MACHINE LEARNING FOR ROAD FOLLOWING BY AUTONOMOUS MOBILE ROBOTS

by

EMILY AMANDA WARREN

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Thesis Adviser: Dr. Wyatt Newman

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January, 2009

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the ~~thesis~~/dissertation of

Emily Amanda Warren

candidate for the Master of Science degree *.

(signed) Wyatt Newman
(chair of the committee)

M. Cenk Cavusoglu

Francis Merat

(date) 9/19/08

*We also certify that written approval has been obtained for any proprietary material contained therein.

Table of Contents

List of Tables

List of Figures

List of Equations

Machine Learning for Road Following by Autonomous Mobile Robots

Abstract

by

EMILY AMANDA WARREN

This thesis explores the use of machine learning in the context of autonomous mobile robots driving on roads, with the focus on improving the robot's internal map. Early chapters cover the mapping efforts of DEXTER, Team Case's entry in the 2007 DARPA Urban Challenge. Competent driving may include the use of *a priori* information, such as road maps, and online sensory information, including vehicle position and orientation estimates in absolute coordinates as well as error coordinates relative to a sensed road. An algorithm may select the best of these typically flawed sources, or more robustly, use all flawed sources to improve an uncertain world map, both globally in terms of registration corrections and locally in terms of improving knowledge of obscured roads. It is shown how unsupervised learning can be used to train recognition of sensor credibility in a manner applicable to optimal data fusion.

Chapter 1: Introduction

Fully developed autonomous vehicles could one day prevent accidents, improve fuel efficiency, give drivers more time to complete other tasks, and increase the mobility of elderly people who can no longer drive themselves [1]. Historically, various nations have believed that smarter cars could benefit traffic safety and efficiency. A good map has long been important to these efforts: early work in Europe relied on beaming high quality map data from control centers into passing vehicles [2], and in 1988 the Japanese Digital Road Map Association formed to standardize a digital map database for vehicle road guidance systems [3]. In 1995 the PANS (Portable Advanced Navigation Support) system ran on the Navlab 5; dealing with a subset of the autonomous driving problem, it used a stored map to aid control when taking curves [4].

Today, GPS* sensors help Personal Navigation Assistants find a vehicle's location [5]; Personal Navigation Assistants only address a subset of the autonomous driving problem (i.e., determining where the car is and where it should go), but they are marketable. More advanced are Cybercars (also called Cycabs); similar in appearance to golf carts, these can drive autonomously but only on their own roads, and not mixed in with normal traffic [6]. In the United States, the DARPA Grand Challenge began renewed interest in intelligent highway systems and autonomous vehicles [7] starting with the first DARPA Grand Challenge in 2004 [8]. The two Grand Challenges took

---

**Note that in this paper, the phrase "GPS" at times indicates the sensors themselves, and in other cases represents an estimate of orientation within the GPS coordinate system.*

place in the desert, with the competitors more isolated than human drivers usually are. Nonetheless, the self-driving cars in the Grand Challenges faced their share of problems, with the second DARPA Grand Challenge's results more successful than the first [9].

In 2007 DARPA hosted its first Urban Challenge, where the contestants faced additional traffic hazards such as intersections and other vehicles. Schools such as Carnegie Mellon, with their converted 2007 Chevy Tahoe [1], and Stanford, with their 2006 Volkswagen Passat [9], entered the race. Case Western Reserve University shared such company thanks to our robot DEXTER, the Deployable EXtreme Terrain Enabled Robot. When ENSCO first built DEXTER; he (for it quickly became more common to say "he" rather than "it") placed sixth in the 2005 Grand Challenge [10].

DEXTER then came to Case Western Reserve University. Team Case included over fifty people, mainly students and faculty, but some people in industry as well [11]. We altered and maintained DEXTER's physical parts while installing a new code base. The vast majority of this code, including all the code discussed in this paper, was written in the LabVIEW environment from National Instruments [12], which organizes code in files known as VIs and compels the programmer to draw logical wiring diagrams. Through Team Case's efforts, DEXTER reached the 2007 DARPA Urban Challenge semi-finals in Victorville, CA, on our university's first try.

Figure 1-1. DEXTER, Team Case's entry into the 2007 DARPA Urban Challenge. Image credited to [11].

A project of DEXTER's size—both in number of files and in number of team members—requires a modular allotment of duties. This paper specifically covers DEXTER's Global Mapper, which built and maintained DEXTER's memory of routes and belief concerning where to drive, referred to as DEXTER's Global Map. Other modules could plan routes or perform localization with the help of this Global Map. Localization generally requires some environmental representation, whether it be grid-like, based on the line segments from range measurements, in terms of the robot's constraints in movement, or, as with DEXTER's Global Map, a sequence of beads [13]. Localization may involve large corrections every so often, or nearly continuous, faster corrections [14]; the density of points in the Global Map (generally one meter apart) allowed other DEXTER modules the option of taking the latter approach.

One of the most common localization problems is Simultaneous Localization And Mapping (SLAM), which builds a map while observing the environment; an example in [15] builds a landmark-based relative map. Landmark matching can be difficult in general if image processing is necessary, landmark data is incomplete, or the dynamic nature of an outdoor environment interferes [16]. The beads for DEXTER's map are in absolute GPS world coordinate system terms, with no dependency on landmarks; regular landmarks may be difficult to find in the barren desert of the competition. While the Global Map can potentially improve online, the Global Mapper calculates an *a priori* Global Map from DARPA-specified goals, so DEXTER relies less on exploration than a robot in a typical SLAM scenario.

This thesis first discusses the construction and communication of information in

the Global Map, describing the actual methods used when DEXTER competed in the Urban Challenge. Then the paper covers the extreme cases of how the Global Map might be modified. On one end of the spectrum, an all update, every point in a map comes purely from incoming sensor data. Such teach-and-playback algorithms have applied to unmanned dump trucks working in mines [17] and supplemented the remote-controlled operation of unmanned heavy tracked vehicles [18]. In the present research, the author tailored a teach-and-playback method to run on DEXTER. At the other extreme is the nothing update; while driving the competition route, DEXTER's Global Map never changed. In this case, DEXTER's initial Global Map must be of high quality, higher than what can be deduced from DARPA's set of specific GPS coordinates contained in a Route Network Definition File (RNDF). To provide such high quality, DEXTER's Global Map was corrected and improved using aerial photography. The RNDF and the Global Map must have some agreement for localization reasons, but either one may be moved to fit the other.

Though testing time did not allow for it, the author also implemented a way for DEXTER to choose whether to overwrite a bead with sensor data or to keep a bead exactly the same while he drove; this all-or-nothing update method was tested in DEXTER's simulator and suggests that additional intelligence may be used in the maintenance of the Global Map.

When DEXTER became a permanent resident of the Crawford Auto-Aviation Museum [19], work continued in LabVIEW simulations. In the first, simplest case, the simulations evaluated map registration improvements aided by presumed human driving. In this scenario, a human driver would competently steer an instrumented vehicle down

the center of a road contained within an *a priori* map. The measurements acquired in this process would be used to alter the initial map. In this process, it is recognized that available GPS sensors (or other means of state estimation) may be imprecise. It is also recognized that initial estimates of the coordinates of beads intended to represent points along the centers of lanes are flawed. By combining both sources of information (measurement and *a priori* mapping) instead of choosing one over the other, bead coordinates can be improved, even with significant uncertainty in both information sources.

With the approach presented in this thesis, the mapping algorithm performed sensor fusion of bead data (cumulative best-fit map data), physical state estimates (absolute localization in the GPS coordinate system), and (eventually) road detection values (relative localization from direct sensing of road surfaces). While cost increases with the number of sensors, multiple sensors provide redundancy when one sensor fails [20], or similarly, when one sensor's reported data is poor. There are many methods of sensor fusion, including equations describing physical systems, Kalman filters, some form of logical reasoning, averaging [21], and even human interaction for complicated needs [22]. In the present research, sensor fusion was performed using optimal combination of signals based on presumed Gaussian-distributed uncertainty. [23].

In Victorville, Team Case found the aerial imagery of the competition routes to be shifted from DEXTER's sensor readings; hence DEXTER's map had to be moved to be used correctly. Because of this incident, work in this paper attempts to show, in simulation, how a coarsely-registered map and collected GPS coordinates can be used to discover map shifts in latitude and longitude. At first, this process also works under the

assumption that the GPS points correspond to driving precisely centered in lanes. In an extension, the presumption of accurate driving is relaxed and information from (imperfect) direct road sensing is incorporated. Road detection can be used for both global map registration and improving the map model in local regions. To combine information from multiple imperfect sources, it is necessary to know the credibility (equivalently, statistical variance) of each source. Among the information sources, direct sensing of the road has the greatest variation in credibility, and it is also typically difficult to estimate the variance associated with a given measurement. Work in the final chapter describes a method to learn how to infer an appropriate measurement variance from indirect cues, leading to more trustworthy sensor fusion.

Chapter 2: All about Beads

Section I: Perceptions of Space

As DEXTER drove through space, he needed some concept of that space—some sense of when Point A became Point B. DEXTER also had goals of GPS points he had to reach, increasing his need for spatial memory. Programmers wishing to help a robot perceive and recall the world around it have many options. At a high level of abstraction, the robot may record space as a collection of excitations in a neural network, for example in a Self-Organizing Feature Map [24], but this approach would have added unneeded processing complexity to DEXTER.

Perhaps the robot is in a scenario where it simply wishes to learn about its environment through its sensors as it moves about. The robot may use feature-based mapping techniques and form its global map out of observations about the objects around it, leaving its own pose implicit but not used directly [25]. The robot may remember a place based on a set of views (sensory measurements) taken from that place [26]. The robot may also learn about its environment from polylines generated by sensors reading its surroundings [27]. However, DEXTER had goals to reach, so he could not function solely as an observer.

A robot's movements on the ground are essentially 2-D (latitude and longitude, with elevation often of less importance to navigation). In a 2-D world, the robot may be told that the world is a series of 2-D partitions, e.g. rectangles that may be algorithmically split in order to help see past obstacles [28]. Cycabs use *a priori* 2-D maps represented as a set of polygonal obstacles [6]. The robot's internal representation may break out of

2-D in order to remember movements on a 2-D ground as a manifold, a technique developed to avoid confusion when learning online about a loop [29]. Or the robot may save survey data in databases of the land arranged in a grid format; the team from Ohio State University used this approach in the 2005 DARPA Grand Challenge [30].

The Urban Challenge problem and its rules greatly influenced the representation used for DEXTER's Global Map. The Urban Challenge came with *a priori* information, a set of latitude and longitude coordinates that not only helped DEXTER know where the roads were, but also gave DEXTER his very specific goals. DEXTER's road detection software was important for avoiding obstacles and staying on the road, but with the biggest obstacles (other cars) constantly in motion, objects could not be used reliably as reference points. A specific latitude value matched with a specified longitude coordinate can be considered a single, 1-D point in space; hence DEXTER perceives his map not as a 2-D grid, but as a series of 1-D beads. The manifold approach in [29] was not required in DEXTER's situation, since we could use our *a priori* coordinates to define a start and an end to any loop.

For DEXTER, a noteworthy point in space becomes its own distinct entity; "noteworthy" here means that the point is useful for telling DEXTER about the shape of a road. Such points may be too dense, requiring more memory storage than truly necessary, or too sparse, causing DEXTER to know little about where to drive in large, in-between spaces. DEXTER deliberately chose points at a density between these two extremes. We called these points "beads," so named because within a lane or intersection the beads have a specific order from start to finish, like beads on a string. This arrangement resembles the representation for Personal Navigation Assistants where

nodes (endpoints) and so-called "shape points" define a road as the segments between those points [5].

Each bead contains latitude, longitude, and the heading that instructs DEXTER which direction is forward. When Bead A is followed by Bead B within any set of beads, Bead A's heading points towards Bead B. Beads are extendable and they can (and did) hold additional information. Beads may be stored and read in various file formats, such as XML, so the work done to generate initial beads can be saved, updated beads can be used for future runs, and hand-picked beads can be fed into the Global Mapper initially.

Section II: Generating Beads and Creating an Initial Map

To understand bead generation, it is necessary to understand the organization of the Global Map. Beads are contained in a four-part structure. First, there are lanes, contained in the Array of Lanes and sorted by Segment ID, then Lane ID, where one segment may have multiple parallel lanes. A complete street with all its lanes is a segment. Next, there are Intersection Beads, containing bead paths from one lane to either another lane or to a zone (parking lot). Intersections are arranged first by the segment where they begin, then by the lane where they begin, then by an Exit ID. Each intersection contains information concerning the lane or zone the intersection leads to, and it is this information that classifies the different intersections that start in the same lane. Imagine a four-way stop with three legal paths (excluding U-turns). To DEXTER, continuing in the same lane does not require an intersection, but entering a new lane does. The two paths shown in Figure 2-1 are two separate intersections, despite the fact that they start from the same place. Zone exits are arranged just like intersections, except that they start from zones instead of lanes. Finally, the Zone Array lists the perimeter points

and parking spots of each zone (parking lot).

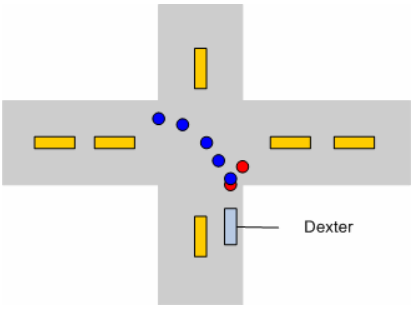


Figure 2-1. The light blue box represents DEXTER. Each set of colored dots is a different intersection in DEXTER's memory, distinguished from each other by their destinations. The path forward is not an intersection, but instead counts as more beads in the same lane.

The data in the initial map depends on the information available from the DARPA Urban Challenge RNDF (Route Network Definition File) [31]. RNDFs give the GPS coordinates for DEXTER to follow and IDs which arrange those points in order and in different pieces of the map. However, RNDFs tend to be sparse, so while surveying efforts may provide detailed maps to Personal Navigation Assistants [5], there is no such luxury here.

Once the RNDF is loaded, generating initial beads starts with Add_Interpolated_Points.vi. From there the LabVIEW code has two main branches: Initial_Beads.vi and Init_Intersection_Interpolation.vi. Initial_Beads.vi iterates through all the segments in the RNDF data, through all lanes within segments, and through all initial waypoints within lanes. The VI collects the three layers of IDs (segment, lane, and waypoint) along with the given GPS points to create the Array of Lanes. Each waypoint becomes its own bead in this process. In parallel, Initial_Beads.vi calls Process_RNDF_Zones.vi, which simply copies all the spots and perimeter points of every zone into the Zone Array.

With just the waypoints from the RNDF, the Array of Lanes is too sparse.

Naively connecting the dots between these waypoint beads could give DEXTER off-road paths and pointed corners to drive. So, Add_Interpolated_Points.vi adds points to the lanes by splining between waypoints. (The spline code was written by Scott McMichael). Generate_Spline.vi interpolates points between but not including existing RNDF waypoints, using a spline calculation that allows for curves. Splining requires the boundary points to have headings; since this data is not provided in the RNDF, it is calculated by comparing the relative positions of consecutive waypoints.

The number of beads generated by splining depends on the distance between the boundary waypoints, with approximately one bead per meter, though an exact number of splined beads may be forced. Fill_Out_Beads_from_GPS.vi then fills beads with default values for some data, Insert_Bead_Headings.vi gives the beads headings based on comparisons with neighbor beads, and Reset_Bead_IDs.vi gives each bead a unique, in-order ID.

Init_Intersection_Interpolation.vi handles the intersections in the map structure. The RNDF data name the lane exits and their entry points; from this, an Entry Waypoint ID can locate either a zone perimeter point or a lane waypoint. The GPS locations and headings of the exits and entrances provide boundaries for an intersection. As with the Array of Lanes, Generate_Spline.vi finds points between entrances and exits, and again this data is packaged into beads with unique, in-order IDs. The entry information is preserved and bundled for each of the lane exits in the Intersection Beads. To help DEXTER drive, any intersection that enters a zone is extended into the parking lot so that DEXTER will not just drive to the perimeter. The extension consists of beads (usually about ten) placed a meter apart following the heading established for the zone perimeter

point. Init_Intersection_Interpolation.vi generates zone exits in the same way as regular intersections, except the "from" boundary point always comes from a zone perimeter point instead of a lane exit point.

Figure 2-2 summarizes the process of bead generation, while Figure 2-3 illustrates how each discussed piece of the overall bead structure fits into an example map.

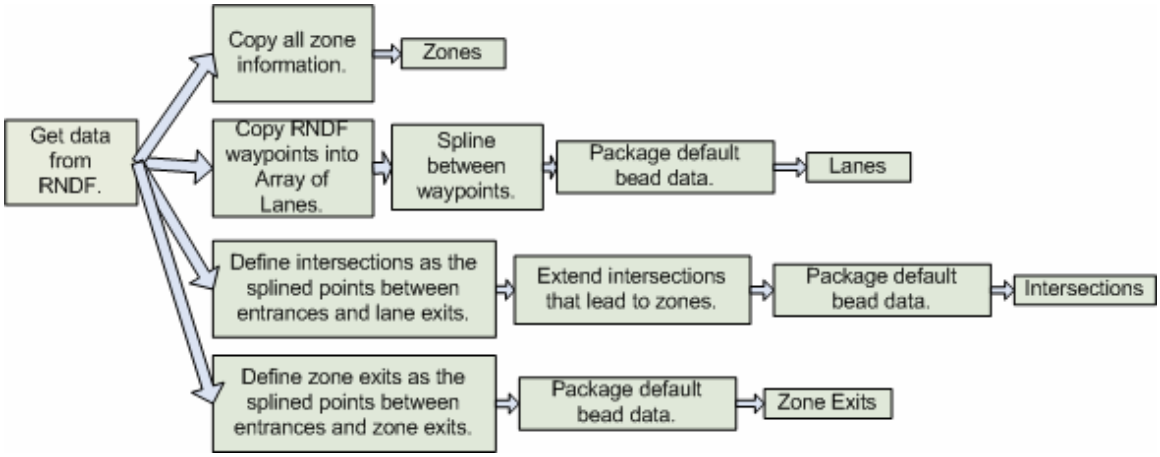


Figure 2-2. The flowchart simplifying the generation of each of the Global Map's four parts.

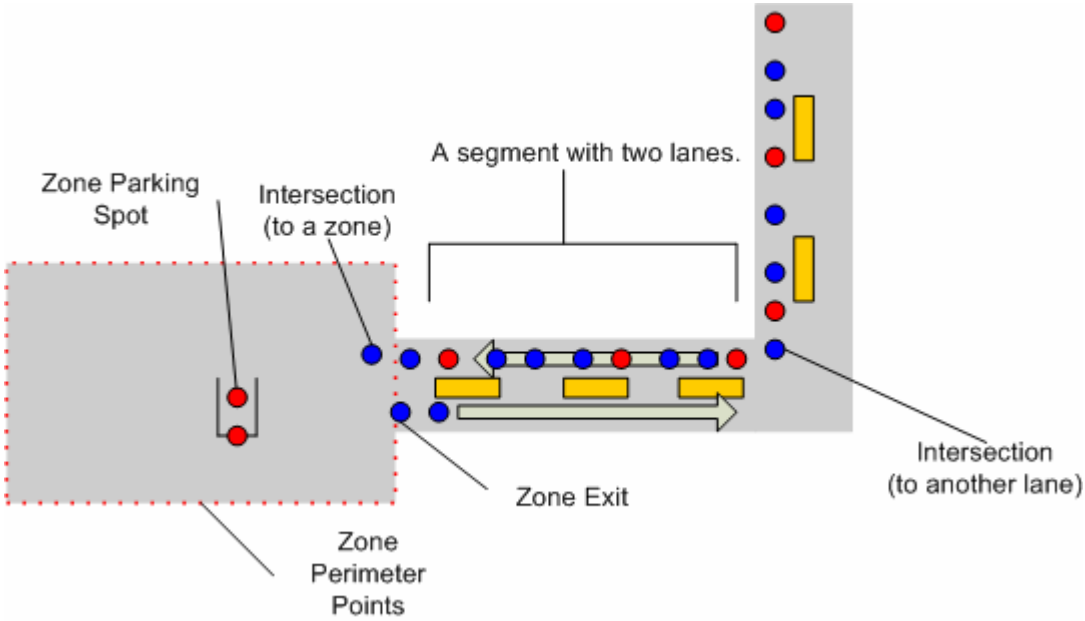


Figure 2-3. Zones have two features (perimeter points and parking spots). Zone exits lead from zones to lanes; Intersections start in lanes and may go to other lanes or zones. The red dots give an idea of what information is in the RNDF, such as the red waypoints

within the lanes. The blue dots show where beads are interpolated, i.e. between lane waypoints and through zone exits and intersections. Note that the difference between an intersection and a zone exit is a matter of traffic flow (shown by arrows on the road).

All interpolations occur once at the beginning when beads are generated. The method of interpolating once contrasts with Ohio State University's method of interspersing points as they were needed during the Grand Challenges [30]. DEXTER avoids the cost of having to spline anywhere on the map more than once, and of having to spline while moving, when many other modules demand processor time.

Section III: Answering Queries

Beads are constructed to support navigation; therefore, communication must exist between the Global Map, which contains and maintains the bead data, and the rest of DEXTER's software. Typically, any one module in the system only needs a subset of the total Global Map at one time. While it is possible to broadcast the entire Global Map constantly and let other modules pick the information they need, the Global Map may be huge in some cases, making such large communications expensive. Thus the Global Mapper must internally locate portions of its data in its multiple bead arrays and make some intelligent decision about what data to publish. If the Global Mapper were solely responsible for that decision, it would have to watch the robot's progress and deduce its location. However, this would blur the responsibility of the Global Mapper, which was intended to behave as memory rather than part of awareness. To make the Global Mapper focus on handling bead management, its communication responsibility was to reply to queries. This approach completely hid the localization problem from the Global Mapper, thus keeping the code more modular. Scott McMichael wrote some of the code that answers queries, while the author of this paper wrote other sections.

A query asks for a certain piece of the map. Queries come rapidly, and a module may re-ask similar or identical queries as much as required. Query answers must be timely in order to provide help with navigation. For these reasons, the less reliable but faster UDP protocol transmitted both queries and query responses. Queries entered a queue upon arrival and the Global Mapper processed them in FIFO (first in, first out) order.

Each query was phrased using the following guidelines. First and perhaps most importantly, the Query Type tells the Global Mapper how the query should be processed. The Query ID distinguishes queries, and can be used by other modules to match questions with answers, since the Query ID carries through to a query's response. Two sets of segment, lane, and waypoint IDs, A and B, give localization information; some queries do not use Location B. The GPS Coordinate gives the location from which the answer beads should start; this is not necessarily the current position of the robot. For instance, the GPS Coordinate in a query may be some distance ahead if the module making the queries is looking towards the future. The GPS Coordinate must ultimately resolve to a valid bead on the road, similar in theory (though not purpose) to third generation Personal Navigation Assistants where the current location of a vehicle must coincide with some street [5].

The Distance variable in a query gives a measure of how many meters forward ahead of the GPS Coordinate the answering beads should cover. In the query response code, each meter in Distance corresponded to one bead to report. This served as a good rule of thumb, since beads were spaced about a meter apart at the outset and bead updates were not used in the final version of DEXTER. If bead updates change the bead spacing,

the query might not get the number of points it expects, though navigation modules could call additional queries to fetch more points. Distance Back works the same way as Distance, only in the reverse direction of an Array of Beads. Finally, the "Reverse?" flag, when true, lets the Global Mapper know DEXTER is traveling backwards in a lane, meaning that the beads should be traversed in the reverse order.

The reverse situation can occur when DEXTER is driving on a two-lane road with lanes going in opposite directions. If there is an obstacle in DEXTER's current lane, he has to drive the parallel lane backwards in order keep going in the correct direction and reenter his true lane past the obstacle. See Figure 2-4 for this example.

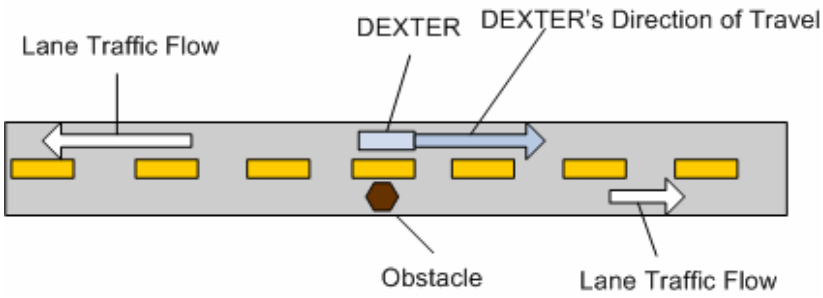


Figure 2-4. In this example, the white arrows show the directions of the lanes. DEXTER must ask for beads in the reverse order so that he can navigate around an obstacle.

Every query in Global_Mapper_Main.vi went through Global_Mapper_Query_Check_v2.vi in order to catch malformed queries. The VI calls Valid_Waypoint_ID.vi in order to check that the first segment, lane, and waypoint ID in the query are in range for the current RNDF. (The second set of segment/lane/waypoint IDs is also checked for Query Types where it is used.) Global_Mapper_Query_Check_v2.vi also calls GPS_Coordinate_Check.vi to make sure that the GPS Coordinate in a query is some reasonable distance from a representative point in the current map. A global variable stored the reference coordinate, chosen as the first bead in the first segment's first lane. This step is important, for a GPS point can

often come in from a sensor as zero degrees latitude, zero degrees longitude; this indicates a drop in reliable data, unless the robot actually is operating off the coast of Africa. Any query errors were eventually returned in the query's answer without beads, since knowledge of these errors prevented the Global Mapper from searching for beads it could not find. A de-queuing error found separately by Global_Mapper_Deque_Query.vi, which happened (for example) when the queue was empty, skipped the query-processing steps and returned to de-queuing.

The author worked on processing a few of the Query Types. The first and most basic, "Get Upcoming Road Points," indicates that the querying module wants information pertaining to a lane. If the incoming query is well-formed, Return_Lane_Beads.vi retrieves the relevant bead list. Segment and Lane IDs in the query locate the bead array corresponding to the appropriate lane. A more refined search typically starts at either the beginning of the Array of Beads or the end; the latter case is used when DEXTER intends to drive the lane in reverse. However, if the Array of Beads for this search and this query matches that requested by the previous query, the refined search begins at the current query's expected Waypoint ID. The assumption in this special case is that DEXTER uses the first query to find out about the lane the robot wishes to enter, but then uses subsequent queries to continue through the middle of that lane. DEXTER's guess of where in the middle of the lane to look is more sophisticated than assuming a lane endpoint.

From the starting point of the search, Find_Nearest_Bead_v2.vi finds the closest bead (in pure distance) to the query's GPS Coordinate. Every point may be checked, even if tests must loop over in the bead array to do so. As a way to save time,

Find_Nearest_Bead_v2.vi stops searching when the beads stop getting closer to the GPS Coordinate and start getting farther away—provided that the winning bead is within some minimum distance. This extra test can be necessary depending on the shape of the lane. Imagine a lane shaped like a "U." The GPS Coordinate is near one side of the "U," but the search starts on the opposite side. In this case, the list of beads would have their distances to the GPS Coordinate shrink then grow before even reaching the curve of the "U." Obviously the search should continue, but for that to happen the lateral distance to the point must be considered, here in the form of a distance cap. See Figure 2-5 for an illustration of this concept. Note that even if it has to search an entire Array of Beads, Find_Nearest_Bead_v2.vi always returns a winner. Overall, Find_Nearest_Bead_v2.vi performs a version of Point-to-Point Map Matching, but since the queries unambiguously name the lane of interest, common pitfalls of Point-to-Point Matching that lead to road confusion are easily avoided [5].

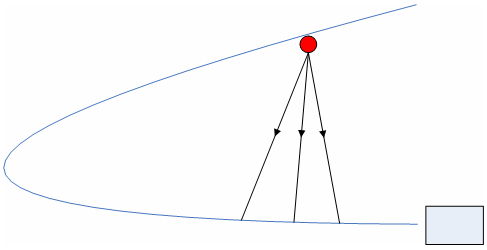


Figure 2-5. The rectangle represents a robot. The red dot is the GPS Coordinate magnified. The black lines show that the robot will get closer to the red dot and then farther away before it even reaches the curve. Hence, the search should continue in this case.

Return_Lane_Beads.vi then finds the bead index containing Waypoint ID B. Either the number of beads between the query waypoints or the number of meters in the query Distance, whichever is lower, is used to decide how many beads to report. Select_Beads.vi finds the beads to return by iterating through the Array of Beads and picking out the bead index returned by Find_Nearest_Bead_v2.vi and the beads after it

(or before it, if in reverse). If the VI reaches the end of the Array of Beads before getting all the beads it was meant to report, Select_Beads.vi passes along the shorter list; most lanes lead into intersections, so looping over in the Array of Beads would give incorrect data in the query response. Imagine a perfectly straight, vertical lane; with looping in Select_Beads.vi, beads in a query response could lead out the bottom and back into the top, jumping a distance which would only confuse a robot and send it in the wrong direction.

After Return_Lane_Beads.vi finds the beads, Global_Mapper_Main.vi calls Beads_to_Road_Description.vi. This VI packages the beads found by Return_Lane_Beads.vi into the approved query response format, which wraps query errors and the beads themselves with a host of other information about the RNDF. The confidences of all beads in the response are included. The official Query Response structure then passes back to the requesting port and address via UDP. The Query Type "Get Road Points Forward and Back" calls Return_Lane_Beads_(forward_and_back).vi, which works the same as Return_Lane_Beads.vi but does the extra work of selecting beads behind the GPS Coordinate as well and stringing all the beads together. This query can be useful if the GPS Coordinate in the query is approximate and therefore a more complete answer comes from looking on both sides of it.

The Query Type "Get Intersection Points" progresses like "Get Upcoming Road Points," only it calls Return_Intersection_Beads.vi instead of Return_Lane_Beads.vi. Here, Location A is the point to reach at the end of the intersection, and Location B is the point where the intersection originates. Hence Location B, compared with the RNDF, decides whether an intersection is in the Intersection List or the Zone Exits. From there,

the code finds an Intersection Bead List that matches the start and end segments/lanes. Find_Nearest_Beads_v2.vi and Select_Beads.vi again take a bead list and return the appropriate subsection. Figure 2-6 summarizes the basic steps needed to answer a generic query.

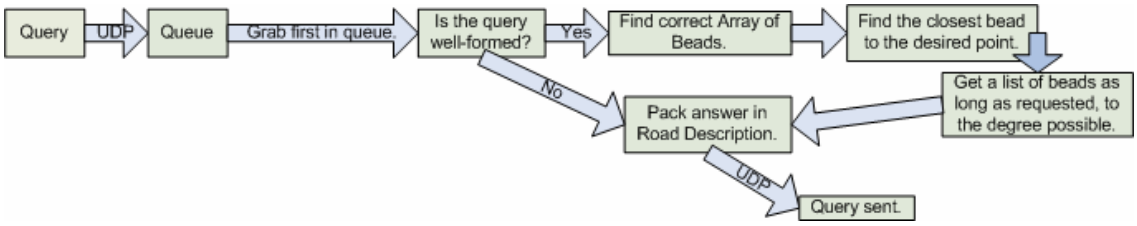


Figure 2-6. This flowchart highlights the important steps taken by the Global Mapper to answer a query.

Zones do not have paths, and are thus much simpler to return; the Query Type "Get Zone Description" calls Get_Zone_Description.vi, which just finds the zone using the Segment ID from query Location A, and returns the entire Zone Description. The parking spots and perimeter points of a zone need not be divided into subsections.

Section IV: Bead Extensions

One advantage of using the bead structure is that information about a point in space can be extended beyond latitude, longitude, heading, and Bead ID. Certainly latitude and longitude locate the point, thus creating a relationship between the bead and real space. The heading defines a relationship to the next bead, which may be helpful to a robot that needs to decide where to go next. Bead IDs also define order between beads in the same lane or intersection. Beads may foster more and more relationships, since an individual bead holds a cluster of data types.

