© 2010 Sundar Subbiah
To my parents, who were my foremost teachers, without whose unwavering support and encouragement, this would not have been possible.
ACKNOWLEDGMENTS

I sincerely thank Dr. Thomas F. Burks for being a great advisor. His academic and moral guidance over the course of three years has helped me shape my life and career in innumerable ways. My special thanks for his patient advice and encouragement through the course of my research.

I would like to thank Gregory Pugh and Mike Zingaro, for helping me out on all sorts of technical problems that arose during research. I also thank them for offering the pleasure of their company in the lab.

I thank Dr. Duke Bulanon and fellow graduate students — Venkat, Anirudh, Vijay and Ian— for coming up with helpful ideas and suggestions during the course of my research.

I would like to acknowledge the Agricultural and Biological Engineering department and staff members for their help and support. Finally, special thanks to all the free-software tools, (especially Qt, \LaTeX, Emacs and Inkscape) that proved immensely useful for research and thesis writing.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>8</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>9</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>11</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>13</td>
</tr>
<tr>
<td>1.1 Citrus Harvesting</td>
<td>13</td>
</tr>
<tr>
<td>1.2 Mechanical Harvesting</td>
<td>14</td>
</tr>
<tr>
<td>1.3 Autonomous Guidance</td>
<td>15</td>
</tr>
<tr>
<td>1.4 Software Systems</td>
<td>16</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>18</td>
</tr>
<tr>
<td>2.1 Scope of Previous Work</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Hardware Configuration</td>
<td>18</td>
</tr>
<tr>
<td>2.2.1 EGator Utility Vehicle</td>
<td>19</td>
</tr>
<tr>
<td>2.2.2 LADAR</td>
<td>20</td>
</tr>
<tr>
<td>2.2.3 Camera</td>
<td>20</td>
</tr>
<tr>
<td>2.2.4 Orientation and Localization Sensors</td>
<td>21</td>
</tr>
<tr>
<td>2.2.5 Additional Computational Units</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Software Configuration</td>
<td>22</td>
</tr>
<tr>
<td>2.3.1 Programming Environment</td>
<td>22</td>
</tr>
<tr>
<td>2.3.2 Software Architecture</td>
<td>23</td>
</tr>
<tr>
<td>2.3.3 Algorithms</td>
<td>24</td>
</tr>
<tr>
<td>2.3.3.1 Machine vision and LADAR based autonomous guidance</td>
<td>24</td>
</tr>
<tr>
<td>2.3.3.2 Headland turning</td>
<td>25</td>
</tr>
<tr>
<td>3 LITERATURE REVIEW</td>
<td>27</td>
</tr>
<tr>
<td>3.1 Agricultural Autonomous Guidance</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1 Contact Methods</td>
<td>27</td>
</tr>
<tr>
<td>3.1.2 Non-Contact Methods</td>
<td>28</td>
</tr>
<tr>
<td>3.1.2.1 Ultrasonic methods</td>
<td>28</td>
</tr>
<tr>
<td>3.1.2.2 Laser based triangulation</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2.3 Dead reckoning</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2.4 GPS based methods</td>
<td>31</td>
</tr>
<tr>
<td>3.1.2.5 Vision based methods</td>
<td>31</td>
</tr>
<tr>
<td>3.1.3 Headland Turning</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Software Architecture</td>
<td>35</td>
</tr>
</tbody>
</table>
## 4 SOFTWARE ARCHITECTURE

### 4.1 Software in Agriculture

### 4.2 Field Automation Software

#### 4.2.1 Requirements

#### 4.2.2 Design

#### 4.2.3 Design Choices

### 4.3 The DashBoard System

#### 4.3.1 The Device

#### 4.3.2 The Controller

#### 4.3.3 The Dashboard

### 4.4 Implementation

#### 4.4.1 GUI

#### 4.4.2 Other Features

### 4.5 Discussion

#### 4.5.1 Advantages

#### 4.5.2 Limitations

## 5 ODOMETRY

### 5.1 Methods

### 5.2 Visual Odometry

#### 5.2.1 Technique Overview

#### 5.2.2 Coordinate Systems

##### 5.2.2.1 Camera model

##### 5.2.2.2 Global coordinates

#### 5.2.3 Feature Point Tracking

##### 5.2.3.1 Feature detection

##### 5.2.3.2 Tracking

##### 5.2.3.3 Displacement estimation

##### 5.2.3.4 Orientation estimation

##### 5.2.3.5 Vision system calibration

#### 5.2.4 Experiments and Discussion

#### 5.2.5 Limitations

### 5.3 Kalman Filtering

#### 5.3.1 Kalman Filter Theory

#### 5.3.2 Enhanced Visual Odometry

##### 5.3.2.1 System model

##### 5.3.2.2 Determination of constants

##### 5.3.2.3 Kalman filter design

### 5.4 Experiments

#### 5.4.1 Distance Measurement

#### 5.4.2 Heading Measurement
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Hardware Specifications</td>
<td>22</td>
</tr>
<tr>
<td>5-1</td>
<td>Effect of feature count on accuracy and computational efficiency</td>
<td>70</td>
</tr>
<tr>
<td>5-2</td>
<td>Effect of terrain on accuracy: Error in the distance traveled</td>
<td>71</td>
</tr>
<tr>
<td>5-3</td>
<td>Distance measurement performance</td>
<td>79</td>
</tr>
<tr>
<td>5-4</td>
<td>Distance measurement performance on featureless surface (Road)</td>
<td>79</td>
</tr>
<tr>
<td>5-5</td>
<td>Heading measurement performance</td>
<td>80</td>
</tr>
<tr>
<td>6-1</td>
<td>Straight line tracking performance</td>
<td>88</td>
</tr>
<tr>
<td>6-2</td>
<td>Loop turn tracking performance</td>
<td>89</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Citrus Grove Illustration</td>
<td>18</td>
</tr>
<tr>
<td>2-2</td>
<td>Schematic showing system configuration</td>
<td>19</td>
</tr>
<tr>
<td>2-3</td>
<td>Software architecture described in [46]</td>
<td>24</td>
</tr>
<tr>
<td>4-1</td>
<td>A hierarchical control architecture</td>
<td>40</td>
</tr>
<tr>
<td>4-2</td>
<td>UML Representation of the Dashboard</td>
<td>45</td>
</tr>
<tr>
<td>4-3</td>
<td>Dashboard Call Sequence</td>
<td>48</td>
</tr>
<tr>
<td>4-4</td>
<td>Dashboard Implementation Schematic</td>
<td>49</td>
</tr>
<tr>
<td>4-5</td>
<td>Device selection</td>
<td>50</td>
</tr>
<tr>
<td>4-6</td>
<td>Main Window</td>
<td>50</td>
</tr>
<tr>
<td>5-1</td>
<td>Visual Odometry in [59]</td>
<td>60</td>
</tr>
<tr>
<td>5-2</td>
<td>Visual odometry overview</td>
<td>61</td>
</tr>
<tr>
<td>5-3</td>
<td>Co-ordinate systems</td>
<td>62</td>
</tr>
<tr>
<td>5-4</td>
<td>Visual odometry— control flow</td>
<td>64</td>
</tr>
<tr>
<td>5-5</td>
<td>Feature detection ROI and features in a typical FOV</td>
<td>65</td>
</tr>
<tr>
<td>5-6</td>
<td>Ground plane motion vector</td>
<td>67</td>
</tr>
<tr>
<td>5-7</td>
<td>Sample images of road and turf used</td>
<td>70</td>
</tr>
<tr>
<td>5-8</td>
<td>Illustration of $K_s$ determination experiment</td>
<td>76</td>
</tr>
<tr>
<td>5-9</td>
<td>Kalman filtered visual odometry— control flow</td>
<td>77</td>
</tr>
<tr>
<td>6-1</td>
<td>Software Implementation</td>
<td>82</td>
</tr>
<tr>
<td>6-2</td>
<td>Alley-way Navigation — Control Flow</td>
<td>83</td>
</tr>
<tr>
<td>6-3</td>
<td>End-of-row Turning illustration</td>
<td>84</td>
</tr>
<tr>
<td>6-4</td>
<td>Point-to-Point driving controller block diagram</td>
<td>85</td>
</tr>
<tr>
<td>6-5</td>
<td>Reaching via-points with the Point-to-Point driving controller</td>
<td>86</td>
</tr>
<tr>
<td>6-6</td>
<td>Path tracking accuracy for a straight line path</td>
<td>88</td>
</tr>
<tr>
<td>6-7</td>
<td>Path tracking accuracy for a loop turn</td>
<td>89</td>
</tr>
</tbody>
</table>
The main focus of this thesis is the construction of robust agricultural guidance systems. This work builds up on previous research on building agricultural guidance in citrus groves and tries to improve on its robustness. The robustness of the system was improved on from two different fronts - control software architecture and headland turning maneuvers.

