CajunBot: Architecture and Algorithms

Arun Lakhotia*
Center for Advanced Computer Studies
University of Louisiana at Lafayette
arun@louisiana.edu

Suresh Golconda
Center for Advanced Computer Studies
University of Louisiana at Lafayette
sxg3148@cacs.louisiana.edu

Anthony Maida
Center for Advanced Computer Studies
University of Louisiana at Lafeyette
maida@cacs.louisiana.edu

Pablo Mejia
Senior Systems Engineer
C&C Technologies, Inc
pjm@cctechnol.com

Amit Puntambeker
Center for Advanced Computer Studies
University of Louisiana at Lafeyette
axp5558@louisiana.edu

Guna Seetharaman †
Air Force Institute of Technology
Wright Patterson Air Force Base
guna@ieee.org

Scott Wilson
Center for Advanced Computer Studies
University of Louisiana at Lafeyette
saw@louisiana.edu

Abstract

CajunBot, an autonomous ground vehicle and a finalist in the 2005 DARPA Grand Challenge, is built on the chassis of MAX IV, a six-wheeled ATV. Transformation of the ATV to an AGV (Autonomous Ground Vehicle) required adding drive-by-wire control, LIDAR sensors, an INS, and a computing system. Significant innovations in the core computational algorithms include an obstacle detection algorithm that takes advantage of shocks and bumps to improve visibility; a path planning algorithm that takes into account the vehicle's maneuverability limits to generate paths that are navigable at high speed; efficient data structures and algorithms that require just a single Intel Pentium 4 HT 3.2 Ghz machine to handle all computations and a middleware layer that transparently distributes the computation to multiple machines, if desired. In addition, CajunBot also features support technologies such as a simulator, playback of logged data and live visualization on off-board computers to aid in development, testing, and debugging.

*Contact author: Arun@Louisiana.edu

†The contents of this paper do not necessarily reflect the official positions ofthe authors respective organizations

1 Introduction

CajunBot is a six-wheeled, skid-steered, Autonomous Ground Vehicle (AGV) developed to compete in the DARPA Grand Challenge. The vehicle was a finalist in both the 2004 and 2005 events. In the 2005 final, the vehicle traveled 17 miles, at which point it did not restart after a prolonged pause. The cause: the motor controlling its actuator burned out due to excessive current for a long duration.

This paper presents the insights and innovations resulting from the development of CajunBot. It is assumed that the reader is familiar with the DARPA Grand Challenge and technical challenges in developing AGVs. There exists an extensive body of research in various aspects of AGVs. This paper does not attempt to provide a survey of the literature; instead it only compares specific innovations to works that are most closely related.

The major innovations in CajunBot have been in its software system, both on-board and off-board software. The on-board software drives the vehicle autonomously and the off-board software facilitates development of on-board software.

- An obstacle detection system that does not require stabilizing of sensors, rather it takes advantage of bumps in the terrain to see further.
- A local path planning algorithm that fuses discrete and differential algorithms to generate vehicle navigable paths around obstacles. The discrete component of the algorithm generates costs in a grid world and the differential component uses these costs to select the best navigable curve from a pre-computed collection of curves.
- A layer of middleware for communication between processes with specialized support for fusing data from multiple sensors arriving at varying frequencies and latencies. The support enables fusion of sensor data based on the time of production of data, thereby ensuring fusion of mutually consistent data.
- A physics-based simulator that generates a simulated clock that may be used to synchronize processes on simulation time, thereby providing the capability to slow down, speed up, and single step the processing in the laboratory environment.
- Decomposing a visualizer as an independent process, rather than as traditionally maintained as part of the simulator, thereby enabling the same visualizer to be used for visualizing vehicle state during field-testing, during simulation, and during post-processing by replaying logged data.
- A software architecture that enables easy replacement of components, thus making it easy to maintain multiple, competing programs for the same task.

The rest of the paper is organized as follows. Section 2 describes the hardware of CajunBot, which includes the automotive, electrical, and the electronics. Section 3 describes the overall software architecture, and describes the middleware, simulator, and visualizer modules. Section 4 presents the core algorithms for obstacle detection, local path planning, and control. Section 5 presents some open problems that are being addressed by the team. Section 6 concludes the paper, and is followed by acknowledgments and references.

Figure 1: CajunBot

2 Hardware: Automotive, Electrical, Electronics

2.1 Automotive

The base of CajunBot is a MAX IV all-terrain vehicle (ATV) manufactured by Recreative Industries. This vehicle was chosen because (1) it can operate on a variety of terrains, including, road, rough surface, sand, water, and marshy conditions; (2) it has a very small turning radius, about 1.2m, making it extremely maneuverable; (3) it is not very wide, just about 1.5m; and (4) its mechanics for throttle and brakes is simple for interfacing with linear actuator and servo motor. The first property was important, given expected conditions on the GC route. The next two properties meant that we had more room to play in the software. The last property made it easy for us to build a drive-by-wire system.

In regards to the drive-by-wire system, it is instructive to know the mechanical points of contact for throttle, braking, and turning. The throttle on the vehicle is pulled by a cable, similar to that in a lawn mower or a motor cycle. Being a skid-steered vehicle, its braking and turning operations are interconnected. The vehicle turns by braking the wheels on one side, see Figure 2. The vehicle has two levers, with each lever controlling the transmission and brakes of the set of wheels on one side. Pulling a lever engages the brakes. Releasing the lever engages the gear. Lastly there is a region in between that represents neutral.

The choice of a MAX IV had its downsides. The top speed of the vehicle, about 45kph, meant it could not be a serious contender for a speed oriented track. The vehicle's five gallon tank was clearly insufficient for a 10+ hours run, and had to be retrofitted with a larger tank. The vehicle did not have any enclosure or roof, thus, a frame had to be built to house the electronics and for mounting sensors. Its power generation capacity was woefully inadequate for our needs requiring us to add a generator. Finally, absence of air conditioning

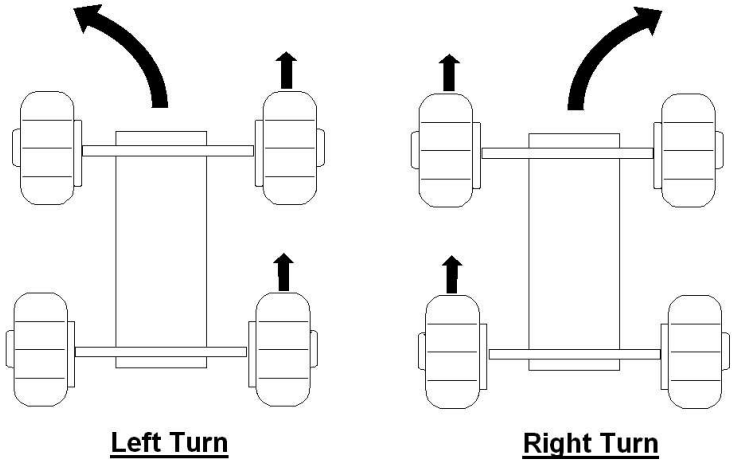

Figure 2: Skid Steered Vehicle

implied we had to improvise the cooling system for computers and electronics.

The most significant drawback of a MAX IV is that it lacks any active suspension. Its wheels are its suspension. Any bump or shock not absorbed by the wheel is transferred to the rest of the body. We used LORD's center-bonded shock mounts (CBA 20-300) to provide vibration and shock isolation to the new frame, and therefore to the sensors mounted on the frame. In addition we used a MIL-spec Hardigg Case with rack mount to further isolate the computers from shocks.

The shock mounts, however, did not change the fact that the vehicle moves like a brick on wheels. Any movements felt by its six axles, as the vehicle travels over bumps, are transferred to the frame. This led to interesting challenges when processing sensors data, as elaborated later.

2.2 Electrical (Power) System

CajunBot requires around 1670W of power for simultaneous peak performance of all the equipment on board. When operating in the field the necessary power is generated by a Honda EU2000i generator, fed to a 2200VA UPS unit, which then conditions the power, and provides four power supplies - 5VDC, 12VDC, 24VDC and 110VAC, as shown in Figure 3. In the lab environment, the power may be switched to a wall outlet.

2.3 Electronics

Figure 4 shows a schematic diagram of CajunBot's electronics, which may be viewed as being composed of three major systems:

- Sensor Systems

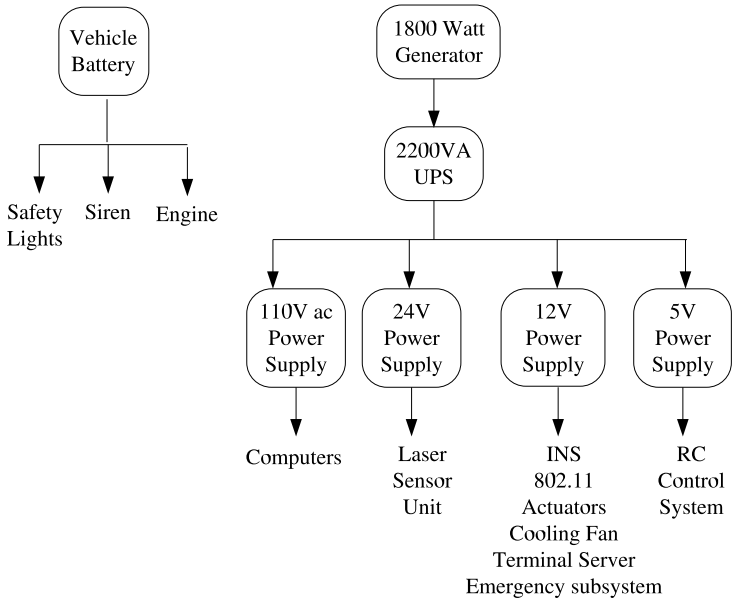

Figure 3: CajunBot E-Power Subsytem

- Drive-By-Wire System
- Computing System

2.3.1 Sensor Systems

The components shown in the left column of Figure 4, identified as INPUT, constitute CajunBot's sensor systems. The sensors may be further classified as those needed for autonomous operation and those for monitoring and emergency control.

CajunBot uses an INS (Oxford Technology Solutions RT3102) and two LIDAR scanners (SICK LMS 291) for autonomous operation. The accuracy of the INS is enhanced by Starfire differential GPS correction signals provided by a C&C Technologies C-Nav receiver. The LIDARs are mounted to look at 16m and 16.3m in the front of the vehicle, see Figure 5. CajunBot also performs well, albeit at a reduced speed, with only one LIDAR.

The sensors for monitoring and emergency control include the DARPA E-Stop, when performing in the Grand Challenge; an RC Receiver to communicate with an RC Controller, used during testing and for moving the vehicle around when not in autonomous mode; a wireless access point to broadcast data for real-time monitoring in a chase vehicle; and two kill switches on each side of the vehicle.

2.3.2 Drive-By-Wire System

The box annotated as 'Control Box' and the components listed on the right column, annotated as OUTPUT, in Figure 4, constitute the Drive-By-Wire system.