Bead relationships to the map structure can be useful to other modules. The IDs for the overall bead structure (segments, intersections, lanes, and zones) are explicit in the way beads are stored, but if necessary beads themselves could also contain these

references. A bead may also record some measure of its placement along its array. For DEXTER, a bead reported its distance from the start of its lane, where such distance was along the lane instead of along the most direct route from a bead to the lane's start. Other modules also wanted the maximum speed that the robot could travel through a bead, which was parsed at initialization from DARPA's Mission Data File (MDF). RNDFs label checkpoints, which indicate possible goal points that should probably not be moved, or may otherwise warrant special attention. Corresponding checkpoint beads contained flags of their status. Another flag noted mutability, based on the idea that some special points, such as the beads at a stop signs, may need to be prevented from moving. This flag could also be used for a lane that should stay constant, or for testing update code.

Again for the sake of other modules, relevant beads held an Original Waypoint ID. All points in an RNDF are waypoints, and the beads derived from that file (as opposed to the beads interpolated by the Global Mapper) kept a record of the waypoints from which they were created. The rest of DEXTER used these waypoints to understand the map, and included Waypoint IDs in queries to the Global Mapper. Hence, keeping an Original Waypoint ID in a bead allowed beads to be matched up to another module's concept of the robot's current map.

When updates are used, beads may hold trending information related to their previous states. The last change in latitude, longitude, or heading may be saved, as well as some form of total change and the original positions. Possible uses of these bead elements include flagging large changes which may be in error and identifying problem areas within the map. Smart updates also require a bead to have some measure of confidence, so that the certainty of starting beads and of past updates can be a part of any

update algorithm. Confidence may be a number, or an enumerable type.

Team Case considered another extension, which was designed but not used. In this extension, a bead records references to notable fiducials visible from its position. In the proposed design, the fiducials would have been listed on their own, so that all of their position information need not travel with the beads, only their unique IDs. While the beads are extendable, they must also pass over UDP, so how the size of a bead cluster affects communication must also be considered. However, should the distance between the fiducial and the bead be required, that distance too may be saved in a bead cluster's array of Fiducial Indices. Again, landmarks may lack reliability in the desert, but such an extension would still maintain a memory of latitude/longitude positions and not use the landmarks as a crutch.

For an example of how to use these indices, assume a robot is located on a particular bead. From there, the robot can see, with cameras or other sensors, a nearby tree. The bead would contain a reference to the tree via a unique ID, and thus the robot would expect to see the tree when driving through that bead. Any given bead may have a list of several trees, telephone poles, or whatever else the terrain may provide, though this method would be limited in flat, desert areas without many features to detect. While driving, the robot could use the fiducials referenced by its current bead(s) to verify its position. In the other direction, fiducials seen by the cameras could be used to help locate the correct beads from the current list, thus performing some measure of localization.

Chapter 3: All Updates vs. Nothing Updates

Section I: The All Case, Teach and Playback

What is meant by an "all" update? In this context, it means that the bead coordinates are only poorly known, but that the robot has reliable information from direct sensing. Because of the disparity in quality between the two, the best course of action is for the beads to remember where DEXTER drove. In the most basic version of an all update, DEXTER records verbatim the latitude/longitude coordinates that he passes while traveling in joystick mode, and then uses those recorded beads to retrace his steps. In this extreme scenario, the beads are as poor as they possibly can be, because until DEXTER drives, the bead coordinates are unknown; alternatively, bead coordinates could exist with such low confidence that they would be completely ignored and overwritten. The saved path is dictated by user choice and therefore no map structure is necessary during recording; essentially, DEXTER just remembers one long lane. Queries still exist to request portions of that lane.

The author wrote code to address this scenario so that it could be used if needed in parking lot demos (defining a map can be difficult in unstructured space). The main file, Record_and_Playback.vi, handles all the main modes of this bead update method, most notably the record (or teaching) phase and the playback phase. These phases are completely separate. There are no queries needed for navigation when DEXTER is under human control, as well as no data to report; hence the map is completely passive during recording. Conceivably, points could be recorded while other points are played back, however the results may be undesirable. DEXTER's bulk in the real world often caused

him to slightly miss his marks, so constantly re-recording those goals would have likely led to drift. Keeping playback completely separate also increases clarity in operation. Since driving DEXTER by joystick can be time-consuming, the main file allows a bead path to be saved to a file and reloaded from one. Record_and_Playback.vi contains a number of helpful buttons for running these different pieces of functionality. The array Recorded Beads contains the path that DEXTER will follow.

The first button in Record_and_Playback.vi is labeled "Record Phase." This button loads new beads into the Recorded Beads array by calling Record.vi. Record.vi reads GPS coordinates from the Physical State Observer and checks to make sure that the coordinates it saves are at least a meter apart (otherwise the beads would be needlessly dense). This provision prevents the string of beads from filling with identical points while DEXTER remains idle. Obviously, the check is not needed for the very first GPS point collected.

During a joystick run, the Physical State Observer may time out, encounter errors, or report invalid points; error checking ignores any information produced under these conditions. The GPS coordinates (each consisting of latitude and longitude) are the information of interest, and each worthy point is copied over to a bead. By recording a list of points as the playback goals, this teach-and-playback algorithm may be classified as "target point following" [17].

For the purposes of record and playback, most of the other bead information uses defaults (headings are ignored, for example). However, headings are reported by the Physical State Observer and could be copied if necessary for some similar application. The record phase runs until the user un-clicks the button of the same name. The

possibility exists that a recorded set of beads may grow too large for the available memory (which will never be infinite); however, this was not an issue for the tested scale.

The other side of recording is the playback phase. Pushing the "Playback Phase" button in Record_and_Playback.vi calls Send_Playback.vi to replay all of the Recorded Beads. During playback, the code listens for queries on a UDP port. At this stage, a map is required for interface reasons. Other modules phrase queries in terms of an RNDF, so an RNDF of the relevant area is needed to check the query. The most relevant query check makes sure that the Recorded Beads are in the same general area as the RNDF, and therefore DEXTER's recorded points are in the general vicinity of where the user would expect.

The points on the dummy RNDF need not have any overlap with the Recorded Beads; the RNDF merely serves as a kind of ballpark figure. The Route Localizer module dictated that DEXTER had to be somewhat near a lane in order to see the Recorded Beads associated with that lane, even if those beads were nowhere on the lane. A solution to this requirement was to put the recorded path inside an RNDF loop. Stop points caused DEXTER to stop when he was only near a lane, not on it, so an RNDF for teach and playback should not contain stops. Once Send_Playback.vi receives a well-formed query, the particular beads to send are chosen by Beads_to_Play_Back.vi.

In Beads_to_Play_Back.vi, a sample of DEXTER's current GPS location is taken by Take_Sample.vi, which simply returns the first valid GPS reading from the Physical State Observer. Find_Nearest_Bead_v2.vi (detailed in Section III of Chapter 2 on answering queries) locates the element of Recorded Beads closest to DEXTER's current position. This functionality means that DEXTER need not be at the beginning of his

recorded path when playback begins. However, DEXTER should start near enough to some position in his recorded path so that he can navigate to the path without having to cross a large, bead-less expanse. Because the RNDF is only approximate, Beads_to_Play_Back.vi must perform this minimal localization.

Beads_to_Play_Back.vi compares the distance from the last element in Recorded Beads and the first; if this distance is less than ten meters, then the points can loop over and all beads may be sent to the Observers; if not, the gap is too far for DEXTER to navigate, and the points to report can go no farther than the end of Recorded Beads. The number of beads possible to report is compared with the number of beads requested by the query, and the smaller of the two becomes the number of beads sent.

Due to the interface used, Send_Playback.vi must use Beads_to_Road_Description.vi to package the bead GPS coordinate data into a format the Observers expect to read. From there, the Road Description travels back to the Observers through UDP. The playback phase halts with the push of a button, or when the Recorded Beads run out of points and the start of the path is too far to reach. Figure 3-1 explains how the path shape makes a difference. Unfortunately, a path like the red circle may cause DEXTER to keep going even after playback is stopped. At least during one point in development, the Road Detector did not clear out when the Observers stopped receiving query answers. If any sort of closed loop were present, this other part of DEXTER unrelated to his map would keep him driving on saved data even with no incoming points. When this other module presented this problem, DEXTER himself needed to be stopped before the playback phase was turned off.

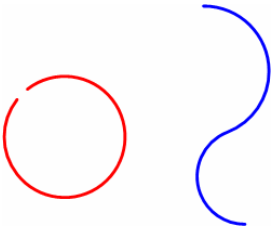


Figure 3-1. In the red path at the left, the end is close enough to the beginning that the beads to play back can loop over, and playback can tell DEXTER to keep driving the circle. In the blue path at the right, DEXTER does not know how to navigate from one end back to the other, so playback stops feeding points to the Observers and DEXTER must stop since he does not know where to drive.

The other three buttons in Record_and_Playback.vi are for storage and retrieval, so that recorded paths can be played back later without DEXTER having to relearn them. Figure 3-2 shows all of the buttons the user has available. The Store in File button calls Record_Points_to_File.vi to write all of the current Recorded Beads into a binary file. The counterpart button, Get from File, reads points from a binary file and loads that data into the Recorded Beads. Finally, the Save to CSV File button calls Recorded_Beads_to_CSV.vi, which takes only the latitude and longitude information from each bead and writes it to a CSV file so that users may observe this data in programs such as MATLAB. Figure 3-3 summarizes how the list of Recorded Beads can be passed around and used.

Figure 3-2. The user has five buttons to choose from when teaching and playing back beads. Output errors are also available in the interface.

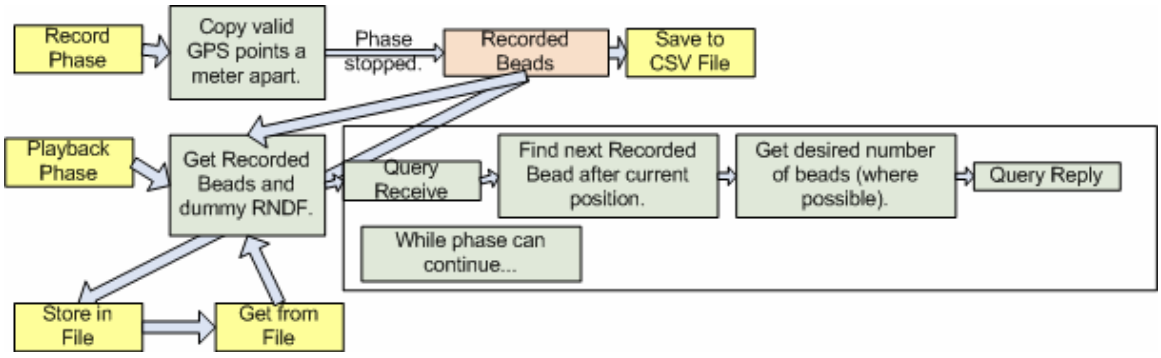


Figure 3-3. How the Recorded Beads, highlighted in pink, are created and used. Buttons used in the teach-and-playback method are highlighted in yellow.

With dexsim, a program written by Christian Miller to drive DEXTER in simulation [32], the user could employ a computer's mouse to simulate the joystick needed for recording, and then watch the simulated DEXTER move on its own during playback. Figure 3-4 shows the simulated DEXTER driving a figure eight that clearly

does not follow the lanes on the map (and in the RNDF); the figure eight was drawn manually. In the spring of 2007, DEXTER himself used this teach-and-playback code in the Mount Sinai parking lot in Cleveland. The RNDF used followed the suggested one loop format with no stops. The code ran exactly as intended on the first try, with DEXTER following a misshapen loop originally drawn by hand.

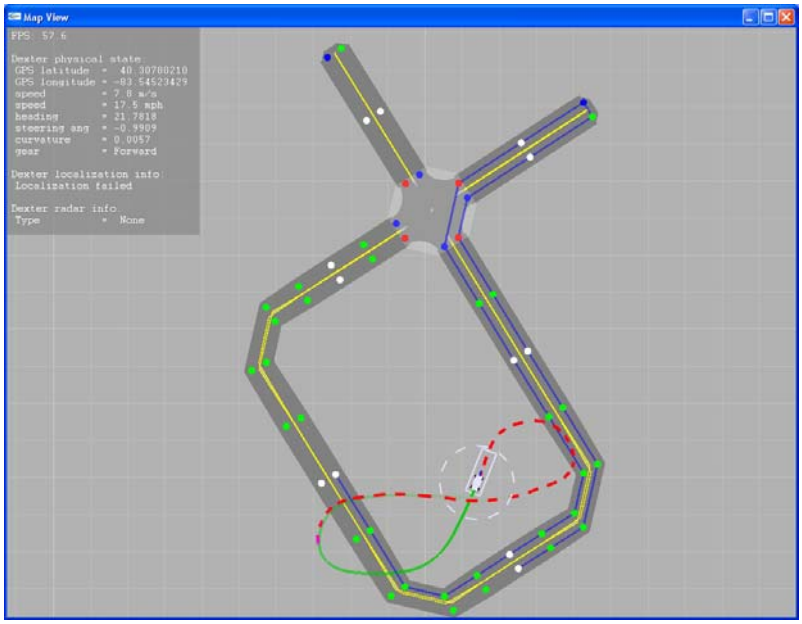


Figure 3-4. DEXTER following a manually drawn figure eight by using Record_and_Playback.vi. The green trail shows where simulated DEXTER has just driven, while the red dotted line shows where the simulated robot is going next.

Section II: The Nothing Case, Altering Maps

What is meant by a "nothing" update? In this context, it means that the beads never move; DEXTER practiced nothing updates in competition. During a nothing update, the Global Mapper still keeps active and reports beads to other modules. But no update code runs as the robot drives, and the beads are constant regardless of what any sensors might say. However, these conditions do not mean that a nothing update involves no more work than generating beads from an RNDF and reporting them to the rest of

DEXTER (see Chapter 2 for detailed descriptions of these procedures). Rather, a nothing update means that the *a priori* beads must have very good quality.

Beads may potentially come from maps, which have never been perfect. In the 1940s, the U.S. National Map Accuracy Standards specified that the accuracy of a map should be within $1/30^{th}$ or $1/50^{th}$ of an inch (depending on map scale), where such accuracy could only be verified by better surveys [33]. Map accuracy has improved since then, and even the terms have changed—now commercial satellite imagery can be as accurate as a half meter resolution [34], while aerial photography offered by the U.S. Geological Survey can reach a resolution of 0.3 meters in some areas [35].

The RNDF format further sabotages bead confidence level. First, RNDFs are sparse, and many beads must be interpolated between them as a guess. Second, if RNDF points come from measurements with a GPS antenna (instead of from a map), these measurements may not align with what DEXTER's antenna sees on a different day. Third, when RNDF points come from satellite (or aerial) imagery, DEXTER's GPS antenna may again not agree with those results. The second and third problems are really just two specific versions of the alignment problem; it is more important for DEXTER's sensors to match the bead map than for that bead map to be absolutely correct.

Hence, a nothing update requires work before DEXTER runs in order to provide acceptable beads. Generated beads have no knowledge of the look of the road, and therefore their confidence is restricted. For DEXTER, Andrew Horchler addressed this problem by writing Java code that displayed beads (written to an XML file) and then allowed manual editing. Several team members spent time using his software to place beads on the middles of roads before DEXTER's semi-final events. This ability was

invaluable for cleaning up curves and intersections.

Yet, not all bead shifting must be manual. In the National Qualifying Event (NQE) that DEXTER participated in at Victorville, Team Case observed shifts (in both latitude and longitude) between RNDF points and the nearby points that the DARPA-distributed aerial imagery believed was on the road. Beads come from RNDF points, so the beads were off the centers of roads as well. The author wrote Shift_All_Beads.vi to move all beads globally.

Shift_All_Beads.vi considers all five global movements: latitude offset, longitude offset, horizontal scaling, vertical scaling, and rotation. For the user's convenience, latitude and longitude offsets need not be entered directly. Instead, the user may specify some number of meters to move east or west (for longitude) and north or south (for latitude). Each meter offset has a matching heading in polar degrees to decide direction; for example, a value of ninety degrees for the longitude heading is due west.

To convert these meters to latitude and longitude, a reference point is required. Find_Gathered_GPS_Midpoint.vi allowed Team Case to choose a particular GPS coordinate at the middle of the existing bead positions. The midpoint may alternatively be calculated with Find_Gathered_GPS_Midpoint_(Calculate).vi, which calculates the midpoint latitude and midpoint longitude from an incoming list of beads, an RNDF, or sensor logs. Once Shift_All_Beads.vi has the GPS midpoint, GPS_Location_from_Given_Position.vi finds the latitude of a new point the requested number of meters away from the midpoint and in the requested direction of latitude shift; the VI is called again for longitude. The differences between these moved coordinates and that of the midpoint give a latitude offset and a longitude offset.

A user can easily understand latitude offset, longitude offset, horizontal scaling, vertical scaling, and rotation. However, moving points is easier to do in matrix math, so first Shift_All_Beads.vi converts variables using Rotation_Scales_and_Offset_to_Matrix_Variables.vi. The offset variables only have to be added to one latitude/longitude pair in order to produce another, so the offsets may be kept in their own vector. However, the angle of rotation becomes a rotation matrix (see Equation 3-1) and this matrix carries the scaling factors. Rotation_Scales_and_Offset_to_Matrix_Variables.vi performs the math in Equation 3-2, and copies the offsets to output variables. The output variables are the following: xi, xj, yi, yj, x offset, and y offset. The angle *x* must be in radians.

$$\begin{bmatrix} \cos x & -\sin x \\ \sin x & \cos x \end{bmatrix}$$

Equation 3-1. The rotation matrix of an angle x, where x is in radians.

$$\begin{bmatrix} \cos x*(\textit{vertical scale}) & -\sin x*(\textit{horizontal scale}) \\ \sin x*(\textit{vertical scale}) & \cos x*(\textit{horizontal scale}) \end{bmatrix} = \begin{bmatrix} xi\ xj \\ yi\ yj \end{bmatrix}$$

Equation 3-2. The matrix produced by Rotation_Scales_and_Offset_to_Matrix_Variables.vi. The rotation angle is x radians.

Once Shift_All_Beads.vi has the matrix result in Equation 3-2 and the two offsets, it parses out each and every bead from the overall bead structure: beads in lanes, in intersections, in zone exits, and in zones must be found separately. Parking spots are not moved, since these must be considered goals to be reached and are not connected with any path. All other beads move by a three-step process. First, the GPS midpoint is subtracted from the latitude and longitude of the bead in order to put the bead in relative terms. Second, Matrix_Math_for_Image_GPS_to_Real_GPS_(Inverse).vi changes the relative position of the point. Last, the GPS midpoint is added back in so that the moved

bead's position will be in absolute terms.

Matrix_Math_for_Image_GPS_to_Real_GPS_(Inverse).vi has a long name for legacy reasons; consider the "Image GPS" as a guess that has to be converted to reality. The relative bead positions in the inputs, *i* and *j*, are added to the *x* and *y* offsets. The inverse (found by LabVIEW) of the matrix in Equation 3-2 is multiplied by the vector created by adding in offsets. The resulting vector is the new relative position which will be put back into absolute latitude/longitude terms and repackaged into a bead by Shift_All_Beads.vi. Shift_All_Beads.vi resaves all the beads it moves; Figure 3-5 offers an overview of the VI.

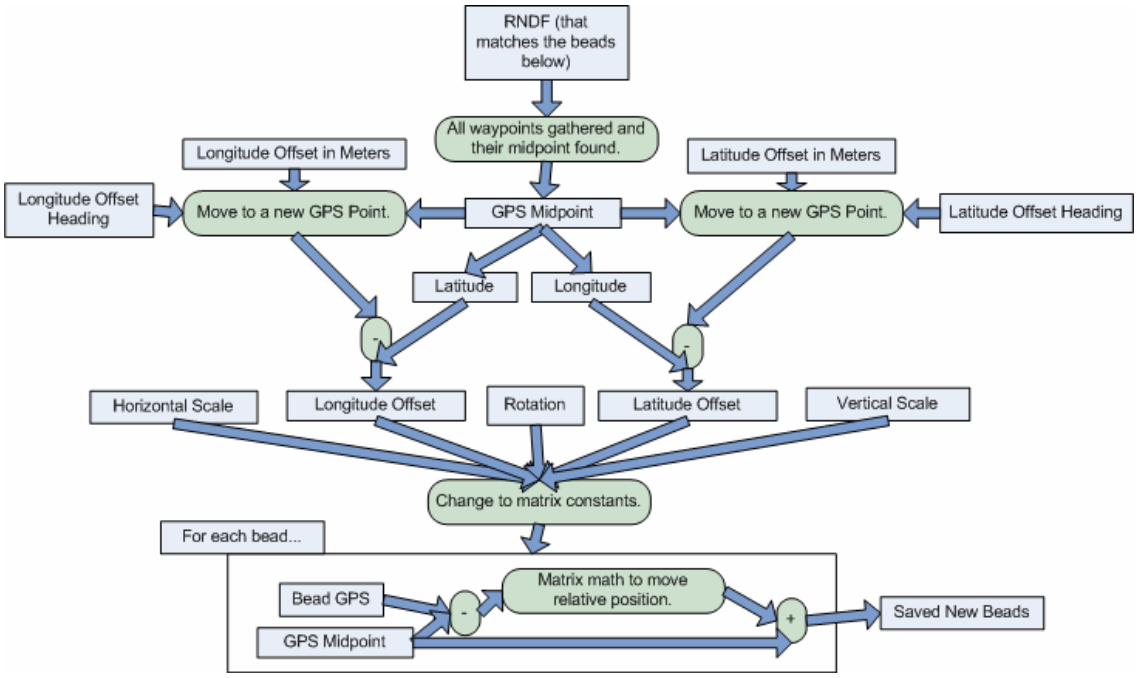


Figure 3-5. A flowchart of Shift_All_Beads.vi, with the GPS midpoint calculated from an RNDF as an example.

Figure 3-6 shows Shift_All_Beads.vi in action, using other Team Case code to display beads on Google Earth. The beads with rotation moved counterclockwise, following the direction in which polar degrees flow. Points are rotated individually relative to a midpoint, which is an estimate, so some change in the path's shape occurs.

Scaled beads show some rotation; since the scale factors roll into the rotation matrix, this is an expected consequence of using stretches and shrinks. "Horizontal" vs. "vertical" scaling is a matter of perspective, especially when used on a diagonal path. Scaling is perhaps the trickiest parameter adjustment, and the user must play with these values in order to meet some goal (as is true in general). The scene in Figure 3-6 is of Case Western University's Engineering Quad.

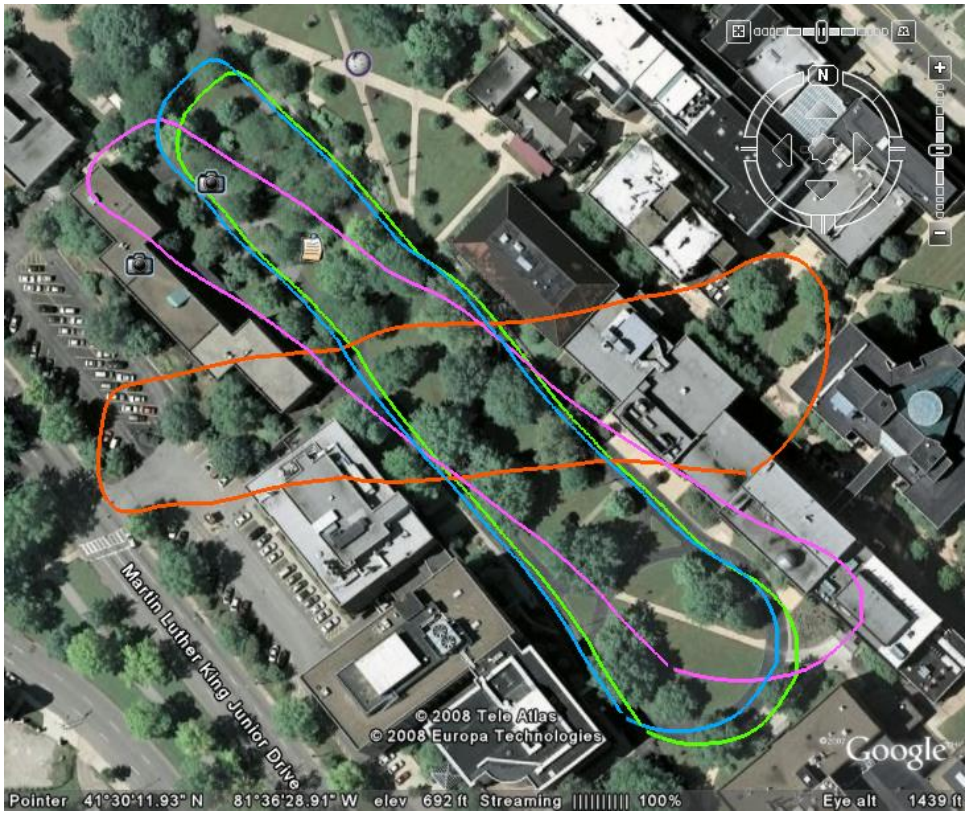


Figure 3-6. The original beads created from an RNDF are in blue. The beads shifted 5.5 meters east and 3.5 meters south, with only offsets used, are in green. The scaled beads (1.2 vertical and 0.8 horizontal, where one would mean no change) are shown in pink. Finally, the orange path is the beads after a forty-five degree rotation.

Shift_All_Beads.vi uses parameters to move the beads in order to try to get a general fit to what DEXTER believes. Our team was not allowed to gather data with DEXTER's GPS sensor on any of the tracks before the events, but we were allowed to drive DEXTER in the area, which confirmed that DEXTER's perceptions matched the

aerial imagery, not the RNDF. (The aerial imagery included surrounding areas that we used for this analysis.) DEXTER's GPS antenna's measurements, and therefore his belief about where the middle of a road was, agreed with the aerial imagery. RNDF points were never guaranteed to be exact, even as goals, but it was certain that DEXTER needed to stay on his road. So to deal with the global shift in DEXTER's map registration, the author moved the RNDF to suit the beads (DEXTER's Route Planner required similarity between the two).

Match_RNDF_to_Beads.vi solves a parsing problem. In one corner sits a set of beads read from an XML file. The beads in intersections or zone exits have no corresponding points in the RNDF, but zone beads are copied directly from an RNDF and lane beads take waypoints from the RNDF. RNDFs have a very precise format, and with knowledge of it, Match_Beads_to_RNDF.vi parses out each zone and each lane's waypoints. Segments, lanes, waypoints and zones have IDs that the beads copy into their organizational structure, and the IDs provide precise identification.

For zones, the perimeter is not worth changing; DEXTER is never meant to drive outside the bounds of a parking lot, and these points are not among his goals. Parking spots are important, however, as goals, and lane waypoints are important to drivability. Using IDs to correctly match the right spots and waypoints (see Figure 3-7), Match_Beads_to_RNDF.vi converts the latitude and longitude information in the beads to text and replaces the previous GPS coordinate text in the RNDF file. Non-GPS coordinate parts of the RNDF (such as IDs) remain the same, and after the latitudes and longitudes are replaced, the RNDF is resaved. Interpolated beads lie within the moved RNDF. Team Case used these moved RNDFs in the semi-finals, instead of the original

RNDFs distributed by DARPA.

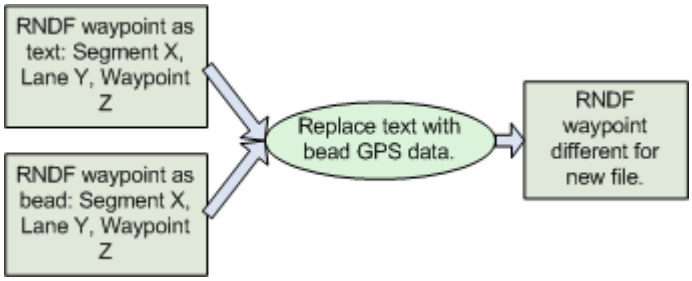


Figure 3-7. The same waypoint is defined in an RNDF as in the bead structure, but the RNDF's GPS coordinate data is replaced as above. The same is done for parking spots.

Figure 3-8 shows how well Match_Beads_to_RNDF.vi works. For this test, Shift_All_Beads.vi created a poor set of beads, which led to a poor RNDF. The original good beads (before shifting) then became the goal for the poor RNDF. Match_Beads_to_RNDF.vi created a new and improved RNDF from the poor RNDF and the goal beads. Paths on the figure vary in thickness in order to show when paths overlay.

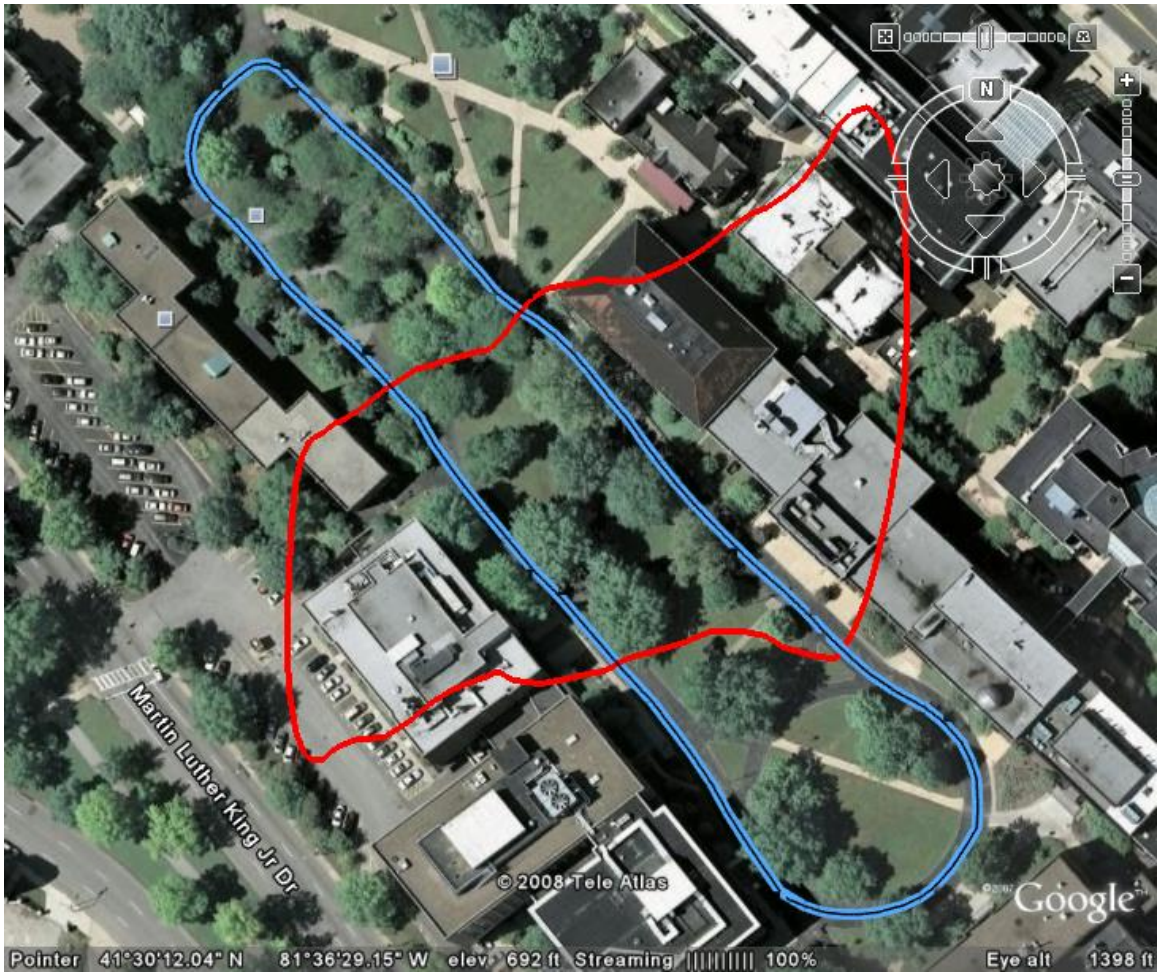


Figure 3-8. The red path is the poor RNDF, which is clearly impossible to drive. The dark band around the quad indicates the good beads. Match_RNDF_to_Beads.vi moved the red RNDF to the light blue RNDF that follows the exact same course as the good beads.