The prime limiting factors in the existing control software were studied and compared with software approaches taken in other contemporary non-agricultural systems. As a result, a robust software framework called the DashBoard has been arrived upon. This new software framework, improves robustness by laying a foundation that adequately isolates software components and channelizes the communication between them. This isolation is key to abstracting the complexity of software and ensuring that the assumptions of one component do not adversely affect the others. Thus, the resulting control applications are robust and predictable. Apart from robustness; the complexity, scalability and re-usability characteristics of the framework are analyzed.

Headland turning behavior of the existing system, under practical grove conditions was studied and uncertainties hampering robust turning were identified. A new method, which takes a closed-loop approach toward turning, using Kalman filtered visual odometry to localize the vehicle in a 2-D space, has been developed. This eliminates
the need for a GPS, and hence allows the system to be robust under poor GPS signal conditions. A simple guidance algorithm, for maneuvering in the headland, based on via-points has been developed. This could facilitate the system deal with issues like insufficient turning space, rough terrain and irregularities in the tree row. The approach was tested under typical field conditions and the results are presented.
CHAPTER 1
INTRODUCTION

Citrus fruits are being produced and consumed in countries around the world. Presently, 80% of world’s citrus production is geographically concentrated in Florida, United States and São Paulo, Brazil [44]. Thanks to global trade liberalization and improvements in citrus transportation and packaging technologies; citrus consumption and thus production has been experiencing strong growth since the 1980s. While consumption is on the rise, citrus production facilities around the world compete with each other in keeping up quality and bringing down production cost. The Florida citrus industry is the largest in the United States and is a major economic force in the state.

1.1 Citrus Harvesting

Rising land values, climatic variations, diseases and labor availability are the major challenges faced by the Florida citrus industry in recent years. Though labor availability does not contribute to production losses as much as diseases and the recent freezes, it tends to place the industry in a competitive disadvantage with other producers like Brazil. Citrus harvesting has been historically challenging both in terms of cost and labor availability. For the 2003-04 season, harvesting expenditure accounted for an average of 41% of the total production cost in farms across Florida [44]. Harvesting is comprised of three activities - Picking, Road-siding and Hauling. Picking is the removal of fruit from the tree, road-siding is the transport of fruit from the trees to the truck near-by and hauling is the mass transport of fruit from the farms to the packer or processor. The first two operations are the most labor intensive and they represent the major cost component. Fruit picking and road-siding alone, costs to about $1.60 per box of harvested fruit in Florida against $0.384 in São Paulo, Brazil [27]. Adding to the labor costs is the fact that this labor requirement is seasonal. The industry typically employs around 5000 workers in year round farm maintenance and for the harvesting season, up
to 20,000 more workers are needed. Finding temporary labor at such huge volumes is difficult and tends to become expensive.

Novel harvesting practices, that either reduce the labor requirement or improve harvesting productivity, can be expected to have a significant impact on the production cost and thus help counter market forces. The $9.3 billion\(^1\) Florida citrus industry is dominated by technology savvy, agglomerated corporations equipped for mass production. Over the past decade, the industry also saw growth of third party service providers who, for example, provide only harvesting services to the farm owners. Given the scale and global impact of this industry, it is reasonable to expect that the industry will welcome integration of today’s cutting edge technologies into harvesting. Over the years, numerous researchers have ventured into improving farm yields, disease identification, use of abscission chemicals and mechanization of harvesting, with varying degrees of success.

### 1.2 Mechanical Harvesting

The Florida industry started showing interest, during the late 1960s because of labor shortage. Importing labor was restricted and labor demands from the growth of high profit, non-agricultural sectors choked labor supply [57]. Responding to calls for improvement in citrus harvesting practices, various mechanical harvesting technologies have been developed. Mechanical harvesting is effective because it helps reduce harvesting costs and at the same time help farm owners depend on fewer people in the harvesting season. Techniques like canopy and trunk shaking have been successfully adopted in the citrus farms of Florida. As a result, the use of mechanical harvesters has seen a steady increase since 1999. Mechanical harvesting has been proved to have saved an estimated 10 to 20% in harvesting costs [37] in the current state of the art.

---

\(^1\) 2003-04 estimates
Robotic citrus harvesting, as an emerging technology in this area holds promises for improved harvesting efficiency and reduced tree damage. Multidisciplinary research efforts in investigating robotic harvesting problems are in progress. Machine vision for fruit identification and disease detection, advanced robotic manipulation techniques for harvesting and autonomous vehicle guidance are among the chief research avenues [8].

One difference between design of industrial and agricultural machinery is that, agricultural machinery should be designed to work with nature; and that introduces a wide range of uncertainties. In the case of citrus harvesting, harvesting machinery is required to be able handle various fruit varieties and work in groves with varying dimensions. Different citrus fruit varieties have major differences in terms of color, size and weight. These differences, for instance, are the main challenges in design of automated vision based fruit identification systems for harvesting. Likewise, grove dimensions and tree heights of existing groves dictate the size and behavior of designed machinery. While industrial material handling robots enjoy at least a semi-structured working environment, agricultural robots must find their way through ill-structured tree branches.

1.3 Autonomous Guidance

As various harvesting technologies have been in development over the past 30 years, the idea that guiding the harvesting vehicle can be automated has started to emerge [51]. Autonomous guidance can be viewed as completely replacing the human operator, or assisting an operator in driving. The advantage of having such a system is evident - the operator can now concentrate on other tasks while the vehicle moves. Given the present state of technology, autonomous driving capabilities have not found enough ground in road transport, primarily because of safety concerns. In case of agricultural applications, the challenges are entirely different. For applications like spraying, planting, harvesting, etc., the vehicle has to move at a very slow and constant pace. Sometimes the vehicles are big and require more effort for a driver to operate.
With long hours of slow and constant driving, it becomes a menial task, which the human operator would like to avoid. This makes guidance an attractive candidate for automation.

The guidance problem can be phrased as the continuous positioning of a moving vehicle or implement, through a desired path, while maintaining a safe relationship with surrounding objects. There exists a desired state, an actual state, and an error at any point of time. Autonomous guidance can thus be viewed as a tracking problem, with position as the controlled parameter. For closed loop control, we would require the automatic control system to know the precise state of the vehicle in real-time. Assuming a two-dimensional world, the state can be minimally expressed with cartesian co-ordinates and the heading angle. Now the controller should be able to determine the action necessary to minimize the error in position and orientation. A trivial control strategy would be to come up with the error and use it to drive the actuator, but for purposes of optimal steering, the system might have to consider the path lying ahead in making control decisions. Also, decisions like when to start turns, moving to the next row and how to turn, complicate the control.