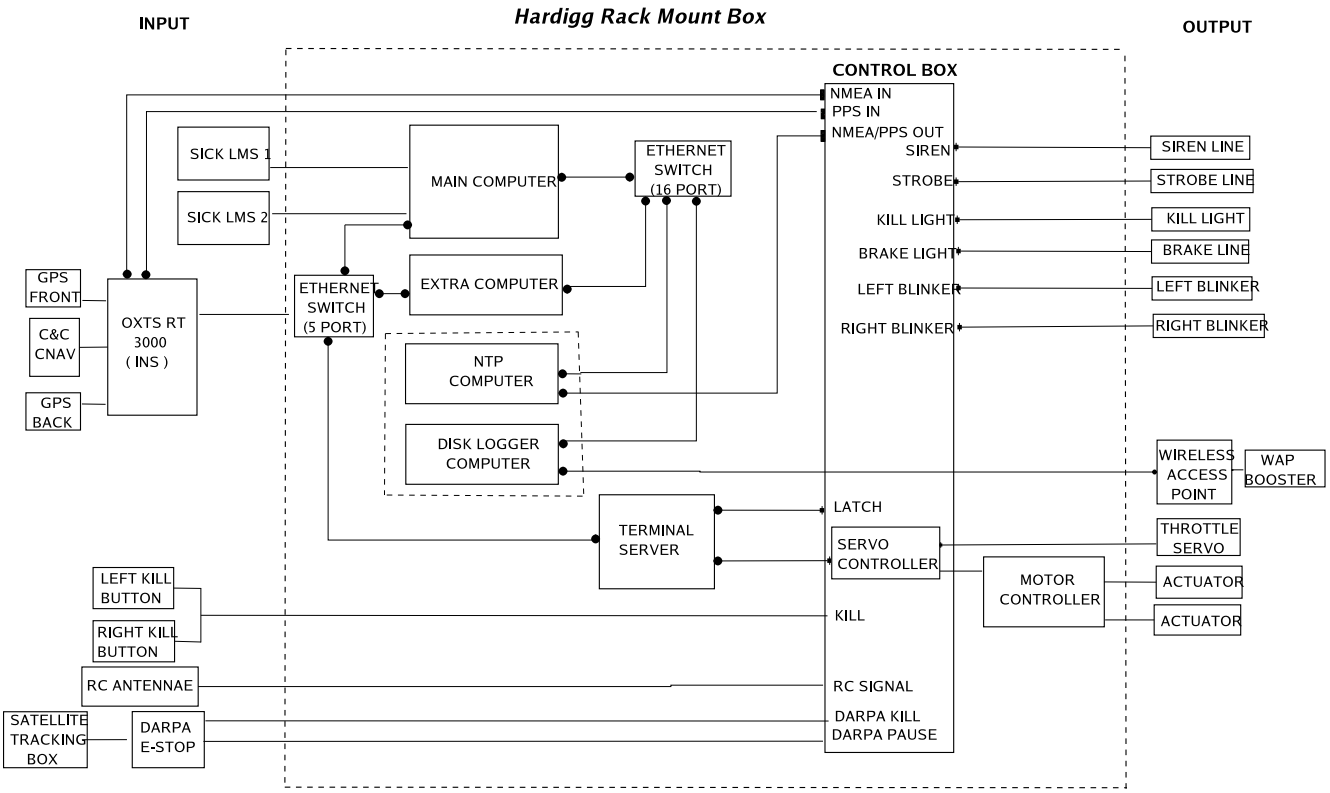

Figure 4: CajunBot Electronic System

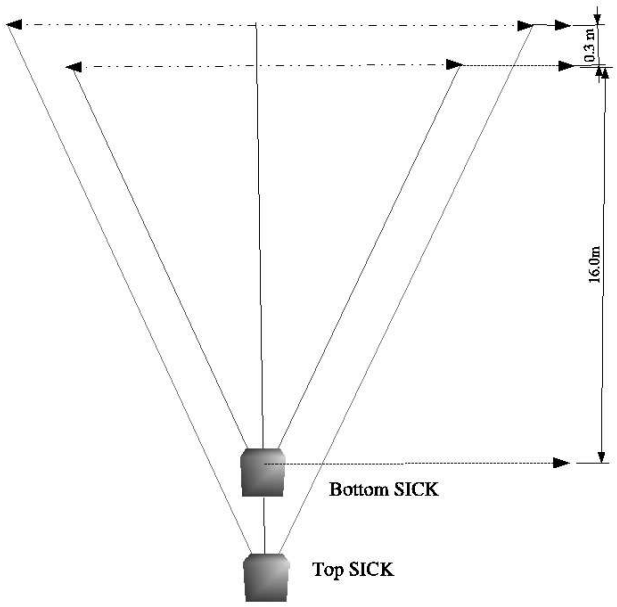

Figure 5: The Top View of the Top and Bottom LIDAR's

The Drive-By-Wire System provides (1) an interface for computer control over the vehicle's mechanics, i.e., throttle and levers, and signals, i.e., siren, safety lights, and brake indicators; (2) emergency control of the vehicle in autonomous mode; and (3) manual control of the vehicle when not in autonomous mode.

Computer control. The drive-by-wire system provides serial communication interfaces for controlling the vehicle through software. There are five serial interfaces, one each for servo, left lever, right lever, signals, and controls.

The servo interface receives a single byte value representing the position of the servo connected to the throttle cable of the vehicle. The control box translates the value into appropriate servo signals. The interfaces for the left and right lever also behave similarly, except that a motor controller is used to communicate the commands to two actuators, one connected to each lever.

The serial interface for signals uses one bit for each of: left and right turn signals, brake lights, kill lights, strobe lights, and siren. The control box translates the bits received into an appropriate electrical signal to activate/decativate a relay for each device.

While the other interfaces principally receive commands from the computers, the control interface provides input to the computers. The control interface currently provides single bit status indicators for pause and kill signals.

Emergency control. The system supports three emergency control operations: disable, manual override, and pause. The first two operations are implemented entirely in the Control Box, and override the computers. The vehicle may be disabled by the kill buttons on the

vehicle, DARPA Kill-switch, or kill button on the RC Controller. When a kill signal is received, the drive-by-wire system pulls the left and right levers to the brake position, cuts the throttle, kills the engine, turns on a flashing light, and also turns on the kill signal on the control interface to inform the control software. For safety reasons when the vehicle goes outside the RC controller's range, a kill signal is issued internally, unless the RC kill is disabled.

The kill signal brings the vehicle to an abrupt halt, which could be detrimental when a vehicle is traveling at a high speed. Depending on the dexterity of the operator in the chase vehicle, it may sometimes be preferred to take manual control of the vehicle, and bring it to a safe state. This is achieved by toggling a button on the RC controller. The drive-by-wire system then ignores the computer commands and takes commands from the RC controller.

Finally, there is the pause signal, which is simply passed on to the software, which then stops the vehicle with maximum safe deceleration. In the pause mode, both the software and the hardware continue to operate. When the pause mode is removed, the vehicle resumes autonomous operation.

Manual control. In non-autonomous (or manual) mode the vehicle is operated using an RC controller. This operation is completely at the hardware level, and does not require the computers to be turned on.

2.3.3 Computing System

The computing system, the collection of four computers as shown in Figure 4, provides the computational power of CajunBot. The computers labeled "Main Machine" and "Extra Machine" are Dell PowerEdge 750s. CajunBot performed in the GC with only the main machine. The extra machine was in place if there was a need to distribute the computation. The other two computers "NTP Machine" and "Disk Logger Machine" are mini-ITX boards, used because of their low cost and small physical size. Both the boards are mounted on the same 1-U case. The computers were mounted on a shock proof MIL-spec Hardigg cases to dampen the shocks

The NTP Machine provides Network Time Protocol service, a service necessary to synchronize data from multiple sensors and computers. Though NTP is a light process a separate machine is dedicated to it for pragmatic reasons. To setup a Linux machine as an NTP server requires applying a PPS patch. The patch was available for Linux 2.4 Kernel, not for Linux 2.6 Kernel, the base of Fedora Core 2 OS used on our main computing machines. The Disk Logger Machine is used for logging data during a run, typically for post analysis. The logging operation is moved to a separate machine so that a disk failure, a very likely possibility in a 10 hour run, does not interfere with the autonomous operation. Using flash media for logging data would have circumvented the need to have a separate machine for this purpose. However, we were unable to boot the Dell PowerEdge 750s from flash media, and had to use machines with disks.

Though all the devices could potentially be connected on the same network, the system is configured with three networks, once again for pragmatic reasons. The first network, connects

all the computers and the control box (via a Digi Terminal Server) through the 16-port gigabit Ethernet switch. This network carries data required for distributed processing. The INS is not connected on the same network because it required a certain network configuration for optimal performance. Since the INS data is used for all the phases of the processing, a second network consisting of the INS and all the computational machines is configured. A third network is configured to support real-time monitoring of the system from a chase vehicle. This network consists of the Disk Logger Machine connected to a wireless access point. The access point is not connected to the first network because the amount of data flowing on it saturates the wireless device, thus disrupting real-time monitoring. To overcome this problem the Disk Logger Machine samples the data before broadcasting.

3 Software Architecture

Figure 6 depicts CajunBot's software architecture. The system is decomposed into several components along functional boundaries. Each component, except the Middleware, runs as an independent process (program). The modules "Obstacle Detection", "Planner", and "Navigator" implement the *Core Algorithms* for autonomous behavior, and are discussed in the next section. All other modules are considered *Support Modules*, and are described in this section along with the design criteria that influence the software architecture.

The Drivers, Simulator, and Playback modules are mutually exclusive. While the Drivers module provides an interface to a physical device, the Simulator and the Playback modules provide virtual devices, as described below. Only one of the three modules can be active at any time. The mutual exclusion of the three modules is annotated in the architecture diagram by the walls between these modules, akin to the '|' symbol used in regular expressions.

3.1 Design Criteria

The following design criteria influence the software architecture:

Device independence. There are multiple vendors for sensors, such as, GPS, INS, IMU, LIDARs, and so forth. The core algorithms of the system should not depend on the specific device. It should be possible to replace an existing device with another make/model or to introduce a new device while making only localized changes to the system.

Algorithm independence. Development of the system is an iterative process, which involves choosing between competing algorithms for the same task. It should be possible to develop each algorithm in isolation, that is, in a separate program, and switch the algorithm being used by selecting some configuration values.

Scalability. The computational requirements of the system may vary as the system's design evolves. For instance, if CajunBot did not perform adequately with two LIDARs and there was a need to add a third, it would require more computational power. It should be easy to add additional sensors and also seamlessly distribute the application on multiple computers.

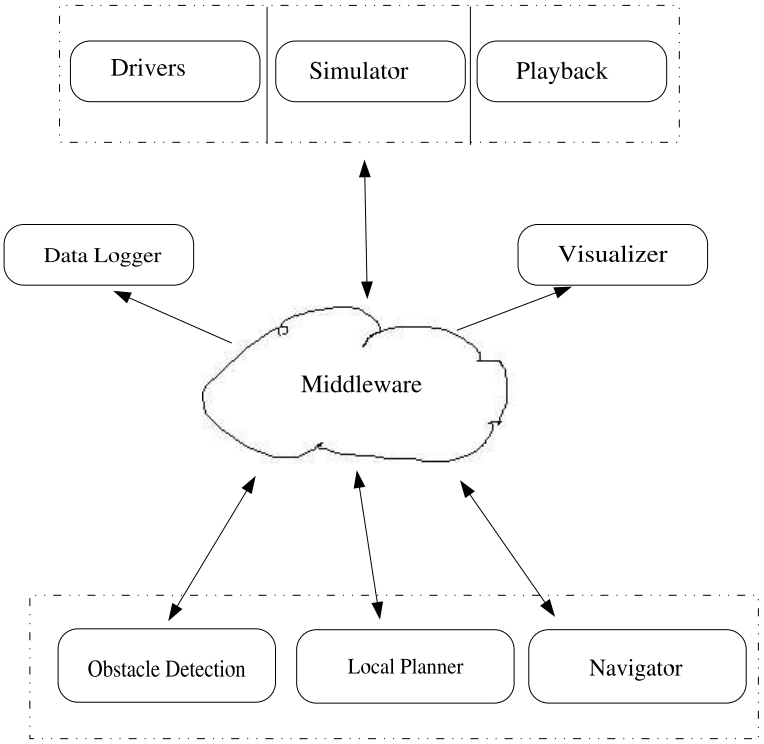

Figure 6: Software Architecture

Off-line Testability. The definitive way to test an AGV is to run it in the field. However, it is expensive to test each submodule of the system and every change by running the vehicle in the field. The system should enable off-line (in the lab) testing of various components and compositions of components.