Chapter 4: An All-or-Nothing Update

Section I: Deciding between All and Nothing Online

High quality aerial and/or satellite photography compared with and matched to the behavior of DEXTER's own sensors can provide excellent beads to DEXTER before he ever takes on a course. In practice, Team Case used this ability at the semi-finals to avoid ever having to update a single bead online. The idea of using *a priori* position information is not unique; for example, a feature-based mapping approach may require that the absolute positions of some "anchor objects" be known before extensive map building [25]. Yet, not all robots can rely on *a priori* information. Some so-called "reactive systems" do not require any sort of global map [28]. The popular SLAM problem also defines a robot's environment as a nebulous, unknown quantity [15], and other scenarios employ a robot that must learn about its world by "walking about" inside it [26]. In future DEXTER endeavors, reliable photography may not be provided, and the robot may be forbidden from using its sensors in the area for pre-competition testing.

Though it might be time-consuming, DEXTER's sensors can navigate through areas, and DEXTER's Global Map could memorize his movements. But this extreme is insufficient as well: if DEXTER only used such all updates on a map, then there would be no way to take advantage of *some* known good beads. For example, aerial and satellite photography may be clear in some areas but obscured in others. Hence, our team foresaw a need to let DEXTER make up his own mind about when to replace beads and when to keep his memory intact.

Section II: Guidelines for Updating a Bead

An Array of Beads may define the path of a lane, an intersection, or a zone exit. Zones do not have paths like the rest of the map: their parking spots must be believed as goals to be reached, and robots do not usually drive along zone perimeters. For these reasons, zone points are never updated. Find_Array_of_Beads.vi uses DEXTER's Route Localization information to locate where in the map structure DEXTER last drove, so any update algorithm can locate the correct bead array.

Replacing beads requires DEXTER to collect GPS points in some systematic way. The robot may make maneuvers that invalidate the data it collects with regards to any update operation. When the robot avoids an obstacle, it may not take the path a second time (especially if the obstacle can move). If the robot is driving a lane in reverse, then the robot is either in error, or disobeying the normal motion of traffic in order to move forward at all. Reverse driving is unreliable in terms of repeating paths, so all update code assumes the forward direction and ignores data collected while DEXTER is in reverse.

Collecting raw GPS data is not accurate enough in the field. GPS signals drop and shift at random intervals. And in the case of DEXTER, who had a "wagon handle" to direct him, the robot would not drive exactly over the path his handle followed on curved roads. If DEXTER remembered raw GPS coordinates, and beads were ever updated more than once, then DEXTER could potentially move his remembered path farther and farther from his real goal as the wagon handle forced him to take wider and wider turns. To solve this problem, any update bead method needs a filtered, smooth path of GPS points that should be traveled but are not necessarily driven over directly. In DEXTER's case, the Lane Description from the Observers contained GPS points derived

from combining regular GPS signals, road detection sensors, and the current beads; the Lane Description publishes its own confidence for each point based on the confidences of these pieces of data. The Lane Description is not perfect, however, for it is a list of points that does not update as quickly as DEXTER moves. When collecting latitude and longitude values for updates, choosing the point from the Lane Description closest to DEXTER's current physical position allows the collected points for updates to move forward even when the Lane Description does not.

Another consideration when updating beads is how dense or sparse the collected GPS points that inspire updates should be. The choice translates directly into the density of the updated beads. Beads that are too dense are unnecessary and a memory and computation burden, but beads that are too sparse do not lead to a well-defined path. While other spacing choices are possible, a distance of roughly one meter (and at least that) between examined GPS points (and therefore the beads they affect) was suitable for avoiding the problems at either end of the spectrum. This choice furthers a convention, since the Global Mapper also spaces beads one meter apart upon initialization.

Since the robot's memory of its internal map is always in use, any update method must be careful about when the beads change. For example, one possibility would be to change beads according to the current plan of the path up ahead. The timing of this can adversely affect other modules; when the upcoming path is constantly reevaluated, a rapid and drastic change in the memory for upcoming beads may lead to choppy driving. Combined confidences, as found in the Lane Description, pose another problem. The bead confidences could increase when altered by the Lane Description, but the Lane Description looks at the same points multiple times in succession. Hence, the Lane

Description confidence could increase because of the Lane Description's own previous results (as repeated by the beads). This feedback may continue with the beads believing the Lane Description, and the Lane Description believing the beads, until the modules' confidences unjustifiably approach unity from consistently repeating the same assumption (of possibly low confidence at the start) back to each other.

Replacing beads behind the robot after it has passed them solves these two problems, but not others. The method presented in this paper updates beads as a group (once DEXTER has passed a lane or intersection) in order to avoid the issues inherent in updating beads one at a time. For example, since DEXTER's map was segmented, moving beads while driving past them can erase the distinction of where a lane was supposed to end. Extra memory is then required simply to remember the old boundaries for the purpose of restricting the new ones. These boundaries must be strictly maintained, or a single GPS point may belong to both a lane and its adjoining intersection (for example) simultaneously. Worse, without boundaries the robot may need to back up after finishing a lane's points in order to start an intersection's points. When beads are updated individually, it is also very tempting to update beads in order (based on lane direction). This myopic view ignores the best fit of where new data should replace old. However, searching for the best bead to update with each incoming GPS coordinate could get costly. Moreover, updating beads on an individual, out-of-order basis allows beads to be skipped, which can cause discontinuities.

For these reasons, DEXTER's proposed update method works on a self-contained portion of the map with clear start and end points—such as a lane, or an intersection—after the robot leaves that Array of Beads behind for the time being. The update method

updates a section of a given lane or intersection so that discontinuities created by the update can be restricted to either end of the updated section. These discontinuities occur when DEXTER only drives part of a lane (or other structure), and is therefore only able to update part of that lane. However, since the discontinuities occur in expected places, they can be dealt with systematically.

Updating a lane or intersection all at once allows for the update section that best fits the incoming data to be found once within the lane. It is also simpler to keep that update section in the predetermined bounds, since the original endpoints will not move until they can be compared with the new data. For DEXTER, a VI named Get_Rid_of_GPS_past_Bead_Array.vi drops collected GPS points for the update that are ahead of the last point in an Array of Beads. Since the update method waits for DEXTER to change his lane or segment before recording points for the new lane or segment, the method does not have a chance to grab GPS data before the start of a lane or intersection.

Replacing a section of an Array of Beads requires identification of such a section. Find_Bead_Subsection_to_Replace.vi locates which beads in an Array of Beads should be altered by updates, checking that the last bead index to replace is after the first, and that both the first and last indices are valid for the Array of Beads. Match_GPS_to_Bead_Index.vi finds both the first and last beads of the subsection to be updated (in separate calls). The VI takes in a GPS coordinate ready to cause a bead update and finds the bead just behind it. In this context, a bead that is "behind" a GPS point means that were DEXTER sitting on that bead facing in the direction of the bead's heading, the GPS point would be somewhere in the half-plane ahead of DEXTER, though perhaps off to the side as well. The algorithm does not search for simply the closest

bead, because that scheme does not prevent some beads from being pulled forward in the update while others are pulled backward..

Paths can twist and turn, so if more than one bead in a bead list is directly behind the GPS point (recall the definition of "behind" here), the bead closest to the GPS point wins. Imagine a serpentine path (see Figure 4-1); in that case, one bead would be behind the GPS point (while the following bead would not be) before and after each turn. Hence the choice of closest bead is needed to find the most appropriate match. If no bead index can be matched to the GPS point, Match_GPS_to_Bead_Index.vi returns an error.

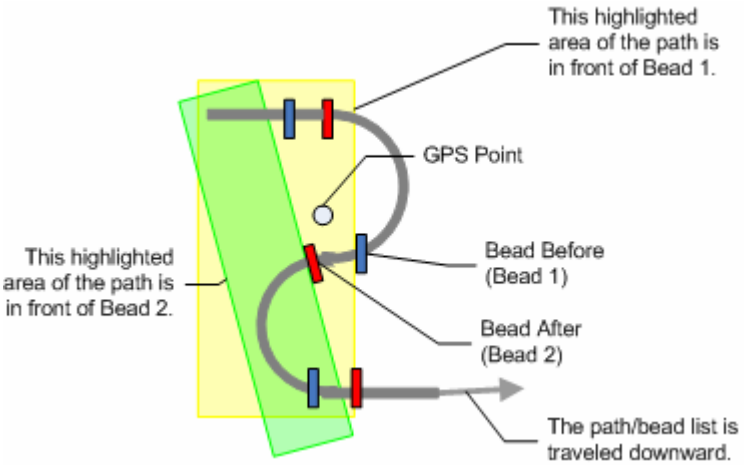


Figure 4-1. The serpentine path is traveled downwards. There are three blue beads (marked) that are "behind" the GPS point; that is, when DEXTER is positioned on top of one of the blue beads and oriented tangent to the road, the labeled GPS point is in front of the vehicle, albeit to the side as well. Each of the three pairs of beads shown contains two beads (one red and one blue) that are sequential along the path. The red bead positions are "in front of" the GPS point, because the GPS point is not within the half-plane in front of DEXTER when the vehicle sits on top of one of the red beads at an orientation aligned with the bead's heading. Match_GPS_to_Bead_Index.vi must use absolute distance to choose among the beads in the blue positions. In this example, the GPS point should match with Bead 1 (marked).

If finding the beads to change throws an error for any reason, no update should occur, since trying to do so would only propagate the error. But this does not mean that beads must always be updated when there are no errors. Finally, LabVIEW offers multi-

threading capabilities, so the update method can record GPS points for the current lane or intersection while updating the last visited set of beads in parallel.

Section III: Running All-or-Nothing Updates

In DEXTER's all-or-nothing update method, every bead, each a representation of a single point in space, can be assigned two levels of confidence: absolute confidence, or none at all. As the robot's memory is "born," some information is instinctual; beads given high confidence are accepted as unassailable, unchangeable fact, a product of centuries of evolution, or in this case, a human judgment call. For example, if a mission states that the sparse points distributed by the officials of the contest were measured with similar equipment to the robot under similar conditions, such points are likely to be believable. These beads maintain their certainty, and they will be used exactly the same every time.

The beads with no confidence serve an opposite purpose. These beads are generated by producing an inferred path between two known points. But for whatever reason, the human with the vested interest in the robot's performance does not trust these beads. Imagine two known points, A and B, on either side of a black box (e.g. a warehouse whose insides are concealed from satellite and aerial imagery). The robot must navigate through that box from one point to the other, as shown in Figure 4-2. With no knowledge of the hidden terrain, the beads generated from A to B assume the most direct route; there is no advantage to predicting a more time-consuming, winding path.

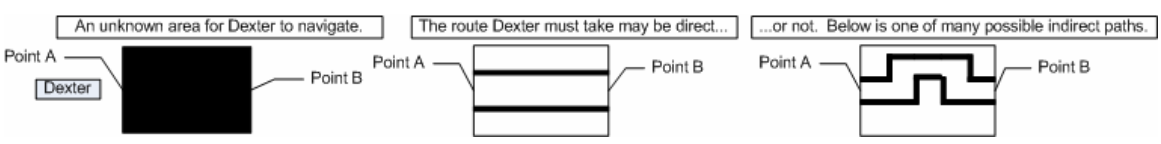


Figure 4-2. When faced with the black box in the first figure, DEXTER must assume the most direct route from Point A to Point B, as shown in the middle figure. However,

obstacles inside the black box may force DEXTER to take a more convoluted path, an example of which is shown in the third image.

Now imagine that the black box contains a maze that totally thwarts the direct route. The robot must rely on non-GPS sensors such as vision, radar, and LIDAR in order to avoid the maze obstacles. As it does so, the robot traverses unexpected locations (physical state estimates) that could reduce its confusion on its next journey from A to B. These physical state estimates are worth remembering. Hence, the original, no-confidence beads move to directly match the path actually traveled. Provided the robot has some confidence in its measurements for the new path, the bead confidences are set to high—this path worked once, so the robot assumes it will work again.

Note that now these moved beads are equivalent to the ones that the robot's managing human believed in, and they will never change again. The robot's overall map will only be dynamic in this update method until all of the no-confidence beads have been visited and updated. After that, the robot's internal map is generally static, and the robot ceases to learn until it is given new paths to follow. There are exceptions where no-confidence beads may be created by the update method in order to smooth out discontinuities in a lane or intersection that result from the update of a subsection.

In addition to relying on the update method properties described in the previous section, this update process runs Replace_Bead_Array_(Decide).vi to determine whether beads should be updated or not. In a situation with binary confidence levels, generally an entire lane or intersection or zone exit either has total confidence or none. The code tests a sample of an Array of Beads (by default five beads): if all of them have no confidence, the bead array can be altered. Alternatively, a larger sample of the array and a lower threshold percentage of no-confidence beads could be used. This adjustment would make

a more informed decision for bead arrays with very mixed confidences, but it would also require more processing time. If the original sparse points of a map are given high confidence, then a larger sample will likely see those points, and in that case a less-than-perfect threshold percentage will be crucial if updates are to occur.

A low confidence lane (or other bead path) may have high confidence endpoints—designated points that must line up with intersections, for example. To still allow a test for a complete lack of confidence, the first two beads in the "to update" bead list are ignored, provided that the array is long enough to do so; zone exits are very short but also do not have high confidence endpoints. Ignoring the first two beads is sufficient for a test using a small number of bead samples, because such a test will not reach original map positions or the bead array's other endpoint.

For each bead in the section of beads to replace, the code calls Update_Bead_All_or_Nothing.vi. Many of a bead's attributes are kept the same, but the latitude and longitude are completely replaced, and the heading and confidence are copied from values reported by the Physical State Observer and the Lane Description, respectively, at the time the update point was recorded. Normally the Lane Description has four levels of confidence. But with this method, if a point from the Lane Description had any confidence at all, the confidence of the bead is effectively absolute (not none). The VI also inserts into the bead data a record of how much the latitude and longitude have each changed in the update.

But what if the number of update points meant to replace a section of beads is more or less than the number of beads originally in that section? While the density of beads must be acceptable, the exact number of beads in a particular bead list need not

remain constant. Because a whole section of beads is being replaced, the problem can be examined from a higher level than that of just individual beads. By finding the section of beads to replace at the start of the process, the beads stay within bounds no matter how sparse or dense they are. The code iterates through solely the new data points. If there are fewer new points than old, the new points simply change however many beads they require and ignore the rest; the extra beads are deleted.

If more beads need to be created in order to fill the section with all the new data, the code cannot use Update_Bead_All_or_Nothing.vi, which requires that some of a bead's fields exist already. This condition of extra beads is identified when the next bead index required for an updated point exceeds the last bead index originally contained in the replaced section. When the condition occurs, a different function is called, New_Bead_from_Updates.vi, which creates an entirely new bead from latitude/longitude position, heading, confidence, distance from lane start, and max speed. These quantities are collected from the Lane Description and the physical state when the robot drives through a lane/intersection/zone exit. The other bead fields are set to defaults.

New_Bead_from_Updates.vi is also used if the newly created beads are too sparse. Fill_Out_Bead_Gaps.vi has a value for a minimum spacing (in meters) between consecutive beads that it will tolerate. If a gap is too large, the VI linearly interpolates points in order to fill the gap. Linear interpolation can work for creating drivable paths, as shown in map matching algorithms that use linear interpolation in order to model vehicle trajectory [36]. Each interpolated bead receives a default confidence. This confidence can be tweaked; if none (which was used in the experiments), the interpolated beads can be used for navigation but possibly replaced later. If the gap beads are given

some confidence, then this qualifies them as good enough to keep forever.

The section of new beads is inserted into the Array of Beads at the same position that the old beads were removed; the code does not touch beads to either side of the updated section at this stage. This result can often lead to discontinuities. Imagine a lane with several other lanes crossing it from start to finish. The robot can enter the lane partway through, and leave before the end (see Figure 4-3). The updated section may be a noticeable distance from the initially guessed positions, such that if the robot next drives the lane from end to end, the path suggested by the beads jumps sideways at two points, even as it is moving forward. Even when the robot starts out by driving straight through a lane, delays in various modules of DEXTER's code can cause the update points collected to miss either end, leaving a few points in their original positions.

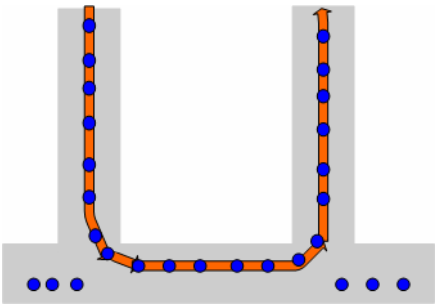


Figure 4-3. The blue dots are beads. The orange lines show the path taken by the robot, which forced updates on each one lane road. In the bottom lane, the difference between the old beads and the beads that were moved due to the orange path of travel is very noticeable.

When driving, a smooth path can be more important than a completely accurate one. In DEXTER, downstream modules wanted the beads from the Global Mapper to give a relatively smooth path as the initial guess. With that in mind, Interpolate_Replaced_Section_to_Borders.vi moves some points on either side of the replaced section, so that while the updated part of the path may be clearly different from

the original part, the connections between them will be easier for the robot to follow. As with initial bead generation, the interpolation here employs a spline instead of a purely linear method like that implemented by Fill_Out_Bead_Gaps.vi. Interpolating to borders need not be as finely grained as simply filling in a gap often is, so it is more likely for curves to be part of the interpolation. The spline method handles curves better than the linear method.

Interpolation requires two points to confine either end of the trail to be generated. One of the two endpoints comes from an edge of the route of updated beads. The farther away from the updated beads, the less likely the update pattern has a real bearing on where the un-updated beads should be, so the other bordering reference point should generally be a small number of beads (five in these experiments) from the updated section. However, if not enough beads exist between an updated section and an end of the current Array of Beads, the number of splined beads shrinks to what the current Array of Beads can accommodate. Interpolate_Replaced_Section_to_Borders.vi creates one-to-one replacements for beads. The interpolated points are given a default confidence; since these points are like the gap-filling beads that are created to help path drivability, they receive no confidence in the experiments.

The number of new beads at this stage may be less than the number of old beads in the same section, a consequence of receiving sparse data that is only partially fixed by filling in gaps with entirely new beads. Sometimes, beads are waypoints. These waypoints are important because they are the original points that define the map. Furthermore, some waypoints are checkpoints (which have to be reached to properly complete a mission) or stop points (which tell where the robot must wait before crossing

an intersection). Reassign_Data_from_Deleted_Array_of_Beads.vi preserves these important characteristics. The VI examines the beads that were deleted—truly deleted, and not just moved. These are the beads that were never updated, and if any of them is a waypoint, Reassign_Data_from_Deleted_Array_of_Beads.vi transfers that waypoint ID to the nearest (in terms of GPS proximity) surviving bead. If the nearest surviving bead is itself a waypoint, the rest of the Array of Beads is examined to find an empty host. If no such host exists, the data must be truly abandoned.

If the all-or-nothing method were to be pursued further, more error checking could be added to this step. In practice, this reassignment of data was not really needed given the density of points DEXTER can grab, nor was the order of the waypoints an issue. But if the robot needed the waypoints in order, this can be forced by checking existing waypoint IDs. A different approach would be to assume that any updates whatsoever could take the waypoints too far off their original positions to be trusted, and then reassign all waypoints and waypoint data within the replaced section. Or, updates of waypoints could be strictly prohibited.

New beads may have been added in this entire process, and the bead index of each bead cluster should match the bead's place in its Array of Beads (other code not connected to updates has such an expectation). To insure any bead index issues are avoided, Reset_Bead_IDs.vi renumbers each bead index in the newly updated Array of Beads.

Reinsert Array_of_Beads.vi takes the location of the Array of Beads in the overall map structure (saved from when the original Array of Beads was first pulled from that structure) and places the new Array of Beads back into the overall map as a whole. The

new map structure passes back to Global_Mapper_Main.vi, which can now use these updated beads as it answers queries about upcoming bead paths. Since sometimes updates do not happen, Update_Mapper_Beads_All_or_Nothing.vi informs Global_Mapper_Main.vi if it should expend the effort to update its memory. Update_Mapper_Beads_All_or_Nothing.vi is the main bead update file that calls each of the update VIs described according to its purpose. Figure 4-4 illustrates the structure of Update_Mapper_Beads_All_or_Nothing.vi.

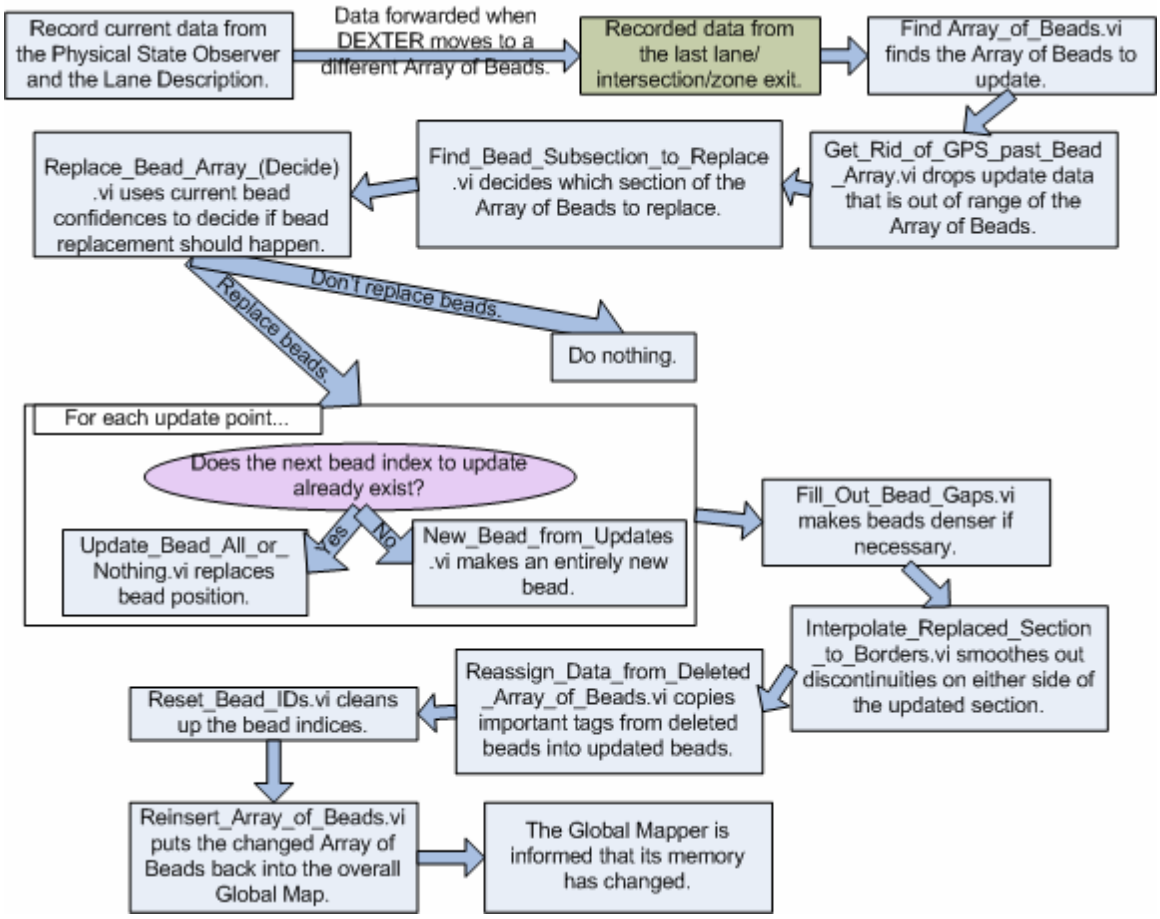


Figure 4-4. This flowchart summarizes Update_Mapper_Beads_All_or_Nothing.vi and describes how and when various helper functions are used in the update process.

Section IV: Results in Simulation of the All-or-Nothing Approach

When the 2007 DARPA Urban Challenge ended, so did testing on Team Case's

hardware. With DEXTER retired, the all-or-nothing update method needed to be analyzed on a simulated DEXTER. Fortunately, a program called dexsim (written by Christian Miller) served such simulation needs. However, simulated DEXTER did not have the lane detection powers that real DEXTER did, so in testing the all-or-nothing update method the simulated DEXTER had to be encouraged to drive off of the RNDF by more direct means. All-or-nothing tests rely on driving away from the RNDF; otherwise, DEXTER will drive over the original beads and update them to very similar positions.

To force a departure from the RNDF, first the human running the simulation must make a log of an RNDF path simulated DEXTER followed and the corresponding Lane Description he produced. Once this record is stored, the Python script move_RNDF.py can move the original RNDF by set latitude and longitude offsets. Using the shifted RNDF, dexsim runs again in playback mode such that the simulated DEXTER drives the path in the log. Yet, the RNDF has changed, so the initial beads in the current test are located on the new roads, not on the roads used to make the log. Hence, the beads change from matching the current roads to matching the Lane Description in the log. Any figures that show DEXTER as driving off a road actually show DEXTER ignoring a bad map for better, stored data.

Figure 4-5 shows an example scenario that clearly illustrates how the all-or-nothing update method can provide some benefit to DEXTER. The red dashed lines are DEXTER's logged breadcrumbs that he must follow in playback. The breadcrumbs are not the same as the Lane Description, as the former comes from the latter, but the two are very close; in fact, these experiments display the Lane Description as a set of black dots, but the red breadcrumbs cover them since the two are in the same place. In Figure 4-5,

simulated DEXTER's map told him to drive over the green and white dots (both waypoints), but his stored sensor log placed him more towards the road center.

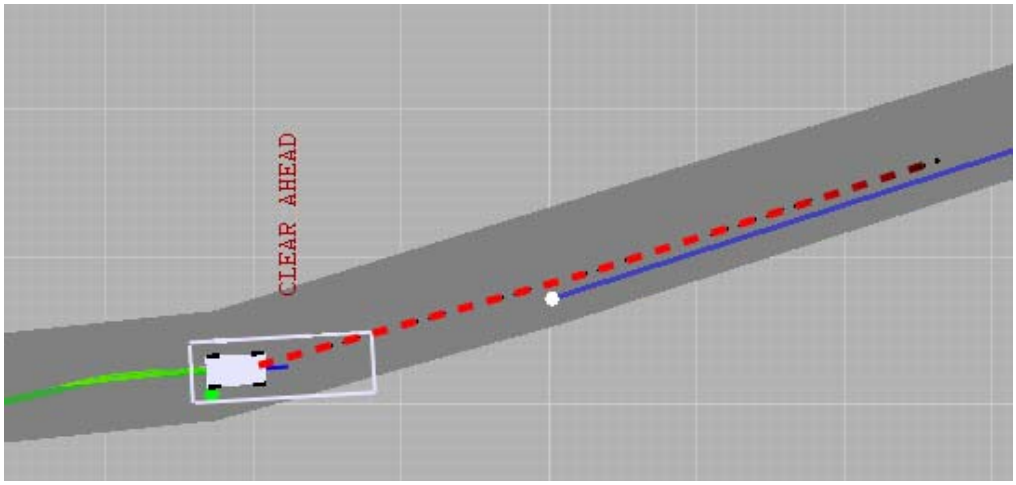


Figure 4-5. Using a playback log, a simulated DEXTER drives more centered than the proposed route in blue. The red breadcrumbs show where DEXTER is about to drive next. The green trail shows where DEXTER has just driven.

Figure 4-6 shows that the beads (marked by blue dots) have been moved in line with where DEXTER drove in Figure 4-5. Though waypoints started with some confidence, the other beads in the lane did not, so the update code replaced a whole section of the lane. Note the gaps in the beads. In these experiments, Fill_Out_Bead_Gaps.vi dealt with gaps more than one meter. Observing the code showed that many points were created by this VI; Figure 4-6 shows that the VI results in beads spaced similarly to their initial spacing (which is also one meter). For comparison, the bottom beads (below the intersection) in Figure 4-7 are original beads that have not been traversed and updated.

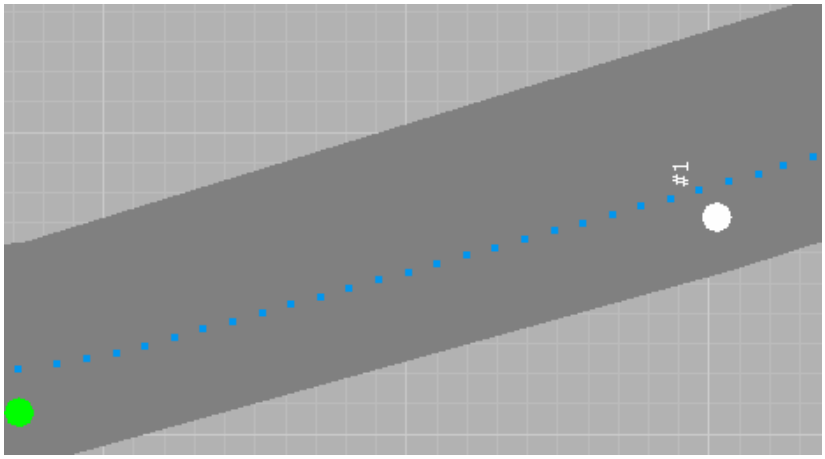


Figure 4-6. The beads (marked by blue dots) have been moved more towards the center of the road (and placed along simulated DEXTER's log data) because of the path driven by the simulated DEXTER in Figure 4-5.

Despite the potential for improvement, there are localization hazards when performing the all-or-nothing update method. Remember, DEXTER is not actually off the road in the following figures, rather the current map belief (shown)—and therefore the localization software—thinks he is. Figure 4-7 shows one example of how belief in a poor map can confuse localization. Because DEXTER can drive straight through the intersection shown, the straight path is not actually an intersection, but rather a part of the lane. When DEXTER actually entered an intersection, but was close to the lane path, the localizer suffered confusion and DEXTER's travel was seen as part of the lane. Hence some of the lane's beads were updated to fill that curved intersection. Notice the beads headed back from the curve towards the original beads at the bottom; Interpolate_Replaced_Section_to_Borders.vi determined their placement.

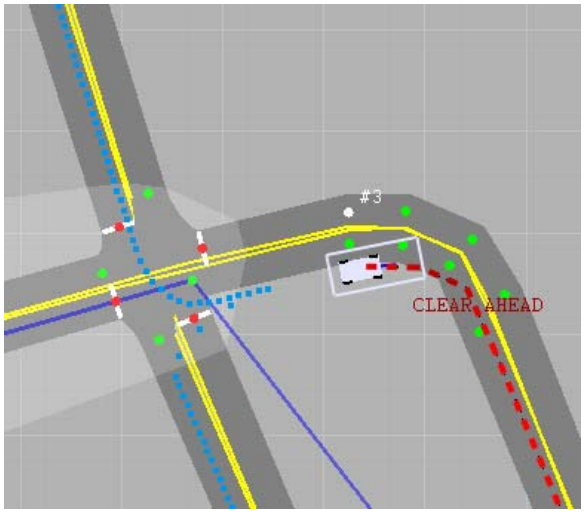
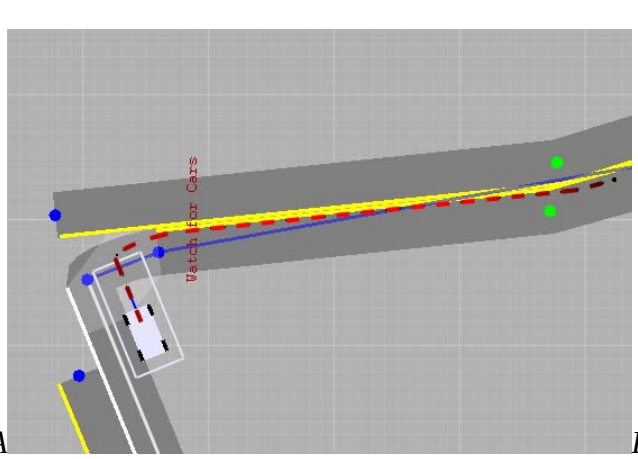
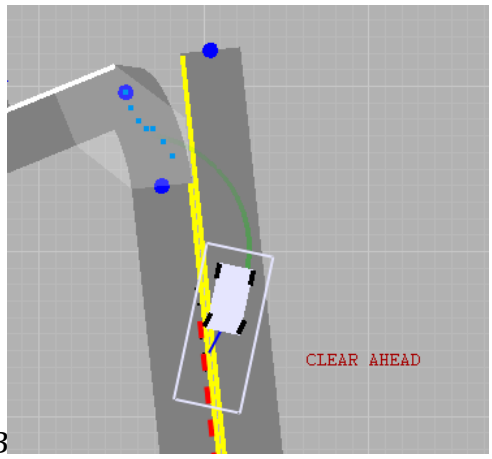


Figure 4-7. Due to simulated DEXTER's displacement, the curve through the intersection was seen as part of the lane, and updated accordingly. The yellow lines divide lanes in dexsim. DEXTER's movements clearly ignored the poor map; the problem is that his localization software did not.