In addition to the challenges discussed in the previous sections, absence of road-like markers, uneven and marshy terrain with sporadic GPS reception can pose significant difficulties. Given the complexity of control involved and the uncertainties faced, its not hard to fathom that the solution would require a significant amount of software. Software for such control systems is inherently concurrent and multi-threaded in nature; thus requiring a significant amount of attention.

1.4 Software Systems

Complex autonomous guidance systems, will require multiple control loops which are robust to noise and appropriately responsive. In software terms this translates to an application that is multi-threaded and real-time in nature. When multiple concurrent threads of execution are used, it is often required to synchronize and ensure consistency.
of the shared data. Also, quite often the need for using multiple computing units for solving a problem, arises. This could be due to computational requirements or due to data availability to individual computers. In such cases, it is beneficial to have a distributed software system that is capable of using the network of data exchange. Unfortunately, software involving such mechanisms becomes fragile and vulnerable to uncertainties. While construction of 100% reliable software remains unrealizable in today’s systems, engineering practices can help boost the overall reliability of the system considerably. Construction of such a system for autonomous guidance is one of the themes of this research.

The main objective of this research effort is improving the robustness of an existing guidance system developed in [46]. In looking for ways to enhance the robustness of the system, it was identified that a flexible software foundation is necessary. This foundation was required to enable the control system designer build robust systems on top of it. The requirements of such a foundation were initially identified as,

1. Support concurrently running software components
2. Enable interfaces for communication between components
3. Allow distribution of data across a network

With such a foundation in place, the software can continue to grow in a controlled framework, thus making the overall system more reliable.

Another main objective of this research is to enhance the turning performance of the existing guidance system. An earlier approach for the autonomous headland turning maneuver by [49] was successfully implemented on a test vehicle platform under certain grove conditions. This approach was reimplemented to produce robust results under varying grove conditions. This new approach enjoys robustness mainly due to the software framework. The framework enabled development of multiple, concurrent, controlling agents that compete to control the turn behavior. Each agent is unique in terms of accuracy and relevance. A fusion algorithm then fuses the control efforts of these agents, to produce the most relevant control effort for the given context.
CHAPTER 2
BACKGROUND

Since this work is built up on a previous work, establishment of boundary between the current and previous research becomes necessary. In this chapter, a brief account on the work of [46] discussing the overall scope, hardware and software configurations and the results obtained, are presented in the following sections.

2.1 Scope of Previous Work

The main objective of [46] was to build a basic autonomous guidance system for citrus grove applications. The development started on a regular John Deere tractor, after an electro-hydraulic retrofit for steering control. The steering was controlled by an on-board computer. The tractor was found to follow alley ways with an error of less than 1% at a maximum speed of 3.1 m/s.

Figure 2-1. Citrus Grove Illustration :Alleyways, headland and turn procedures

2.2 Hardware Configuration

For further development, an electrically-powered vehicle platform has been used. The vehicle was fitted with the required sensors and computational units. With this platform set, the development focused mainly on two aspects of guidance - Alley way navigation and headland turns. Alley-way is the space in between tree rows, and is primarily used for maintaining the trees themselves. Headland is a wider strip of land at the end of tree rows. The headland is used for hauling and movement of other heavy equipment. The dimensions of alley ways and headlands differ in different groves. The
vehicle was programmed to move through relatively straight alley ways and at the end of alley, turn into the next alley. U-turns and switch-back turns were programmed into the turning algorithm.

The hardware components that constitute the vehicle platform are briefly described in the following sections. In each of the sections, the working principle, basic specifications and the utility of the component is explained. A table of selected specifications relevant to this research are presented at the end.

2.2.1 **EGator Utility Vehicle**

The *EGator* is an electrically powered agricultural utility vehicle, capable of carrying light loads. It is powered by eight 6 V batteries connected in series with a 48 V DC motor. The motor is controlled by an embedded motor controller [21], which gets position feedback through an encoder attached to the motor shaft. The motor drives the rear wheels through a drive train. The front wheels free roll and they are used for steering. An electronic steering controller controls the front wheel angle through a servo mechanism. The steering controller allows for electronic setting of wheel angle by means of a CAN interface. For autonomous control of steering, an on-board, PC/104 [12] compliant industrial grade, general purpose embedded computer is used. The computer commands the steering controller via the CAN and is also interfaced to the
traction motor controller. It is fully programmable and thus allows complete control of
the vehicle’s motion through software. However, it is a uni-processor machine and has
limited computational power. A powerful wireless LAN access point on the vehicle allows
networking the on-board computer with other stationary systems. The vehicle dashboard
controls allow switching between manual or autonomous modes. In manual mode, the
vehicle allows driving with a steering wheel and pedal.

2.2.2 LADAR

A LADAR or Laser RADAR is an optical distance measurement device. The
standard SICK LMS 200 type LADAR used in this work has a maximum scan width
of 180 degrees and maximum range of 80 meters. A laser emitter-receiver assembly
shoots short pulses of light to the object in front. The reflected pulse is then received
and the time of flight is used to estimate the distance of the object. A rotating mirror
housed inside the device directs the pulse to different directions from 0 to 180 degrees
in increments of 0.5 degrees. Thus the device collects distances at 361 different
directions, producing a two dimensional map. An array of 16 bit distance measures,
representing the map is available through a serial interface. The LADAR was mounted
on a swivel mount and attached to the front of the vehicle. This mount allows the device
to pitch up and down by ±30 deg from the ground plane. Since, every scan from the
device can yield a two dimensional map of objects lying around, it is possible to obtain
a three-dimensional map of the surrounding by swiveling. A standard RS-232 serial
interface can transfer a maximum of only up to seven frames per second. Consequently,
a special high-speed serial interface - RS-422, with a baud rate of 500 Kilobaud, is used
to communicate at frame rates of around 35 Hz.

2.2.3 Camera

A Sony industrial machine vision camera, enclosed in metal casing was mounted to
the front, to get a full view of the path ahead. Unlike the LADAR, the camera’s angle is
fixed. Standard 640 × 480 NTSC images with 24-bit pixels are available from the camera
through a special frame-grabber at a maximum frame rate of 38 Hz. The frame grabber is a PCI card with communication hardware to assist transfer of video data from the camera to the system’s physical memory, by direct memory access (DMA). Thus images could be acquired in real-time without slowing down the CPU.

2.2.4 Orientation and Localization Sensors

To measure the vehicle orientation, a Microstrain 3DM orientation sensor, capable of tri-axial inclination and acceleration measurement is used. The digital orientation sensor measures the absolute orientation of the vehicle in three axes, with respect to the earth’s magnetic axis. It is also capable of measuring rotations in terms of rotation matrices. The sensor has its own embedded micro-controller and can do basic filtering. Communication with the sensor is enabled through a regular RS-232 serial interface.

A John Deere Differential GPS (DGPS) receiver is available to get the absolute position of the vehicle in three dimensions, when necessary. However, this is only used as a backup device during navigation. A DGPS is precise up to a meter distance in two co-ordinates. A GPS works by receiving time signals from 4 different satellites and triangulating the position of the receiver. A DGPS in addition to triangulating, reduces the error in position by comparing its position with a terrestrial reference station. The DGPS can measure latitude, longitude, altitude and average velocity, provided that there are at least 4 satellites in range.