Ease of debugging. To debug a system one needs access to internal data of the system. When debugging an AGV, it is most valuable if the internal data is available in real-time, when the vehicle is running. Debugging also requires performing the same operation over and over again, for one may not observe all the cues in a single run. The system should support (1) real-time monitoring of internal state of its various components and also (2) the ability to replay the internal states time-synchronized. The system should also support (3) presenting the data, which is expected to be voluminous, in a graphical form to enable ease of analysis.

3.2 CajunBot Middleware

The Middleware module, CBWare, provides the infrastructure for communication between distributed processes (CajunBot programs), such that the producers and consumers of data are independent of each other. Except for the properties of the data written to or read from CBWare, a module in the system does not need to know anything else about the module that has generated or will consume the data. This decoupling of modules is central to achieving the design criteria listed above.

CBWare provides two types of interfaces. A typed queue interface, CBQueues, for reading

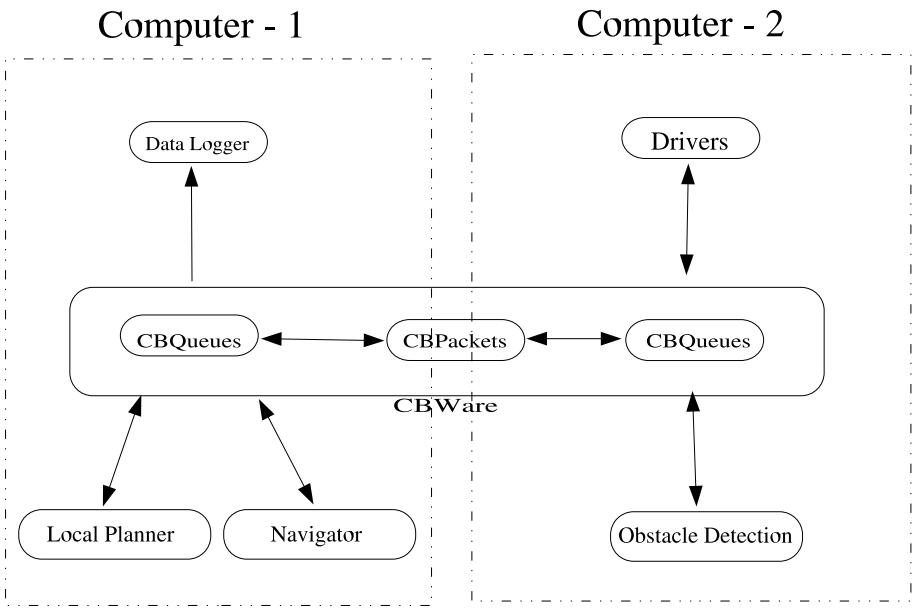

Figure 7: Distributed Onboard Software Architecture

and writing messages. And a typed message packet interface, CBPackets, for only writing messages. A typed message packet may also find its way into a queue, from where it may be read.

CBQueues provides distributed queues using a combination of POSIX Shared Memory (Marshall, 1999) and UDP communication. On an individual machine the queues are maintained as circular lists in the shared memory. The data written to a queue is distributed to other computers using a UDP broadcast. Figure 7 depicts how distributed interprocess communication is achieved by replicating shared memory queues across machines. This feature of CBWare to distribute queues over other machines allowed easy porting of programs over to multiple machines, achieving easy scalability of computational power, one of the design criteria.

CBQueues imposes an important constraint: each queue can have only one writer, but there is no limit on the number of readers. The single writer (producer) restriction ensures that the data in each distributed queue can be temporally ordered on the time the data was produced. If multiple producers of similar type of data exist, such as multiple LIDARs, a separate queue is maintained for each producer.

Besides providing the usual interfaces to access a queue, CBQueues also provides an interface to find in a queue two data items produced around a particular time. This capability, made possible due to temporal ordering of data in the queues, provides support for fusion of data from multiple sources based on the time of production. When two sources generate data at different frequencies, it may not always be appropriate to use the most recent data from both sources. Doing so may lead to the fusion of mutually inconsistent data. For instance, when a LIDAR scan is mapped to global coordinates using the INS data, the resulting coordinates would have significant error if the vehicle experienced a sharp bump immediately after the

scan. In such cases it is better to fuse data in close temporal proximity. Along the same lines, instead of using the data generated directly by a source, sometimes it is preferred to interpolate the data for the specific time when data from another source is produced. In our LIDAR and INS example, it may be preferred to interpolate the position of the vehicle to match the time of LIDAR scan.

The CBPackets interface provides support for multiple writers and multiple readers. However, in so doing it cannot support temporal fusion of data. This interface is most useful for distributing status, warning, and error messages. Such messages are used in isolation, that is, they are not fused with other messages, and are mostly used for monitoring, not control.

CBWare serves the same purpose as NIST's Neutral Message Language (NML) (Shackleford et al., 2000), Simmons & Dale's CMU-IPC (Simmons and James, 2001), or RTI's NDDS (Pardo-Castellote and Hamilton, 1999), to cite a few middleware frameworks for real-time, distributed systems. While CBWare shares several similarities with each of these systems, such as publish-subscribe communication, the fundamental, and most crucial, difference is that CBWare supports fusion of sensor (and other data) based on the time of production. This support has been an important contributor in CajunBot's ability to leverage rough terrain to increase its sensor performance.

3.3 Data Logger

The Data Logger is run on the Disk Logger machine. Besides logging the data on disk, the Data Logger broadcasts data on the wireless network for real-time monitoring in a chase vehicle. CajunBot communicates with the chase vehicle on an 802.11G wireless network. The variety of wireless communication equipment we have tried tends to crash when all data produced in the queues is put in the air. This is particularly true when, for the purpose of debugging, the internal states of Obstacle Detection and Local Planner modules are broadcast. To accomodate for the shortcomings of the wireless network, the data logger has provision to sample the data at some prescribed interval. If disk space is an issue, it also provides support to save only a sample of the data.

3.4 Drivers

Device independence is achieved by having a separate program, referred to as a Driver, to interact with a particular device. The Drivers are divided into two classes. 1) Sensor drivers, which read input data from sensors, such as the INS and LIDARs. 2) Control drivers, which control devices, such as throttle, left and right levers, safety lights, siren, kill lights, brake lights, and indicator lights.

Besides hiding the details of communicating with the device, a driver also transforms the data to match units and conventions used by the rest of the system. For instance, the CajunBot system measures angles in the anti-clockwise direction, with East as zero. If an IMU or INS uses any other convention for measuring angles, its corresponding device driver transforms angles from the device's convention to CajunBot's convention. Similarly, most

GPS and INS equipment tends to provide vehicle position in latitude/longitude, which is translated by the driver to UTM coordinates, as used by the CajunBot system.

3.5 Simulator

Off-line testability is a direct outcome of device independence. Since the core algorithms are unaware of the source/destination of the data, the data does not have to come from or go to an actual device. The algorithms may as well interact with virtual devices and a virtual world. The Simulator module (and the Playback module, discussed later) creates a virtual world in which the core algorithms can be tested in the laboratory.

CajunBot's Simulator, CBSim, is a physics-based simulator developed using the Open Dynamics Engine (ODE) physics engine. Along with simulating the vehicle dynamics and terrain, CBSim also simulates all the onboard sensors. It populates the same CBWare queues with data in the same format as the sensor drivers. It also reads vehicle control commands from CBWare queues and interprets them to have the desired effect on the simulated vehicle.

While CBSim is a physics-based simulator like Stage (Gerkey et al., 2003) and Gazebo (Vaughan, 2000), it has two interesting differences. First, CBSim does not provide any visual/graphical interface. The visualization of the world and the vehicle state is provided by the Visualizer module, discussed later. Second, CBSim also generates a clock, albeit a simulated one, using the CBWare queues.

The simulated clock helps in synchronizing the distributed programs when running in a virtual world. The distributed programs of CajunBot have a read-process-output-sleep loop, as elaborated later. The frequency at which a program is scheduled is controlled by choosing the duration for which it sleeps. When operating in the real-world, the duration of 'sleep' is measured in elapsed real-time. However, it is not beneficial to use elapsed real-time to control the sleep duration of a process when operating in a virtual environment. In particular, using the real-time for controlling sleep forces the simulation to execute at the same pace as the real program, even when one may be using faster and better computers. Thus, when operating in virtual environment we use the elapsed simulated time to determine how long a processs sleeps.

By maintaining a system wide simulated time, the CajunBot system is able to create a higher fidelity simulation than that provided by Stage and Gazebo. The computation in the entire system can be stopped by stopping the clock; and its speed can be altered by slowing down or speeding up the clock. This also makes it feasible to run the application in a single step mode, executing one cycle of all programs at a time, thereby significantly improving testing and debugging.

3.6 Playback

Offline-testing and debugging is further aided by the Playback module. This module reads data logged from the disk and populates CBWare queues associated with the data. The order in which data is placed in different queues is determined by the time stamp of the data. This

ensures that the queues are populated in the same relative order. In addition, the Playback module, like the Simulator module, generates the simulator time queue representing the system-wide clock.

This simple act of playing back the logged data has several benefits. In the simplest use, the data can be visualized (using the Visualizer module) over and over again, to replay a scenario that may have occured in the field or the simulator. It offers the ability to replay a run after a certain milestone, such as a certain amount of elapsed time or a waypoint is crossed. In a more significant use, the playback module can also be used to test the core algorithms with archived data. This capability has been instrumental in helping us refine and tune our obstacle detection algorithm. It is our common operating procedure to drive the vehicle over some terrain (such as during the DARPA National Qualifying Event), playback the INS and LIDAR data, apply the obstacle detection algorithm on the data, and then tune the parameters to improve the obstacle detection accuracy.

3.7 Visualizer

Real-time and off-line debugging is supported by CBViz, the Visualizer module. CBViz is an OpenGL graphical program that presents visual/graphical views of the world seen by the system. It accesses the data to be viewed from the CBWare queues. Thus, CBViz may be used to visualize data live during field tests and simulated tests, as well as visualizing logged data using the Playback module.

Since communication between processes occurs using CBWare, CBViz can visualize data flow between processes. We have also found it beneficial to create special queues in CBWare to provide CBViz with data that is otherwise internal to a process. This capability has been very effective, and almost essential, in achieving the Ease of Debugging design criterion as listed above.

4 Algorithms

This section presents the algorithms underlying CajunBot's autonomous behavior.

Figure 8 presents the system level data flow diagram. Although CajunBot's system uses the blackboard architecture, see Figure 6, the system level DFD of Figure 8 shows the actual data flows. Each step in the system level DFD is implemented by one or more independent programs. Each program has the following pattern:

```
while (true) {
   read inputs;
   perform processing;
   generate outputs;
   sleep for a specified time
}
```

Table 1: Operating frequencies of programs of Figure 8

Program	Frequency (Hz)
INS Driver	100
LIDAR Driver	75
Obstacle Detection	15
Local Planner	5
Navigator	20
Control Drivers	20

The overall steps are depicted in Figure 8. The INS data read in Step 1 gives the vehicle's state, which includes position in UTM coordinate space, orientations along the three dimensions (i.e., heading, roll, and pitch), speed over ground, and accelerations along the three dimensions. The LIDAR generates a sequence of scans, with each scan containing a collection of beams. Each beam is a value in polar coordinate space. The LIDAR scans are read in Step 2. In Step 3, the LIDAR scans and vehicle state (at the time of the scan) are used by the Obstacle Detection Module to create a Terrain Obstacle Map (TOM). The TOM is a map of obstacles in the vicinity of the vehicle. The Local Path Planner Module (Step 4) uses the TOM and the vehicle's state to generate a Navigation Plan. This consists of a sequence of waypoints and the recommended speed along each segment. The Navigation Plan is used by the Navigator Module, Step 5, to generate steering and throttle commands. In executing the plan the Navigator takes into account safety of the vehicle. Finally, the Control Drivers (Step 6) map the steering and throttle commands to physical devices. This includes the throttle servo position and actuator positions for brake control. The Pause and Kill Signals are read by the Signal Driver (Step 7). These signals are used by the Obstacle Detection and Navigator Modules, as described later. The Navigator Module also controls the emergency signals, siren and flashing lights.