Figure 4-8 shows simulated DEXTER navigating an intersection. Note how the resulting beads tried to fit as best as they could within the localization software's belief in the intersection location while still following the update data. Because the update path started in the middle of the believed intersection on the poor map, this created a kink in the result. Interpolate_Replaced_Section_to_Borders.vi still tried to reach the original intersection endpoints from the poor map. In real life, endpoints of a structure need to be fairly accurate, but dexsim cannot replicate that condition.



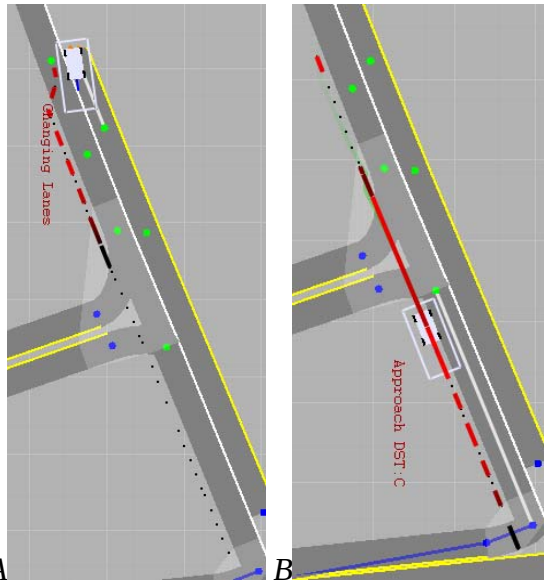
A



B

Figure 4-8. On the left (Figure A), the simulated DEXTER drove through an intersection offset from the intersection on the poor map. The poor map's intersection curves between the two dark blue dots directly ahead of DEXTER. On the right (Figure B), the light blue dots show an updated intersection that still tries to keep the same endpoints.

Updated beads must be kept in numerical order. For example, Bead 2 in an Array of Beads cannot be after Bead 3, where "after" is defined based on the normal forward progression of the path. Even if the overall shape of the path is good, out-of-order beads confuse the robot, which may become convinced that it needs to backtrack to hit some beads and leap to reach others. DEXTER could not handle such a situation. To verify that the update code keeps beads in the correct order, a new simulation ran on a set of previously updated beads from a previous test. The simulation again used a moved RNDF for visualization purposes, so that the reader can still see that the poor map is ignored in favor of beads created by the all-or-nothing update method. Figure 4-9 shows the results, with the Lane Description (black dots) and breadcrumbs (red dashes) built from the previously updated beads. The simulated DEXTER drove on these updated beads, meaning their order was sound. Note that the Lane Description does not always start right in front of any DEXTER, simulated or otherwise.



A *B*

Figure 4-9. In this simulation, DEXTER loaded beads previously created by the all-or-nothing update method and found those beads to be drivable. The red dashes represent the breadcrumbs while the black dots represent the Lane Description. Figure A comes before Figure B in time. The poor map in the image is ignored for the updated beads.

Recall the kink problem in the intersection, as illustrated by Figure 4-8. Figure 4-10 is the result of driving on this crimped data; simulated DEXTER was confused not by bead order, but by trying to reach the localization software's believed end of the intersection. DEXTER handled the problem by attempting to start the intersection and then reaching what beads were next available to him within the realm of his allowed movements. Since the beads were in order and DEXTER did not need to backtrack, he constantly moved forward until he hit the path.

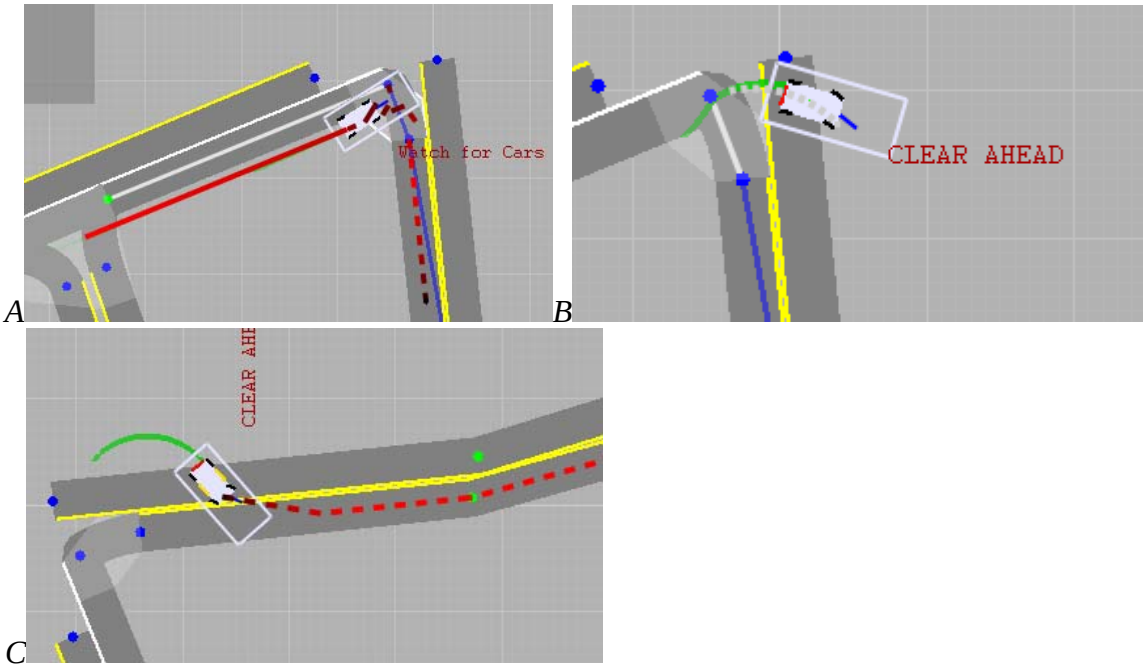


Figure 4-10. DEXTER recovers from a kink in the intersection data. The green trail shows where DEXTER drove moments before. The letters (A-C) indicate how the figures progress through time.

The all-or-nothing update method, while a bit simplistic, effectively introduced updates into DEXTER's Global Mapper. This simple method does not use the finer granularity of confidence available in DEXTER: none, low, medium and high. With such

information, a bead could be moved partially instead of fully replaced, in order to take into account the previous bead position. This idea of creating new beads by combining sources of uncertain information is explored further in Chapter 5.

Chapter 5: Improving Beads by Merging Noisy Data

Section I: The Theory for an Improved Estimate from Noisy Measurements

Assume that all the measurements influencing the Global Map come packaged with known variances. In such a world, a bead need not be substituted with sensor data in an all update, nor kept constant in a nothing update, nor decide between the two extremes. Instead, each bead can be partially moved in a tug-of-war among all the relevant inputs, where the data with the lowest standard deviation and variance (and therefore the highest confidence) pulls hardest. In *Applied Optimal Estimation*, Gelb et al. present a way to find an optimal estimate given erroneous data with "random, independent, unbiased measurement errors" [37]. Following from these criteria, the simulations will assume sensors with errors in a Gaussian distribution and zero mean variance.

The measurement sources in the simulations represent pieces of the DEXTER problem. First, there are *a priori* data, e.g. from RNDF information, with all the errors therein. In reality, the RNDF points come from a GPS antenna or a human interpreting satellite or aerial imagery, and any of these methods may introduce errors. Existing RNDF data may be further augmented by additional information (e.g. road maps or more aerial photography). Errors in the *a priori* information may include human transcription errors, map or photographic registration errors, and measurement or interpretation imprecision. The *a priori* information is used to initialize all beads. Second, there is

physical state data, representing DEXTER's Physical State Observer, which captured online GPS sensor data and kept DEXTER aware of his position [38]. The concept of building an observer to enhance raw physical data has served other, less complicated mobile robots in the past, such as one trained to follow a guide wall [39], or another deciding the heading of a unicycle robot [40]. DEXTER's Physical State Observer combined GPS data with other sensory and kinematic information and strengthened its assumptions with a Kalman filter. A Kalman filter's usefulness to a mobile robot is not specific to GPS readings, as it can be used to filter the noise from other positional data, such as that derived from sonar sensors [41]. Positional information can be absolute (GPS coordinates) or relative (inertial and odometry data); gathering some form of it is essential in an autonomous robot like DEXTER and his contemporaries [1], [30].

DARPA defined goals in terms of GPS coordinates [31], so the GPS sensor was of particular importance in referencing DEXTER's physical state to a GPS-defined coordinate system. In general, errors in GPS data provide issues in even simpler systems than DEXTER, such as in map matching algorithms [42]. Tree cover and tall buildings obstruct GPS signals [43]. Errors may also be introduced into GPS signals on purpose, which the U.S. military implemented historically to the detriment of civilian receivers [44]. In 2003, after the lifting of some military restrictions, GPS devices could give readings within ten meters of the true position, with Differential GPS far better at +/- 1 m [43]. A current, example civilian application, the Garmin, claims that its GPS receivers are accurate within an average of fifteen meters normally, three to five meters using Differential GPS, or less than three meters with a Wide Area Augmentation System—yet "atmospheric factors" are still a concern [45]. The physical state measurements in the

simulations provide information about latitude, longitude, and heading elements. GPS positions have Gaussian errors in latitude and longitude, a model also used in the map matching algorithms in [36].

DEXTER had road detection abilities, which reported his lateral offset from the identified center of the road and a heading correction meant to help DEXTER match his heading to the road's forward direction. However, road detection values are not in absolute GPS coordinate terms; as a measure of relative quantities, they cannot be combined with the other two measurements via the optimal estimation method. The introduction of a road detection system into the simulations is still possible via other techniques, as discussed in Chapter 7.

If the physical state estimate has high confidence but beads do not, the update will resemble an "all" update. Conversely, if the beads are known to be reliable but the physical state estimate is not, the update will resemble a "nothing" update. In the general case, even low-confidence data affects the results, so no source of information is completely ignored as with truly all updates or nothing updates. Furthermore, shades of gray in the confidence of each of the measurements can be explored with this new system.

In *Applied Optimal Estimation*, the equations for the optimal estimate and its new variance follow from two measurement sources [37]. The *a priori* latitudes, longitudes, and headings in the beads are separate quantities with separate optimal estimates. The beads are responsible for holding improved estimates across repeated updates along the same path. The outcome of the two-measurement equation can be considered a new measurement itself, and the equations may be applied iteratively as the robot "drives"

through the simulated path again and again.

Equation 5-1 shows the estimate *z* that results from two measurements, *x* and *y* (*see* [37], pg 6). The estimate *z* has a corresponding variance σ_z^2, computed per Equation 5-2.

$$z = \frac{\sigma_y^2}{\sigma_x^2 + \sigma_y^2} x + \frac{\sigma_x^2}{\sigma_x^2 + \sigma_y^2} y$$

Equation 5-1. Combining two measurements, x and y, into a new estimate z.

$$\sigma_z^2 = \left(\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2} \right)^{-1}$$

Equation 5-2. The variance of the new estimate z.

Assume the robot re-drives the same path. The simulation takes in its third measurement, *a*, and combines it with estimate *z* to get a new estimate, *b*. Equation 5-3 is Equation 5-1 reused with different variables.

$$b = \frac{\sigma_a^2}{\sigma_a^2 + \sigma_z^2} z + \frac{\sigma_z^2}{\sigma_a^2 + \sigma_z^2} a$$

Equation 5-3. The technique in Equation 5-1 is reapplied.

Equation 5-4 substitutes *z* from Equation 5-1 into Equation 5-3 and uses the distributive property of multiplication to produce three terms.

$$b = \frac{\sigma_a^2}{\sigma_a^2 + \sigma_z^2} \left(\frac{\sigma_y^2}{\sigma_x^2 + \sigma_y^2} x \right) + \frac{\sigma_a^2}{\sigma_a^2 + \sigma_z^2} \left(\frac{\sigma_x^2}{\sigma_x^2 + \sigma_y^2} y \right) + \frac{\sigma_z^2}{\sigma_a^2 + \sigma_z^2} a$$

Equation 5-4. The terms of z are substituted into Equation 5-3 to produce this new equation.

The presence of σ_z^2 in Equation 5-4 can be rewritten in terms of σ_x^2 and σ_y^2 using Equation 5-2. The result of these replacements is Equation 5-5.

$$b = \frac{\sigma_a^2}{\sigma_a^2 + \frac{1}{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}}} \left(\frac{\sigma_y^2}{\sigma_x^2 + \sigma_y^2} x \right) + \frac{\sigma_a^2}{\sigma_a^2 + \frac{1}{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}}} \left(\frac{\sigma_x^2}{\sigma_x^2 + \sigma_y^2} y \right) + \frac{\frac{1}{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}}}{\sigma_a^2 + \frac{1}{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}}} a$$

Equation 5-5. The instances of σ_z^2 in Equation 5-4 are replaced using Equation 5-2.

Equation 5-5 can be greatly simplified. Equations 5-6 and 5-7 show how two relevant fractions can be rewritten by multiplying each by a fraction equal to one.

$$\frac{1}{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}} \bullet \frac{\sigma_x^2 \sigma_y^2}{\sigma_x^2 \sigma_y^2} = \frac{\sigma_x^2 \sigma_y^2}{\sigma_x^2 + \sigma_y^2}$$

Equation 5-6. The meaning of σ_z^2 can be rewritten.

$$\frac{\sigma_a^2}{\sigma_a^2 + \frac{\sigma_x^2 \sigma_y^2}{\sigma_x^2 + \sigma_y^2}} \bullet \frac{\sigma_x^2 + \sigma_y^2}{\sigma_x^2 + \sigma_y^2} = \frac{\sigma_a^2 \left(\sigma_x^2 + \sigma_y^2 \right)}{\sigma_a^2 \sigma_x^2 + \sigma_a^2 \sigma_y^2 + \sigma_x^2 \sigma_y^2}$$

Equation 5-7. A fraction found twice in Equation 5-5 can be simplified using Equation 5-6 and the identity property of multiplication.

The results of Equations 5-6 and 5-7 can be substituted into Equation 5-5 for simplification, producing Equation 5-8.

$$b = \frac{\sigma_a^2\left(\sigma_x^2+\sigma_y^2\right)}{\sigma_a^2\sigma_x^2+\sigma_a^2\sigma_y^2+\sigma_x^2\sigma_y^2}\left(\frac{\sigma_y^2}{\sigma_x^2+\sigma_y^2}x\right)+\frac{\sigma_a^2\left(\sigma_x^2+\sigma_y^2\right)}{\sigma_a^2\sigma_x^2+\sigma_a^2\sigma_y^2+\sigma_x^2\sigma_y^2}\left(\frac{\sigma_x^2}{\sigma_x^2+\sigma_y^2}y\right)+\frac{\frac{\sigma_x^2\sigma_y^2}{\sigma_x^2+\sigma_y^2}}{\sigma_a^2+\frac{\sigma_x^2\sigma_y^2}{\sigma_x^2+\sigma_y^2}}a$$

Equation 5-8. Equation 5-5 simplified using Equations 5-6 and 5-7.

The last shown fraction (on the *a* term) may be multiplied by $\frac{\sigma_x^2+\sigma_y^2}{\sigma_x^2+\sigma_y^2}$ in order to perform cancellations in both its numerator and denominator. Similar cancellations apply to the fractions on the other two terms (for *x* and *y*) as already written. The result of these simplifications is Equation 5-9.

$$b = \frac{\sigma_a^2\sigma_y^2}{\sigma_a^2\sigma_x^2+\sigma_a^2\sigma_y^2+\sigma_x^2\sigma_y^2}x+\frac{\sigma_a^2\sigma_x^2}{\sigma_a^2\sigma_x^2+\sigma_a^2\sigma_y^2+\sigma_x^2\sigma_y^2}y+\frac{\sigma_x^2\sigma_y^2}{\sigma_a^2\sigma_x^2+\sigma_a^2\sigma_y^2+\sigma_x^2\sigma_y^2}a$$

Equation 5-9. A simplified form of an estimate b formed from three measurements.

When processing measurements iteratively, it may appear that the algorithm favors earlier measurements over later ones. The repeated application of Equation 5-2 leads to smaller variances in new estimates [37], and Equations 5-1 and 5-9 both show that measurements with smaller variances influence the outcome to a greater degree. Yet, Equation 5-9 shows that the third measurement has as much of a chance to affect the outcome as the first two, regardless of order. For example, if *x* came last and *a* came first into the iterative process, all appearances of *x* and *a* would be switched in Equation 5-9 (and throughout the process), but the result would be equivalent. Because of this, the order in which the simulations process incoming measurements is irrelevant. However, the repeated application of the same measurement on a point can weigh the results in favor of one outcome. If, for example, the physical state consistently saw the same

results every time the robot drove down a path, then the beads would become more and more likely to believe the many measurements in agreement over their own starting assumptions—depending, still, on the relative variances involved.

The new estimate *b* requires its own variance, σ_b^2. If σ_z^2 represents the error before measurement *a*, then Equation 5-10 represents σ_b^2.

$$\sigma_b^2 = \left(\frac{1}{\sigma_z^2} + \frac{1}{\sigma_a^2} \right)^{-1}$$

Equation 5-10. The variance after incorporating three measurements together into estimate b.

From Equation 5-2, the value of σ_z^2 is known and can be substituted into Equation 5-10 to derive Equation 5-11.

$$\sigma_b^2 = \left(\frac{1}{\frac{1}{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}}} + \frac{1}{\sigma_a^2} \right)^{-1}$$

Equation 5-11. A rewrite of Equation 5-10 with σ_z^2 replaced from Equation 5-2.

The first fraction can be multiplied by a version of one for the purposes of simplification. The result is in Equation 5-12. Each previous error contributes equally to the new estimate error σ_b^2, reinforcing the idea that the order of the measurements does not matter.

$$\sigma_b^2 = \left(\frac{1}{\frac{1}{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}}} \bullet \frac{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}}{\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2}} + \frac{1}{\sigma_a^2} \right)^{-1}$$

$$\sigma_b^2 = \left(\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2} + \frac{1}{\sigma_a^2} \right)^{-1}$$

Equation 5-12. An illustrative simplification of Equation 5-11.

The mathematics involved can be performed as a batch, or the calculations can occur continuously as a robot runs. The equations do not change; in either event, the equations incorporate the sensor readings in some series. Since the order of contributing quantities does not change the results, there is no difference in using sensor readings as they are taken versus collecting a host of readings and (potentially) reordering them after data collection. Examination of the sensor values sooner rather than later merely provides the advantage of making partial answers available.

Since the equations are abstract, each new data point may be a new measurement from a sensor used earlier, or a new measurement from a completely new sensor; regardless, the values for *x*, *y*, and *a* can be used in the same way. In an attempt to approach the DEXTER system, the number and type of sensors in upcoming simulations follow from DEXTER's own setup. However, the method is extendable for any number of data acquisition devices.

Section II: Implementing the Combination of Gaussian Measurements

A road-following robot has a Path Driven, a set of points traversed in reality. The robot also has a Center Lane Path, which is the desired path at the center of the road

where the robot should be driving. When the two paths are one and the same, road detection may be ignored. If a human is behind the wheel of the robot (either literally or via a joystick, as was necessary with DEXTER), then such a condition may indeed be enforced. For the simulations in this chapter, the Center Lane Path is the only path under discussion. The physical state (consisting of latitude, longitude, and heading) and the bead list (with the same three values) are the only types of measurements used.

In the present analysis, physical state measurements and the bead list are both assumed to contain Gaussian-distributed errors with zero mean (i.e. no bias), constituting deviations from the true Center Lane Path. A single path can provide sufficient coordinates for testing, and the data-fusion algorithm is independent of lane/intersection distinctions, so simulations performed for this analysis do not require an entire Global Map. Time necessary to generate a path (measured at one millisecond for a path of 300 coordinates) does not provide a noticeable delay to the user, so the simulations generate a new path for each test.

The main simulation file (Basic_Test_Path_Driven_on_Center_of_Lane.vi) first calls Generate_True_Path.vi to generate a Center Lane Path with a user-specified Number of Coordinates. Generate_True_Path.vi produces realistic paths comprised of points spaced 1 m apart with heading changes no greater than +/-5 degrees from point to point. The Center Lane Path needs a seed location and heading. Unless stated otherwise, the beginning heading is 45 degrees in every simulation where a Center Lane Path is generated (including those in future chapters). The seed coordinate is always 41.502566 degrees latitude, -81.607586 degrees longitude, a point on the engineering quad of Case Western Reserve University.

Once the Center Lane Path exists (and can therefore be used for comparison with results), code injects Gaussian errors to produce simulated GPS and heading measurements. Generate_Gaussian_Distribution_of_Lat-Lon-Head_Variable_Std_Dev.vi performs this service for the simulated physical state. Distributed with LabVIEW, Gaussian_White_Noise_PtByPt.vi produces Gaussian white noise of a given standard deviation. The VI for introducing errors receives two sets of standard deviations: one for headings, in units of degrees, and one for GPS coordinate shifts, in meters.

Though expressed in the same units, latitudes and longitudes could each have their own set of driving standard deviations. However, since these two pieces of data come from the same sensor, it is likely that the two quantities will be measured about equally well, so the same standard deviations are used for both. This assumption should not be taken to imply correlation between latitude and longitude. A change in one does not drive a change in the other: when moving north, a robot may at the same time move east, move west, or stay at a constant longitude. Furthermore, a latitude error towards the north does not imply that a longitude error (even one of equal magnitude) should necessarily be east versus west.

Generate_Gaussian_Distribution_of_Lat-Lon-Head_Variable_Std_Dev.vi uses specified standard deviations to introduce errors to each point in a path. The standard deviations of noise introduced may be varied throughout the path to emulate realistic variations in path uncertainty. For each point in the Center Lane Path, the chosen standard deviations create three errors; one error is added to the original heading and reset to be within the 0-360 range. The latitude error moves the original coordinate north

or south (depending on error sign), and the longitude error moves the coordinate east or west. The new latitude and new longitude join the new heading in one cluster of path data, along with each of their three variances. By definition, each variance is the square of the standard deviation used to add error to that specific measurement (latitude, longitude, or heading). Though the latitude and longitude variances will be the same at this step, the two variances are stored separately; road detection offsets, if and when they are used, may change the certainty of one differently from the other. For beads, a slightly different VI is used. The theory of operation is the same, but the outputs are packaged into actual bead clusters with unique Bead IDs.

Once errors have been introduced, the simulator must be able to combine two uncertain measurements in an effort to identify the true Center Lane Path. To do that, the code applies Equations 5-1 and 5-2 from Section I of this chapter. Average_Two_Measurements.vi employs both equations literally and simultaneously. The code assumes that the two measurements it receives are of the same type and the same units; latitude, longitude, and heading are processed independently and do not affect each other.

Average_N_Measurements_(of_One_Quantity).vi sits a level above Average_Two_Measurements.vi. Although not necessary with only two data sources, the use of this VI makes the solution more flexible; the simulation will readily accept any number of unbiased, flawed data sources. Average_N_Measurements_(of_One_Quantity).vi deals with n measurements by calling Average_Two_Measurements.vi iteratively, with the new estimate from one call passed as a measurement to the next. If $n = 1$, Average_N_Measurements_(of_One_Quantity).vi

returns the same data it received.

For latitudes or longitudes, Average_N_Measurements_(of_One_Quantity).vi can produce the new estimate on its own. Polar headings, however, are periodic; 360 degrees wrap around to map onto smaller angles. Average_N_Headings.vi addresses this problem. Average_N_Headings.vi chooses the first heading it receives as the Reference Heading. This choice is arbitrary. All that matters is that the headings are combined relative to one of their members, so that the new estimate can be computed as the distance that the Reference Heading should move.

Since Average_N_Headings.vi must convert headings into relative quantities, and since headings are periodic, Closest_Heading_Difference.vi finds the shortest distance, positive or negative, between the Reference Heading and another heading *h*. The shortest distance between two headings either crosses the zero to 360 line once or not at all. The heading *h* has three relevant forms: *h* itself, *h* + 360, and *h* – 360. Once Closest_Heading_Difference.vi finds these three sums, each has the Reference Heading subtracted from it. The difference among the three with the lowest absolute value is returned by the function as the closest difference.

As an example, consider a Reference Heading of zero degrees with $h = 359$ degrees. The heading *h* may be written as 359 degrees, 359 + 360 = 719 degrees, or 359 – 360 = -1 degrees. Subtracting zero degrees from each sum yields the same sums back again. The distance of -1 degrees has the smallest absolute value, so as a human would expect, Closest_Heading_Difference.vi reports -1 degrees as the shortest distance to travel from zero degrees to 359 degrees.

Average_N_Headings.vi sends all relative data (with the Reference Heading as

zero degrees, relative to itself) into Average_N_Measurements_(of_One_Quantity).vi. The result measures how much to move the Reference Heading; the variance associated with the zero degrees heading fights to keep the Reference Heading the same. Once the Reference Heading moves to a new heading in absolute terms, the resulting new estimate is placed back in the 0-360 range.

With these tools, Basic_Test_Path_Driven_on_Center_of_Lane.vi iterates through the entire path for a user-specified number of times. Bead updates are kept across such iterations, so that potentially a path may become more and more improved as it is driven multiple times. Physical state errors are created for each path traversal. Given the assumption of Gaussian errors, a sensor is unlikely to repeat the same exact information each time it takes a reading in the same spot. If this repetition occurred, the method used to combine measurements would become more and more convinced that the physical state was simply correct.

When exploring a path, the code finds a new estimate for each bead's latitude, longitude, and heading using previous bead data and incoming physical state data. The *n*th point in the bead list is always generated from the same Center Lane Path data as the *n*th point in the physical state data. The results for latitude, longitude, and heading estimates (including the new variances) are packaged back into the current bead, overwriting previous data with better estimates. Path data before and after updates is saved to .CSV files for later analysis.

Figure 5-1 presents an overview of the architecture of Basic_Test_Path_Driven_on_Center_of_Lane.vi. The three basic stages, each of which has already been covered in greater detail, are to create the Center Lane Path, to introduce

measurement errors, and to improve the path through the application of the optimal estimation method described by Equations 5-1 and 5-2. Figure 5-1 also depicts key times in the simulation procedure where the code saves results for later examination.

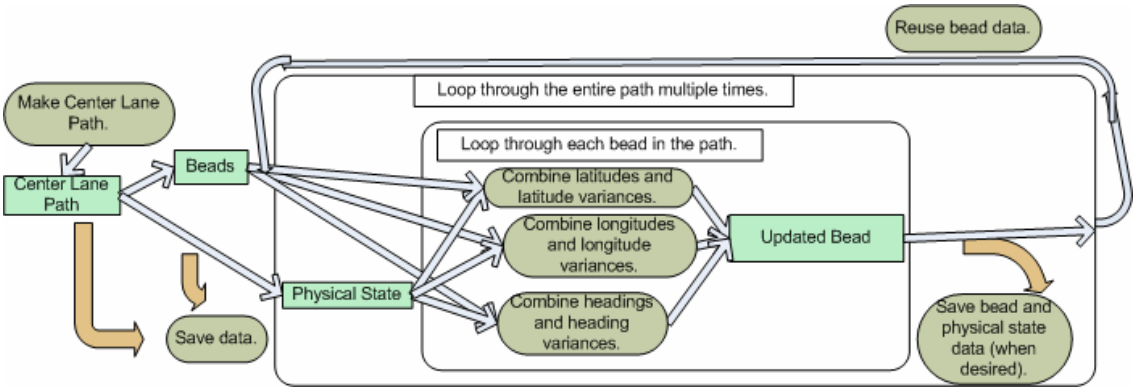


Figure 5-1. A simplified picture of Basic_Test_Path_Driven_on_Center_of_Lane.vi's modus operandi. Data may not be saved for insignificant path traversals.

Section III: Running Basic Simulations

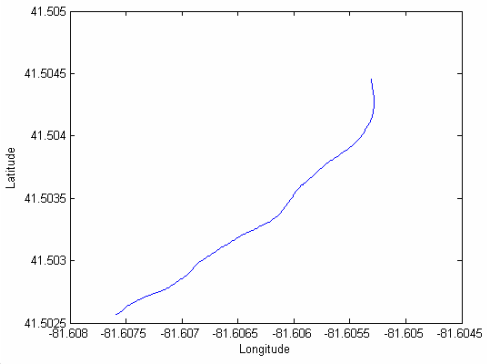
For comparison with other chapters, Table 5-1 shows what path data and error data are used in the following tests. Note that road detection errors are ignored; there is no road detection, since the Path Driven equals the Center Lane Path. The errors in initial bead placements and simulated physical state measurements have no bias.

Type	**Data**
Path	Path Driven = Center Lane Path, number of points in Center Lane Path = 300
Bead Errors	$\sigma_{b,GPS} > 0$, $\sigma_{b,heading} > 0$, bias = 0
Physical State Errors	$\sigma_{ps,GPS} > 0$, $\sigma_{ps,heading} > 0$, bias = 0
Road Detection Errors	$\sigma_{rd,offset} = 0$, $\sigma_{rd,heading\,correction} = 0$

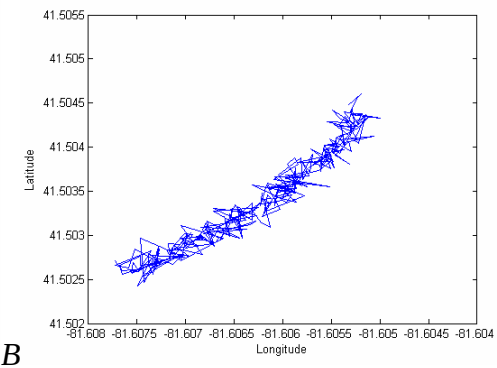
Table 5-1. Summarizing the data used for the following tests in bead improvement with no road detection. Biases and standard deviations are only factors in the experiments when they are not equal to zero.

Beads have an inherent order due to their relative arrangement within a list. The simulation generates beads on the Center Lane Path such that each subsequent bead continues a drivable path from its previous neighbor. However, once Gaussian noise is introduced, the simulations are able to show extreme cases with enough noise to make the path look as if it has wild backward and forward movements. This is allowed in order to more clearly illustrate the behavior of the optimal estimation algorithm.

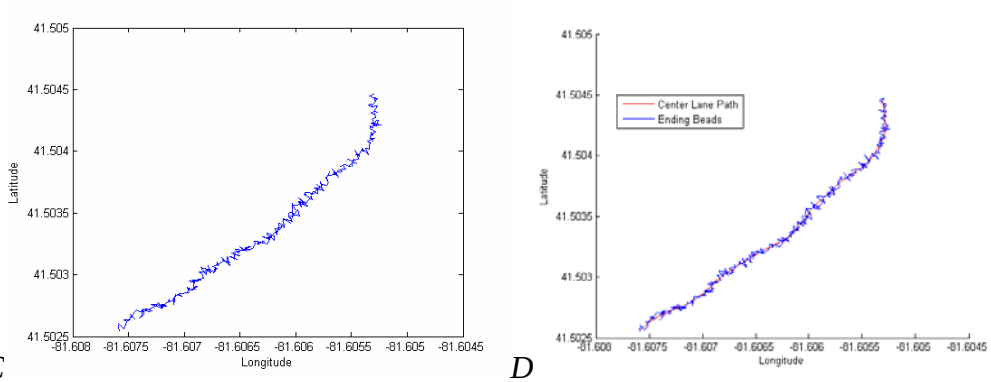
The simulator can approximate all updates or nothing updates when changing the initial latitude/longitude positions of the beads. First, consider the conditions for an all update: the physical state must have a much higher confidence than the starting beads, so much so that the best course for improvement is for the beads to practically copy what the physical state reports. Figure 5-2 shows the results after driving once through a path, with beads at a starting standard deviation of ten meters and the physical state measurements starting at a lower standard deviation of two meters (each standard deviation applies to both latitude and longitude).