2.2.5 Additional Computational Units

The on-board computer was found to be inadequate for handling sensory inputs from all the above sensors. Thus an additional, general purpose, industrial form-factor computer was added to the system. All the sensors were connected to this computer, as it was used for making decisions on heading. [46] calls this the “Shoebox”, due to the small form-factor. Serial communication was set up between this computer and the on-board computer for transmission of heading values.
### Table 2-1. Hardware Specifications

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Camera</strong></td>
<td>Max. Framerate</td>
<td>24 Hz</td>
</tr>
<tr>
<td></td>
<td>Resolution</td>
<td>400x300 pix</td>
</tr>
<tr>
<td></td>
<td>Output</td>
<td>RGB</td>
</tr>
<tr>
<td><strong>LADAR</strong></td>
<td>Max. Framerate</td>
<td>17 Hz</td>
</tr>
<tr>
<td></td>
<td>Resolution</td>
<td>0.5°</td>
</tr>
<tr>
<td></td>
<td>Scanning Range</td>
<td>180°</td>
</tr>
<tr>
<td></td>
<td>Range</td>
<td>80 m</td>
</tr>
<tr>
<td><strong>IMU</strong></td>
<td>Angular Resolution</td>
<td>0.25°</td>
</tr>
<tr>
<td></td>
<td>Pitch Range</td>
<td>180°</td>
</tr>
<tr>
<td></td>
<td>Yaw Range</td>
<td>360°</td>
</tr>
<tr>
<td></td>
<td>Roll Range</td>
<td>360°</td>
</tr>
<tr>
<td></td>
<td>Update Frequency</td>
<td>100 Hz</td>
</tr>
<tr>
<td><strong>GPS</strong></td>
<td>Resolution</td>
<td>1 m</td>
</tr>
<tr>
<td></td>
<td>Update Frequency</td>
<td>5 Hz</td>
</tr>
<tr>
<td><strong>Wheel Encoder</strong></td>
<td>Resolution</td>
<td>3.5 clicks/mm</td>
</tr>
<tr>
<td></td>
<td>Update Frequency</td>
<td>40 Hz</td>
</tr>
<tr>
<td><strong>Steering Controller</strong></td>
<td>Angular Resolution</td>
<td>$\frac{1}{100}$°</td>
</tr>
<tr>
<td></td>
<td>Range</td>
<td>±30°</td>
</tr>
<tr>
<td></td>
<td>Update Frequency</td>
<td>40 Hz</td>
</tr>
<tr>
<td></td>
<td>Interface</td>
<td>CAN</td>
</tr>
<tr>
<td><strong>Speed Controller</strong></td>
<td>Resolution</td>
<td>1 mm/s</td>
</tr>
<tr>
<td></td>
<td>Range</td>
<td>0 – 2000 mm/s</td>
</tr>
<tr>
<td></td>
<td>Update Frequency</td>
<td>40 Hz</td>
</tr>
<tr>
<td></td>
<td>Interface</td>
<td>CAN</td>
</tr>
</tbody>
</table>

### 2.3 Software Configuration

#### 2.3.1 Programming Environment

The system consists of two computing units - the PC-104 and the Shoebox, both general purpose and programmable. The PC-104 was embedded into the vehicle platform and directly communicates with actuators that control the vehicle’s speed and direction. This embedded computer was pre-installed with the Linux operating system (SUSE Linux 10). For programming, the C++ programming language was used, for
reasons of performance and ease of programming. The standard free distribution of the GNU C++ Compiler was used.

On the other hand, the Shoebox ran on the Windows XP operating system. Programming was done with the C++ programming language but, using the Microsoft Visual Studio 2003 environment and the Microsoft C++ compiler. This configuration allows for programming GUI applications for Microsoft Windows operating systems. It should be noted that the above configuration does not guarantee real-time execution. Since the system is constantly under development, the need for deterministic execution was less pronounced than the need for ease of programming.

2.3.2 Software Architecture

The problem being solved, i.e., autonomous guidance has been broken into two parts. They are,

- Driving the vehicle actuators at a set speed and direction
- Processing sensory input to calculate heading

The first part of the problem has been solved in the PC-104, by means of a loop that continuously commands the vehicle hardware in response to speed and heading commands received in one of its serial ports. The devices in this case are speed controller, steering controller and serial input from the Shoebox. These devices are abstractly represented by a C++ class, which contains serial communication primitives. The main loop makes use of these classes to read and write to the respective devices. The heading input received from the Shoebox is converted to the actuator’s acceptable range and is filtered with a moving average filter.

The Shoebox runs the second part of the computation. It collects data from the sensors - Camera, LADAR, GPS and Orientation sensor, and determines the heading angle. The exact algorithms used are described in the next section. Since all sensors (except the camera) communicate through a serial interface, a generic serial device class encapsulates all the serial port functionality. The algorithms use device classes
to read and write data to ports. Unlike the PC-104, here the system interacts with a user; hence a GUI that shows the sensor readings and allows user control of devices has been created. A guidance thread continuously reads the sensors and produces a heading angle. A headland detection thread continuously monitors the environment for approaching headland and triggers a turn sequence when the headland is reached.

Figure 2-3. Software architecture described in [46]

2.3.3 Algorithms

As discussed previously, the main objectives of the system are guidance in alleyways and headland turning. Alleyway guidance is the continuous positioning of the vehicle in between nearly-straight rows of trees. The width of alleyways can vary from grove to grove. Headland turning is a special guidance requirement at the end of row; the vehicle has to be guided into the next alleyway by means of a 180 deg turn. The algorithms that accomplish these two objectives are briefly explained in this section.

2.3.3.1 Machine vision and LADAR based autonomous guidance

Machine vision in an alleyway works by differentiating between tree rows and path by means of color segmentation. Under normal daylight conditions, the tree row, the horizon and the ground have distinct colors. Thus, using a fixed camera, the tree rows on either side can be identified and approximate tree-row-lines can be fitted on the image. In [47], a median line in between the left and right tree-lines was calculated.
Using this line as a reference, a PID controller was used for guiding the vehicle in between the tree-lines.

When using the LADAR, the same basic principle as the vision is used. The LADAR was tilted to scan a fixed look-ahead point in the alley way. The LADAR recorded the trees on either sides, and a median point in between the obstacles was obtained. This median was used as a reference to guide the vehicle. It was concluded that the LADAR based guidance was more accurate than vision based guidance for both straight and curved paths. However, both the systems were found to be accurate enough to be compared with human driving.

Since LADAR guidance is strictly based on a single look-ahead point, missing tree rows and uneven tree shapes posed significant difficulty in following straight lines. On the other hand, the vision was less accurate and had a chance of being rendered useless by obstructing branches in the field of view. In [48], a novel sensor fusion based approach that combines the advantages of both LADAR and vision was developed. This approach used a Fuzzy Logic Enhanced Kalman Filter to fuse the sensor readings and model a robust sensor.

2.3.3.2 Headland turning

To negotiate a headland turn, the vehicle is expected to make a continuous 180 degree turn, once the end of row is reached. In [49], the turning problem has been broken into two sub-problems - headland detection and turning. The detection of end of row is done by constantly monitoring the visual field of view for receding tree rows. The LADAR alone is not sufficient because it was susceptible to false positives rendered by missing trees in the alley way. This is due to the fact that the LADAR guidance is based on a single look ahead point, whereas the visual guidance take its entire field of view into consideration. However, to make sure that the vehicle initiates the turn precisely at the end of row, the LADAR was used only toward the end of row. Also, to get a better view of the headland, the LADAR was swept to an angle of 15 deg above horizontal.
and 30 deg below horizontal. This helped establish the end of row precisely using the
three-dimensional view obtained.

To start turning, the IMU is constantly monitored till the vehicle’s orientation changes
to 150 deg. The steering was maintained at a fully turned position in the direction of
interest. After the required orientation, the vehicle switches back to alley way mode to
enter into the next alley way. To be able to get as much space as possible for turning, a
slight turn in the opposite direction was required before the U-turn. While executing the
turn, the LADAR constantly monitors the field for any obstacles; in event of obstacles
in the turn path, the vehicle comes to a stop. The vehicle was able to make a loop-turn
and a three-point or switch-back turn in this fashion. In case of the switch-back turn, the
vehicle drove to the reverse for a fixed distance at an orientation of 90 deg, and then
proceeded to turn into the next alley way.
CHAPTER 3
LITERATURE REVIEW

A detailed account of developments in the field of autonomous guidance are presented in this chapter. The ability of an autonomous system to localize itself is key to navigation; however, for agricultural navigation, the main focus is often following a crop row or navigating in-between crop rows. Thus, a globally referenced localization system is unnecessary in most cases. Researchers have tried to use both GPS based and local feature based guidance systems. Localization methods and crop-row sensing methods adopted for guidance, through the years has been presented and a survey of how these methods have been used for in-row and headland guidance is presented. Following guidance, a detailed account on software architectures used for autonomous systems in general is presented. Some recent developments in adoption of software frameworks for agricultural automation has been included at the end.

