

```
<error.ErrorConfig>
  <name>testErrorConfig</name>
  <errorList>
    <error.ErrorSpecification>
      <type>DROP_READINGS</type>
      <strength>1</strength>
      <probability>0.5</probability>
    </error.ErrorSpecification>
    <error.ErrorSpecification>
      <type>X_POSITION_BIAS</type>
      <strength>10</strength>
      <probability>0.6</probability>
    </error.ErrorSpecification>
  </errorList>
</error.ErrorConfig>
```

Appendix B – Test Plan

Test Plan

Our test plan consists of a list of packages that comprehensively tests our code to ensure that the desired result is produced by our code. This document is somewhat abridged, as many of the test cases are repetitive or self-evident. Thus, we aim to explain the more confusing and complex test cases as well as provide a high level view of our overall testing scheme.

Utilities Package

The utilities package stores all of the utility functions that are required for testing. The specific functions in the Testing Utility Functions class are described in detail below.

Testing Utility Functions

The first function is the `makeTwoPointTrack` function, which creates the `twoPointTrack` structure, a simple `TrackStruct` containing one track of two points. The second function is `makeTwoTrackData`, which creates the `twoTrackData` structure, a simple `TrackStruct` containing two tracks of three points. The third function is `makeCrossStruct`, which creates the `crossStruct` structure, a `TrackStruct` with a large number of points in it. It is generated using the `cross1l` example `sumocfg` file. The fourth function is `makeBurlingtonStruct`, which creates the `Burlington` structure, a `TrackStruct` based on the city of Burlington, MA. It was generated using a sumo simulation based on a Shapefile of the city of Burlington. The next three functions create `ErrorConfigurations` with specific properties that are clearly outlined in their Javadoc comments.

Sumo Converter Test

This class exists to test the `SumoConverter` class. It is fairly straightforward, and only has a single test to verify that it still works properly.

Error Package

The purpose of the error package is to test the functionality of our error generator. The error generator is required to turn truth data into modeled data. In order to do this, the error generator adds, removes, or modifies points based on error configurations. There are some elements of randomness to the modifications performed by the error generator, so we test mainly for the quantity of data points, rather than the actual data points.

Error Generator Test

The first step in this file is to set up data used during testing by calling functions from the Testing Utility Functions class. The purpose of this class is to test the different ErrorTypes that can be passed to the Error Generator. To that end, there are two tests for each ErrorType, one with zero probability of modification, and one with full probability of modification. This is done to reliably test both cases of the probability's if statement.

Framework Package

The purpose of the framework package is to execute full runs of our testing framework. The class that handles this testing is the Testing Controller Test.

Testing Controller Test

This class tests a full run of the framework and ensures that the proper output files are generated. To ensure that the run goes smoothly, the test manually adds a truth data set, an error configuration, a tracker, and a series of metrics to the necessary directories prior to the run through.

Metrics Package

The purpose of the metric package is to test the many metrics that we have included in our framework. The class that handles this testing is Metric Test.

Metric Test

This class tests the many metric calculations that are included in the framework. It also tests writing metrics to files. There are individual tests for each of the metrics in the framework, as well as a few generic tests that write and read metrics to files.

Track Package

Coordinate Position Test

This class tests coordinate position data structure used in the framework. There are individual tests for each of the functions included in the CoordinatePosition class.

Covariance Generator Test

This class tests the covariance generator used within the framework. There are individual tests for each of the functions included in the CoordinatePosition class. The test first sets up a trackStruct used during testing. The first test checks that the constructor without an inputted seed works properly. The next test checks that the name of the trackStruct changes after running through the generator by

creating a covariance and then running the covariance and trackStruct through the generator. The next test gives zero probability that the covariance will be manipulated from the added base covariance. The last test has a one hundred percent chance of manipulation to check both branches.

Covariance Test

This class tests the Covariance data structure used in the framework. The tests create different matrices and check that they are being translated into the Covariance structure correctly. There are also tests for the InverseMatrix function that check example matrices against the expected output.

Time Test

This class tests Time data structure used in the framework. There are individual tests for each of the functions included in the Time class.

Track Test

This class tests the Track data structure. There are individual tests for each of the functions included in the Track classes including a test for the sortPoints function that sorts a track's data points by their time values.

TrackStruct Test

This class tests the TrackStruct data structure used in the framework. There are individual tests for each of the functions included in the TrackStruct class.

TrackDataPoint Test

This class tests the TrackDataPoint data structure used in the framework. The tests are primarily unit tests for the different fields and equality conditions in the data structure.

Tracker Package

Identity Tracker Test

This class tests the identity tracker that was used during the testing of the framework. The main test simply makes sure that the TrackStruct inputted into the IdentityTracker is equal to the one outputted. There is also a quick check that the getName() method returns its expected value.

Parameterized Tracker Test

This class tests the ParameterizedTracker used by the framework. To facilitate this test we use a modified instance of the twoPointTrack data structure. The primary modification we make to the structure is we do not explicitly specify the pointIDs, so they default to negative one. We do this so the

parameterized tracker will believe that the points may have been added by the ErrorGenerator. The first two tests correspond to the function that removes points added by the ErrorGenerator. The next two tests correspond to the function that adds points that were removed by the ErrorGenerator. The final two tests correspond to the function that incorrectly splits tracks into multiple smaller tracks. For each of these sets, there is one test where the probability of fixing the points is 100%, and one test where it is 0%.

Truth Association Package

Global Neighbor Test

This class tests that the Global Nearest Neighbor points association algorithm is return the expected results. The test first sets up a truth and tracker data set of TrackStructs to be inputted into the algorithm. The first test runs these through the algorithm and checks that it made the expected matches. The next test adds an extra data point to the tracker data set to test the expected performance of the algorithm when the tracker includes noise. The third and fourth tests add points outside of the algorithm's gating range for distance and time respectively to assurance that the algorithm discards points correctly during the gating step.

Nearest Neighbor Test

This class tests the Nearest Neighbor truth association algorithm. It sets up two simple TrackStructs, an output TrackStruct and a truth TrackStruct, and then checks the associations for expected values. There are also tests to validate the calculations for Mahalanobis distance and time differences.

Track Score Test

This class tests the RealTrackScore system used by the framework to generate true and false inclusions and exclusions. It sets up simple TrackStructs and then runs them through the RealTrackScore calculateInclusionsAndExclusions function and then checks all the inclusions and exclusions against expected values.