A



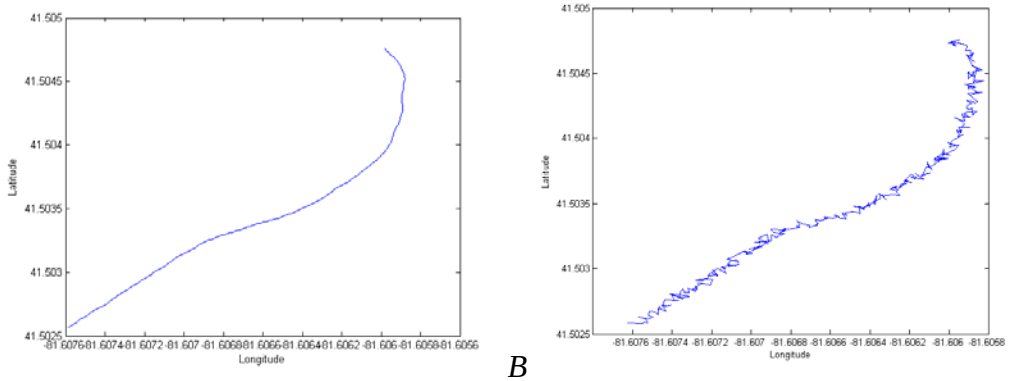
B



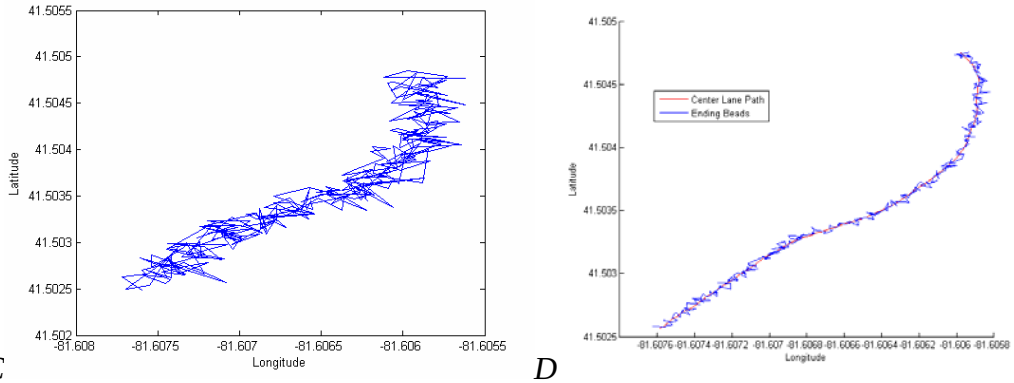
C *D*

Figure 5-2. Graph A shows the true Center Lane Path. Graph B shows the starting beads generated from Graph A with the introduction of noise with $\sigma_{b,GPS}$ *= 10.00 m. Graph C shows the simulated physical state measurements with noise* $\sigma_{ps,GPS}$ *= 2.00 m. Graph D shows the beads after running the simulator once through the entire path, resulting in an ending standard deviation of ~1.96 m in both latitude and longitude. Note how much the ending beads resemble the physical state, much like a typical all update.*

The simulator can also approximate a nothing update. Recall that a nothing update means that the bead position estimates start out with better quality than the sensors, so the beads have no reason to change. Figure 5-3 show an update much like a nothing update after one traversal through a path, where the confidences in the physical state and in the beads were reversed from the situation in Figure 5-2.



A *B*



C *D*

Figure 5-3. Graph A shows the true Center Lane Path. Noise of $\sigma_{b,GPS}$ *= 2.00 m,* $\sigma_{ps,GPS}$ *= 10.00 m were added to the bead estimates and to the simulated physical state measurements, respectively, as shown in figures B and C. Graph D shows the beads after running the simulator once through the entire path, with an ending standard deviation of ~1.96 m in both latitude and longitude. Note how the ending beads closely follow the starting beads, as in a nothing update.*

The previous two figures show that, unlike a true all update or a true nothing update, the resulting standard deviation in the ending bead positions is slightly better than the single best confidence present in the incoming data. The resulting standard deviation for bead positions is determined mathematically by the optimal estimation method (Equation 5-2). Whether the initial bead positions or the physical state measurements were more confident in the beginning of a simulation, the other, inferior source of data still improved the confidence in the result. Thus, using all sources of noisy data makes better use of the information available than previous methods that only used the best incoming source.

The simulator may also show a middle ground, wherein the physical state and the beads start out with equal confidences/standard deviations. Figure 5-4 shows an example using latitude/longitude position standard deviations of 5 m for both beads and physical state. The beads did show some change after one path traversal, but with only two samples of data to work with, neither one particularly reliable, bead improvement was

limited. However, after ten path traversals, and therefore ten physical state readings per bead, the beads in Graph F of Figure 5-4 started to approach the Center Lane Path. The power of the optimal estimation method is shown here, as beads improved when no one good data source existed.

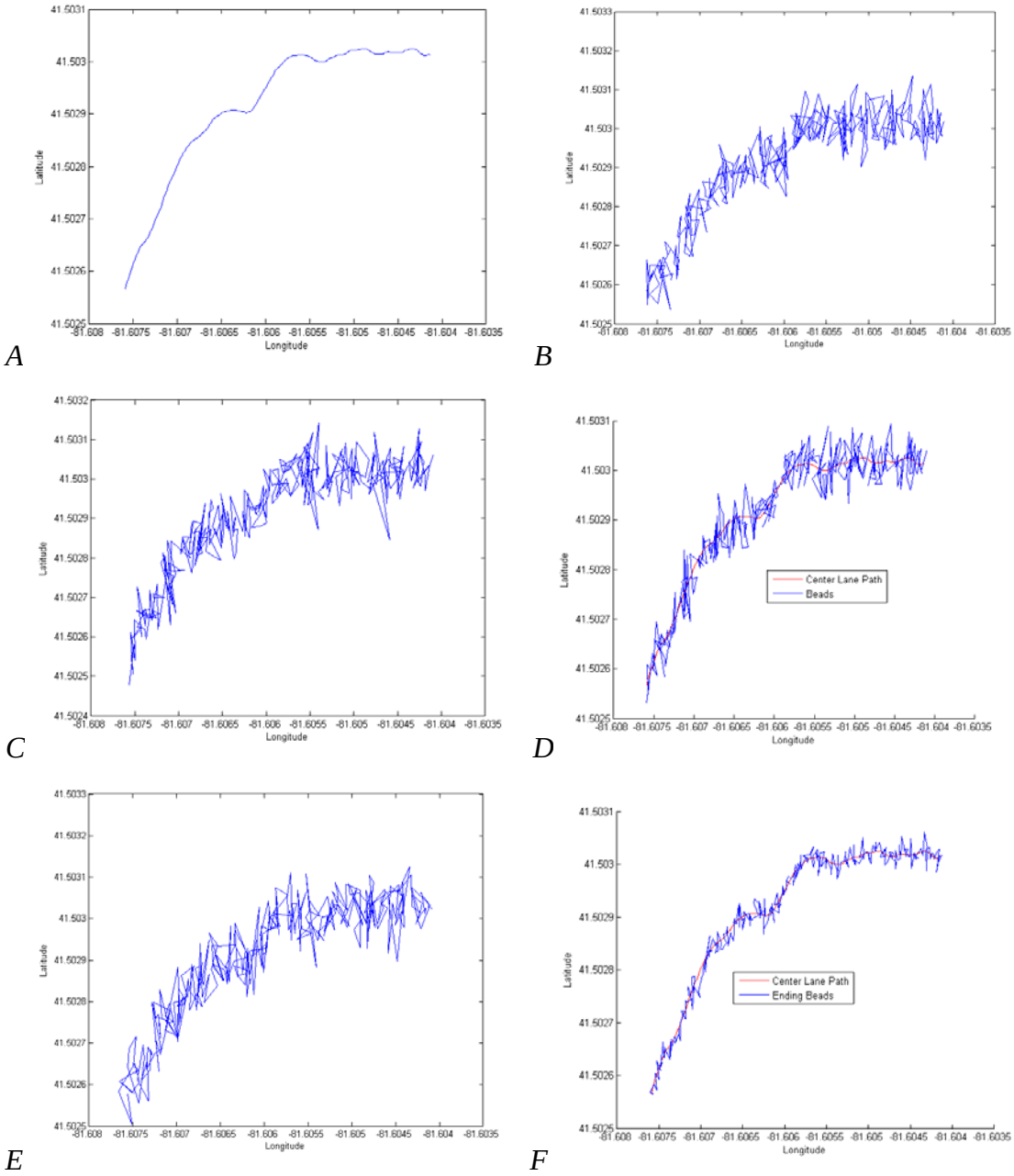
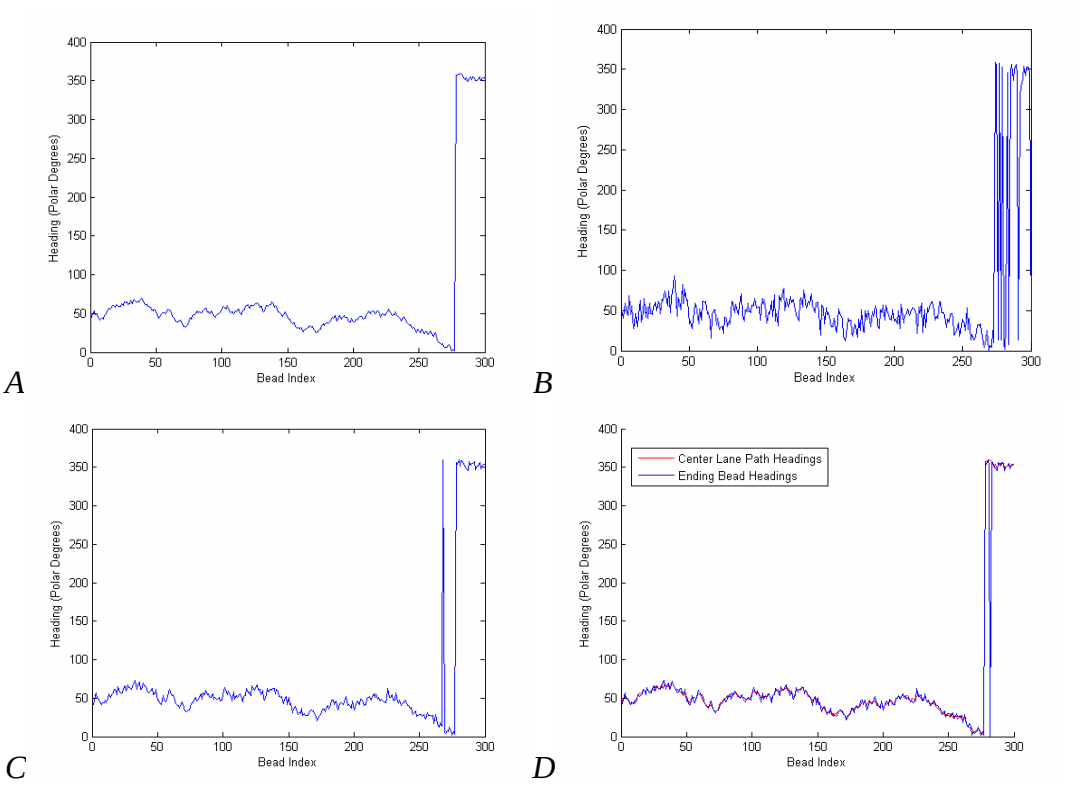


Figure 5-4. Graph A shows the Center Lane Path before adding simulated noise to the a priori bead location estimates and to the simulated physical state measurements of $\sigma_{b,GPS}$ *= 5 m,* $\sigma_{ps,GPS}$ *= 5 m, respectively. Graph B shows the starting beads generated*

from Graph A, while Graph C shows the data in the simulated physical state for the first path traversal. Graph D shows the beads after running the simulator once through the entire path. Graph D is a combination of Graphs B and C, and is not the same as either one. However, the beads can be improved with more physical state data. In the tenth path traversal, the physical state data (Graph E) for that iteration was still poor, but the fusion of data from 10 traversals resulted in the beads shown in Graph F. These beads had a standard deviation of ~1.5 m in both latitude and longitude (as opposed to ~3.5 m after a single path traversal).

The simulator combines headings as well, as outlined in Section II. The behavior of the algorithm in regards to headings is the same as with the latitude/longitude portions of the bead and physical state data; examining a graph of GPS coordinates is just more intuitive. Improving bead positions is also more important, since the headings of a path can be derived from the path's latitude/longitude bead locations, but the reverse is not true. As an example of heading improvement, Figure 5-5 shows an all-like update from the heading's perspective.



A *B*

C *D*

Figure 5-5. Graph A shows the Center Lane Path's headings before $\sigma_{b,heading}$ *= 11.0 degrees,* $\sigma_{ps,heading}$ *= 3.0 degrees. Graph B shows the starting bead headings generated from Graph A, while Graph C shows the heading data in the simulated physical state. Graph D shows the bead headings after running the simulator once through the entire path, with an ending standard deviation of ~2.9 degrees. Note the result headings follow the better data in the physical state, like a typical all update. The large spikes in the graphs come from the fact that for headings, zero degrees is equal to 360 degrees.*

Recall that the simulator is built to handle more than one standard deviation for a series of measurements, whether those measurements are latitudes, longitudes, or headings. On a real course, the reliability of predetermined beads may vary due to what imagery is available beforehand, especially if parts of that imagery are obscured. The confidence in GPS sensors may also vary with weather and nearby trees, etc. The behavior when using more than one standard deviation per measurement series follows from the behavior observed when using only one standard deviation per series. Using path-varying standard deviations creates sub-problems identical to the original problem; each sub-problem, like the original, has one standard deviation to consider for each data source. Figure 5-6 illustrates this concept.

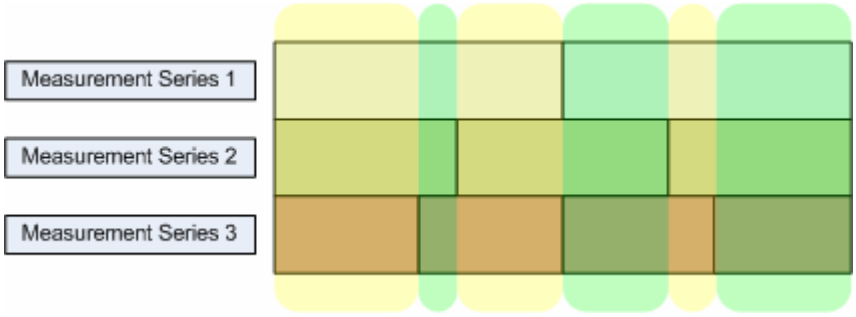


Figure 5-6. Consider three measurement series from three information sources (for illustrative purposes—the concept does not change based on the number of sources involved). Each measurement series has a time (or path)-varying standard deviation. In this instance, the standard deviation of each measurement source varies stepwise (the change from one to another is marked by a vertical black line). Each unique set of three standard deviations (highlighted in yellow or green) produces a smaller version of the original problem, where each measurement source has a constant standard deviation.

Figure 5-7 shows how the same path may have sections more like an all update

(where the beads try to match what is in the physical state) or a nothing update (where the beads change little). Figure 5-7 also shows that sections of a path with at least one source of good data improve more quickly than sections where both sources lack confidence.

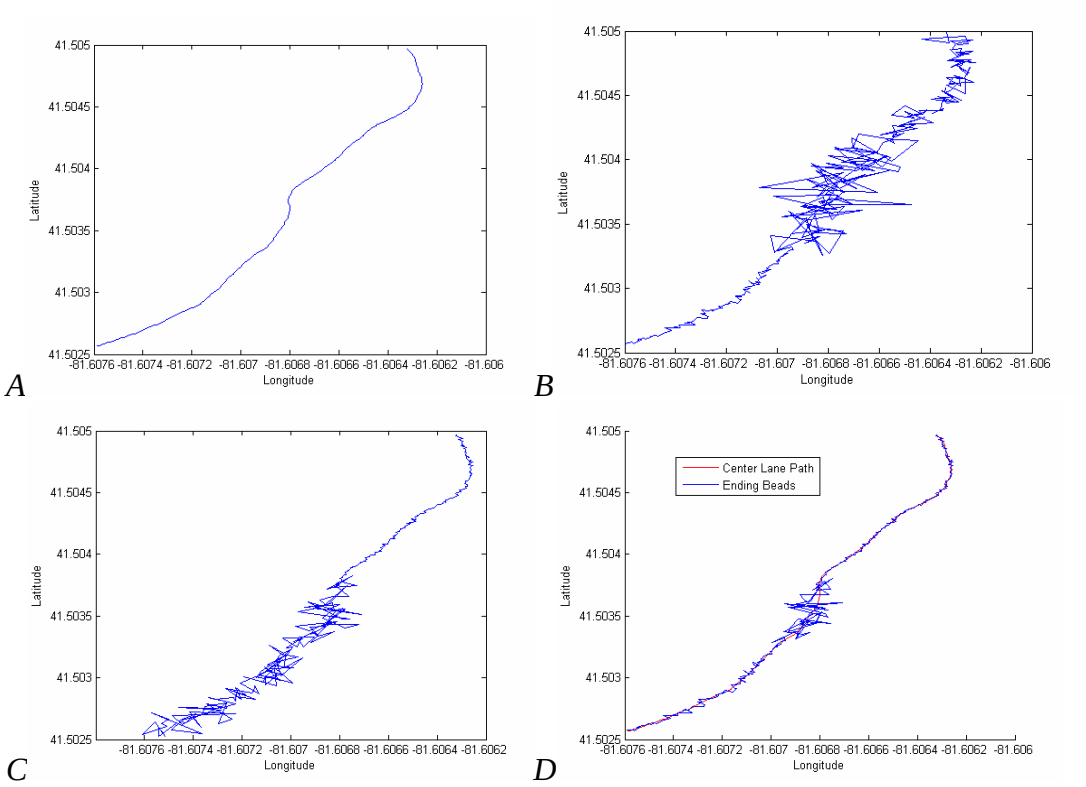


Figure 5-7. Graph A shows the Center Lane Path. Graph B shows the starting beads generated with $\sigma_{b,GPS}$ *= 1, 8, and then 3 m. Graph C shows the physical state generated with* $\sigma_{ps,GPS}$ *= 5 and then 0.5 m. Graph D shows the beads after running the simulator once through the entire path. Note that Graph D has patches of different improvement levels based on the data available to each portion of the path.*

The optimal estimation method of combining measurements has an advantage over setting bead updates into all update mode versus nothing update mode for the entire map. By combining the variances involved, the system itself can decide how the update will behave on different parts of the map. Compared with the DEXTER system that chose between all updates or nothing updates, these simulations can produce middle ground when neither an all update nor a nothing update is appropriate.

Chapter 6: Discovering Bead Bias

Section I: Batch Processing to Determine Bead Bias Estimates

The method the simulator in Chapter 5 uses to combine beads and physical state data relies on the erroneous measurements having no bias, only Gaussian errors. However, at the Urban Challenge National Qualifying Event, Team Case found a positional shift between DEXTER's measurements and DARPA's RNDF, and we had to move the RNDF accordingly to compensate. Since beads come from RNDFs initially, the next experiments attempt to estimate bead bias in simulation. The bead bias is known to have only a potential north-south shift and a potential east-west shift; any warping or rotation that might be present in the map is assumed to be too small to be noteworthy, an assumption justified by Team Case's Urban Challenge experience. The simulations rely on the assumption that the beads alone contain a bias. Physical state measurements from a GPS sensor, while noisy, are presumed to have zero bias.

For a simulation to eventually discover a bias, first it must add in such a bias when generating beads. Generate_Beads_with_Global_Bias.vi still introduces Gaussian errors as before, and still accommodates the application of a series of standard deviations when creating a list of beads. The added functionality comes from two new inputs: North-South Bias and East-West Bias. North and east biases are expressed in positive meters, while south and west biases are written as negative meters. Before a bead receives Gaussian errors, it first moves due north (0 degrees) or south (180 degrees), then due west (270 degrees) or east (90 degrees) by the distances in the two biases. The same bias quantities affect each and every bead the same amount, creating a truly global bias

like the one Team Case saw in Victorville, as opposed to any inconsistent bias found on only some of the roads.

As in past simulations, No_Offsets_Batch_Bead_Bias.vi creates a Center Lane Path, a set of physical state data, and a bead list. The code then examines each bead and its corresponding physical state data exactly once. The simulations have the advantage of knowing which points in the beads and simulated physical state readings correspond to each other; in a scenario that had to deduce such pairs, path areas roughly following a cardinal direction may be more likely to miss shifts along the same direction they travel. For example, a road that runs west to east may miss some of the east-west shift when trying to match points.

DEXTER utility functions find the heading from the simulated physical state point to the corresponding bead point and the number of meters between the two. The heading found is relative to 0 degrees at due north and increases counterclockwise. The code considers the distance between the two points as a hypotenuse of a right triangle, and the angle from the physical state point to the bead as one of the acute angles in the same right triangle. Then, using trigonometric rules, the cosine of the angle between the two points multiplied by the distance between them is the north-south component (in meters) of the distance from the physical state point to the bead. Similarly, using the sine instead of the cosine yields the number of east-west meters involved in the shift. See Figure 6-1 for an illustration of the values used in these trigonometric calculations.

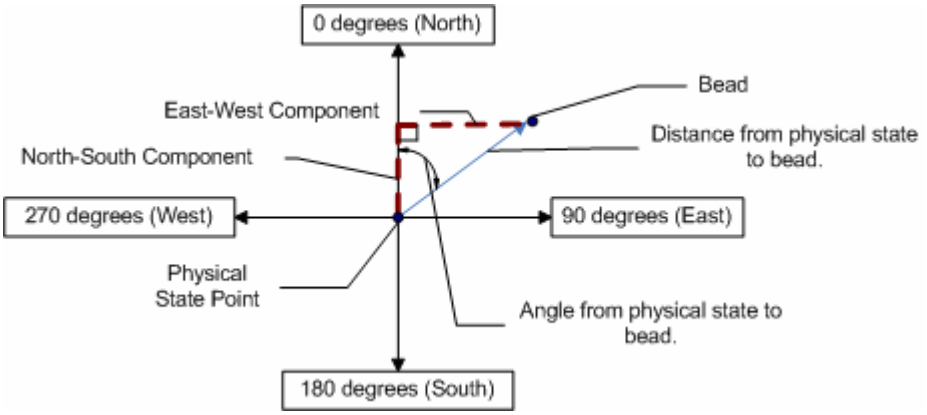


Figure 6-1. Assume a physical state point is at the origin of the graph. Then the associated bead can be in any of the four quadrants at some angle from the physical state point. Four of these possible angles lie on the axes and indicate shifts in pure cardinal directions. The blue arrow shows an example distance to a bead at an example angle, and labels the two components of that distance.

Of course, some bead shifts are negative, and some are positive. The cosine and sine functions correspond nicely to the convention used to insert biases (west and south negative, north and east positive). The cosine will be negative if its argument is greater than ninety degrees but less than 270 degrees—or when its argument points south. As expected, 270 degrees and ninety degrees have no north-south component because the cosine of both measurements is zero, and cosines are positive at all northern degrees (the two top quadrants). The same argument holds for the sine and the east-west convention; the sine gives zero for directions directly north or directly south, negative numbers to the west (the two left quadrants), and positive numbers elsewhere (east). Because of these mathematical properties, using the cosine and sine produces bias estimates with the same signs as the biases added.

The bias components discovered from a bead and its physical state reading are both ultimately the result of a subtraction between those two points. Because two independent quantities are combined, the variances associated with the new north-south

and east-west biases are the sums of the appropriate variances in the bead point and in the physical state point [46]. The code packages the sum of the latitude variances with the north-south distance and the sum of the longitude variances with the east-west component. By using the variances, more confident areas in the paths have added ability to improve the estimates produced by the batch process.

Once the simulator processes the physical state and bead paths, it calls Average_N_Measurements_(of_One_Quantity).vi twice; once to calculate the end east-west bias estimate, and once for the final north-south bias estimate. The estimates produced should resemble the driving biases in both magnitudes and signs. Figure 6-2 shows visually how No_Offsets_Batch_Bead_Bias.vi calculates an estimate of the global bead bias.

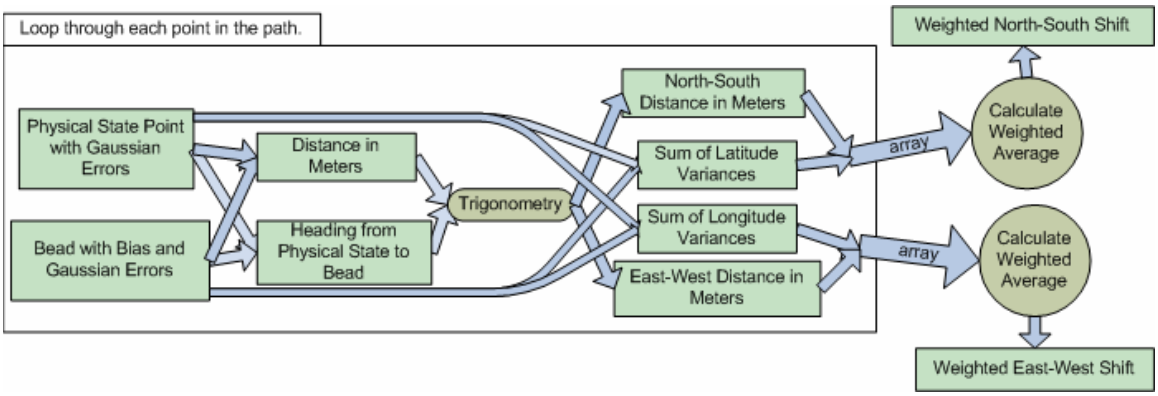


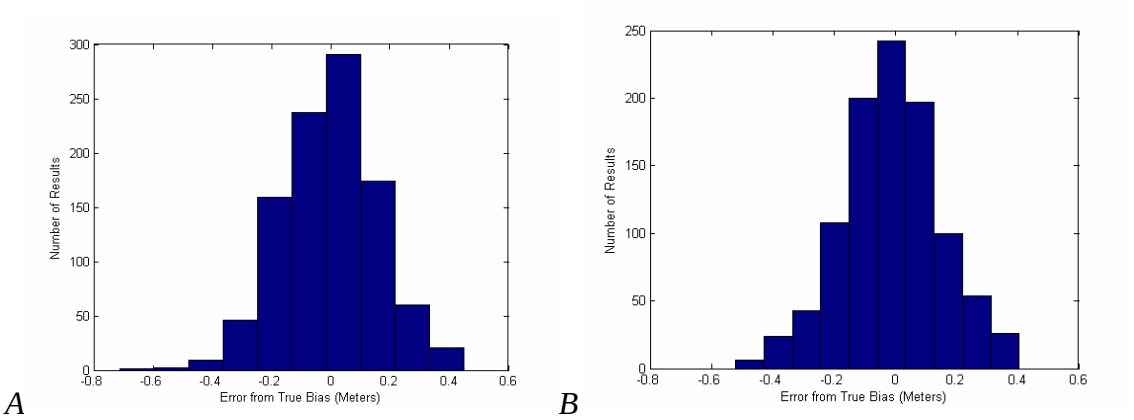
Figure 6-2. A flowchart of the method that No_Offsets_Batch_Bead_Bias.vi uses to calculate estimates of a global north-south bead shift and a global east-west bead shift.

In order to amass statistics, No_Offsets_Batch_Bead_Bias.vi computes bias estimates for a large number of independent runs. For each run, the bead bias works with a new random path and a new random set of Gaussian errors, but with the same driving bead biases and the same standard deviations creating the Gaussian errors. Table 6-1 summarizes the sources of error shared by each run.

Type	Data
Path	Path Driven = Center Lane Path, Coordinates in Center Lane Path: variable
Bead Errors	$\sigma_{b,GPS} > 0$ $\sigma_{b,heading} > 0$, bias $\neq 0$
Physical State Errors	$\sigma_{ps,GPS} > 0$, $\sigma_{ps,heading} > 0$, bias = 0
Road Detection Errors	$\sigma_{rd,offset} = 0$, $\sigma_{rd,heading\ correction} = 0$

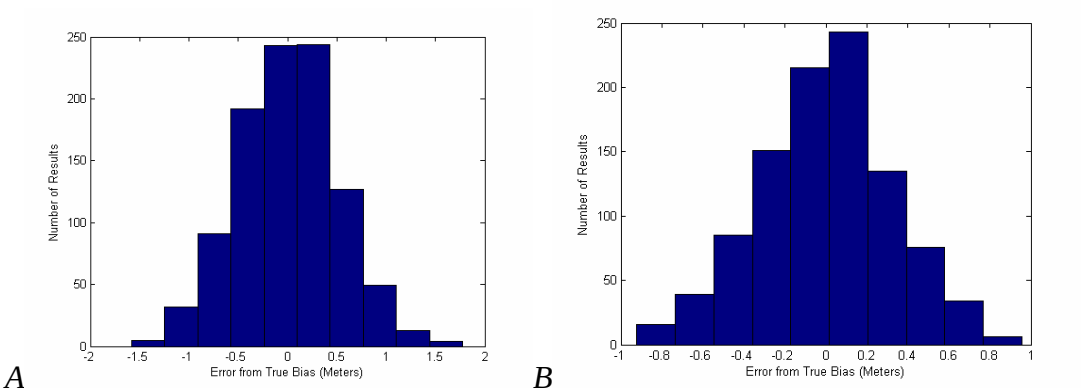
Table 6-1. Summarizing the data used for the following tests in correcting bead biases.

Each run produces an estimate of the map registration biases. These estimates are imperfect, and their distribution is another Gaussian. Errors in bias identification can be calculated and graphed, as shown in the histograms in Figures 6-3 and 6-4. Since those figures graph the errors, they are all centered on zero (truth). However, note that the behavior does not change if the tests run on negative, positive, or zero biases; the actual value of a bias does not affect the accuracy of the algorithm. Note that all of the histograms have Gaussian-like curves whose variances are close to the variances for the bias estimates as calculated in each run of algorithm. As the number of runs approaches infinity, the distribution of bias-identification errors would have a variance that converges identically on the theoretical value.



A *B*

Figure 6-3. Each run used 300 beads and 300 physical state points, $\sigma_{b,GPS}$ = 2 m and $\sigma_{ps,GPS}$ = 2 m. There was a 2.6 m bias north in these tests but no bias east or west. The graphs show the results from running 1000 such experiments. Graph A shows the north-south results with variance 0.0257 m^2, and Graph B shows the east-west results with variance 0.0256 m^2. The calculated variance for both directions at the end of each individual test was 0.0267 m^2.



A *B*

Figure 6-4. Both of the graphs show the discovery of a bias 4.3 m west (a negative number to the algorithm) for 1000 different runs, with $\sigma_{b,GPS}$ = 7 m and $\sigma_{ps,GPS}$ = 5 m. In Graph A, 300 coordinates were used, ending in a predicted variance of 0.247 m^2 per test and a measured variance of 0.259 m^2 for all 1000 tests. In contrast, Graph B used 700 coordinates and ended with a variance of 0.106 m^2 across all experiments (where 0.106 m^2 were expected for each test).

Figure 6-4 highlights which factors do affect the accuracy of the results—the same factors that mathematically reduce the predicted variance of any one result. Graph A in Figure 6-4 has less confident data (i.e., initial bead estimates and physical state measurements have higher standard deviations) than Graph B in Figure 6-3, and the increased variance in Figure 6-4, Graph A, shows this cost. However, Graph B in Figure

6-4 shows that adding more points to a path with little confidence increases confidence in the bias estimates. Once again, the equations used to compute new measurements from uncertain data improve with each new data point (however uncertain). Therefore, when using this method with sensors of low confidence, driving the robot farther to collect more sensor readings can aid the quest to determine the global bead bias.

Section II: Running Updates to Determine Bead Bias Estimates

Bead shift estimates need not be decided at the end of a run. Instead, estimates can be improved upon as the simulation "drives." By doing so, a robot (and anyone watching its progress) can learn about the current map registration online. The robot does not learn how to drive better in this process; it just learns more about the map. For No_Offsets_Running_Bead_Bias.vi, the simulator evaluates the bead bias into the bead list in the same manner as in the batch processing tests. The general path data and errors follow the same scheme as that for the batch method (see Table 6-1).

For No_Offsets_Running_Bead_Bias.vi, the user specifies initial guesses for both bias quantities and their variances; in order to constantly improve an estimate, a starting estimate must be provided. The experiments here use zeroes for the initial guesses; with no idea what a bias might be, zero bias is assumed. The variances for the guesses start at 1.0^{13} m^2 each, essentially giving the starting guesses no confidence so that actual data will quickly replace these numbers.