3.1 Agricultural Autonomous Guidance

Automation has had a considerable impact on the agricultural industry in the past decades. The agricultural engineering community has been constantly absorbing newer technologies from other fields. Emergence of the latest autonomous agricultural guidance systems like the John Deere AutoTrac™ and Case-IH AccuGuide™ stand as evidence to this development. The idea of automatic guidance is fairly old. Evidence suggests an automatically guided tractor being patented as early as 1929 [20].

3.1.1 Contact Methods

All early efforts of guidance used some kind of contact method to sense the path. In one approach, a mechanical feeler slides over the crop row or ridge while the vehicle moves. The feeler input is used for controlling the hydraulic steering system. This guidance mechanism was usually available as a retrofit for existing vehicles. The disadvantage to this method being; misguidance due to local deformities in an otherwise straight row. Also the machine was found to have worked less effectively with curved
rows [36]. In another approach [40], an overhead track guided a small spraying vehicle. A cable connecting the overhead guide and the vehicle, acted as a feeler. The angle produced by the cable was sensed to estimate the lateral position of the vehicle with respect to the guide. The system was shown to have worked with a maximum error of 12 cm for speeds under 0.5 m/s.

The main disadvantage of contact methods is that, the mechanical nature of the sensing mechanism limits the speed of operation. In most cases, it requires crops to be planted in a compliant fashion or an elaborate rail-like external arrangements to guide.

3.1.2 Non-Contact Methods

Guidance methods that do not require mechanical contact of the machine with the crop, for the purpose of sensing alone, are classified under non-contact methods. Almost all of the today’s successful commercial methods fall into this category. One primitive non-contact method was using electrical inductance. An electrical cable buried underground, carrying an AC signal was continuously sensed by an on-board electro-magnetic coil. Since the strength of the electromagnetic signal depends on the position of the receiver, the vehicle could estimate the error in the path. This method is popular in industrial material handling applications, but might hinder agricultural operations that require digging the ground.

3.1.2.1 Ultrasonic methods

Ultrasonic sensors measure distance by bouncing an ultrasonic pulse on obstacles. The distance is estimated by time of flight measurements made on the pulse. Singh et al. [43] developed a six-wheel, differential steering, green house sprayer platform, for spraying potentially harmful chemicals in greenhouses. The vehicle platform used six ultrasonic sensors optimally placed to get maximum range information possible. The vehicle could successfully navigate the intended environment with a maximum speed of 1.8 \( m/s \) with an accuracy of up to 3 cm. [43] has noted that the technology may not
be entirely suited for highly unstructured environments, since the ultrasonic sensors required the target surface perpendicular to the ultrasonic beam, for the beam to return.

3.1.2.2 Laser based triangulation

Holmqvist [18] used a laser based triangulation method for determination of position and orientation. A rotating laser emitter was attached to the vehicle, on an elevated support. Fixed reflectors were placed around the field, such that the laser light emitted from the vehicle could be reflected back to it. From the distance and angle of reflected light, the position and orientation of the vehicle was determined. Using this method, it has been reported that an error of 5 cm in each of the axes and 1 milliradian along each of the orientation axes, was observed. Positioning the vehicle enabled precise guidance and turning within the field. However, this method required installation of reflectors within 300 m of the vehicle, and may be limited for crops that are tall enough to block the vehicle’s line of sight.

Several others including [51] used LADAR as a principal source of distance measurement. The LADAR has proved to be very useful for accurate determination of crop rows or obstacles. The absence of any other useful information other than distance, has driven researchers to use the LADAR in conjunction with other techniques like vision to produce more reliable guidance methods.

3.1.2.3 Dead reckoning

The method of determining current position by integration of velocity and heading over time is called dead reckoning. Dead reckoning has traditionally been used in navigation, especially in marine navigation due to the fact that the method works without using landmarks. It demands accurate measurement of velocity and heading because errors in measurements tend to accumulate. Dead reckoning has been used by many agricultural navigation systems in the past and present [36]. Several sensing mechanisms available for accurate measurement are discussed below.
**Rotary Encoders.** Rotary encoders are ubiquitous among autonomous navigation equipment or any form of position measurement. They are cheaper than other methods of measurement and are considerably reliable. However in case of agricultural applications, where wheel slippage, wet soil and uneven terrain are common, wheel encoders may not be accurate enough. Since position measurement is cumulative, small errors rapidly reduce position accuracy.

**Geomagnetic Direction Sensor (GDS).** GDS uses magnetometers for sensing the earth's magnetic field and thus establish the orientation of the sensor with respect to the earth. Benson et al. [3] evaluated the effectiveness of a flux-gate magnetometer based GDS for agricultural applications and found that GDS could guide through straight lines with average errors of less than 1 cm. The magnetometer was found to be susceptible to disturbance from the vehicle’s magnetic interference and had to be mounted on an special aluminum mount, holding the sensor unit away from the vehicle’s body.

**Inertial Measurement.** An inertial measurement unit (IMU) usually consists of tri-axial linear accelerometer and a tri-axial angular rate sensor (gyro). A combination of measurements from these sensors is used to calculate the pitch, yaw and roll of the vehicle. Linear accelerations in the three axes could be used to estimate the velocity of the vehicle. Inertial measurement is widely used in the aviation industry for dead reckoning. [26] used a combination of GDS and gyroscopes to obtain mean position errors of 5 cm. The gyroscope measurements had to be improved using signal processing techniques for isolation of noise.

Akira et al. [1] developed a low-cost sensor module composed of three vibratory gyroscopes and two inclinometers for precise determination of heading angles. This sensor package was used in conjunction with a DGPS to provide autonomous guidance. The system tested against a ground truth IMU sensor produced r.m.s error of 1.59 deg in the heading angle. It was established that the developed unit was practically applicable for navigation purposes as much as an RTK GPS and precise IMU system.
3.1.2.4 GPS based methods

With the availability of GPS for civilian users by 1996, it has proved to be an accurate and cheap replacement for all navigational needs. O'Connor et al. [32] of Stanford University demonstrated the use of GPS on farm vehicles. In a closed-loop tracking control arrangement, using a GPS, microprocessor and hydraulic steering controller, the system tracked paths with maximum lateral error of 10 cm. This 1996 demonstration replaced the use of multiple heading and velocity measurement sensors with a single, cheap and reliable GPS. However the guidance through curved paths was not addressed in this paper. Also, some of the early GPS receivers had latencies up to 1 s. This presented limits on the speed at which vehicles could travel.

Various researchers [35] then found the RTK-GPS with high update rate to be accurate and fast enough for agricultural applications. Stombaugh et al. [45] used a 5 Hz RTK-GPS for guidance of a 2WD Case 7720 tractor for a speed of 4.5 m/s and a maximum position error of 16 cm. Noguchi et al. [30] developed an RTK-GPS and IMU guided robotic tractor system, with an r.m.s error of within 5 cm. The tests were conducted for diverse agricultural operations like spraying, tilling, planting etc., in human operated speeds.

GPS remains the preferred sensor among most research efforts in the agricultural navigation area [36]. However, GPS receivers could not be entirely reliable due to factors like multi-path effects and signal reception errors. This could pose significant problems in citrus groves where, trees might often obstruct GPS receivers [51].