The system does not use any explicit real-time primitives. Each program runs in an endless loop, reading from its input CBWare queues and writing to its output CBWare queues. When explicit syncronization is needed between the producer and consumer of some data, the consumer uses CBWare primitives that block it if no new data is available from the respective producer. Further, the sleep step of each program is tuned to have the program run at a certain frequency. The rate at which various programs of Figure 8 operate are provided in Table 1.

The algorithms for the Obstacle Detection, Local Path Planner, and Motion Controller Modules are key to providing the autonomous behavior and are described below.

4.1 Obstacle Detection Module

This section summarizes CajunBot's obstacle detection algorithm and highlights the specific features that enable it to take advantage of vibrations along the height axis, i.e., bumps, to improve its ability to detect obstacles.

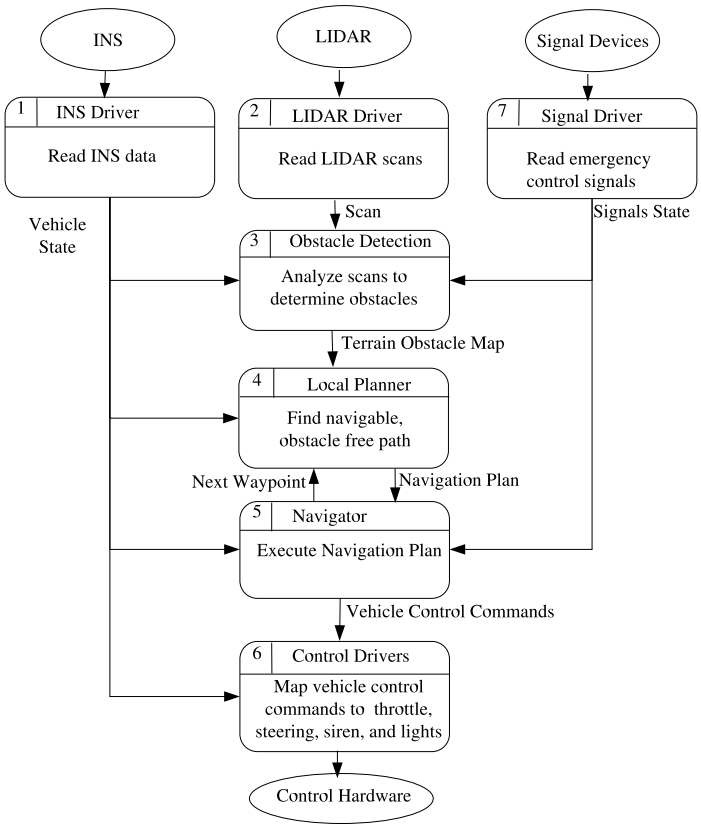

Figure 8: System Level Data Flow Diagram

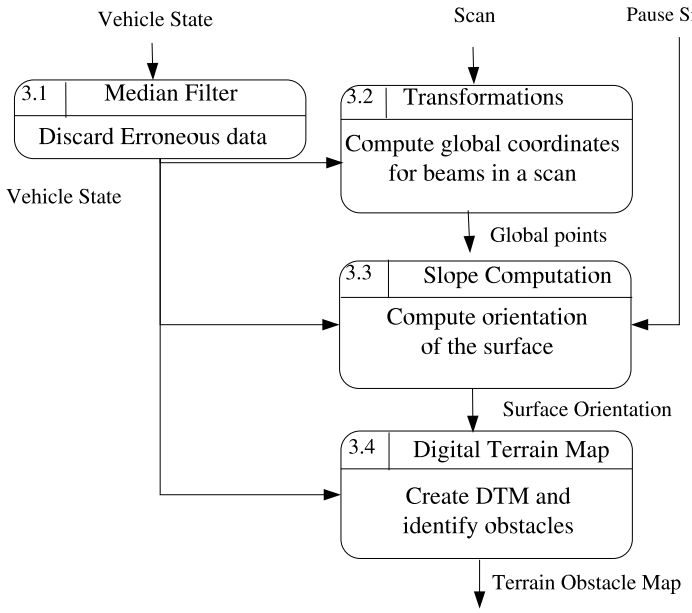

Figure 9: Data Flow Diagram for the Obstacle Detection Module

4.1.1 The Algorithm

The data flow diagram in Figure 9 enumerates the major steps of the obstacle detection algorithm. The algorithm takes as input the vehicle's state and LIDAR scans. The vehicle state data is filtered to attend to spikes in data due to sensor errors (Step 3.1), and then used to compute the global coordinates for the locations from which the beams in a LIDAR scan were reflected (Step 3.2). The global coordinates form a 3-D space with the X and Y axes corresponding to the Easting and Northing axes of UTM Coordinates, and the Z axis giving the height above sea level. Virtual triangular surfaces with sides of length 0.20m to 0.40m are created with the global points as the vertices. The slope of each such surface is computed and associated with the centroid of the triangle (Step 3.3). A vector product of the sides of the triangle yields the slope. The height and slope information is maintained in a digital terrain map, which is an infinite grid of 0.32m $\times$ 0.32m cells. A small part of this grid within the vicinity of the vehicle is analyzed to determine whether each cell contains obstacles (Step 3.4). This data is then extracted as a Terrain Obstacle Map.

Figure 10 graphically depicts data from the steps discussed above. The figure presents pertinent data at a particular instant of time. The grey region represents the path between two waypoints. The radial lines emanating from the lower part of the figure show the LIDAR beams. There are two sets of LIDAR beams, one for each LIDAR. Only beams that are reflected from some object or surface are shown. The scattering of black dots represent the global points, the points where LIDAR beams from some previous iteration had reflected. The figure is scattered with triangles created from the global points. Only global points that satisfy the spatio-temporal constraints, discussed later, are part of triangles. There is a lag in the data being displayed. The triangles shown, the global points, and the LIDAR beam

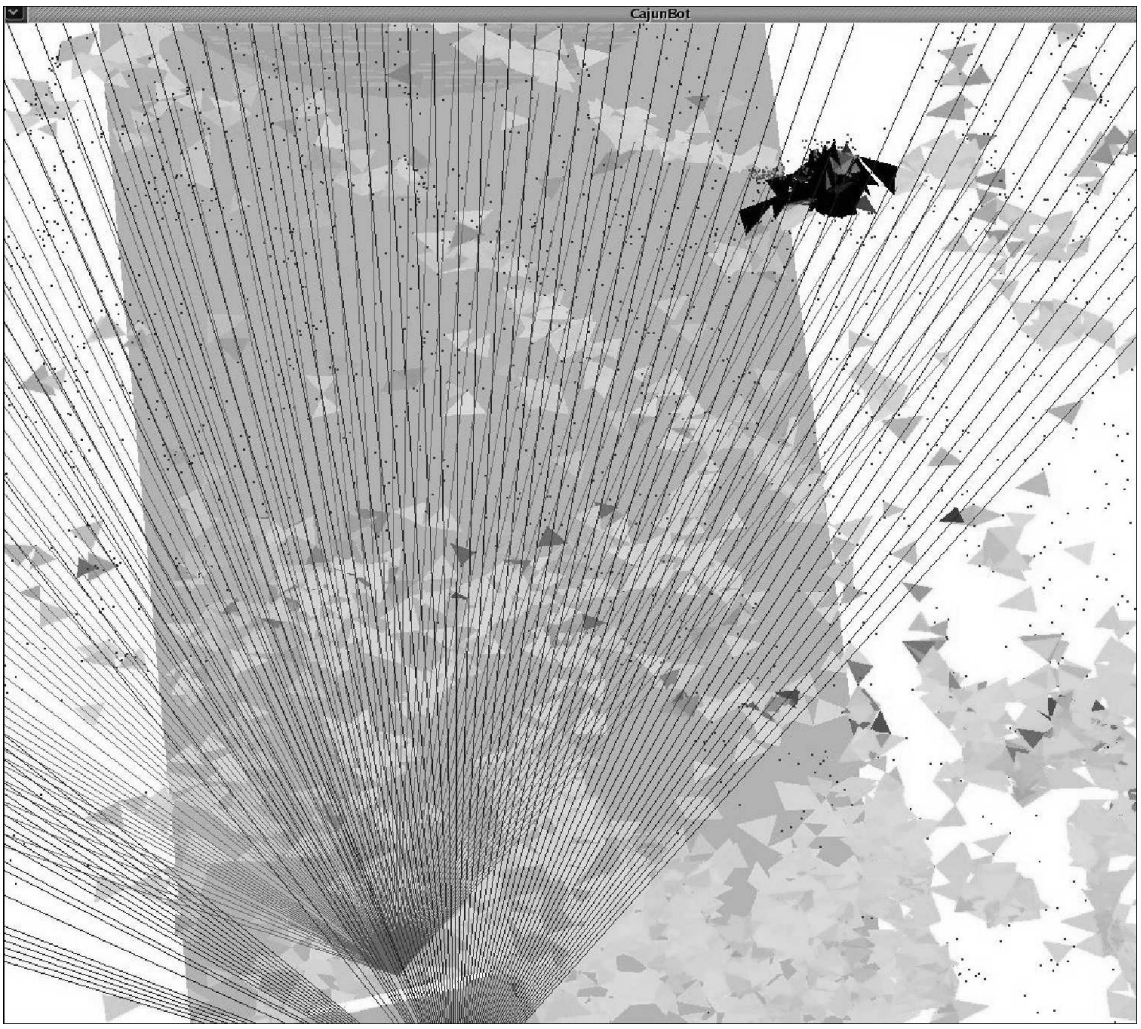

Figure 10: Virtual Triangle Visualization

are not from the same instant. Hence, some points that can make suitable triangles are not shown to form triangles. The shade of the triangles in Figure 10 represents the magnitude of slopes. The black triangles have high slope, +/- 90 degrees, and the ones with lighter shades have much smaller slopes. In the figure, a trash can is detected as an obstacle, as shown by the heap of black triangles. The data was collected in UL's Horse Farm, a farm with ungraded surface. The scattering of dark triangles is a result of the uneven surface.

An obstacle is classified as an obstacle using the following steps. First, a cell is tagged as a 'potential' obstacle if it satisfies one of three criteria. The number of times a cell is categorized as a potential obstacle by a criterion is counted. If this count exceeds a threshold–a separate threshold for each criterion–it is deemed an obstacle. The criteria used to determine the classification of a cell as a potential obstacle are as follows:

High absolute slope. A cell is deemed as a potential obstacle if the absolute maximum slope is greater than 40 degrees. Large objects, such as, cars, fences, and walls, for which all three vertices of a triangle can fall on the object, are identified as potential obstacles by this criterion. The threshold angle of 40 degrees is chosen because CajunBot cannot physically climb such a slope. Thus, this criterion also helps in keeping CajunBot away from unnavigable surfaces.

High relative slope. A cell is deemed as a potential obstacle if (1) the maximum difference between the slope of a cell and a neighbor is greater than 40 degrees and (2) if the maximum difference between the heights of the cell and that neighbor is greater than 23cm. This criterion helps in detecting rocks as obstacles, when the rock is not large enough to register three LIDAR beams that would form a triangle satisfying the spatio-temporal constraint. The criterion also helps in detecting large obstacles when traveling on a slope, for the relative slope of the obstacle may be 90 degrees, but the absolute slope may be less than 40 degrees. The test for height difference ensures that small rocks and bushes are not deemed as a potential obstacle. The height 23cm is 2cm more than the ground clearance of CajunBot.

High relative height. A cell is deemed as a potential obstacle if the difference between its height and the height of any of its neighbor is greater than 23cm. This criterion aids in detecting narrow obstacles, such as poles, that may register very few LIDAR hits.