While iterating through a path, the code calculates north-south and east-west components using the same process and trigonometric rules as in the batch processing case. The variances of the components are also the same sums as in the batch process. However, now the two calls (one for east-west and one for north-south) to

Average_N_Measurements_(of_One_Quantity).vi occur each time a new bead and physical state point pair is processed. For the first pair, Average_N_Measurements_(of_One_Quantity).vi combines the initial guesses of shifts and variances with the results for that pair. Then the two new weighted averages and their variances replace the initial guesses and their variances. As the path is "driven," the result of the last step is always combined with the new data.

As with the batch process, No_Offsets_Running_Bead_Bias.vi handles multiple runs, each with new paths and new specific errors but with the same driving biases and driving standard deviations. The initial guesses of bias quantities also start out the same in every run. Figure 6-5 shows, in the scope of one run, the organization of the simulation code that uses the running update method (contrast with Figure 6-2, which illustrates the batch processing method). Figure 6-5 shows that the progress of each bead bias estimate in each run is recorded for later analysis.

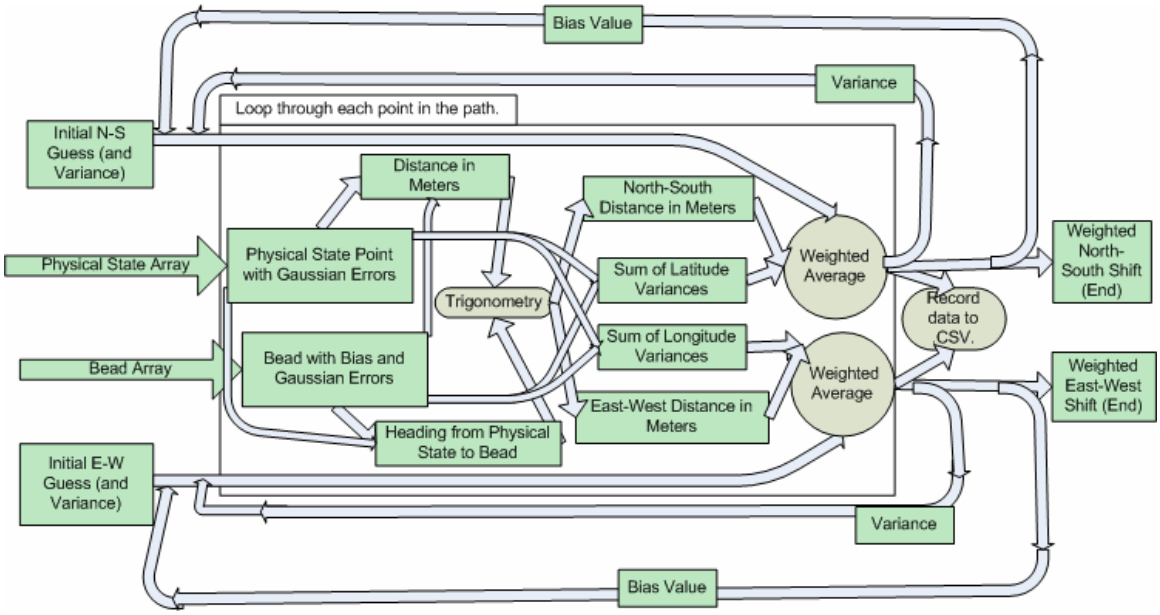
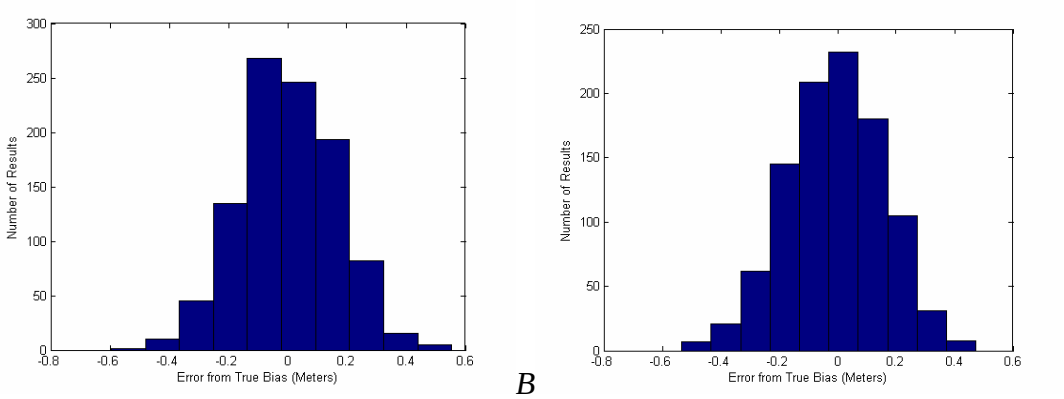


Figure 6-5. A flowchart of the method that No_Offsets_Running_Bead_Bias.vi uses to calculate estimates of a global north-south bead shift and a global east-west bead shift.

The math in the running update method is the same as that used to arrive at bias estimates in the batch processing method. Figure 6-6 shows histograms created under similar conditions to those in Figure 6-3. Once again, the algorithm's calculated variance for each test result is verified by the variance of the errors in the results after 1000 tests. And once again, the shape of the errors after many runs is roughly Gaussian.



A *B*

Figure 6-6. Each run used 300 beads and 300 physical state points, $\sigma_{b,GPS}$ = 2 m and $\sigma_{ps,GPS}$ = 2 m. There was a 2.6 m bias south (a negative bias) in these tests and a 4.3 m bias east. The histograms show the results after running 1000 such experiments. Graph A shows the north-south results with variance 0.0254 m^2, and Graph B shows the east-west results with variance 0.0270 m^2. The calculated variance for both directions at the end of each individual test was 0.0267 m^2.

Though the results are the same in both methods, the running update method is more like learning than the batch processing method, because it makes intermediate results available (if only to the user) in real time. Such intermediate data may be useful if the robot or user has to decide when enough points have been collected. Figure 6-7 illustrates this feature by showing progress for several bias estimate experiments. As the running update method moves through the beads, the bias estimates level off and becomes more stable. The simulator code places the initial guess at the beginning of each plot for comparison.

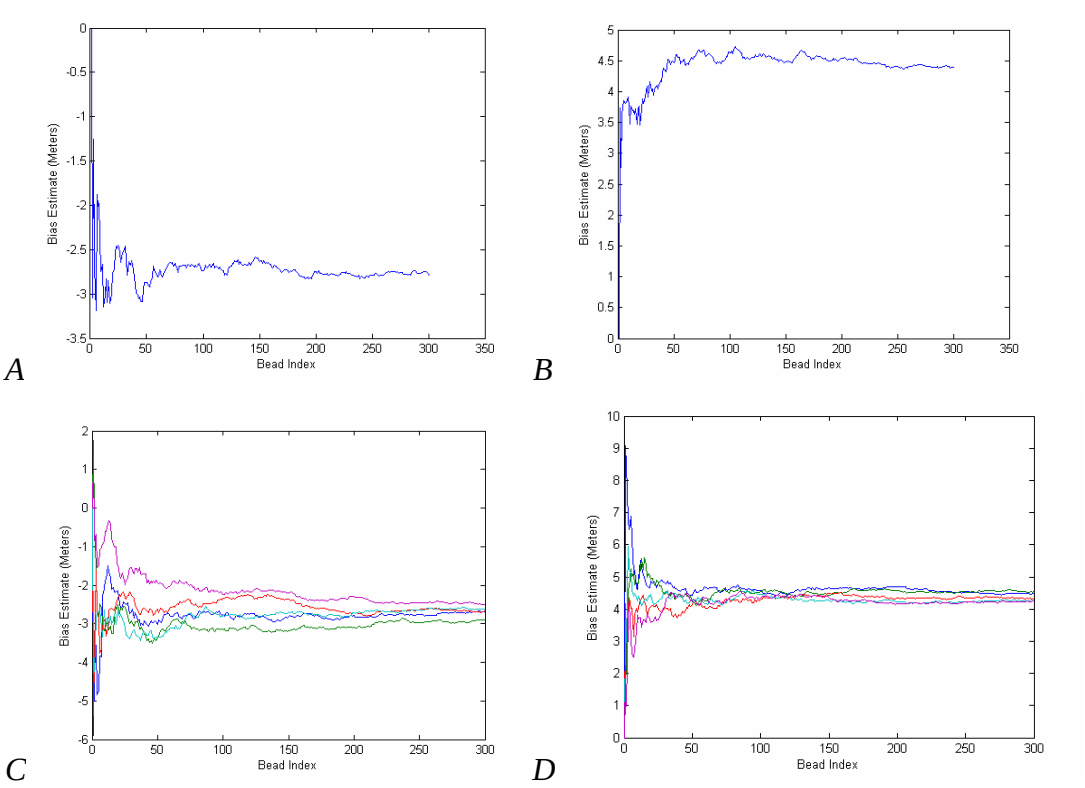


Figure 6-7. These graphs show some detail from the 1000 runs in the previous figure (Figure 6-6). Graphs A and C show the discovery of a bias 2.6 m south, while Graphs B and D show estimates approaching a bias 4.3 m east. Graphs A and B each show one example experiment, while Graphs C and D show five runs each.

Section III: Running Updates Used with Bead Improvement

The running update method can be used directly to help bead-improvement simulations, namely those outlined in Chapter 5. In Chapter 5, the bead errors were assumed to be unbiased, but this may not always be the case. The following bead-improvement simulator follows the scheme in Table 6-1 over the scheme in Table 5-1. To simulate the improvement of biased beads, first No_Offsets_Running_Bead_Bias.vi creates its own path data to estimate the biases, with its own bead data, its own physical state data, and its own Center Lane Path. The VI that uses bias estimates creates a second set of path data in parallel. In general, the robot could drive around a competition area to discover map registration values before the event. However, the true biases need to be

consistent for both paths (the Bias Discovery Path and the Bead Improvement Path) in order for the results of the former to apply to the latter. Also, this simulator assumes that the qualities of the robot's sensors do not change, so that both path sets use the same standard deviations to drive errors.

When the simulator arrives at the bead-improvement code, it pulls in both bias estimates from No_Offsets_Running_Bead_Bias.vi. Using these bias estimates, every bead is shifted north or south and east or west before being combined with physical state data as per the optimal estimation equations. Because the bias estimates are imperfect, these movements introduce more variance into the original bead. Fortunately, No_Offsets_Running_Bead_Bias.vi calculates the variances of its estimates. The estimate variances (one for latitude, one for longitude) are added to the original bead variances to arrive at the new bead variances; as before, variances are added when two independent sources act on a measurement. Figure 6-8 illustrates how two sets of path data are used for these simulations.

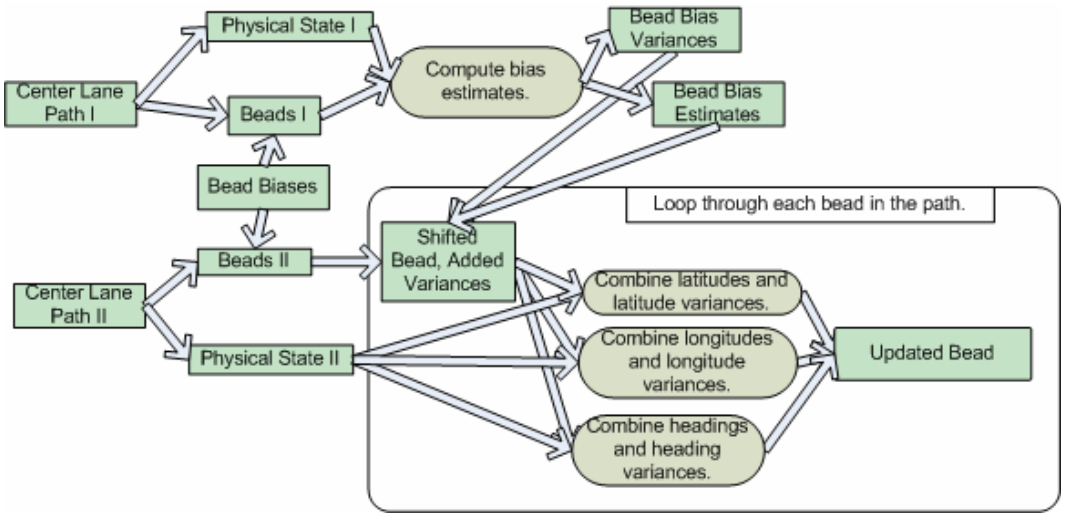


Figure 6-8. From the two Center Lane Paths, two sets of beads and of physical state data are created. Though individual errors vary, $\sigma_{b,GPS}$ and $\sigma_{b,heading}$ are the same for both sets of beads, and the same is true for the physical state data sets.

Figure 6-9 displays the results of not only introducing bead biases to the bead-improvement algorithm, but also estimating those biases and adjusting for them. Only one run of No_Offsets_Running_Bead_Bias.vi was used to get both bead bias estimates, as opposed to using the means of many runs. The former case is faster to accomplish, and as Figure 6-9 shows, still helpful. In Victorville, Team Case had to estimate the map registration offsets in order to improve the quality of beads; this work shows that the map registration offsets may be calculated algorithmically.

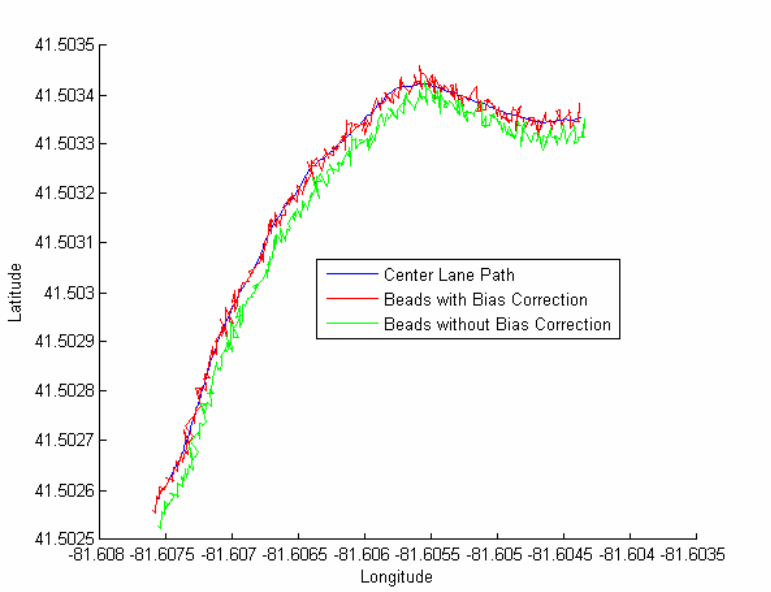


Figure 6-9. Both the path used to discover bead bias estimates and the path used to improve beads were 300 coordinates long, but they were not the same. Both of these paths were generated with a true global map shift of 7 m south, 6 m east. The paths shown above were all created by the bead-improvement algorithm. The tests in green and red above used all of the exact same data in terms of physical state and starting beads, and each used exactly one iteration through the path in order to improve. Both identified paths are noisy due to using $\sigma_{b,GPS}$ *= 2 m and* $\sigma_{ps,GPS}$ *= 2 m. The only difference between the red and green paths is that the red path corrected its bead data using bias estimates, and the green path did not. The red path is thus noticeably better, as it is centered on the Center Lane Path and not shifted from it.*

Chapter 7: Using Road Detection Information

Section I: Introducing Road Detection

Mobile robots can gather non-GPS information; other sensors, such as cameras or LIDAR scanners, can aid in navigation. Typically a mobile robot's vision may find features (such as lane markings) to follow, or features (such as obstacles) to avoid [47]. In the 2005 DARPA Grand Challenge, Ohio State University used digital cameras (and a host of other sensors) to generate points to keep their robot driving [30]. DEXTER's road detection system for the 2007 DARPA Urban Challenge used color cameras, infrared cameras, and laser scanners known as LIDAR (LIght Detection And Ranging) units [38]. Other teams at the 2007 DARPA Urban Challenge also employed LIDAR units. Therefore, instead of "vision," these tests use the phrase "road detection." The individual sensors that make up a road detection system are immaterial, and indeed vision may be included. Road detection as a whole observes the road while a robot is driving and helps the robot know where it is in relationship to the center of its lane.

As with any sensor data, road detection quantities include errors, which may arise from various factors. Hardware itself can be the source; for example, vision systems make demands on processing speed and power consumption [48]. Restricted camera views may also limit accuracy, especially for a robot that needs to make tight maneuvers or to travel at higher speeds [49]. The environment can fool road detection with changes in illumination, defects in road surfaces, pedestrians, the absence of lane markings at intersections, and more. Illumination particularly can make it very difficult for lane detection to tell yellow lane lines from white ones [50]. DEXTER's cameras had issues

with changeable pavement reflections, missing lane lines, obstructions, and camera vibrations caused by both the robot and the road [38]. The simulator combines these sources of error into overall variances in the road detection measurements. As in previous simulations, the errors introduced into road detection follow a zero-mean Gaussian distribution. With the addition of road detection, the simulations can remove the restriction that the Path Driven—where the robot travels—must be the same as the Center Lane Path—where the robot should drive. When running autonomously, it is rare for a robot to be able to follow such a restriction, due to errors in the system and the kinematics of driving. DEXTER, for example, definitely deviated from the centers of his lanes. The bead data, since it is a set of goals, is still based on the Center Lane Path. Physical state measurements, on the other hand, correspond to the robot's actual coordinates, and as such their errors must center on the Path Driven. Road detection must reconcile these two sets of data, recognizing that the Path Driven may not correspond to the center of the intended lane.

Section II: Simulated Road Detection Values

Each point of road detection data contains a lateral offset, which gives the simulation an estimate of how far to the left or right the robot is displaced from the center of its lane. The road detector can also estimate a heading correction relative to the direction of the road. Ideally (with no errors involved in the system), the heading correction from the road detector added to the robot's physical state heading would equal the tangent angle (road heading) of the intended center of lane. In practice, measuring a lateral offset may be done with projections, by looking ahead or behind at the robot's direction of travel and the lane's direction of travel and projecting how those two paths

would relate at the robot's current location. Alternatively, a lateral offset may be estimated by looking ahead at time *t* at what the offset will be if the robot continues its current path for *x* time steps ahead of *t*, then recalling that offset when the robot reaches time *t* + *x*. In another approach, sensors such as vision cameras and LIDAR units may be placed to look sideways from the robot, so that offsets might be observed as they occur.

In the simulations, offsets must be calculated by comparing the Center Lane Path with the Path Driven. The Center Lane Path is generated as in previous simulations. The Path Driven, now generated separately, must roughly—but not exactly—follow the same course as the Center Lane Path. The Path Driven, in order to make the simulator's output more realistic, should also appear smooth enough to be drivable. Generate_Path_Driven.vi creates splines between points near the Center Lane Path in order to create the Path Driven.

The first spline's origin is the first point on the Center Lane Path. The destination of that spline is a point near the point on the Center Lane Path *p* points away from the start of the Center Lane Path, where *p* is specified according to the user's judgment call. In order for the Path Driven to look drivable, the value of *p* must encompass enough points to keep the splines from being choppy in relationship to each other, but not so many points that the Path Driven can veer too far away from the Center Lane Path. The destination point of the spline comes from moving the point designated by *p* on the Center Lane Path in a random direction by some random number of meters *m*, where *m* has a standard deviation of 2 m. The spline results replace the exact number of points between both endpoints, and these interim points, along with their destination, begin the start of a new path. Since headings are needed to continue splining, the headings of the

new points are recalculated.

The destination point (*after* its movement) becomes the next starting point for the next spline, and the process continues. The Spline Spacing *p* chooses the next destination by jumping along the Center Lane Path *p* points from the last destination (*before* its movement). Since spline origins come from previous spline destinations, and no point needs to be moved twice, no origin is moved. As a result, the first points in both the Center Lane Path and the Path Driven match. The last set of splined and replaced points may be shorter than previous sets, because the number of points a spline can replace cannot exceed the number of points left in the Center Lane Path. The last point in the Path Driven is the same as in the Center Lane Path. Once the Path Driven is complete, its headings are completely recalculated based on the relative positions of its member points. Note that the Path Driven and the Center Lane Path have exactly the same number of points. Figure 7-1 gives more details on the process in Generate_Path_Driven.vi, while Figure 7-2 shows an example of how a possible Path Driven is derived from its associated Center Lane Path.

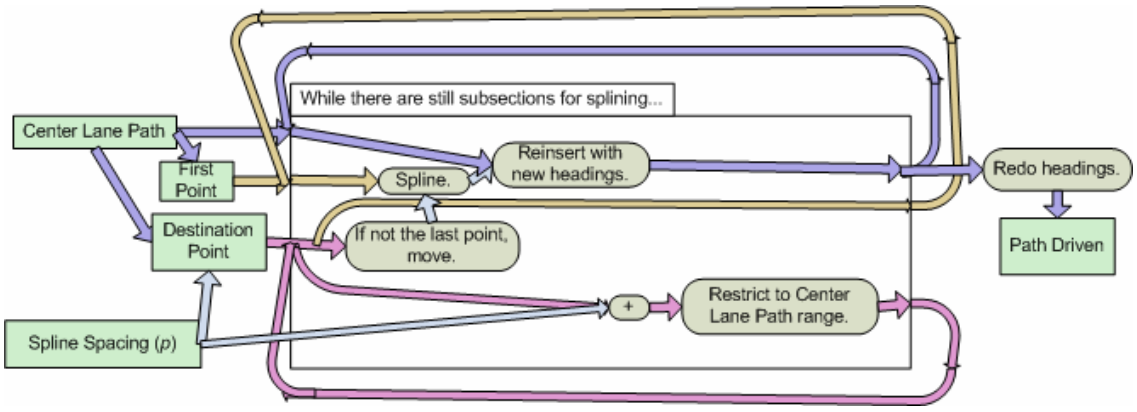


Figure 7-1. The flowchart of the code in Generate_Path_Driven.vi. The pink arrows follow the spline destinations, while the yellow-orange arrows highlight the spline origins. The purple arrows show how the Center Lane Path is replaced in parts to become the Path Driven. Blue arrows are used for miscellaneous transfers of data.

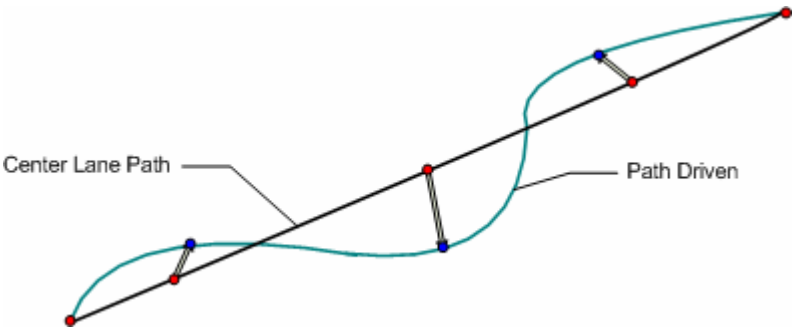


Figure 7-2. The spacing of points on this diagram is coarse in order to show the points of interest, as opposed to showing every point on either path. The red dots on the Center Lane Path move randomly to positions marked by the blue dots, and the Path Driven is splined between those blue points. Note that both paths start and end at the same places.

Once both paths exist, code calculates the true lateral offsets between them before any errors become present in the system. The definition of what a lateral offset means must be consistent and chosen carefully. A lateral offset is a distance between two points, but not any pair of points can be considered "lateral" from each other. The definition of the word "lateral" means "to the side." But there are two paths with two different sets of headings, and thus two concepts of "to the side." See Figure 7-3.

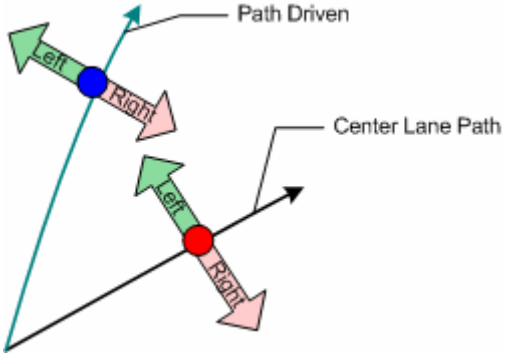


Figure 7-3. The blue dot and the red dot above have different headings, and thus different concepts of what it means to go forward. From this, two definitions of left and right follow. The blue dot on the Path Driven is left of the red dot on the Center Lane Path, but the red dot is not directly right of the blue dot.

Since the Center Lane Path represents the goal of a simulation, the lateral offsets are each defined with one endpoint resting on a point in the Center Lane Path, and with

left and right defined relative to the Center Lane Path heading at that point. However, note from Figure 7-2 that the known Path Driven points may not be directly left or right of the corresponding known points on the Center Lane Path. There is not a simple correspondence between sample points on the Center Lane Path and sample points on the Path Driven. To define consistent lateral offsets, the Path Driven is resampled to create a simple one-to-one correspondence between Center Lane Path and Path Driven points. In a real-time situation, path, road detector and physical state samples may be synchronized either by triggering or by interpolation (as is done in simulation).

For each known point on the Center Lane Path, the line going through the left and right directions (as shown on Figure 7-3) can be calculated as the line perpendicular to the line through the Center Lane Path's heading at that point. This perpendicular line intersects with the Path Driven in one of the Path Driven segments. In the special cases found at the endpoints of both paths, the line intersection occurs at the Center Lane Path point. For the other cases, the point lateral to the Center Lane Path point intersects with the perpendicular line within a line segment (endpoints included) of the Path Driven. These intersections define the resampled Path Driven. The distance from the Center Lane Path point to the intersection on the Path Driven is the Euclidean distance between corresponding pairs of points. Finding the distance from a point to a line segment is also used in the Geometric Point-to-Curve Map Matching algorithm, but here finding the correct segment is less ambiguous because the lane of interest is automatically known [5]. To distinguish between left and right offsets, by convention the offset value is defined to be negative when the Path Driven's intersection is to the right of the Center Lane Path point. Figure 7-4 illustrates the graphical calculations.

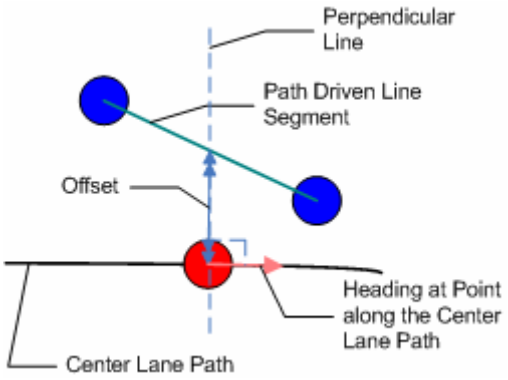


Figure 7-4. The offset is found along a line perpendicular to the Center Lane Path at a particular point. The distance along this perpendicular line from the Center Lane Path to the intersection with a segment of the Path Driven becomes the offset. In this case, the Path Driven is left of the Center Lane Path at the position of interest, so the offset is a positive value.

The code enforces offset calculations to use monotonically increasing Path Driven segments, where "increasing" segments go forward through the path and the starting and ending indices of a segment each increase numerically. Offsets can be found using the same Path Driven segment for two or more sequential Center Lane Path points. However, an offset calculation should never jump backwards to intersect a previous section of the Path Driven, otherwise the points in the new definition of the Path Driven would become out-of-order. To enforce these rules, the calculation (found in Find_Lateral_Distance_Redo_Path_Driven.vi) of the offset for a single point in the Center Lane Path receives as input the first index of the last Path Driven line segment used (as reported by the last call to the same VI). The segments tested can only advance as long as the Path Driven continues. The search for the Path Driven line segment to use ends with the first Path Driven segment that the line perpendicular to the Center Lane Path intersects. Doing so prevents skipping Path Driven segments that could otherwise be used (see Figure 7-5).

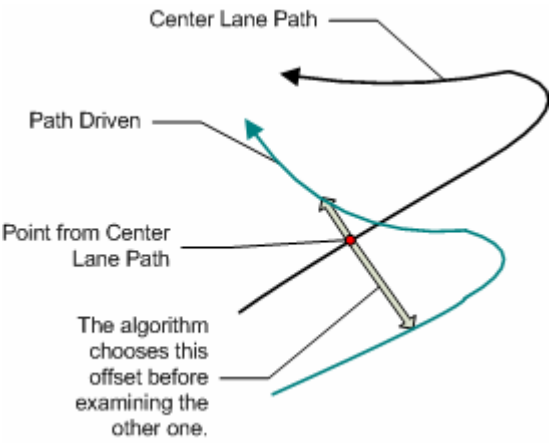


Figure 7-5. The point shown in the Center Lane Path is positioned laterally from two places along the Path Driven. However, the offset comes from the earlier Path Driven intersection, so that all the offsets that will be calculated through the turns of both paths are not skipped.

When all the offsets have been calculated, the Path Driven is redefined as a polyline connecting the resampled points. Since those points were already on the original Path Driven, the shape of the resampled Path Driven is still reasonable. This new definition of the Path Driven is as good as the last in terms of shape; the only reason the Path Driven is not generated from the beginning from a set of lateral offsets is because doing so makes it difficult to produce a smooth, drivable path. The new Path Driven has its headings redefined. The road detector must report heading corrections. To simulate a road detector, heading corrections are calculated by subtracting each new Path Driven heading from the heading of the corresponding point on the Center Lane Path. The file that handles redefining the Path Driven and finding lateral offsets is named Find_True_Offsets_Redo_Path_Driven.vi; Figure 7-6 summarizes its major features.

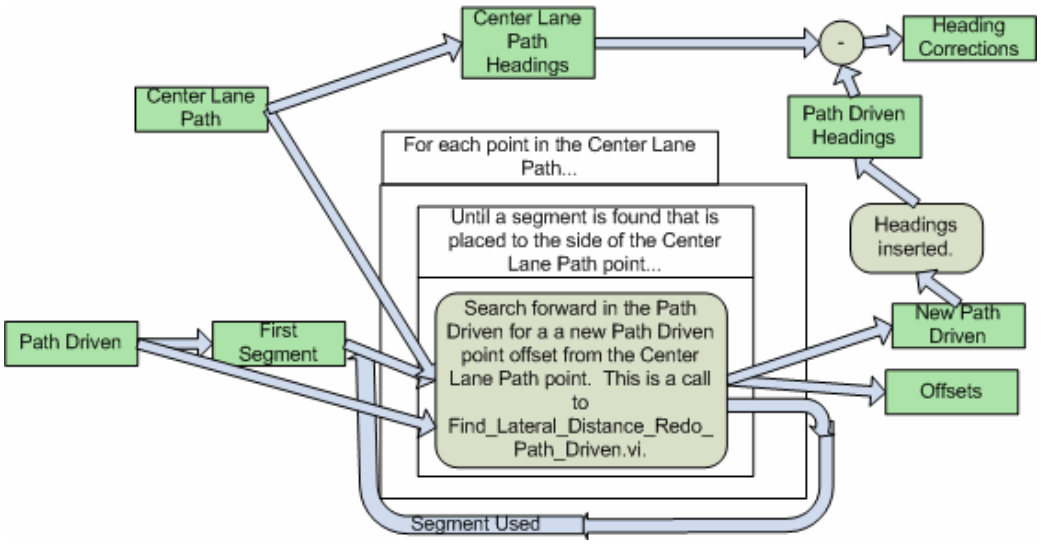


Figure 7-6. The flowchart of Find_True_Offsets_Redo_Path_Driven.vi. Find_Lateral_Distance_Redo_Path_Driven.vi computes each individual offset.

Once the simulator computes the true lateral and heading offsets, a road detector can be simulated by adding zero-mean Gaussian errors to these values. Generate_Gaussian_Distribution_of_Offsets.vi works very similarly to the other VIs that generate Gaussian errors. But since the offsets are already measured as relative displacements, the errors in offsets are directly added to the true offsets without having to move any points. Heading corrections have errors added in just like headings do, but heading corrections are not set into the 0-360 degrees range. As with other data sets, the errors in the offsets or the in the heading corrections may come from a series of standard deviations.

Because road detection values are relative (in offsets and heading corrections) and not absolute (in latitude/longitude positions and polar headings), they cannot be combined with bead and physical state data via the optimal estimation equations. For readings to be fused at all, they must relate to each other in the same coordinate system [51]. To achieve this, the simulator combines road detection values with physical state

estimates (as well as combining their variances), thus producing a virtual road sensor that expresses belief about the Center Lane Path in absolute coordinates. The beads separately represent a different belief concerning the same quantity. The virtual road sensor estimates of the center of the lane can then be reconciled with the Center Lane Path estimates contained in beads to compute optimal corrected estimates of the center of the lane. Latitude_and_Longitude_Plus_Lateral_Offset.vi combines physical state values and road detection values to produce the virtual road sensor. Since two stochastic sources are added to produce the new heading estimate, the variance of each of the virtual road sensor's headings is the sum of the variances of its two components.