3.1.2.5 Vision based methods

Vision based methods rely on one or more imaging devices on the vehicle to sense the environment. Machine vision capabilities can provide local information relating to crop rows or obstacles. In [14], development of a vision based guidance system that followed row crops is discussed. A battery powered golf cart was tested on corn fields at speeds of about 4 km/h and was found to follow crop rows with maximum
error of 5 cm. Subsequently, in [15], a J.I. Case-IH 7110 tractor was fitted with a color video camera and an Intel 80286 computer to process the images. The system segmented the crop row from the background based on user selected target color. The system was found to be as ‘skillful’ as a human operator up to speeds of 4.8 km/h. The accuracy of vision systems were typically affected by differences in ambient lighting, shadows and environmental effects like dust, fog, etc., Benson et al. [2] used artificial lighting in an attempt to avoid ambient lighting problems, but encountered problems with shadows. As computing became cheaper, it became possible to run complex image processing algorithms. Methods like hough transform, stereo image processing, principal component analysis and k-means clustering exist in contemporary systems, for robust classification of features [35]. [33] describes an unsupervised classifier for segmenting cut and uncut regions of alfalfa crop. The algorithm was used to guide a harvester to speeds of up to 4 mph. Takahashi et al. [52] used stereo vision to obtain 3D distances from the cameras to target objects. It was observed that stereo vision requires thin and sharp features for distinction in the left and right images. An error of −10% to 5% for objects within a range of 20 m was reported. The errors were found to be dependent on the width of a feature in pixels.

The main problem with vision based control is that, the algorithms involved are very complex and a number of input conditions affect the system’s control decision. Under uncontrolled outdoor conditions, it becomes impossible to build a vision system that makes the correct decisions for all possible inputs. The same is true with other crop-row sensing mechanisms like LADAR. Given that the agricultural environment is highly unstructured, decisions based only on a single sensing mechanism is not satisfactory [35]. With sensor fusion, it becomes possible to build a robust system with a set of unreliable sensors. Zhang et al. [61] tested a fusion system using two Extended Kalman Filters (EKF) to fuse the outputs of three sensors - RTK-GPS, GDS and machine vision. It was found that the fused system performed better than using the
RTK-GPS alone. Subramanian & Burks [48] used fuzzy enhanced kalman filter fusing data from vision, LADAR, IMU and encoder, to guide a tractor in curved tracks. Using vision-only guidance sometimes fails when the camera is accidentally obstructed by vegetation. On the other hand, the LADAR measurement fails when trees rows miss one or two trees. So, using continuous-valued numeric reliability functions for these sensors was not practical. A fuzzy logic supervisor was used to decide on the reliability factors of sensors in different contexts. It was found that with the fuzzy-EKF scheme the average error reduced to 1.9 cm from 2.5 cm for LADAR only guidance, for a speed of 3.1 m/s.

### 3.1.3 Headland Turning

There is a wealth of literature available on guidance along crop rows. This requires the guidance algorithm to recognize the crop row and guide the vehicle by following it. In case of GPS guided systems, local adjustments are made on globally planned paths to slightly alter an otherwise straight line path. For citrus groves, alley way guidance is achieved by detecting tree rows and guiding the vehicle in between them. Completely autonomous guidance would require the capability to make turns in the headland and enter the next alley way. Crop rows or any kind of markers are not available for guidance in the headland, and thus there is a need for the vehicle to deliberately plan a path. Several factors like space availability, vehicle limitations, obstructions, time taken to turn have to be taken into account when planning a turn.

Noguchi et al. [31] used a third order spline function to plan a curved trajectory and guided the vehicle along the path. RTK-GPS and Fiber optic gyroscope (FOG) were used to localize the vehicle during guidance. The vehicle was able to generate a turn in order to reach a specified point with an orientation of 180 deg from initial. The error measured between the commanded target point and the actual target point was measured to be a maximum of 5 cm. Also the algorithm was programmed to generate both u-turns and switch-back turns. [55] proposes a point-to-point navigation algorithm for navigation of turns rather than planning in terms of geometric primitives.
The proposed algorithm plans paths between any two given points, considering the location of obstacles. The generated path is then optimized by means of numerical optimization, subject to constraints of vehicle’s turning capabilities, time and space required. Smooth trajectories resulting in smooth velocity and steering profiles were generated and the algorithm was tested on an autonomous weeder. It was noted that the computing requirements of the algorithm was prohibitive for long distances, because the number of possible paths could be huge. [10] recommends modeling paths with geometric primitives (lines and circular arcs connected with clothoids \(^1\), results in curvature continuity and demand less computational cost compared to point-to-point trajectory generators like that suggested by [55]). A steering controller using a precise kinematic model extended with sliding parameters, is used to guide the vehicle through the generated path. Also, unlike other experiments that, are mostly based on a constant velocity assumption, a model-predictive velocity control is considered. Experimental results showed that the generated paths for a switch-back turn were tracked with an accuracy of ±5 cm.

It should also be noted that such accurate turns from all the presented work could be achieved partly because of accurate feedback from GPS used. Considering the absence of GPS, [50] proposes a simple strategy based on the pure-pursuit algorithm. A fixed target position is fed to the algorithm and the algorithm drives the vehicle by continuously trying to minimize the position error between current and target positions. U-turns and switch-back turns were made using dead-reckoning instead of GPS. The details of the turning behavior were discussed in the previous chapter.

---

\(^1\) A clothoid is a continuous-valued function that represents the transition curve between a straight line and a circular path. It is used in highway engineering for design of transition curves between straight and curved road segments.
3.2 Software Architecture

The need for software architectures for use in control systems was felt when early robot designers found control algorithms grow increasingly complex. The early control architectures were classified as robot control architectures and were usually dealt as a subject of Artificial Intelligence. Most early control frameworks were coupled with autonomy, and usually enforced a certain hierarchy or dependency between modules. With improvements in control systems, control frameworks started to be used for automatic control applications other than robotics. As a result, control architectures started to drop the implied autonomy and became more generic. Improvements in software technology like introduction of new languages and programming paradigms has also had significant impact on trends in control architectures. This section is devoted to the historical developments starting from early robot control architectures to the more recent generic control system architectures.

As control systems grew in complexity, the need for organized development of its software was more pronounced. It is the very requirements of an autonomous system that makes the software complex. Brooks [7] tried to establish the basic qualities of an ideal robot control system as,

- Able to handle multiple, conflicting goals
- Able to fuse data from multiple sensors
- Robustness and tolerance toward failures
- Extensibility

In his paper [7], Brooks suggests a robust control architecture called subsumption. The main aim of this architecture is to model a control framework that has robustness built-in. In case of subsumption, behaviors are defined as individual control loops, and range from primitive to advanced. Advanced behaviors subsume primitive behaviors, and thus layers of control loops are built on top of one another to model a complex behavior. Robustness is introduced in the system by the fact that, a partial failure in one of the control loops still rendered the system functional with primitive behaviors. In [42], the
task control architecture for a mobile robot is considered. The central characteristic of this work is task decomposition. A central controller, received tasks from the user or other sources and decomposed into a task tree with primitive tasks, and dispatched the tasks to the corresponding components. The components were in turn, software modules programmed to receive and execute tasks. The architecture allowed enforcing sequential task execution, resource sharing, exception handling and concurrent planning and execution.

Another architecture called ORCCAD developed in [4] simplifies the process of control software development by special programming language. Unlike the TCA, ORCCAD has only two layers of task decomposition; any task can be decomposed to primitive ROBOT-TASKs and individual ROBOT-TASKs are defined using ROBOT-PROCEDUREs. A ROBOT-TASK can be specified using blocks of ROBOT-PROCEDUREs in a GUI based editor, and individual ROBOT-PROCEDUREs have to be specified using the Maestro programming language. The ORCCAD also marks the dilution of autonomy and hierarchy being enforced in the control architecture.