The threshold counts of 5, 5, and 12, respectively, are used for the three criteria to confirm a potential obstacle as an obstacle.

As a matter of caution, Step 3.3 disables any processing when the Pause Signal is activated. This prevents the system from being corrupted if someone walks in front of the vehicle when the vehicle is paused, as may be expected since the Pause Signal is activated during startup and in an emergency.

4.1.2 Utilizing Bumps to Enhance Obstacle Detection Distance

Figure 11 presents evidence that the algorithm's obstacle detection distance improves with roughness of the terrain (bumps). The figure plots data logged by CajunBot traveling at

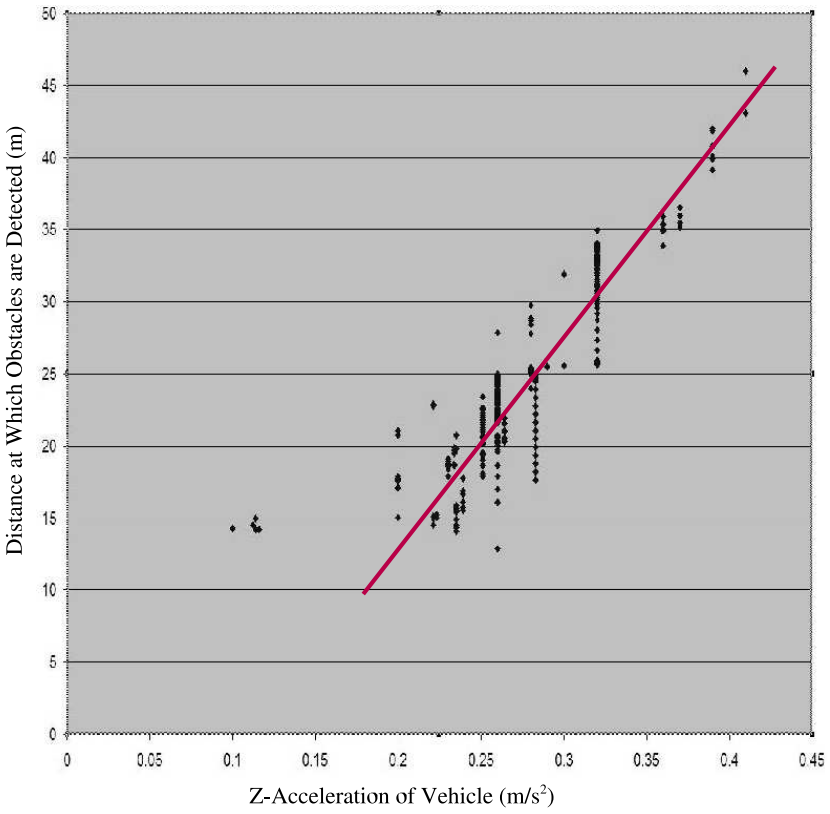

Figure 11: Graph Showing Distance to Detected Obstacles Versus Z-Acceleration

7m/s through a distance of about 640m of a bumpy section during the 2005 GC Final. The X-axis of the plot represents the absolute acceleration along the height (Z) axis at a particular time. Greater acceleration implies greater bumps. The Y-axis represents the largest distance from the vehicle at which an obstacle is recorded in the Terrain Obstacle Map. The plot is the result of pairing, at a particular instance, the vehicle's Z acceleration with the furthest recorded obstacle in the Terrain Obstacle Map (which need not always be the furthest point where the LIDAR beams hit). The plot shows that the obstacle detection distance increases almost linearly with the severity of bumps experienced by the vehicle. The absolute vertical acceleration was never less than 0.1 m/s^2 because the vehicle travelled at a high speed of 10 m/s on a rough terrain. That the onboard video did not show any obstacles on the track and that the obstacle detector also did not place any obstacles on the track leads us to believe that the method did not detect any false obstacles.

Bumps along the road have impact on two steps of the algorithm, Step 3.2, where data from the INS and LIDAR is fused and, Step 3.3, when data from beams from multiple LIDAR scans are collected to create a triangular surface. The issues and solutions for each of these steps are elaborated below.

In order to meaningfully fuse INS and LIDAR data it is important that the INS data give orientation of the LIDARs at the time a scan is read. Since it is not feasible to mount an INS on top of a LIDAR, due to the bulk and cost of an INS, the next logical solution is to

mount the two such that they are mutually rigid, that is, the two units experience the same movements. There are three general strategies to ensure mutual rigidity between sensors: (1) Using a vehicle with a very good suspension so as to dampen sudden rotational movements of the whole body and mounting the sensors anywhere in the body. (2) Mounting the sensors on a platform stabilized by a Gimbal or other stabilizers. (3) Mounting all sensors on a single platform and ensuring that the entire platform is rigid (i.e., does not have tuning fork effects). Of course, it is also possible to combine the three methods.

CajunBot uses the third strategy. The sensor mounting areas of the metal frame we created is rigid, strengthened by trusses and beams. In contrast, most other GC teams used the first strategy and the two Red Teams used a combination of the first two strategies.

Strategy 3 in itself does not completely ensure that mutually consistent INS and LIDAR data will be used for fusion. The problem still remains that the sensors generate data at different frequencies. Oxford RT 3102 generates data at 100Hz, producing data at 10ms intervals, whereas a SICK LMS 291 LIDAR operates at 75Hz, producing scans separated by 13ms intervals. Thus, the most recent INS reading available when a LIDAR scan is read may be up to 9ms old. Since a rigid sensor mount does not dampen rotational movements, it is also possible the INS may record a very different orientation than the time when the LMS data is recorded. Fusing these readings can give erroneous results, more so because an angular difference of a fraction of a degree can result in a LIDAR beam being mapped to a global point several feet away from the correct location.

The temporally ordered queues of CBWare and its support for interpolating data help in addressing the issue resulting from differences in the throughput of the sensors. Instead of fusing the most recent data from the two sensors, Step 3.2 computes global points by using the vehicle state generated by interpolating the state immediately before and immediately after the time when a LIDAR scan was read. Robots with some mechanism for stabilizing sensors can fuse a LIDAR scan with the most recent INS data because the stabilizing mechanism dampens rotatonal movements, thus ensuring that the sensors will not experience significantly different orientations in any 10ms period.

Absence of a sensor stabilizer also influences Step 3.3, wherein triangular surfaces are created by collecting global points corresponding to LIDAR beams. Since CajunBot's sensors are not stabilized, its successive scans do not incrementally sweep the surface. Instead, the scans are scattered over the surface as shown in Figure 12. This makes it impossible to create a sufficient number of triangular surfaces of sides 0.20m to 0.40m using points from successive scans (or even ten successive scans). It is always possible to create very large triangles, but then the slope of such a triangle is not always a good approximation for the actual slope of its centroid.

If the GPS/INS data were very precise then triangles of desired dimensions could be created by saving the global points from Step 3.2 in a terrain matrix, and finding groups of three points at a desired spatial distance. This is not practical because of Z-drift, the drift in Z values reported by a GPS (and therefore by the INS) over time. When stationary, a drift of 10 cm - 25 cm in Z values can make even a flat surface appear uneven.

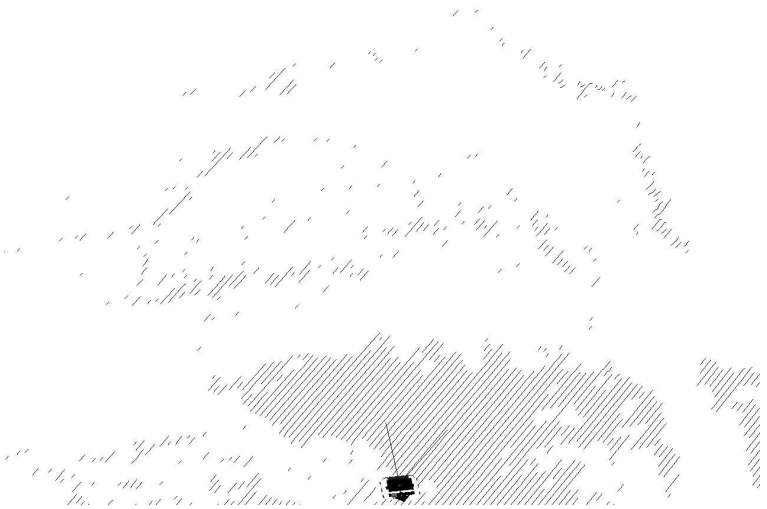

Figure 12: LIDAR Beams Scattered Due to Bumps

The Z-drift issue can be addressed by taking into account the time when a particular global point was observed. In other words, a global point is a 4-D value (x, y, z, and time-of-measurement). Besides requiring that the spatial distance between the points of a triangular surface be within 0.20m and 0.40m, Step 3.3 also requires that their temporal distance be under three seconds.

To recap, the following features of the Obstacle Detection Module enables it to utilize bumps to improve obstactle detection distance.

- A rigid frame for mounting all sensors.
- Fusing mutually consistent LIDAR scan with INS data based on the time of production of data.
- Using 4-D space and spatio-temporal constraints for creating triangles to compute the slope of locations in the 3-D world.

4.1.3 Performance Evaluation on 2005 GC Data

We now turn to the question of efficiency and scalability of the algorithm. Tables 2 and 3 present the following data for a single LIDAR and LIDAR pair configuration.

CPU Utilization. The average 'percentage CPU utilization', as reported by the Linux utility `top`, sampled every second.

Increase in CPU. The percentage increase in CPU utilization going from one LIDAR configuration to a configuration of two LIDARs.

Table 2 gives the data when the terrain was not very bumpy, whereas Table 3 presents data for bumpy terrain in the actual Grand Challenge Final Run. In both the situations,

Table 2: Effect of number of LIDARs, with low average bumps: $0.11m/s^2$

# LIDARs	1	2
CPU Utilization	12.4%	15.9%
Increase in CPU		28.23%

Table 3: Effect of number of LIDARs, with high average bumps: $0.24m/s^2$

# LIDARs	1	2
CPU Utilization	11.2%	13.7%
Increase in CPU		22.32%

adding another LIDAR reduces the obstacle detection time at a higher rate (38-48%) than the increase in the CPU utilization (22-28%). This implies our algorithm scales well with additional LIDARs, since the benefits of adding a LIDAR exceeds the costs.

Comparing data across the Table 2 and Table 3 further substantiates that our algorithm takes advantage of bumps. Compare the data for the single LIDAR configurations in the two tables. The CPU utilization is lower when the terrain is bumpy. The same is true for the dual LIDAR configuration. The more interesting point is that adding another LIDAR does not lead to the same increase in CPU utilization for the two forms of terrain. For the bumpy terrain the CPU utilization increased by 22.32%, which is significantly less than the 28.23% increase for the smoother terrain. The efficient and scalable implementation is due to two factors. First, in Step 3.3 it is not necessary that the triangles be created using global points observed by the same LIDAR. The data may be from multiple LIDARs. The only requirement is that the triangles created satisfy the spatio-temporal constraints. The second factor is that we utilize an efficient data structure for maintaining the 4-D space. Though the 4-D space is infinite, an efficient representation is achieved from the observation that only the most recent three seconds of the space need to be represented. This follows from the temporal constraint and that one point of each triangle created in Step 3.3 is always from the most recent scan.

4.2 Local Path Planner Module

The RDDF (route data description format) file provided as input to the vehicle may be considered a global plan, a plan computed in response to some global mission. The Local Path Planner Module's role is to create a navigation plan to advance the vehicle towards its mission taking into account the ground realities observed by the sensors. The navigation plan is a sequence of *local* waypoints that directs the vehicle towards an intermediate goal, while staying within the lateral boundary and avoiding obstacles. The intermediate goal–a point on the RDDF route at a fixed distance away from the vehicle–advances as the vehicle makes progress. Also associated to each local waypoint is a speed that is within the speed limits prescribed in the global plan and one that can be safely achieved by the vehicle.