The magnitude of a lateral offset, as reported by the road detector, determines the Euclidean distance between the physical state point and the new virtual road sensor latitude/longitude position. The sign of the offset tells the code whether the new position is left or right of the physical state measurement; by convention, a negative sign means the result should be left of the physical state point in order to get closer to a perception of the Center Lane Path. Recall that "left" and "right" as terms cannot by themselves define the direction from one point to another. Left and right are defined in terms of the current heading of the Center Lane Path, and the best simulated knowledge of that Center Lane Path heading comes from the bead that exists laterally (before Gaussian errors) from the current physical state point. The virtual road sensor's heading could also be used, but doing so would mean that the heading used to determine directions of left and right would have two sources of error instead of just one. For the purpose of ignoring its influence in most of the tests, the bead heading's standard deviation will be set very low.

When combining physical state and lateral offset data, care must be taken in

computing the resulting uncertainty (variance) in terms of latitude and longitude. For example, if the lateral offset corresponds to a due north correction, the resulting virtual road sensor coordinate estimate would have a latitudinal variance equal to the sum of the physical state estimate's latitudinal variance plus the road detector's lateral offset variance. However, the longitudinal variance would be equal to the physical state estimate's longitudinal variance, unaffected by the lateral offset variance. It is thus important to take into account the direction of the lateral offset to correctly compute the longitudinal and latitudinal variances contained in the virtual road sensor.

To compute the latitude/longitude variances for the virtual road sensor estimate, the road detector's lateral offset variance is multiplied by the square of the cosine of the heading from the physical state point to the new virtual road sensor point. In parallel, the same lateral offset variance is multiplied by the square of the sine of the same heading. These quantities are added to the physical state longitudinal and latitudinal variances, respectively. Figure 7-7 illustrates how the position of the virtual road sensor estimate is computed. Figure 7-8 summarizes the process through which the calculation happens.

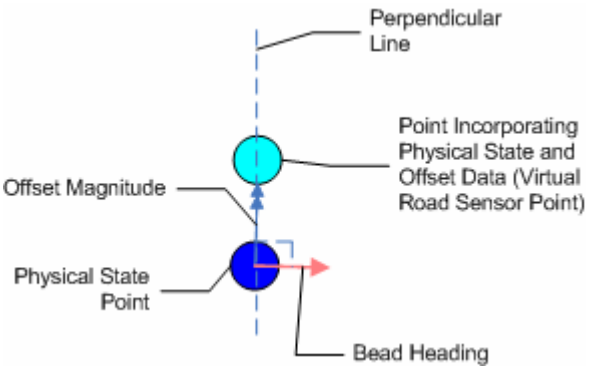


Figure 7-7. The virtual road sensor point created by Latitude_and_Longitude_Plus_Lateral_Offset.vi is positioned laterally from the physical state point based on the heading tangent to the center of the lane and the offset magnitude. The sign of the offset decides the relative direction from the physical state point to the virtual road sensor point.

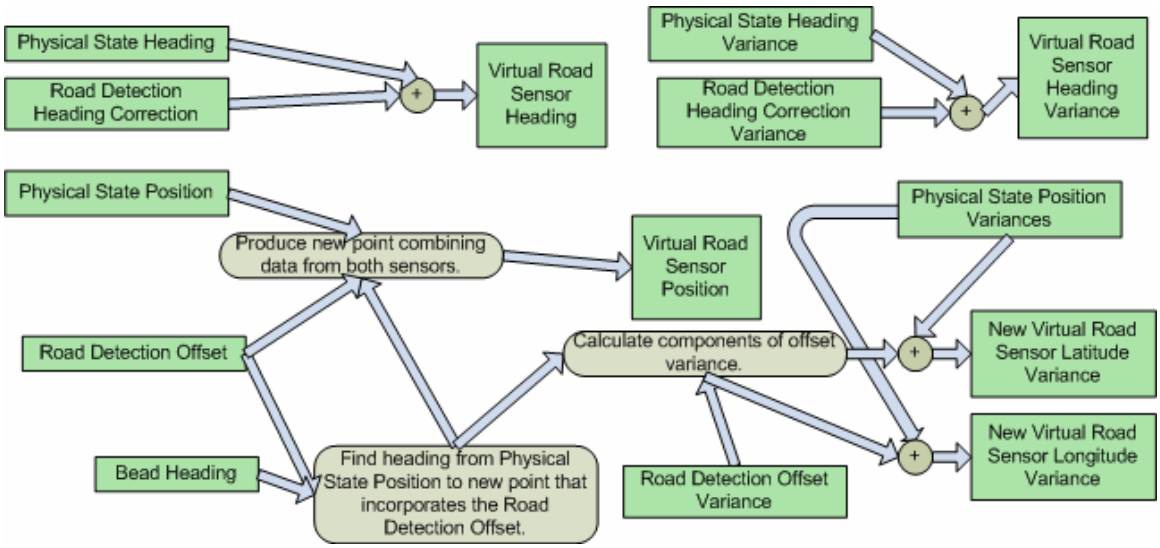


Figure 7-8. Latitude_and_Longitude_Plus_Lateral_Offset.vi uses road detection and physical state data to produce a new latitude, longitude, and heading, as well as the variances for each of those three pieces of data, thus creating a virtual road sensor.

Section III: Road Detection Experiments

As Team Case learned with DEXTER, a robot is very likely to drive off the center of its lane when controlling itself autonomously. These next experiments introduce a Path Driven different from the Center Lane Path. In some cases, the simulator assumes that the Path Driven is the Center Lane Path, even though this is no longer true. These older methods are performed for comparison with the cases that use road detection data to compensate for the differences between the Path Driven and the Center Lane Path. Table 7-1 summarizes the assumptions for using road detection data. Note that the standard deviation for bead heading is very low, so that the distinction of left vs. right in road detection offsets will introduce negligible error when combining lateral offset data with physical state data.

Type	Data
Path	Path Driven is different from Center Lane Path,

	Coordinates in Both Paths: 300 points, Spline Spacing: 100 points
Bead Errors	$\sigma_{b,GPS} > 0$, $\sigma_{b,heading} = \varepsilon$ (i.e., positive but negligible), bias ≠ 0 in map registration tests, bias = 0 otherwise
Physical State Errors	$\sigma_{ps,GPS} > 0$, $\sigma_{ps,heading} > 0$, bias = 0
Road Detection Errors	$\sigma_{rd,offset} > 0$, $\sigma_{rd,heading\,correction} > 0$

Table 7-1. Road detection data is now a factor in the simulations.

Basic_Test_Offsets_Used.vi greatly resembles Basic_Test_Path_Driven_on_Center_of_Lane.vi, but the former creates and then uses road detection values. The physical state data in Basic_Test_Offsets_Used.vi comes from the Path Driven instead of the Center Lane Path, since the two paths no longer match. After the physical state and road detection data combine to produce the virtual road sensor, estimates concerning the Center Lane Path may be combined using the optimal estimation method. Figure 7-9 shows the structure of Basic_Test_Offsets_Used.vi. For the tests to come, the algorithms in Basic_Test_Offsets_Used.vi and in Basic_Test_Path_Driven_on_Center_of_Lane.vi are run in parallel on the same data in order to compare the results.

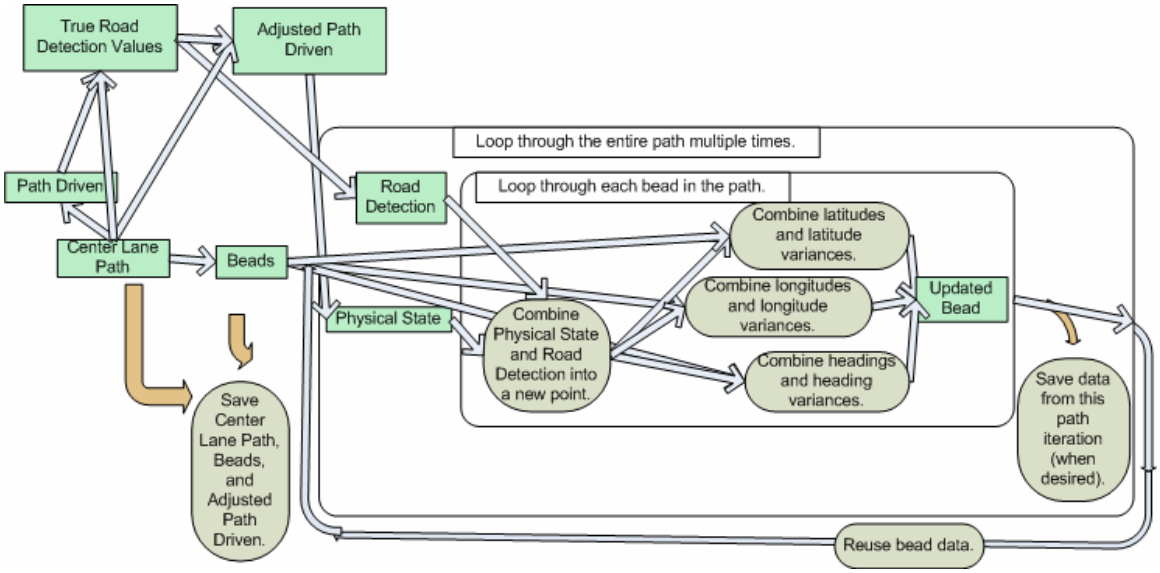


Figure 7-9. Basic_Test_Offsets_Used.vi varies from Basic_Test_Path_Driven_on_Center_of_Lane.vi in that the former, shown here, creates and incorporates road detection data as shown.

Figure 7-10 shows that when beads are confident, more so than the other data, road detection is not really needed, because the beads already know where the Center Lane Path is. However, beads cannot always be guaranteed to be of such good quality. Figures 7-11 and 7-12 show cases when the physical state or the road detection data are most confident, respectively. Recall that using road detection to influence physical state actually increases the physical state variances. Because of this, and as the figures illustrate, using road detection does not increase confidence. What road detection does instead is center the bead results on the Center Lane Path as desired. Without the benefit of the road detector, the robot could only infer that its beads should lie on the Path Driven, as reported by the physical state estimates. Figure 7-13 reinforces the idea that using road detection improves identification of the Center Lane Path by accounting for discrepancies between it and the Path Driven. In Fig 7-13, the physical state and road detection data had equally strong confidences, and were both more confident than the

beads.

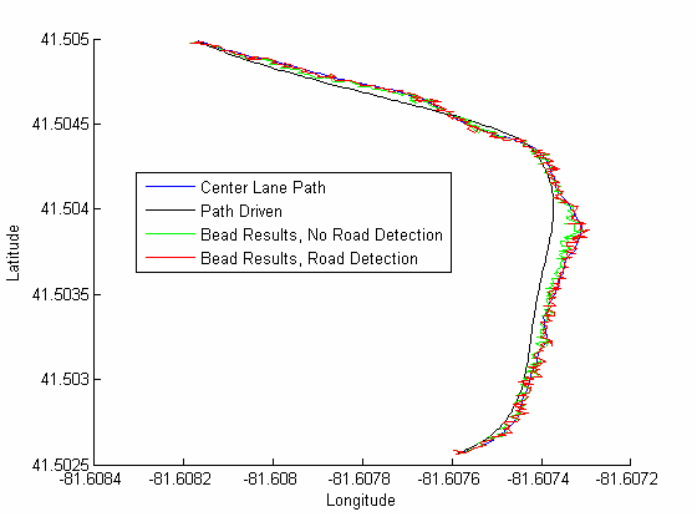


Figure 7-10. This figure shows beads after ten iterations, with $\sigma_{b,GPS}$ = 1 m, $\sigma_{ps,GPS}$ = 6 m, and $\sigma_{rd,offset}$ = 5 m. When the confidence in the beads is noticeably the strongest of all confidences, little input from the other sensors is used, and road detection can be ignored. Notice, however, that after ten iterations a portion of the beads that used no road detection started to drift slightly towards the Path Driven and away from the Center Lane Path.

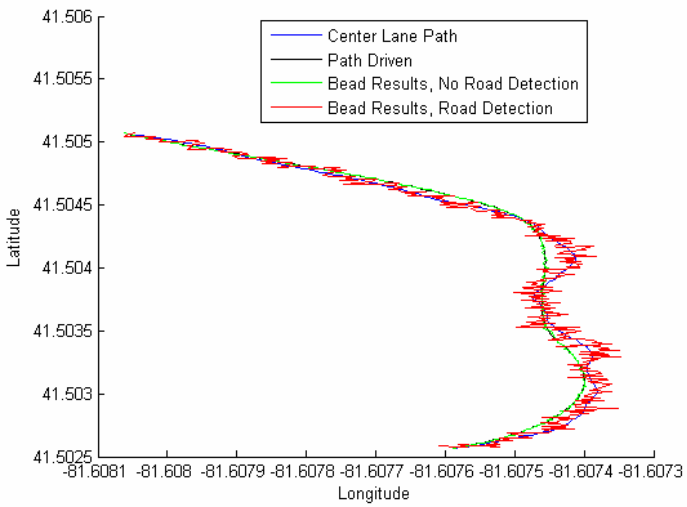


Figure 7-11. This figure shows beads after ten iterations, with $\sigma_{b,GPS}$ = 6 m, $\sigma_{ps,GPS}$ = 0.5 m, and $\sigma_{rd,offset}$ = 5 m. Because using the road detection data added noticeable error to the physical state, beads that used road detection became more variable than beads that ignored it. However, the path that ignored road detection is also wrong, and goes along the Path Driven instead of trying to find the Center Lane Path.

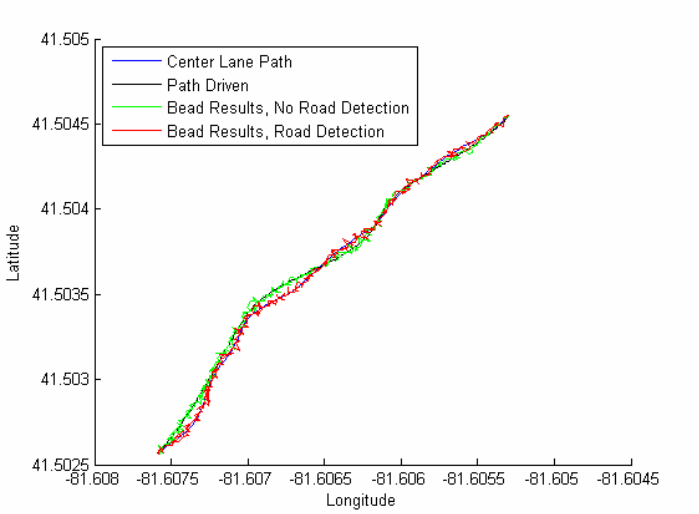


Figure 7-12. This figure shows beads after ten iterations, with $\sigma_{b,GPS}$ *= 6 m,* $\sigma_{ps,GPS}$ *= 5 m, and* $\sigma_{rd,offset}$ *= 0.5 m. Road detection added minimally to the physical state's variances, but its inclusion allowed beads to follow the Center Lane Path as desired; ignoring road detection led to beads following the Path Driven instead.*

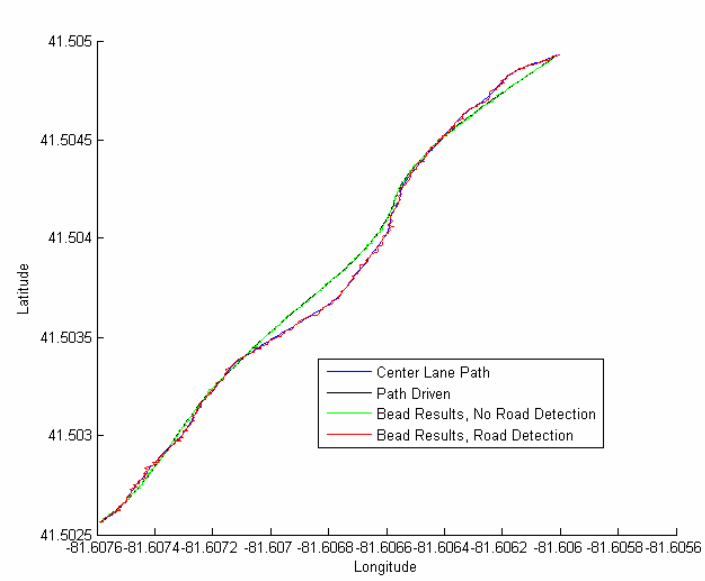


Figure 7-13. This figure shows beads after ten iterations, with $\sigma_{b,GPS}$ *= 6 m,* $\sigma_{ps,GPS}$ *= 1 m, and* $\sigma_{rd,offset}$ *= 1 m. With the combination of the physical state and the road detection (the virtual road sensor) at a relatively high confidence, the identified paths were fairly clean. Using road detection found the Center Lane Path nicely, while ignoring road detection only reinforced the Path Driven.*

In previous experiments in discovering map registration, the assumption was that the robot would need to be forced to drive down the center of its lane. Now assume

instead that this restriction is lifted, and the robot is driving autonomously and likely imperfectly. Without road detection, the robot attempts to drive based on the biased beads and cannot realize that it may be literally running off the road. Furthermore, without road detection, the physical state data remains based on the Path Driven and cannot provide an estimate by itself of where the Center Lane Path should be. Therefore, an autonomous robot without road detection cannot detect its global bias in map registration.

Applying road detection, however, allows the previous algorithms in discovering map registration to be used with a robot driving autonomously. As before, an alternative Center Lane Path estimate is compared with a matching bead in order to determine the two shifts (north-south and east-west) between those two points. But now, the road detection data combines with the physical state data and creates a virtual road sensor in order to compare those estimates of points on the Center Lane Path with the bead positions. Figure 7-14 illustrates how road detection can be used in discovering map registration.

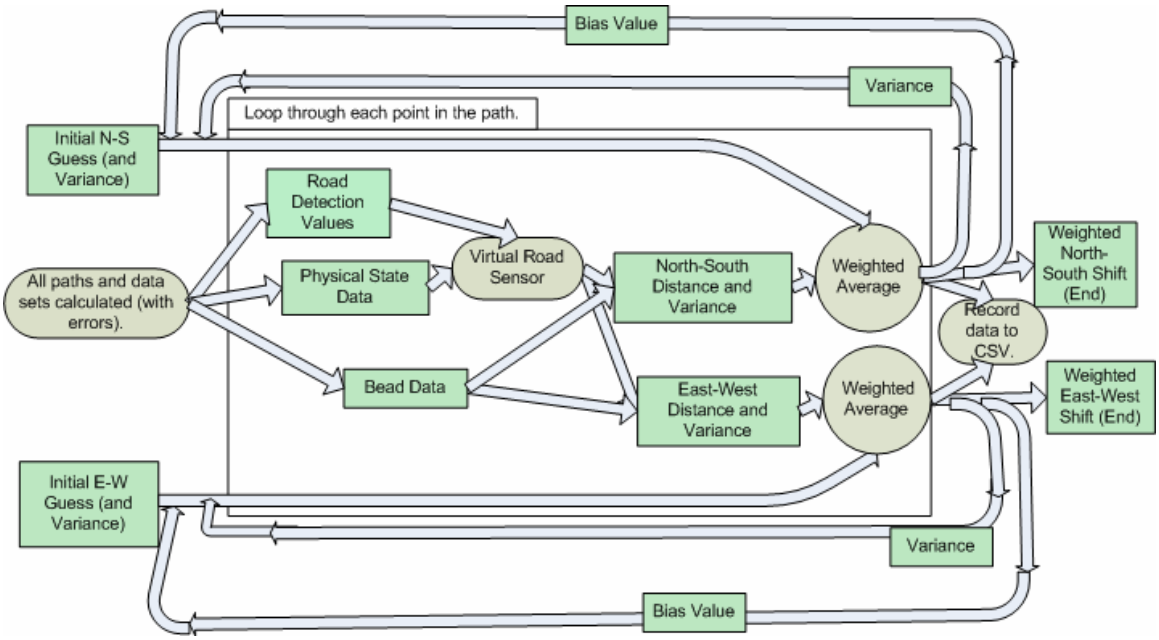
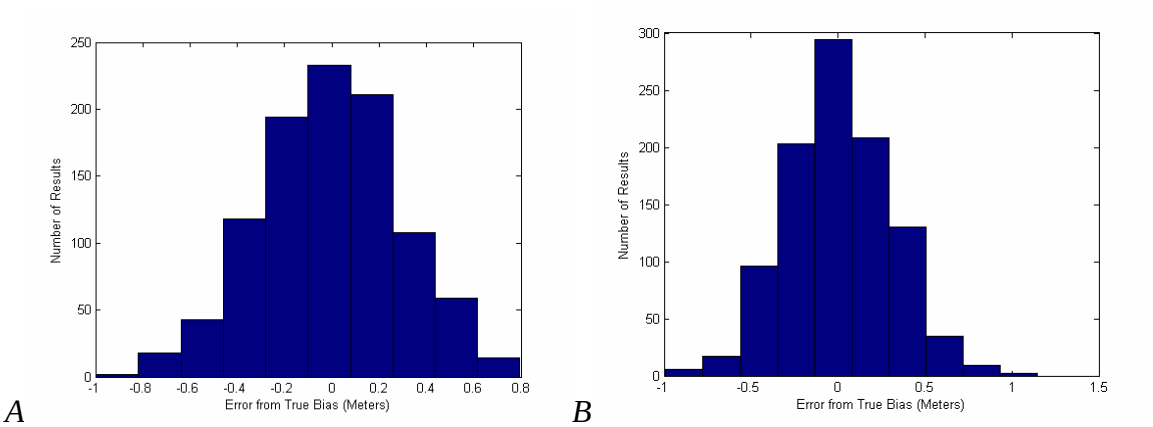


Figure 7-14. Offsets_with_Corrected_True_Data_Running_Bead_Bias.vi discovers bead biases by taking into account the road detection data. For more mathematical details on how to calculate shifts, see the previous chapter on map registration. The generation of the Center Lane Path and the Path Driven is the same as in Figure 7-9.

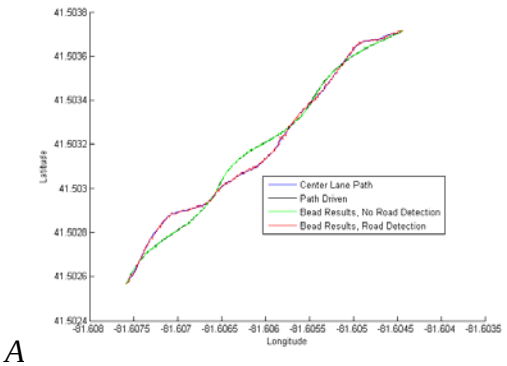
As Figure 7-14 suggests, these experiments use the running bead bias method. The errors in each sensor are recalculated for each run, but the underlying paths (the Center Lane Path and the Path Driven) are not. Figure 7-15 shows that, provided the correcting force of road detection, the global bead bias can be estimated as before, even though the robot is not driving on the center of its lane. Furthermore, the statistical variances in the experimental results follow the algorithm's anticipated confidences in its bias estimates.



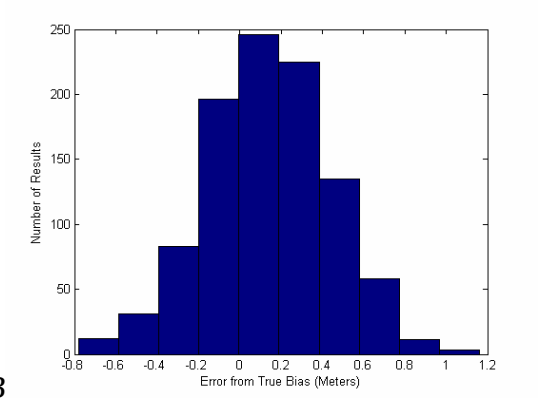
A *B*

Figure 7-15. This data was collected over 1000 runs with $\sigma_{b,GPS}$ *= 4 m,* $\sigma_{ps,GPS}$ *= 3 m, and* $\sigma_{rd,offset}$ *= 2 m. Graph A shows the discovery of an east-west bias 4.3 m west, with each run expecting a variance of 0.089 m*^{2} *in its bias estimate and the graph overall displaying a variance of 0.087 m*^{2}*. Graph B shows the discovery of a north-south bias 2.6 m north, with each run expecting a variance of 0.090 m*^{2} *in its bias estimate and the graph overall displaying a variance of 0.090 m*^{2}*. With road detection, a human need not drive down the Center Lane Path in order for a global bead bias to be discovered.*

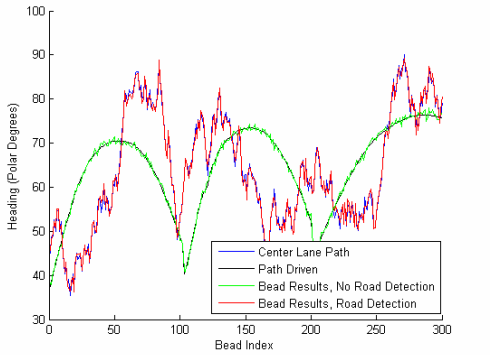
Finally, it should be noted that the directions of perceived lateral offsets were computed based on high-quality estimates of bead heading, and this might not always be practical. Figure 7-16, however, shows that a higher bead heading variance did not have a noticeable effect on the results; their character, when compared with previous, similar figures, remained the same. One possible explanation is that since offsets are generally only a few meters, the effect of moving at an incorrect angle may be minimized. Nevertheless, a practical application may unavoidably degrade the knowledge of left vs. right relative to the Center Lane Path, and additional sensor information may not be able to determine this center lane heading while driving. The challenge in this scenario would be to determine exactly how the variance in the bead heading (in degrees) adds to the variances of position in the virtual road sensor measurements.



A



B



C

Figure 7-16. Graph A shows ten iterations of bead improvement with $\sigma_{b,GPS}$ *= 6 m,* $\sigma_{ps,GPS}$ *= 1 m, and* $\sigma_{rd,offset}$ *= 1 m. This time,* $\sigma_{b,heading}$ *= 20 degrees (with* $\sigma_{ps,heading}$ *= 2 degrees and* $\sigma_{rd,heading\,correction}$ *= 2 degrees). Graph B shows east-west bias estimates collected over 1000 runs using a bias 4.3 m west and* $\sigma_{b,GPS}$ *= 4 m,* $\sigma_{ps,GPS}$ *= 3 m,* $\sigma_{rd,offset}$ *= 2m, and* $\sigma_{b,heading}$ *= 20 degrees. The data on Graph B has a variance of 0.090* m^2*, while each run expected a variance of 0.093* m^2 *in its bias estimate. Overall, the poor bead headings did not have an obvious effect. However, poor starting bead data allows Graph C to illustrate that headings, like GPS data, need road detection to approximate the data in the Center Lane Path instead of in the Path Driven. Graph C is from the same test as Graph A.*

Chapter 8: Discovering Road Detection Offset Variances

Section I: Calibrating Road Detection Offset Variances

In the previous chapter, road detector information (relative heading and lateral offset error measurements) was merged with physical state estimates based on presumed Gaussian noise and knowledge of variances. Bead variances were also used in the overall process; these variances can be approximated by a close examination of maps. For the physical state estimates, a combination of sensor sources may yield latitudes, longitudes, headings, and the variances of each (DEXTER's Physical State Observer, for example, performed such calculations). To incorporate road detector data, it is also necessary to estimate the variance of the road detector uncertainty. This is particularly difficult for vision-based road detection, as errors in road detection vary wildly and these errors are not easily describable in terms of zero-mean Gaussian noise. This chapter offers some suggestions for deriving the variances of road detection offsets as a function of more-easily evaluated indicators. Similar methods could potentially apply to discovering the variances of road detection heading corrections.

A road detection system must have sensory inputs, such as camera images or LIDAR scans. Raw data from these sensors is analyzed in terms of some type of pattern matching. In this process, there are multiple cues indicating quality of the road identification. These may include consideration of how well the current fit conforms to expectation, how strongly lines on the road are identified, the RMS error of line fits,

parallelism and distance between identified curbs, and others. While none of these cues fulfills the role of a variance estimate, each of them can contribute to an estimate of the lateral offset and heading error variances.

In the present analysis, it is assumed that there are multiple indirect cues that have bearing on the resulting measurement variance. Each of these numerical indicators will be referred to as a "feature." A feature can represent any quantity that has some influence on the road detection system. In these simulations, a feature with a higher value (all other features being held constant) translates into a higher variance in the road detection system. Further, the feature values have been scaled to fall in the range from zero to one. Any features may be scaled and inverted to follow this scheme without loss of generality. For example, if a feature measured the visibility of painted road lines in an image, fewer pixels assigned to road lines in one image versus another would assign a higher value (a value closer to one) to the road line feature in the former case.

It is assumed that there is some multivariable and possibly quite nonlinear mapping from feature values onto the lateral offset variance. It should be noted that variance, in this case, is not truly random noise. For a given snapshot of road detection data, the computed lateral offset is repeatable (i.e. deterministic). However, for larger feature values, the computed result is less trustworthy. It is desired to assign an equivalent variance that would be suitable for optimally merging road detection data with physical state and bead location data.

In this chapter, a method is proposed for learning an appropriate mapping from available cues (features) to road detector variances. The method is evaluated in simulation. In these simulations, the road detector is assumed to have three easily-

computed features that should have bearing on the credibility of the road detector lateral offset estimates. (The method applies to any number of features).

A set of features captured at one point in time maps to a standard deviation (and therefore a variance) in the road detection system. Since a smaller feature indicates better data, the road detection standard deviation should increase and decrease in value monotonically with each individual feature. However, some features may be more important to the road detection standard deviation mapping than others. For example, a feature that detects the visibility of road lines may affect the road detection standard deviation more than a feature that tries to determine the color of those lines. Hence, the influences of the features on the road detection standard deviation are not equal.

In the simulated evaluations presented here, the mapping from features onto a standard deviation is defined according to Equation 8-1. This mapping satisfies the presumption that the variance increases monotonically with each feature. More complex, nonlinear mappings could be assumed, but this is adequate for the present analysis. It should be noted that it is only presumed that there is *some* underlying mapping from features onto variance. The exact form of that mapping is not known *a priori* to the learning algorithm, and the algorithm does not assume any model for this mapping.

$$\sigma_{rd} = \sum_{i}^{N} (f_i \bullet c_i)$$

Equation 8-1. For one point of road detection data, each feature f is multiplied by its corresponding constant c up to N features; in these simulations, N = 3. The sum of these products gives the standard deviation of a road detection offset value. Optionally, a constant could be added to the sum above.

For simulation analyses, the influence coefficients in Equation 8-1 are generated at random, but held constant throughout a given simulation. These influence coefficients remain unchanged throughout the calibration and testing phases of any one simulation,

but are regenerated for each subsequent simulation. The random influence coefficients can take values from zero to some user-defined maximum. The larger these coefficients, the higher the road detection standard deviations can be in a given simulation.

In order to perform calibration, road detection data must be gathered under special circumstances. The road detection calibration readings should capture a range of possible sets of features, in order to better characterize the behavior of the system. For similar reasons, calibration data should include many samples. Each simulation generates a random feature set 1000 times in order to perform calibration. By assumption, calibration readings occur while the robot drives on the center of a lane (zero meters offset), so that any offset reported by the road detection system constitutes an offset error. As discussed previously, robots can indeed drive down the center of the lane when forced; calibration need not occur in autonomous driving mode.

Discovering_Road_Detection_Variance.vi generates calibration data in simulation, creating 1000 feature set readings and mapping each feature set to a road detection standard deviation. Each road detection standard deviation then translates into a Gaussian offset error. By this process, errors are added to the virtual road detector lateral offset data. In simulation, only the indirect measures of road detector uncertainty (the three feature values) are exposed to the system; the corresponding variance responsible for generating an offset error is not known. It is desired to discover the mapping from features to variances.