CLARATy is the control architecture used by NASA developed for reuse of algorithms on multiple robotic platforms. In [28], the main features of the architecture are explained. CLARATy defines two layers of software; functional and decision layers. The functional layer contains functional abstractions of low-level vehicle functions like navigation, object avoidance, vision, etc., and the decision layer is responsible for the deliberative actions like planning, task decomposition, etc., The functional layer abstracts the entire underlying robot platform and all platform specific software is contained within the functional layer. When using the same planning algorithms in structurally distinct robots, only the functional layer changes. The decision layer can access the functional layer at various levels of granularity to get the plans executed. The object oriented nature of the functional layer allows reuse and portability among different platforms without requiring changes in the decision layer algorithms.
The Component Based Software Engineering (CBSE) paradigm has found acceptance, in the development of large software systems, over the past decade. CBSE is the compartmentalization of software modules by development of an application into separate binaries and enabling well-defined interfaces for communication between the component binaries. This is useful for development of large systems because, the isolation helps collaboration of experts without the assumptions of one affecting the other. Many control architectures embracing the CBSE approach have emerged through the years [22, 54]. In a typical CBSE based control architecture, individual sensors and control algorithms are wrapped into separate components. Each component is then allowed to expose only the required parameters and control functions. It can be seen that this class of architectures does not enforce any particular hierarchy; the control designer is free to arrange the components to suit the requirement. In addition to specifying components, the Joint Architecture for Unmanned Systems (JAUS) specification by the Department of Defense, specifies messages for communication between components. Thus systems using the JAUS framework must have the components deliberately message each other for exchange of information and control. In [62], a proposal for usage of JAUS for autonomous agricultural equipment has been presented, showing interest for adoption of software frameworks in the agricultural community. In [17], a C# based, modular, object-oriented system is presented. The framework was primarily developed for agricultural navigation and has a GUI pertaining to navigation.
CHAPTER 4
SOFTWARE ARCHITECTURE

Computers have been playing an important role in agricultural operations and management over the past decade. The use of computers has revolutionized the following domains of agricultural production - Data capture, data analysis, automatic control and decision support. In the following section, a brief account on the role of software in agricultural engineering is presented, in order to acquaint the reader with the very range of applications computers and software find here. In the subsequent sections, a detailed account on the software used in this research effort is presented.

4.1 Software in Agriculture

In situations where the data acquired from a sensor or a group of sensors may not be readily usable, a computer can act as an intermediate processor to extract useful information from the data. A range of data processing techniques ranging from simple filtering, to machine vision algorithms, are in use today. Also computation brings the ability to extract information out of noisy sensors. For instance, multi-sensor fusion algorithms fuse data from multiple noisy sensors to compute an accurate estimate of the measured quantity, thus avoiding the need for using a single expensive sensor. In agricultural applications where uncertainties are at large, such data capture techniques are enablers of automation. In addition, as cheap computational power is becoming commonplace; this helps reduce the overall cost of a system.

Novel computer networking technologies like wireless sensor networks are being widely researched in the agricultural community. The inherent abilities of a wireless sensor network, enables close monitoring of wide geographical areas. A set of wireless sensors in agricultural farmlands could collect a variety of information from, soil conditions, atmospheric conditions, pests etc. The main challenge here is power consumption in sensor nodes as wireless transmission itself is power consuming.
Special network management software is required to maintain the balance between power consumption and frequency of wireless transmissions [63].

The agricultural industry is also beginning to see the benefits of management information systems, through integrated farm management software products offered by major agricultural equipment companies like John Deere, Inc and Case New Holland. Farm management systems, with the help of Global Positioning System (GPS), can map various crop variables like soil condition, yield, disease spread, pests - both geographically and temporally. By being able to see this information at different levels, the farmer can make informed decisions on fertilizer and pesticide use to maximize efficiency. The core to such information systems is a central database management system, and a network of servers, that facilitates storage and retrieval of vast amounts of information.

The variety of crops and specialized expertise required in the cultivation of each crop variety has been perfect applications for expert systems. Experts systems are used for disease diagnosis and treatment of different crops. Expert systems can also make decisions on irrigation and fertilizer requirement over varying climatic conditions [19].

Field automation is another rich area where computers find niche applications. Automation of agricultural equipment reduces the need for man power as discussed in the previous chapters. The unstructured outdoor environment offered by an agricultural setting presents the main challenge in automation. Unlike factory automation, field automation demands more intelligence from the system. This chapter is dedicated to the description of characteristics, design choices and principles governing the construction of such software.
4.2 Field Automation Software

4.2.1 Requirements

The design requirements of automation software is primarily driven by certain common characteristics of automation problems. The main requirements for the given application are identified as follows,

- Simultaneity
- Real-time response
- Hierarchy
- Component communication

Automation problems are inherently synchronous with simultaneous events occurring in the physical world. They are often required to respond to events in a timely and reliable manner for extended periods of time. The primary requirement of this research effort was to build a software control system that will enable the vehicle to guide itself under a finite set of conditions. Guidance in itself is a compound problem and can be divided into subproblems like - obstacle avoidance, path-planning, path behavior etc., depending on the level of sophistication. For citrus grove guidance, the vehicle is required to have two basic behaviors - alleyway guidance and end-of-row turning. Assuming a basic guidance system, at any point, the vehicle is in any one of the two modes. In addition to carrying out behaviors, a higher level behavior arbitrator is required to switch behaviors. The arbitrator constantly monitors the environment and selects the most appropriate behavior. This apparently results in a hierarchical control architecture, where all levels of control are simultaneously active.

Figure 4-1. Hierarchical control Architecture
The vehicle has a suite of different sensors as discussed in chapter 2. The controllers should have simultaneous access to all the devices in the system. The data produced from the sensors vary from a single 32 bit integer to an RGB image. The devices cannot be expected to have any level of uniformity except for the fact that they produce some kind of data. The communication protocols involved in fetching data from the devices are unique to each device and hence would require separate driver programs for each sensor and actuator in the system. For instance, the camera communicates through a unique PCI interface, that facilitates DMA. Unlike the GPS that continuously produces data values in its output port, the camera interface card is an active device that writes the system's physical memory directly. So, a manufacturer supplied, custom driver package has to be used to interface the camera. Device drivers aside, the users of the devices - the controllers have to programmed to run in a continuous loop. The controllers might have to interact with each other or even control each other. As today's computers are inherently sequential machines, the simultaneous controllers would demand concurrency and synchronization from the programming environment. In addition, the user has to be able to monitor and control the behavior using a Graphical User Interface (GUI).

In addition to the above application specific requirements, software engineering requirements like scalability, re-usability, portability, adaptability, interoperability and ease-of-use also apply.

4.2.2 Design

Before attempting to realize the blocks in figure 4-1, it is necessary to have a software control framework that helps build this hierarchy of components - A framework that will provide a hosting environment for components to operate and exchange information. This is essential because it would be impractical and undesirable to include communication functions as a part of every individual component. A central lump of
functions that the components could use for maintenance and communication would be helpful. The requirements of a such generic framework are listed below,

- Supports multiple concurrent components
- Supports communication between components
- Limited run-time overhead
- Flexibility
- Extensibility
- Supports distribution of components
- Quick and easy to use

The framework must consider concurrency at all levels of control. Every sensor and actuator should consistently handle requests from multiple components, i.e., the behavior of a component should be exactly the same when running independently or with other components in the system. Concurrency can be an issue when components are not sufficiently isolated. This is especially true when multiple components work on a single piece of shared memory. For instance, multiple image processing routines reading from the same copy of an image, while the image acquisition component is in the process of refreshing the image, might get different images.