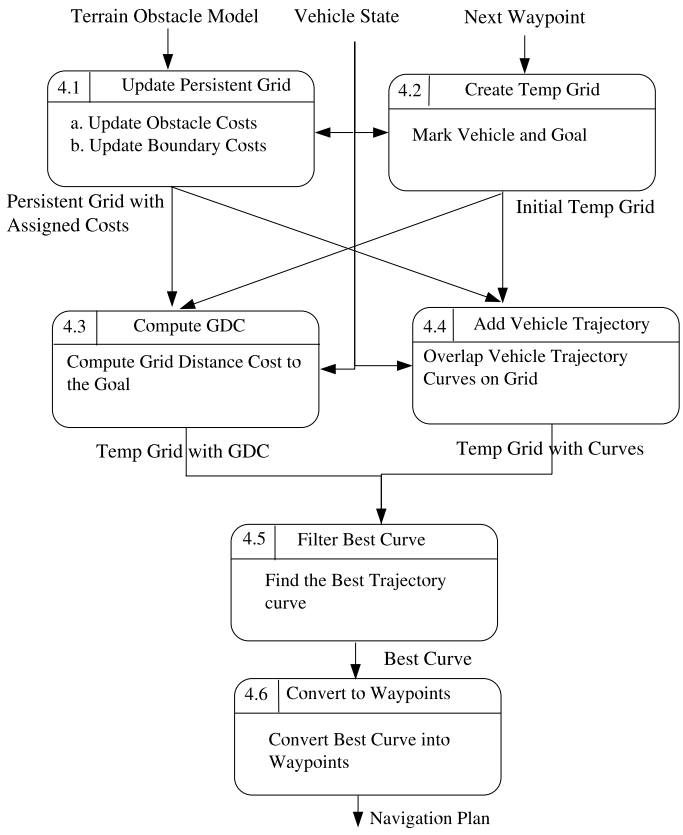

Figure 13: Data Flow Diagram for the Navigator Module

Figure 13 presents the logical data flow diagram of the local path planner[1]. The algorithm maintains two grid representations of the world, a persistent grid and a temporary grid. The persistent grid uses Easting and Northing coordinates whereas the temporary grid uses the vehicle's own coordinate space. As the name suggests, the persistent grid retains data between iterations of the algorithm, whereas the temporary grid is created afresh for each iteration. Following is the explanation of the steps followed by local path planner as shown in Figure 13.

Step 4.1 updates 'assigned costs' in the Persistent Grid. A cell in the persistent grid is assigned two types of costs, *obstacle cost* and *boundary cost*. The obstacle cost is based on the proximity of a cell to an obstacle. The boundary cost is based on its proximity to the boundary. The locations of obstacles are obtained from the Terrain Obstacle Model. To enable treating the vehicle as a point object, each obstacle is expanded in two concentric circles, as shown in Figure 14. First the region within the inner circle is called the hard obstacle expansion region. Each cell inside this circle is assigned an infinite obstacle cost. That the point vehicle is in this region implies the vehicle has crashed into some object. The hard expansion indicates the area the vehicle must avoid at any cost. Second, the region between the inner and the outer circle is called the soft obstacle expansion region. Cells in this region are assigned a smaller obstacle cost, and the cost decreases radially outwards. The soft expansion region discourages the local path planner from picking a path too close to an obstacle, unless it is absolutely necessary. Cells outside the soft expansion region are assigned a zero obstacle cost. Figure 14 also depicts three types of lateral boundary expansion region: warning region, soft expansion region, and a hard expansion region. The warning region is 0.3m wide and about 0.7m distance within the lateral boundary in the direction towards the center of the track. When the (point) vehicle is in this region, its wheels will be barely inside the lateral boundary. The cells in the warning region are given a very small boundary cost, a cost sufficient to keep the vehicle away from the lateral boundary. The soft expansion region starts outside of the warning region and extends past the lateral boundary for about 2m. Cells in this region have a slightly higher boundary cost than those in the warning region. This cost is intended to 'aggressively' prevent the vehicle out from cross the lateral boundary, and if the vehicle happens to be in the lateral boundary it aggressively pushes it back. The region outside of the track, further past the soft expansion region is called the hard expansion region. Cells in this region are assigned an infinite boundary cost. Like the hard obstacle expansion region, this region is considered to be unsafe for the vehicle and should be avoided at any cost. Cells inside the track, between the warning region, are assigned a zero boundary cost.

Step 4.2 uses the Vehicle State and the Next Global Waypoint information to create the temporary grid, which involves allocating memory, marking the cells for the intermediate goal, and marking the position of the vehicle.

Step 4.3 computes the Grid Distance Cost (GDC) (Maida et al., 2006; Barraquand et al., 1992) for the cells in-between the vehicle and the intermediate goal on the temporary grid. Each cell of the temporary grid has three costs associated with it: obstacle cost, lateral boundary cost and cell cost. Obstacle costs and lateral boundary costs are taken from the

[1]Note that branches in a DFD represent branching of data flow, not of control.

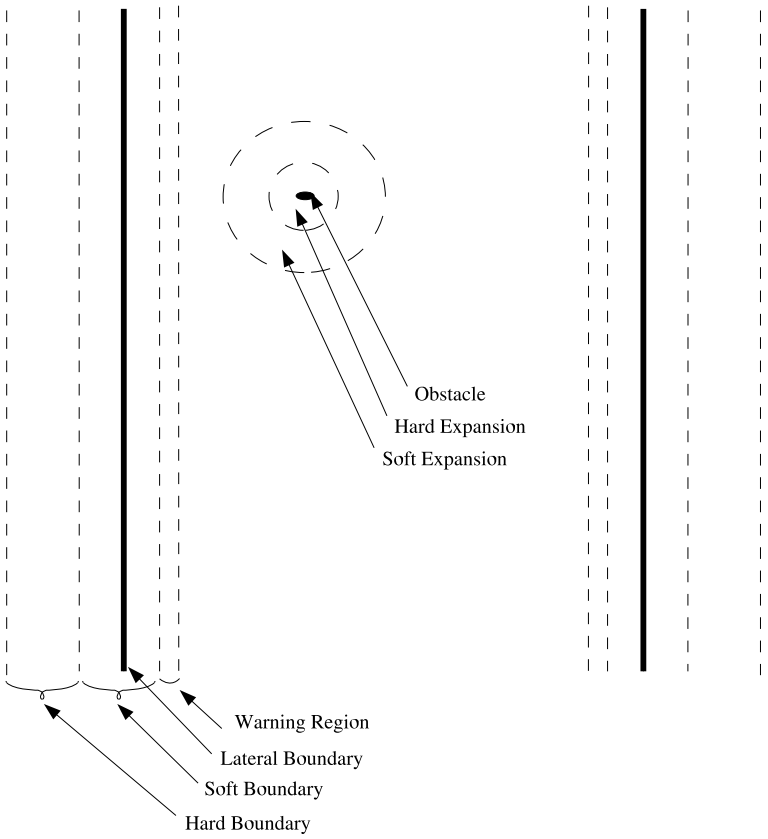

Figure 14: Obstacle and boundary costs

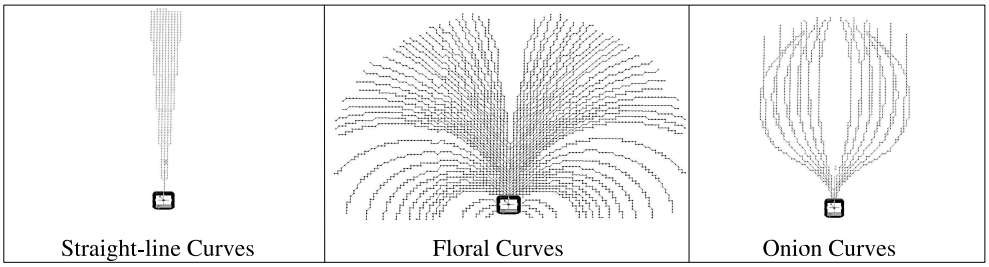

Figure 15: Maneuverable Curves of CajunBot

corresponding cell of the persistent grid, computed in Step 4.1. The cell cost represents the cost of traveling from that cell to the goal cell. It includes distance to the goal, penalty for traveling close to obstacles and lateral boundary.

The following recursive equations describe the relation between the cell cost $C(i)$ of a cell i and the cell costs of its neighbors $nbs(i)$.

$C(i) = 0$, i is a goal cell.

$C(i) = min\{C(j) + T(j,i) \mid j \in nbs(i)\}$, i is not a goal cell

where $T(j,i) = CF(j,i) \times (1 + B(j) + O(j))$, is the cost of traveling from cell j to cell i, $CF(j,i)$ is the Chamfer Factor (de Smith, 2004), $B(j)$ is the boundary cost of cell j, and $O(j)$ is the obstacle cost of cell j.

The fixed point of the above system of equations gives the desired cost. We use the A* algorithm to efficiently direct the propagation of costs from the goal towards the vehicle.

Step 4.4 overlays on the temporary grid a set of pre-computed curves representing acceptable movements of the vehicle. The curves are truncated at the point where they enter an obstacle cell, or a lateral boundary cell. Figure 15 shows the maneuverable curves in an obstacle free world. The curves originate at the vehicle and spread out along the orientation of the vehicle. These curves are pre-computed only once by simulating the vehicle's steering control (Scheuer and Xie, 1999; Scheuer and Laugier, 1999). Unlike curve computations based on Dubin's car (Dubins, 1957), our collection of curves does not represent all possible maneuvers that can be made by the vehicle. Instead, as evident from Figure 15, our collection contains *straight-line curves*, those curves emanating in the direction of the vehicle's heading; *floral curves*, curves that diverge from the current heading of the vehicle like a floral arrangement; and *onion curves*, the curves that diverge from the straight path and then converge back, like layers in an onion. A combination of straight-line, floral curves, and onion-curves is found to be sufficient to go around obstacles and make various types of turns. This is because the Local Path Planner generates a new navigation plan in every iteration. When an obstacle is first seen, an onion curve diverts the vehicle away from the obstacle. In a subsequent iteration, a floral curve brings the the vehicle smoothly back on track.

Step 4.5 uses the temporary grid annotated with the GDC and projected curves to select the Best Curve. This is done in two steps. First, a set of candidate curves are selected from the

projected curves. Second, a Best Curve is selected from the candidate curves. Each curve is evaluated on three properties, namely, a) length of the curve, b) final cost to the goal, i.e., the cell cost of the cell where the curve terminates, and c) the rate of reduction in the cell cost. The latter is the reduction in cell cost from the start of the curve to the end of the curve divided by the cost of traveling through each cell–$T(j, i)$ described earlier–along the curve. The candidate curves are selected from the projected curves by eliminating curves as follows. First, all curves that are smaller than a minimum acceptable length are removed. Next, of the remaining curves the smallest final cost to the goal is found. All curves whose final cost to the goal is greater than some threshold more than the smallest final cost are removed. Finally, the highest rate of reduction in cell cost of the remaining curves is found. Any curve whose rate of reduction is less than some threshold of the highest is removed. Now, the Best Curve of the set of candidate curves is the curve with the end point closest to the centroid of the all the candidate curves. If the set of candidate curves is empty, the path planner reports that it cannot generate any path.

Step 4.6 converts the best curve into a sequence of local waypoints. This step also annotates each waypoint with a recommended speed, which is computed using the RDDF speed, distance to the nearest obstacle, and amount of turn to make at that waypoint.

The key innovation of our algorithm is that it is a hybrid of discrete and differential algorithms (see (LaValle, 2006) for a comprehensive survey on planning algorithms). Step 4.3, the discrete part of the algorithm, computes GDC without taking into account the vehicle's state. Steps 4.4 and 4.5, the differential components of the algorithm, take the vehicle's state and its maneuverability into account to pick the Best Curve. By combining discrete and differential algorithms, we are able to get the best of both worlds. The discrete algorithm is fast and efficient, and hence can be used for planning over a large area. The differential component computes a path only in the vicinity of the vehicle. But, since the path selected is based on GDC, the curve chosen may be influenced by terrain conditions much further away. Thus, the navigation plan created is maneuverable within the vicinity of the vehicle and also brings the vehicle to a position close to an optimal path.