The 1000 offset errors from calibration have their own statistics, including an error standard deviation that describes how these offset errors vary. This error standard deviation, when squared, produces a general estimate of the variance in the road detection

system. This simple approximation of the road detection variance will be referred to as the naïve variance, as produced by the naïve method. In the naïve case, the road detector is assumed to have a constant variance. Since the mapping from features onto variance is unknown, the feature values are ignored. The naïve case thus loses the opportunity to recognize when the road detector data is particularly trustworthy, and at the other extreme, it may put more faith in the road detector than is warranted. An improvement over the naïve case is a learned mapping from features onto variances. The method used here to learn this mapping is Lloyd's k-means clustering algorithm, as described in [52]. First, some user-specified number of clusters is created. The centroid of each new cluster consists of a random feature set, where each feature is between the maximum and minimum values of that same feature that were observed during calibration.

Once clusters exist, the code examines each feature set in the calibration data. Each feature set will be closest, in terms of Euclidean distance, to one of the cluster centroids. If the closest cluster does not already contain this feature set as one of its members, the data point is added, and the feature set moves the centroid (the characteristic feature set) of the cluster. If a specific sample was formerly assigned to a different cluster than the closest cluster, this data point is removed from the former cluster, altering that cluster's centroid. Thus, this process alters the centroids of both affected clusters. At any one time, a cluster's centroid consists of a set of features, where each feature is averaged from the cluster's members' values of that feature.

Available data is re-clustered iteratively until all clusters are stable (or, theoretically, until some maximum number of iterations, though that restriction was not needed for these simulations). After clustering, the statistics of each cluster may be

calculated. Cluster statistics depend on member data, so the code removes empty clusters from the overall cluster list. The final clusters each hold a population of samples that have similar feature sets. Each cluster is analyzed to infer the mapping between features and variance. The representative feature set of a given cluster is its centroid (the average of all feature sets from the population of samples within a cluster). It is desired to associate this average feature set with an appropriate measurement variance. To do so, member properties within the respective cluster are analyzed. Each member of the cluster is a calibration sample with a known lateral offset error. A representative cluster variance is computed as the variance in lateral offset errors using all members of that cluster. In this way, each cluster is associated with a corresponding representative variance.

Once the clusters are trained, the simulation can consider the computable feature values for a road detector sample with unknown error. These features are used to find the closest trained cluster. The variance associated with the closest trained cluster is then assumed to apply to the current measurement. In order to evaluate this method, the simulated robot drives a test path (using the same hidden mapping from features onto variances). The simulator creates new feature sets for some user-specified number of test path points. The feature sets on this test path follow one of three patterns. In the first case, code creates each feature set on the test path randomly, as was the case during calibration. For the second situation, each feature set is a copy of the feature set that produced the maximum road detection standard deviation observed during calibration (emulating poor road detector performance). The final, third case resembles the second, except the feature set copied produced the minimum road detection standard deviation

observed during calibration (emulating good road detector performance). The last two cases represent extremes where, after calibration, the robot drives on roads that are hard to sense (the second case) or easy to sense (the third case).

Each feature set on the test path maps onto a road detection standard deviation per a known (but hidden) mapping—the same hidden mapping that was used for calibration. For validation purposes, the actual standard deviations used to generate the lateral offset errors are recorded, although this information is not exposed to the methods under test. These are called the "true" standard deviations. Each feature set on the test path also lies closest to the centroid of one of the trained clusters. The algorithm assumes that each road detector sample may have a variance (equivalently, standard deviation) equal to the variance associated with the closest cluster. Recall that the naïve road detection standard deviations were already calculated, and also that method assumes that the road detector variance is constant throughout a path. In contrast, the *K*-means clustering algorithm has *K* clusters, each with an associated variance. Thus, rather than assuming a single variance, the clustering method assigns one of *K* variances to each sample point. After Produce_Feature_Standard_Deviations_for_Path.vi completes the simulation of test path uncertainties, there are three lists of road detection standard deviations that could be believed for the road detection offsets in the test path: the true standard deviations from the (hidden) feature mappings, the standard deviations assigned by clustering, and the naïve standard deviations (constant for all samples).

Comparison_of_Lateral_Measurements.vi compares the results of believing in turn each of the three possible lists of road detection standard deviations. First, the VI generates at random the error-free offsets of the test path, which are the lateral offsets that

sensors would report if they were perfect. For simulation purposes, the maximum pre-error lateral offset was three meters; in the U.S., the most common lane width is twelve feet [53], which is a little more than three and a half meters. As described above, random feature values were generated, and these produced corresponding road detection variances per the hidden mapping. These variances were recorded for comparison, and they were used to generate Gaussian-distributed offset errors for (simulated) lateral displacement measurements. The resulting offsets, errors included, represent the offsets reported by the road detection system.

In order to compare the effects of using each of the three different lists of uncertainties, the simulations require some other lateral offset measurement that can be combined in turn with each separate list of uncertainties. This independent lateral offset measurement could be calculated from comparing beads with physical state data, for example. For simplicity and to preserve generality, the simulations do not care about the source of these measured offsets. The user sets the standard deviations for the measured offsets throughout the test path; using these standard deviations and the true lateral offsets, the reported measured offsets contain Gaussian errors, and the variance of each measured offset is known. For simplicity, the same standard deviation is used for every measured offset in the test path.

To invoke the optimal estimation equations, it is necessary for the information sources to satisfy the assumptions of zero-mean Gaussian noise, and it is also necessary to know the variance of the Gaussian noise. In the current evaluations, the virtual road detector measurements include Gaussian noise, but the variance of that noise is unknown to the system. Nonetheless, the variances used to generate the road detector

measurements can be analyzed as if this information were available, and use of this information yields the theoretically optimal use of the road detector data. This computation, using the "true" variances, is used as a benchmark constituting an upper bound on the performance of a variance-learning algorithm. On the other hand, assuming a constant variance for the road detector (the naïve case) constitutes a lower bound for a learning algorithm. The learning algorithm should, at a minimum, perform no worse than the naïve case, and ideally it would rival the ideal case.

For every point in the test path, the measured offset and its corresponding known variance combine with the road detection offset and each of the three beliefs about the current road detection variance. The three combinations occur independently, so that none of the three variance beliefs influences the behavior of either of the others. The true standard deviations, the clustering standard deviations, and the naïve standard deviations all become variances through squaring. Each of the three road detection variances uses the same value for the lateral road detection offset, an erroneous value following the true road detection standard deviation. Therefore, the differences in road detection variances are the only differences when the optimal estimation equations combine two lateral offset measurements three separate times. Figure 8-1 summarizes how calibration and test data are generated and used.

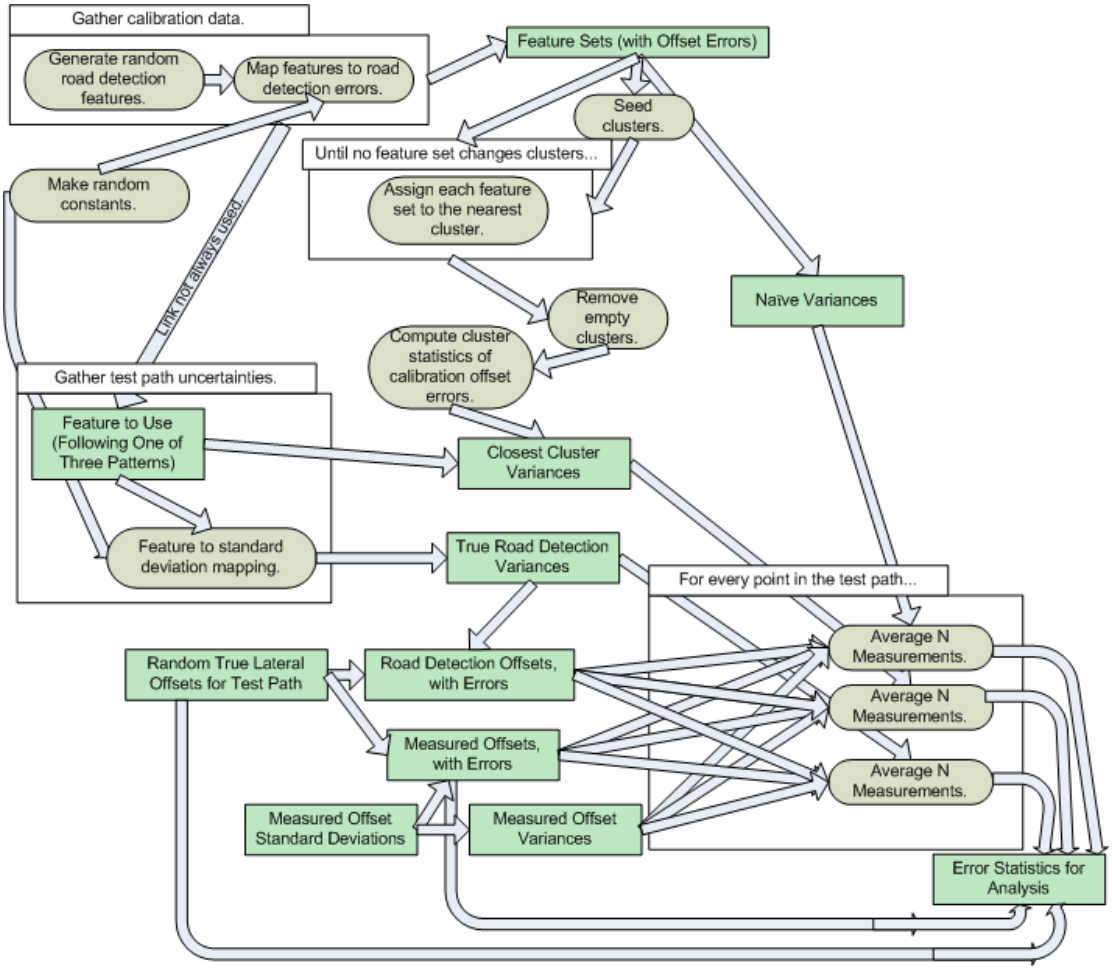


Figure 8-1. Calibrating road detection data makes both the naïve and clustering methods possible. Arbitrary measured offsets allow the naïve and clustering methods to be compared with the effects of using the actual variances present in the road detection system. Recall that "Average N Measurements" refers to a VI that performs optimal estimation.

Traversing the test path produces three outputs, each a list of lateral offset estimates affected by the variances used to produce them. The code subtracts the true lateral offsets from each list of lateral offset estimates. The lateral offset errors following true road detection standard deviations, clustering standard deviations, and naïve standard deviations each have a root mean square (RMS) value. The RMS values of each method can then be compared for analysis. The measured offsets similarly have an RMS value

characterizing how they deviate from the true lateral offsets. Since the true lateral offsets have no errors, any set of true lateral offsets always has an RMS of zero meters.

Section II: Analysis of Clustering and Naïve Calibration Methods

As stated, these simulations assume three features as inputs into the road detection system, and pre-error lateral offsets up to three meters on the imaginary road. Since the calibration process takes 1000 readings, the simulations seeded one hundred clusters in each run. Using too few clusters would have little advantage over the naïve method, while using too many clusters may produce clusters with few members and therefore not very robust member statistics. Further work could experiment with determining the optimal number of clusters to use. Indeed, the clustering performed by the simulations does not always produce variances closer to the true road detection variances than the naïve variances are. However, clustering variances are nonetheless usually better than in the naïve case, and those runs presented in Table 8-1 analyze the usefulness of good clustering.

Table 8-1 provides data from several runs of the simulator. The specific values of the lateral offsets do not matter, since the analysis is concerned instead with the lateral offset errors. In order to make graphical analysis clearer, the test paths used for Table 8-1 were always fifteen points long. The variances in the measured offsets were consistent throughout each test path; if these values varied, the test path would consist of a set of sub-problems, each one characterized by the measured offset variance on the relevant stretch of road. Recall that the three types of road detection offset variances (true, naïve, and cluster) are all attempting to improve (i.e., decrease) the plain RMS value for the measured offsets.

Row #	$\sigma^2_{measured\ offset}$ (m^2)	σ^2_{truth} (m^2)	$\sigma^2_{naïve}$ (m^2)	$\sigma^2_{cluster}$ (m^2)	Measured Offset RMS (m)	True Var. Method RMS (m)	Naïve Method RMS (m)	Cluster Method RMS (m)
1	400	0.30	31.53	0.76	17.31	0.69	1.07	0.68
2	400	259.87	79.35	176.35	16.77	13.31	15.42	13.93
3	169	129.13	46.19	173.87	13.24	7.78	8.82	7.90
4	4	0.071	19.68	0.55	1.52	0.27	1.26	0.29
5	400	Varies	20.11	Varies	24.70	3.90	3.92	3.97
6	9	121.55	33.63	146.45	3.29	3.34	4.22	3.31
7	0.09	0.08	24.97	0.53	0.29	0.20	0.29	0.26
8	0.09	Varies	10.56	Varies	0.32	0.32	0.32	0.34
9	9	Varies	41.39	Varies	2.94	2.23	2.63	2.30

Table 8-1. A set of tests showing how various definitions of the road detection offset variances can hurt or improve the RMS value of errors in an independent measured offset. The three columns on the right all characterize errors in lateral offsets after including some sort of road detection data.

Rows 1-4 show cases where the true road detection variances are less than the variances of the measured offsets (Row 1 drastically so). Since the true road detection variances are consistent, the clustering variances are as well; if Centroid X is closest to Feature Set Y once, that condition will always be true for matching feature sets and the same cluster list. In all four rows, the naïve variances are farther from the true road detection variances than the clustering variances are. This improved accuracy makes the

RMS values using clustering closer than the naïve RMS values to the true road detection variance RMS values. Note that this RMS improvement still occurs in Row 3, where the true road detection variances are more comparable to the measured offset variances.

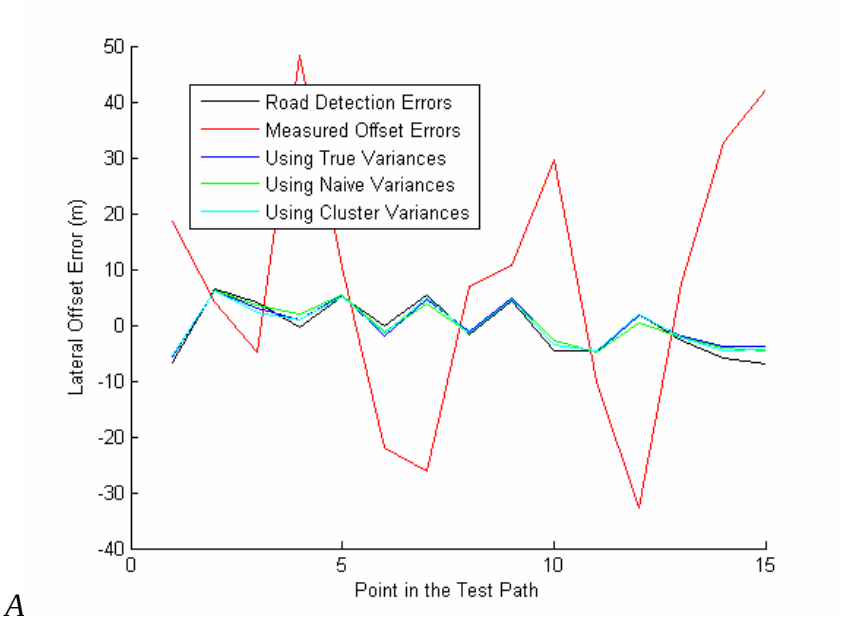
The naïve method in Rows 2 and 3 made the optimal estimation equations incorrectly believe the road detection data too strongly; in Rows 1 and 4, the naïve method's belief was not strong enough. The results of each method can also be observed with regards to each point on the test path. Figure 8-2 graphs information for the test path that produced the data in Row 4. Note that in some cases, believing the naïve lateral offset estimate could lead a robot to drive more than two meters off of the center of the road, while the clustering method would keep the robot driving closer to a half a meter from the center of the lane. Depending on the road, this difference in performance may mean the difference between safety and disaster.

Figure 8-2. Lateral offset error data for Row 4 of Table 8-1. Since this is a graph of errors, the perfect case would be a flat, horizontal line at 0 m. The three lines resulting from combining erroneous data under different beliefs are, as expected, between the road detection errors on one end and the measured offset errors on the other. Note that the

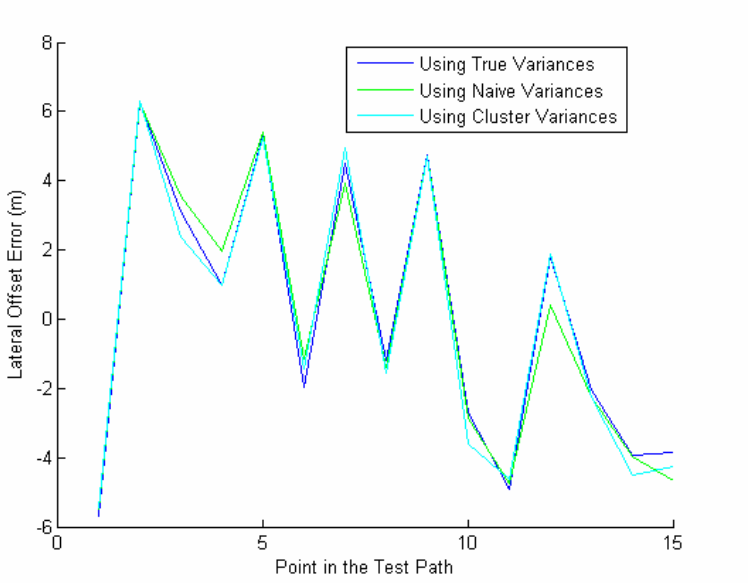
errors using true (road detection) variances are the goals (the best expected behavior), and that the errors using cluster variances are between the true road detection variance errors and the naïve variance errors.

In Row 6, the true road detection variance was consistently poorer than the measured offset variance along the test path. Once again the naïve method believed the road detection data too strongly, but in this case the naïve method actually led to a worse RMS value than the RMS value of the measured offsets. The clustering method, since its variances were closer to the true road detection variances, instead allowed the code to suitably ignore road detection's contribution. However, Row 7 shows that the measured offset confidence can be high enough that the naïve method's high uncertainty may not always have an effect. In Row 7, the true road detection variance was comparable to that of the measured offset variance, so the true and clustering road detection variances caused little change.

The road detection features may vary along a test path; in Rows 5, 8, and 9, those variations were random. When the measured offset variances were large (Row 5), the measured offset RMS value had room to improve, but the clustering method did not noticeably outperform the naïve method in Rows 5, 8, or 9. Using more points on the test path does not change this outcome. When the test path's true road detection variances are random, assuming the average variance (as the naïve method does) works about as well as the clustering method over the course of the test path. Figure 8-3 shows that along the test path, the clustering method may still be better than the naïve method at particular path points.



A



B

Figure 8-3. Lateral offset error data for Row 5 of Table 8-1. There are points on the test path where the clustering method does a better job than the naïve method of improving the lateral offset estimate for that path point; Graph B shows just a subset of Graph A in order to highlight that principle. Yet overall, the naïve variance method's RMS value was close to that of both the true road detection variance and the clustering variance methods.

The naïve method avoids the complications of the clustering method, including the further experimentation that the clustering method desires. Additionally, using the naïve method often improves the measurement offsets, which is a clear step above having

no idea what the road detection variances are. In some cases, the naïve method is just as good as the clustering method, though importantly, the clustering method is never measurably worse. Yet, when a test path has consistent quality in road detection offsets, and that quality is higher than that of the measured offsets, the clustering case makes better use of the calibration information than the naïve method. The same is true when the true road detection variances are much higher (relatively) than the variances in the measured offsets throughout the test path. Therefore, the clustering method is more flexible than the naïve method, a valuable quality since in reality the confidence of road detection data along any test path begins as an unknown.

Chapter 9: Summary and Future Work

The research presented here started with its roots in DEXTER, Team Case's entry into the 2007 DARPA Urban Challenge. Beads were designed specifically for DEXTER. Through repeated tests in preparation for the Urban Challenge, the bead structure itself, as well as bead generation and bead communication, proved accurate and adequate for the DEXTER project. DEXTER also verified the teach-and-playback method outlined in this paper, and ran successfully on beads preprocessed via the described preprocessing method.

A more general solution should perform online bead updates. A different but related problem could instead improve the physical state readings (using the map and potentially other sources), but in this paper the focus remained on updating beads. DEXTER's personal simulator verified one proposed online bead update solution, the all-or-nothing update method. The simulator had some limitations when compared with running on the actual robot, but clearly the errors in bead updates caused by coordinating data with the localization software would need to be addressed in a physical robot. Additionally, the all-or-nothing update method always throws away some information. Making use of all information, and thus performing partial updates, would lead to a more knowledgeable bead map.

Work presented here extended past the Urban Challenge competition through the use of simulations to explore the issue of fused data map updates. Ideally, the algorithms developed in simulation would be tested in physical field trials. One challenge when abandoning simulations will be to gather data from different sources that all correspond

to the same sample point. Simulations started by considering only beads and physical state data, and their respective variances. Since beads may be biased, work was done that (in simulation) could successfully identify both north-south and east-west biases. If needed, future work could investigate the issues involved if a map were shrunk, stretched, and/or rotated from its desired state.

Simulations using only beads and physical state data are ignorant of road detection, and must unrealistically assume that a robot is always driving on the center of its lane. Road detection systems report relative data, contrary to the beads and physical state data that retain absolute latitude/longitude positions. In response, this paper introduced a method of using road detection information in fused data bead map updates. To translate these simulations into practice, additional work would be required for the road detection system to output computable features that are expected to influence road detection credibility.

The online fused data bead updates all relied on knowing the variances in beads, physical state data, and road detection data. Though road detection variances tend to be the most difficult to discover, more work could also address finding variance values for beads and physical state in the best ways available. Because road detection variances are especially tricky, this paper concluded by making suggestions on how to derive such variances from the inputs available to a road detection system. A means was proposed for inferring variances for lateral offset measurements; learning variances in the heading corrections of a road detection system would require additional work.

Clustering was proposed as a method to infer road detection offset variances from available cues. This method was compared to an alternative that assumes a fixed

variance associated with road detector measurements (the naïve method). The clustering method itself, while appearing to provide some benefit over the naïve method, can be investigated further. There may be some optimal number of clusters for certain conditions. Recordings from physical experiments may illuminate what kind of relationship to reasonably expect between calibration and test data. Real-life results may also further restrict the randomness of the calibration data that can be practically anticipated. Finally, while clustering certainly suggests a solution, there may be other unsupervised learning techniques suited to the task.

References

[1] C. Urmson, "Self-Driving Cars and the Urban Challenge," *Intelligent Systems, IEEE,* vol. 23, pp. 66-68, 2008.

[2] I. Catling, "Road transport informatics in Europe-major programs and demonstrations," *Vehicular Technology, IEEE Transactions on,* vol. 40, pp. 132-140, 1991.

[3] W. C. Collier, "Smart cars, smart highways," *Spectrum, IEEE,* vol. 31, pp. 27-33, 1994.

[4] T. Jochem, "PANS: a portable navigation platform," *Intelligent Vehicles '95 Symposium. , Proceedings of the,* pp. 107-112, 1995.

[5] D. Bernstein and A. Kornhauser, "An introduction to map matching for personal navigation assistants," New Jersey TIDE Center, Newark, NJ, August 1996, 1996.

[6] G. Chen, "A Real-Time Navigation Architecture for Automated Vehicles in Urban Environments," *Intelligent Vehicles Symposium, 2007 IEEE,* pp. 1223-1228, 2007.

[7] G. Seetharaman, "Unmanned Vehicles Come of Age: The DARPA Grand Challenge," *Computer,* vol. 39, pp. 26-29, 2006.

[8] Anonymous (2008, 4/21/08). DARPA urban challenge. *2008(6/28),* pp. 15.

[9] S. Thrun, "Why we compete in DARPA's Urban Challenge autonomous robot race," *Commun ACM,* vol. 50, pp. 29-31, 2007.

[10] Anonymous (2005, ENSCO's DEXTER placed sixth out of the 23 robotic vehicles competing in the final run of DARPA grand challenge. *2008(4/16),* pp. 1.

[11] Anonymous (2007, Team case. *2008(4/16),* pp. 10.

[12] Anonymous (2008, NI LabVIEW. *2008(4/16),* pp. 1.

[13] M. Y. Ibrahim and A. Fernandes, "Study on mobile robot navigation techniques," *Industrial Technology, 2004. IEEE ICIT '04. 2004 IEEE International Conference on,* vol. 1, pp. 230-236 Vol. 1, 2004.

[14] B. Yamauchi, A. Schultz, W. Adams and K. Graves, "Integrating map learning, localization and planning in a mobile robot," *Intelligent Control (ISIC), 1998. Held Jointly with IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA), Intelligent Systems and Semiotics (ISAS), Proceedings,* pp. 331-336, 1998.

[15] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte and M. Csorba, "A solution to the simultaneous localization and map building (SLAM) problem," - *Robotics and Automation, IEEE Transactions on,* vol. 17, pp. - 229, 2001.

[16] S. Kotani, K. Kaneko, T. Shinoda and H. Mori, "Mobile robot navigation based on vision and DGPS information," *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on,* vol. 3, pp. 2524-2529 vol.3, 1998.

[17] O. Murayama, "Navigation algorithm based on teaching-playback system for unmanned dumptrucks in mines," in *Intelligent Vehicles '94 Symposium, Proceedings of the,* 1994, pp. 25-31.

[18] G. Xiong, "Development and Implementation of Remote Control System for An Unmanned Heavy Tracked Vehicle," *Intelligent Vehicles Symposium, 2007 IEEE,* pp. 663-667, 2007.

[19] H. Cool. (2008, 2/28/08). DEXTER, case western reserve university's robotic car, to be inducted into crawford auto-aviation museum. *2008(6/28),* pp. 1.

[20] U. Ozguner, C. Stiller and K. Redmill, "Systems for Safety and Autonomous Behavior in Cars: The DARPA Grand Challenge Experience," *Proceedings of the IEEE,* vol. 95, pp. 397-412, 2007.

[21] F. Figueroa and Xiaojing Yuan, "Sensor fusion for a network of processes/systems with highly autonomous sensors," *Virtual and Intelligent Measurement Systems, 2001, IEEE International Workshop on. VIMS 2001,* pp. 4-10, 2001.

[22] M. Chan, E. H. Ruspini, J. Lowrance, J. Yang, J. Murdock and E. Yeh, "Human-aided multi-sensor fusion," *Information Fusion, 2005 8th International Conference on,* vol. 1, pp. 3 pp., 2005.

[23] P. U. Lima, "A Bayesian Approach to Sensor Fusion in Autonomous Sensor and Robot Networks," *Instrumentation & Measurement Magazine, IEEE,* vol. 10, pp. 22-27, 2007.

[24] C. Owen and U. Nehmzow, "Route learning in mobile robots through self-organisation," in - *Advanced Mobile Robot, 1996., Proceedings of the First Euromicro Workshop on,* 1996, pp. - 126.

[25] Takehisa Yairi, "Map building without localization by dimensionality reduction techniques," in *Proceedings of the 24th International Conference on Machine Learning,* 2007, pp. 1071.

[26] S. P. Engelson and D. V. McDermott, "Error correction in mobile robot map learning," in - *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on,* 1992, pp. - 2555.

[27] R. Lakaemper, L. J. Latecki and D. Wolter, "Incremental multi-robot mapping," in - *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on,* 2005, pp. - 3846.

[28] R. Araujo and A. T. de Almeida, "Mobile robot path-learning to separate goals on an unknown world," in - *Intelligent Engineering Systems, 1997. INES '97. Proceedings., 1997 IEEE International Conference on,* 1997, pp. - 265.

[29] A. Howard, G. S. Sukhatme and M. J. Mataric, "Multirobot Simultaneous Localization and Mapping Using Manifold Representations," - *Proceedings of the IEEE,* vol. 94, pp. - 1360, 2006.

[30] C. K. Toth, E. Paska, Q. Chen, Y. Zhu, K. Redmill and U. Ozguner, "Mapping support for the OSU DARPA grand challenge vehicle," in - *Intelligent Transportation Systems Conference, 2006. ITSC '06. IEEE,* 2006, pp. - 1580.

[31] Anonymous (2007, 3/14/2007). Urban challenge: Route network definition file (RNDF) and mission data file (MDF) formats. *2008(4/16),* pp. 14.

[32] C. Miller. (2007, A simulation and regression testing framework for autonomous vehicles. Available: http://vorlon.case.edu/~mcc14/research/papers/ChristianMillerMS2007.pdf

[33] United States, *United States National Map Accuracy Standards.* ,Rev. June 17, 1947 ed.Washington, D.C.: U.S. Bureau of the Budget, 1947, pp. [1].

[34] Ball Aerospace and Technologies Corp. (2008, WORLDVIEW-1. *2008(5/20),* pp. 1.

[35] U.S. Geological Survey. (2006, 8/22/2006). High resolution orthoimagery. *2008(5/22),* pp. 1.

[36] S. Brakatsoulas, D. Pfoser, R. Salas and C. Wenk, "On map-matching vehicle tracking data," in *VLDB '05: Proceedings of the 31st International Conference on very Large Data Bases,* 2005, pp. 853-864.

[37] Analytic Sciences Corporation. Technical Staff: Gelb, Arthur, *Applied Optimal Estimation.* Cambridge, Mass.: M.I.T. Press, 1974, pp. 374.

[38] S. T. McMichael. (2008, Lane detection for DEXTER, an autonomous robot, in the urban challenge.

[39] N. Matsumoto, A. Toyoda and S. Ito, "Mobile robot guidance control with nonlinear observer based state estimation," *Intelligent Robots and Systems '93, IROS '93. Proceedings of the 1993 IEEE/RSJ International Conference on,* vol. 3, pp. 2264-2271 vol.3, 1993.

[40] H. Rodriguez-Cortes and E. Aranda-Bricaire, "Observer Based Trajectory Tracking for a Wheeled Mobile Robot," *American Control Conference, 2007. ACC '07,* pp. 991-996, 2007.

[41] Le-Jie Zhang, Zeng-Guang Hou and Min Tan, "Kalman filter and vision localization based potential field method for autonomous mobile robots," *Mechatronics and Automation, 2005 IEEE International Conference,* vol. 3, pp. 1157-1162 Vol. 3, 2005.

[42] Y. Zhao, "A Map Matching Algorithm in GPS-based Car Navigation System," *Intelligent Information Hiding and Multimedia Signal Processing, 2007. IIHMSP 2007. Third International Conference on,* vol. 1, pp. 77-80, 2007.

[43] M. F. Goodchild, "Geographic Information Science and Systems for Environmental Management," *Annual Review of Environment and Resources,* vol. 28, pp. 493-519, 11/30. 2003.

[44] I. Lachow, "The GPS Dilemma: Balancing Military Risks and Economic Benefits," *Int. Secur.,* vol. 20, pp. 126, 1995.

[45] Garmin Ltd. (2008, What is GPS? *2008(5/20),* pp. 1.

[46] R. A. Fisher, "The Correlation Between Relatives on the Supposition of Mendelian Inheritance," *Philosophical Transactions of the Royal Society of Edinburgh,* vol. 52, pp. 399, 1918.

[47] G. N. Desouza, "Vision for mobile robot navigation: a survey," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* vol. 24, pp. 237-267, 2002.

[48] Peng Lu, Kui Yuan and Wei Zou, "A High Performance Low Power Consumption Robot Vision System," *Natural Computation, 2007. ICNC 2007. Third International Conference on,* vol. 5, pp. 171-175, 2007.

[49] E. D. Dickmanns, "The development of machine vision for road vehicles in the last decade," *Intelligent Vehicle Symposium, 2002. IEEE,* vol. 1, pp. 268-281 vol.1, 2002.

[50] Sunghoon Kim, Jeong-Ho Park, Seong Ik Cho, Soonyoung Park, Kisung Lee and Kyoungho Choi, "Robust Lane Detection for Video-Based Navigation Systems," *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on,* vol. 2, pp. 535-538, 2007.

[51] G. Yin and Q. M. J. Wu, "The multi-sensor fusion: image registration using artificial immune algorithm," *Soft Computing Techniques in Instrumentation, Measurement and Related Applications, 2003. SCIMA 2003. IEEE International Workshop on,* pp. 32-36, 2003.

[52] G. A. Wilkin and X. Huang, "K-Means Clustering Algorithms: Implementation and Comparison," *Computer and Computational Sciences, 2007. IMSCCS 2007. Second International Multi-Symposiums on,* pp. 133-136, 2007.

[53] Anonymous (2007, 4/5/2007). Highway statistics 2005. *2008(4/6),* pp. 1.