There are other non-trivial and interesting aspects of the algorithm that are worthy of explanation: 1) representation of the goal; and 2) representation of robot and obstacles to aid in creation of navigable paths.

Representation of goal. The size and shape of the goal has very significant effect on the paths created by the algorithm. If the goal is a point object on the center line, and there is an obstacle near the goal, the path generated will hug the boundaries of the obstacle even if there is a large amount of free space on the track. If the goal is a straight line encompassing the whole segment, perpendicular to the direction of motion, then the path may hug the corners after a turn, since that will be the shortest path.

We use a V-shaped goal, with an angle of about 150 degrees at the vertex. When there are no obstacles on the track, the tip of the V-goal smoothly brings the vehicle to the center of the track. However if there is an obstacle on the center lane of the track, the path generated will smoothly deflect away from the obstacle aiming for some other point on the V-shaped goal.

Representation of robot and obstacles. The classical approach to path planning for a circular robot with unlimited mobility is to model the robot as a point object and to expand an obstacle cell in a circle the same dimension as the robot (Feng et al., 1990; Stentz and Hebert, 1995). Whether the robot placed in a particular cell will collide with an obstacle can be determined by checking if the cell falls in the expansion region of an obstacle. This model of robot and obstacle leads to a very efficient test for collision.

We extend the above approach for the AGV domain. The Local Path Planner represents the vehicle as a point object and expands an obstacle cell in two concentric circles. The inner circle is called the *hard expansion* and is considered a hard obstacle. Presence of the (point) vehicle in a hard expansion cell implies imminent collision. This is encoded by associating an extremely high cost for being in that cell. The ring between the inner and outer circle is called the soft expansion. It is an area that is not very desirable for the vehicle to be in, unless there is no other option. The inner cells of the soft expansion are considered as far less desirable than the outer cells. This is encoded by using a higher cost for the inner cells than the outer cells.

CajunBot is 1.5m wide, that is, 0.75 m wide from center of the body to a side. Hence the hard expansion radius is set to 1m, giving an extra distance of 0.25m for sensor and vehicle control inaccuracies. The soft expansion radius is set to 2.5m. The combination of hard and soft expansion ensures that the path chosen using the discrete algorithm is navigable by CajunBot even though the vehicle is rectangular, and not circular.

4.3 Navigator

The Navigation Module is responsible for executing the plan generated by the Local Path Planner taking into account the vehicle's dynamics. As shown in Figure 8, the module takes as input the Navigation Plan, the Vehicle State and Safety Signals, and generates Vehicle Control Commands for the Control Drivers Module and the Next Waypoint (index to next global path waypoint) for the Local Path Planner Module.

It is the Navigation Module's responsibility to make the vehicle drive smoothly even when successive straight-line segments in the input plan are at sharp angles and have different speed limits. To provide such a driving experience, the Navigation Module performs four tasks: instantaneous steering guide, instantaneous speed guide, steering controller, and speed controller, as elaborated below.

Instantaneous Steering Guide. The instantaneous steering guide determines the desired heading for the vehicle in order to follow the Navigation Plan provided by the Local Path Planner. If the vehicle is not on the path suggested in the Navigation Plan, the instantaneous steering guide computes the instantaneous desired heading to smoothly bring the vehicle back on to the path. It is the task of the steering controller to achieve the instantaneous desired heading so computed.

The instantaneous desired heading is computed using a variation of the *follow-the-carrot* method (Hebert et al., 1997). The vehicle's position is projected on the segment of the

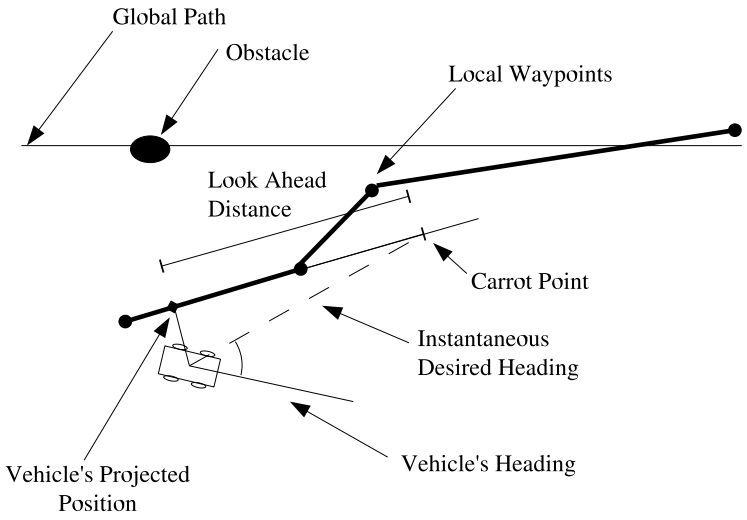

Figure 16: Steering Guide

Navigation Plan being executed (see Figure 16). A carrot point is marked along this segment at a *look-ahead-distance* away from the vehicle's projected position. The orientation of the line joining the vehicle and the carrot point gives the instantaneous desired heading. Our method is different from follow-the-carrot in that carrot point is marked at a *look-ahead-distance* on the current segment, not at a distance on the path. If the length of the segment is shorter than the *look-ahead-distance* we extend the segment for the purpose of placing the carrot point. In contrast, the follow-the-carrot method, travels along the path to find the carrot point.

Instantaneous Speed Guide. The instantaneous speed guide computes the desired speed that vehicle should try to achieve based on the present speed, the vehicle's kinematic limits, and the safe speed limit. The safe speed limit depends on the track speed given in the Navigation Plan, the safe speed for making turns ahead, and, if the pause signal is activated then, the safe speed to bring the vehicle to a stop.

Steering Controller. The responsibility of the steering controller is to ensure that the vehicle maintains the desired heading. It generates low level steering commands for the left and right brakes in order maintain this desired heading. The low level steering commands are generated as a floating point value between -1 and 1. A negative value refers to amount of the left brake to apply and positive value refers to the amount of the right brake to apply.

The controller uses an incremental PID to generate steering commands by using the difference between the desired heading and the present vehicle heading as error input. The controller uses different PID constants at different speeds, as CajunBot, a skid steered vehicle shows varying steering responsiveness for same steering commands at different speeds.

Speed Controller. The speed controller is responsible for achieving the desired speed suggested by the instantaneous speed guide. The speed controller emits a floating point value in the range of -1 to +1, where a negative value represents amount of brakes to apply and a positive value represents the amount of throttle to give. Thus, the controller will never

try to perform both active braking and throttling at the same time. The speed controller is also a classic incremental PD controller that uses the difference between desired speed and present vehicle speed as error input.

5 Field Experience

This section summarizes Team CajunBot's experience during the NQE (National Qualifying Event) and the 2005 GC Final.

The primary goal of the team was to be amongst the GC finalists and travel over 11.8km (over 7.3 miles), the distance traveled by Red Team's Sand Storm in 2004 GC. Due to its inherent low maximum speed 10.35m/s (23 miles/hr), CajunBot was never a contender to win the race based on the speed against the faster competitors like Sandstorm and Stanley. Further, the Max IV ATV, the underlying vehicle, is not normally used to transport for very long distances, running continuously for hours. So we were unsure if the vehicle's mechanics would hold together for the entire run. However, we were assured by the manufacturer and the dealer that the vehicle was rugged enough to last that far. In any case, the team felt that the core problems to be solved were in software, and that our ability to solve them could be amply demonstrated by successfully completing the NQE and traveling a significant distance in the final.

Indeed during the NQE and GC, CajunBot demonstrated the ability to navigate the course, control speed, and detect and avoid obstacles at reasonable speeds. In the GC final CajunBot started at the 21^{st} position and covered about 28.324km (17.6miles) before it was killed due to a mechanical failure. CajunBot was placed 15^{th} on the basis of the distance covered. Along the way it overtook two vehicles that were still in the running, achieved its maximum speed of 10.35 m/s, and covered the 28.324km distance at an average speed of 4.95m/s (11 miles/hr).

5.1 National Qualifying Event (NQE)

Of the six NQE runs, CajunBot completed two runs (Runs 4 and 5), ran into the last car on the final stretch in Run 6, did not compete in Run 3 because of a mechanical failure, suddenly stopped after going through the tunnel due to a GPS/INS related failure in Run 2, and climbed the hay bails when approaching the tunnel due to a rounding error in the path planner in Run 1.

Run 1: In the first run of the qualification round, CajunBot started well passing through the gates and cones, going up the slope, and, through the speed section. The next part of the course was a narrow section bounded by hay bails which lead to the tunnel. While going through this section CajunBot made a sudden left turn to avoid the hay bails on the right, and in so doing climbed up on the bails on the left. All three wheels on left side of CajunBot were off-ground as the vehicle tried to climb the hay bail. Ironically, bringing the vehicle back on the track required spinning the wheels which were in the air and thus without traction. This was the first time we experienced the limitation of skid-steering. CajunBot

had to be taken off the track.

Of course, the incident would not have happened if CajunBot had chosen a straight path along the center of the route. We were puzzled why it did not do so. The reason turned out to be a rounding error in the mapping of a curve path to the grid cell. The effect of the rounding error was magnified because the location that was rounded off was very close to the vehicle, leading to a very high change in instantaneous heading. Ironically, again, the rounding error was not a complete oversight. The single line of faulty code was annotated as 'FIX ME' for later. The approximate calculation was put in place, to be corrected when other pieces were completed. Addressing the 'FIX ME' never became a priority because it never led to any failure until that NQE Run.

Run 2: Very much like the first run, for the second NQE run, CajunBot started confidently, avoiding the initial set of obstacles, passing through the gates and the slope. In the speed section the vehicle reached its top speed of 23 miles/hr. As expected from our testing in the simulator, CajunBot followed a straight-line path through the narrow section bounded by hay bails and the tunnel that followed. It avoided the tires, stationary cars and went swiftly through the gravel section. The navigation system was impeccable as it made a smooth curve by the wall, passing through the narrow section between the cones and the wall. And then CajunBot simply stopped. After waiting for about 10 minutes, the vehicle was manually driven off the track.

The analysis of the logged data revealed a sudden change in the GPS height value in the region where the vehicle stopped. The height value (Z) had jumped by 30 meters in a fraction of a second causing the software to detect a wall-like obstacle in front of the robot. The entire region was filled with the obstacles, forcing the navigation software to stop giving paths.

The reason for the spike in Z value turned out to be due to loss of GPS signal in the tunnel. After the GPS signal was lost the INS started using ded-reckoning to estimate change in its position. Even after passing through the tunnel and regaining the GPS signal, the INS continued ded-reckoning, or so we understand. At some point it switched from ded-reckoning to using the Z-value reading from the GPS. But by then ded-reckoning errors had accumulated and we experienced a sudden spike in the data.

A median filter was added to the software, which monitors the data from the GPS/INS. The filter discards the data if any unreasonable rate of change in the height and position information of the GPS/INS data was detected. Also, upon detecting a spike in data, the obstacle detection module also flushes its spatio-temporal data to ensure that the data before and after the spike are not mixed during analysis.

Run 3: As CajunBot was being brought for the third run, it was observed that the vehicle steering incorrectly while been driven using the RC control. The reason turned out to be a broken transmission. We had to forgo this run as the vehicle was not in operational condition.

This was a good lesson about the consequences of using an uncommon vehicle. The nearest

dealer of Max ATVs was about 320km (200miles) away. We were extremely lucky that "2 The Max ATV" dealer was very magnanimous. She was willing to bring a brand new Max IV ATV to us, and allow us to scavenge it for spares, at no additional cost but to replace the parts when we were done. The team changed the entire transmission of the vehicle in less than seven hours and was ready for the fourth run the next morning.

Run 4: The fourth run was the first successful run for CajunBot at the NQE. The vehicle started smoothly, detected and avoided every obstacle, and successfully completed the run.

For the fourth run, the speed section was shifted to a more bumpy stretch as compared to the earlier runs. The logged data showed that CajunBot saw a line of false obstacles a couple of times when the vehicle was at its maximum speed on a considerably bumpy track. The sensors were pointing way far apart. The top LIDAR was aimed at 25m in front of the robot whereas the bottom LIDAR was aimed at 7m in front of the robot. During testing a few days before the NQE the configuration of the sensors was changed from the one described in Figure 5. The bottom LIDAR was reoriented to reduce the blind spots at turns. This change had the unintended consequence of increasing false obstacles in rough conditions. The configuration of Figure 5 was chosen to ensure rapid creation of triangles that satisfied the spatio-temporal constraints. With only 0.3m separation in their range, triangles of Figure 10 could be created using the global points resulting from the most recent scans of the two LIDARs. Increasing the separation in their range to 18m significantly decreased the chances of grouping beams from the most recent scans from the two LIDARs. When the global points from the two LIDARs were grouped, they would be expected to be temporally a part, and therefore the triangles were prone to errors. By increasing the number of such triangles, we increased the chances of detecting false obstacles. We could address the problem by reverting the sensor configuration. But doing so implied more risk since we did not have the time to test the effect of the change. Hence, the team decided not to make any changes and live with the consequences.

Run 5: The fifth run was also a successful run, with the only difference that the vehicle could not detect the lower leg of the tank trap, placed in the extreme end of the run. It brushed the tank trap on its way to the end of the course.

Low data density on the lower leg of the tank trap caused the software not to detect it as an obstacle. A solution for this was to increase the time for which the LIDAR data is used to from the triangles or to point the LIDAR's more closer such that there is more data density. The first option was ruled out as the GPS signal was not reliable after the vehicle passes through the tunnel. Increasing the time might have led to an increase in false obstacles. The second option was not tested in the recent past, hence, the team decided to keep the same configuration for the rest of the runs.

Run 6: The final run of the NQE, the run 6, was a near perfect run until the very last part. The last 200m of the track had two cars and a tank trap as obstacles in the path. CajunBot detected and avoided the first car perfectly. However, while trying to avoid the second car it hit the car in the corner. Like the first run, CajunBot's left wheels were in the air and it needed to spin those wheels to turn right. CajunBot had to be taken off the track.

Based on the analysis of the logged data, as CajunBot turned to avoid the first car, the second car was in the blind spot of the top LIDAR. By the time the bottom LIDAR could detect the car as an obstacle, CajunBot was dangerously close to it. The delay in detecting the obstacle did not give enough time for the local path planner to steer the vehicle around the obstacle. The vehicle hit the right rear end of the car while trying to steer away from it.

Having completed two runs successfully and one near successful run, CajunBot was selected as one of the 23 finalists to compete in the DARPA Grand Challenge 2005. The team and the bot moved to Primm, Nevada.

In the time between the NQE and the GC final, we were afforded a window of opportunity to fix the configuration of our LIDARs. We reverted the LIDARs to the configuration given in Figure 5 and tested it near Slash X, the starting point of the 2004 Grand Challenge.

5.2 DARPA Grand Challenge Final Run

On the grand finale, CajunBot was the 21^{st} robot to start. She was flagged off at 8:30am. Just about that time the weather took a turn. The winds picked up, blowing through the dry lake bed and causing a big sand storm. In no time, CajunBot was out of sight, in the thick of the storm. We knew that the LIDARs could not see through the sand, and were pretty nervous. To our relief the DARPA scoreboard showed CajunBot was still moving. It took about an hour for CajunBot to complete the 12.874km (8 mile) loop and pass the spectator stand. By then the weather had settled and she was moving really well. For the next hour, CajunBot passed several disabled vehicles and two vehicles still in the run. We were pleased that she was going strong. It was now poised to enter a region that was also part of the tail end of the course. The leading bots, Stanley, Sandstorm, and Highlander, were about to enter this region. To give precedence to the leading bots, CajunBot was paused. It took about 45 minutes for the lead bots to clear that path. When CajunBot was un-paused, she simply failed to start. After attempting to restart for 20 minutes, the vehicle was disabled.

The reason why CajunBot failed to restart turned out to be very mundane, and related to the transmission failure before Run 3. The transmission has two plungers that connect to two levers. One lever is for braking the left wheels or engaging the left transmission. The second one is for the right side. We control each lever using a lead-screw linear actuator. The actuators consume current when the lead screw is tightened to pull a load. However, if power is cut-off it stays locked in a position. We map the thrust of the actuator to a range from 0 to 1, to correspond to the maximum desired movement of the levers. After the transmission failure, we did not calibrate the actuators correctly. Its '1' was mapped to a position that the transmission could not physically reach. When the vehicle was put in to pause mode, to engage the brakes the levers had to be moved to '1'. However, this position could not be reached and the motor controller continued to attempt to move it. In the process, for 45 minutes the motor was fed its peak current, a current it can withstand only for short duration. That caused the motors on the actuators to burn out.

Further analysis of the logged data revealed how CajunBot weathered the sand storm. The

on-board video show CajunBot completely engulfed in the sand. That led it to see 'false obstacles', forcing it to go out of the track to avoid them. When the vehicle strays too far outside the lateral boundary we disable path planning and force it back to the track. Once it was back on track it would repeat the same behavior, thus appearing as though it is wandering aimlessly. However, after the sand storm cleared, the video shows CajunBot running very much along the middle of the track, and passing stalled or stopped vehicles. The logged data shows absolutely zero false obstacles throughout the run after the storm, even in areas where the vehicle experienced severe bumps.

Regardless of the outcome of the race, Team CajunBot came away with a much better understanding of autonomous navigation. We are very confident that, but for the mechanical failures, the vehicle would have completed the track, especially since there was no weather related disturbance in the later part of the day. The team is looking forward for the next challenge and is working on the new proposed entry, the RaginBot - a 2004 Jeep Rubicon.

6 Future work

Our experience suggests that field testing is one of the most expensive parts of developing an AGV. To field test, one must have a fully operational vehicle, a field for testing it, correct weather conditions, and a significant size staff. Unless the procedures for bringing the vehicle to the field are very well-defined, small issues, such as insufficient gas in the generator, can consume significant time.

Having a fully operational vehicle is no small requirement, given that an AGV has linear dependencies between the automotive, the electromechanical components, the electrical, electronics, sensors, and the software. Failure in any one of the components can hold back the testing.

Yet testing in the current generation of simulation environments, such as CBSim, Stage (Gerkey et al., 2003) and Gazebo (Vaughan, 2000) are quite limited. While these environments are good for doing integration testing, their simulation abilities are quite limited providing information about how the vehicle may perform in the real-world, such as, in different terrain and weather conditions.

We are working on developing a higher fidelity simulation and visualization of the real-world and the vehicle. The environment will utilize a cluster of computers and a 6-surface cave to create an immersive visualization of real-time simulation.

There is another aspect of field testing that can be improved. How does one evaluate the performance of a vehicle in the field? A report of observations by the testing team while useful leaves room for subjectivity and human error. A 10 hour run can be pretty long for someone to correctly recount and to pay attention for taking notes.

There is ongoing work in our lab to develop automated, objective methods to analyze logged data from a field run and to evaluate a vehicle's performance.

In addition we are also working on improving the system further, such as, to introduce camera vision capability; improve reliability by introducing the ability to restart individual software components or the whole system; and porting the system to a completely new vehicle, Ragin'Bot, a 4x4 Jeep Wrangler.

7 Conclusion

Participating in the DARPA Grand Challenge has been an intense experience, unlike anything we have experienced during normal research. The most significant difference was to have an end-to-end system that would perform in a real-world situations, and in the course of the development solve some significant research problems.

The requirement to develop an end-to-end system implied we could not have tunnel vision and get carried away improving only one component of the system. That the vehicle would perform in real-world situations required us to consider solutions for a multitude of scenarios, rather than be content with developing solutions for some simplified scenarios.

This paper presents the overall hardware and software architecture of the CajunBot. More importantly, it highlights the specific innovations resulting from the effort. It is hoped that these innovations in some small way help in advancing the overall field of AGV.

Acknowledgements

Special thanks to following Team CajunBot members for their contribution: Adam Lewis, Adrian Aucoin, Christopher Mire, Daro Eghagha, Joshua Brideveaux, Muralidar Chakravarthi, Patrick Landry, Santhosh Padmanabhan, Scott Wilson, Vidhyalakshmi Venkitakrishnan.

The project was supported in part by the Louisiana Governor's Information Technology Initiative, and benefited from the sponsorship of C&C Technologies, MedExpress Ambulance Service, Lafayette Motors, Firefly Digital, Oxford Technical Solutions, and SICK USA, Inc.

References

Barraquand, J., Langlois, B., and Latombe, J. C. (1992). Numerical potential field techniques for robot path plannning. *IEEE Transactions on Systems, Man, and Cybernetics*, 22:224–241.

de Smith, M. J. (2004). Distance tranforms as a new tool in spatial analysis, urban planning, and gis. *Environment and Planning B: Planning and Design*, 31:85–104.

Dubins, L. E. (1957). On curves of minimum length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79:497–516.

Feng, D., Singh, S., and Krogh, B. H. (1990). Implementation of dynamic obstacle avoid-

ance on the CMU NavLab. In *Proceedings of the 1990 IEEE Conference on Systems Engineering*, pages 208–211.

Gerkey, B., Vaughan, R. T., and Howard, A. (2003). The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics (ICAR'03)*, pages 317–323, Coimbra, Portugal.

Hebert, M., Thorpe, C., and Stentz, A., editors (1997). *Intelligent Unmanned Ground Vehicles: Autonomous Navigation Research at Carnegie Mellon University*. Kluwer Academic Publishers.

LaValle, S. M. (to appear in 2006). *Planning Algorithms*, chapter 1, page 3. Cambridge University Press. `http://msl.cs.uiuc.edu/planning/`.

Maida, A. S., Golconda, S., Mejia, P., and Lakhotia, A. (2006). Subgoal-based local-navigation and obstacle-avoidance using a grid-distance field. *International Journal of Vehicle Autonomous Systems*.

Marshall, A. D. (1999). *Programming in C, Unix System Calls and Subroutines using C*, chapter IPC: Shared Memory. `http://www.cs.cf.ac.uk/Dave/C/node27.html`.

Pardo-Castellote, S. S. G. and Hamilton, M. (1999). NDDS: The real-time publish-subscribe middleware. Technical report, Real-Time Innovations, Inc.

Scheuer, A. and Laugier, C. (1999). Planning sub-optimal and continuous-curvature paths for car-like robots. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, volume 1, pages 25–31, Victoria, Canada.

Scheuer, A. and Xie, M. (1999). Continuous-curvature trajectory planning for manoeuvrable non-holomorphic robots. In *Proceeding of IEEE-RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 1675–1680.

Shackleford, W. P., Proctor, F. M., and Michaloski, J. L. (2000). The neutral message language: A model and method for message passing in heterogeneous environments. In *Proceedings of the World Automation Conference*, Maui, Hawaii. `http://www.isd.mel.nist.gov/documents/shackleford/Neutral_Message_Langu%age.pdf`.

Simmons, R. and James, D. (2001). *IPC – A Reference Manual, version 3.6*. Robotics Institute, Carnegie Mellon University. `http://www.cs.cmu.edu/afs/cs/project/TCA/ftp/IPC_Manual.pdf`.

Stentz, A. and Hebert, M. (1995). A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots*, 2(2):127–145.

Vaughan, R. T. (2000). Gazebo: A multiple robot simulator. Technical Report IRIS-00-394, Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California.