Components should be able to communicate all kinds of data, and should not be limited to just numerical values. Also, any component in the system should be accessible to all other components in the system. When all components are running in a single address space, communication can be enabled by means of shared memory. A producer component produces a value and multiple consumers read from the same memory location. Communication can become a challenge when distributing components across the network. Multiple copies of a single data value have to be made in order to serve requests of consumers. This is accomplished by means of passing data messages across the network. The two main communication paradigms in use in various components are, request-reply and publish-subscribe. In a request-reply type architecture, the consumer has to request the producer for data each time a value is required. This might require a lot of network bandwidth, but is very simple to implement
and manage failures. In a publish-subscribe type architecture, the consumers subscribe for data only once, or with a predetermined periodicity. The producers maintain a list of subscribers and keep sending update messages whenever a value changes. The JAUS architecture is designed to be a publish-subscribe type architecture [53].

The run-time overhead of the framework should not rise with the number of components. Frameworks might have to use specialized arbitration algorithms to route calls between components. For real-time applications, the run time of the framework has to be deterministic under all conditions. For use with safety critical distributed systems, the Common Object Request Broker Architecture (CORBA) has been extended with real-time guarantees in [25].

The framework should not limit the control system designed in using a preferred order or hierarchy. Using an uniform interface for all components will help arranging the components in arbitrary order. Limitations can surface when a particular component is given priority or access rights over the others. On the other hand limiting the roles of a component can also help in improving performance. For example, differentiating components on the basis of data flow direction can help manage concurrency.

An open framework that supports effortless addition of components will help build the system in small increments. Extensibility, can also mean inclusion of existing software into the new system with minimal effort. In [6], a Component Based Software Engineering (CBSE) approach towards building robots is described. The CBSE paradigm is widely used in large software systems, for its scalability. CBSE advocates building of large software systems from simple re-usable, pre-produced software components as in electronic circuits. Each component is a compiled binary by itself and the framework has no means to access the internal data structures of a component, except for the ones, the component exposes. Thus, CBSE allows for components to be immune from internal assumptions of other components and at the same time allowing safe collaboration between experts.
Distribution of components across the network is essential in today’s control systems, as problems often get too big for a single computer to handle. Distribution means, components are allowed to talk through a network and have to have additional code to accomplish this. However, if the programmer had to program the network interactions when programming each component, it would defeat the purpose of having a framework. Distributed frameworks avoid this by letting the framework abstract the network. So individual components are unaware of where the resources they access are located. This means, the programming semantics for requesting values from different components are the same regardless of the location of the component. This reduces complexity on the part of the component programmer as well as enables portability of components in the network.

Even though, the construction of frameworks is in the domain of software engineering, the users of such frameworks are typically mechanical or electrical engineers. Therefore, it is desirable to hide details of component communication, concurrency and distribution. Better abstractions lead to ease-of-use and scalability of the framework.

4.2.3 Design Choices

Many such control frameworks have been discussed in the previous chapter, starting from the early ones like subsumption [7], Task Control Architecture [42] to the latest frameworks like CORBA [25] and JAUS [53]. Some of the older control architectures like subsumption and TCA were built for autonomous control of robots and hence did not differentiate AI from control architecture. These designs typically enforced a certain hierarchy, rather than the software engineering aspects of building autonomous systems. Though it is desirable to use one of these frameworks, most frameworks are aimed at control systems that span across hundreds of nodes. The very scale and generality of some of these frameworks may be irrelevant for simple systems. With high-level, object oriented programming languages like C# and Java, building of small and medium sized systems from scratch have become relatively
easy. Many small system designers opt for custom frameworks rather than go through learning and implementing components of some standard framework \([13, 17]\). In our problem, the system was very simple with less than 10 components that run in two different computers. Thus, a custom framework based on object-oriented principles was chosen. The most popular language options available are C++, C# and Java. C++ was selected over Java and C# because of run-time, future real-time requirements, backward compatibility and portability.

4.3 The DashBoard System

To meet the demands of the agricultural guidance system, we derived an object oriented framework called the Dashboard. The Dashboard is an object capable of hosting a number of heterogeneous component objects. The primary responsibilities of the Dashboard include - initializing component objects, managing communication between the objects and object lifetime management.

Figure 4-2. UML Representation of the Dashboard
4.3.1 The Device

A Device, in context of the Dashboard, is an object that is used to model a sensor or actuator. A sensor continuously produces measurements and an actuator provides control functions. A device, being the generalization of the two, can produce any number of measurements or have any number of control functions. The CDevicePlugin class provides a generalized abstraction of any sensor or actuator in the system. This class is just a model and it doesn’t necessarily model an actual hardware device; even virtual, software-only devices can be used for the purpose of modularity. In the real-world, devices either are self updating, i.e., continuously produce measurements or need an explicit request from the reader. Usually, measurements that are sparingly used would require an explicit request, to save the device from continuously working when it is not being read. This concept is internally modeled into a device object in the form of active and passive measurements.

For the device to be a part of the Dashboard system, the programmer has to wrap the device driver implementation into the the CDevicePlugin class and specify which member variables and functions to export from the class. The device class can be either programmed to continuously measure and update a parameter or to measure only upon a request from the user. CDevicePlugin offers the option of declaring a parameter as passive or active. This differentiation between active and passive measurements is done only inside the boundaries of the device and the users need not be aware of the type of measurements. Abstracting such details away from the user makes the Dashboard interface simple and easy to learn. The details of the semantics are given in Appendix A. Also, while specifying the exportable resources, the programmer has to associate each resource with a resource ID, for identification by users.

4.3.2 The Controller

The CControllerPlugin class provides an abstraction for controllers. Controllers can also export their internal parameters and functions. Every controller object has to
implement controllerInit(), controllerMain(), controllerDeinit() functions. The controllerMain() function represents the main control loop and will be called repeatedly at a frequency set by the user. The CControllerPlugin class also derives all properties from the CDevicePlugin class. Thus, a controller can also be treated as a device that is capable of producing measurements and being controlled. This property enables the controllers to be read and controlled by other controllers. Controllers can access parameters and functions of each other and that of other devices. For instance, consider a simple climate control system with two controlled variables - temperature and humidity. Using the Dashboard system, the temperature and humidity sensors and corresponding actuators can be modeled as CDevicePlugin objects and the independent temperature and humidity controller can be modeled as CControllerPlugin objects. The sensors would export temperature and humidity values and the actuator would expose control functions that set fan speed and water flow. The controller also export control functions that allow the user to set the controlled parameters and other configuration parameters. This also enables the individual controllers to interact or share information with each other. Thus the Dashboard together with device and controller plug-ins creates an object oriented platform for programming multiple simultaneous control loops.

4.3.3 The Dashboard

Instantiation. The user has to create the Dashboard and the Dashboard in turn, instantiates all the devices and controllers attached. During instantiation, the Dashboard queries each device and controller for exported resources and maintains a database of exported resources with their IDs. At this stage the Dashboard maintains the handles to all the devices and controllers in the system. Through selective instantiation functions - attach() and detach() the user can choose to keep or drop individual devices. This feature helps in running the system with fewer devices in event of partial failures.
Resolution. When a user or controller requires a particular resource the user calls the `get(<ResourceID>)` or `set(<ResourceID>)` functions of the Dashboard. The Dashboard internally maintains pointers to these resources and resolves the call during runtime using the resource ID. Once called, the Dashboard checks whether the resource is available; if available, the requested resource is returned. In case of sensors, the requested resource is a measured parameter and in case of actuators, the requested resource is a control function. Also, only the resources, i.e., the measurements and control functions and not the device itself is exposed to the user. Thus the user is not required to address resources using a specific hardware device. This could be useful in a scenario when multiple sensors provide information on the same measurement. At the
instantiation stage, the user can select which device to instantiate and can use the same function calls to achieve a different effect. This abstraction of devices also revokes the need to change device user code with changes in the underlying device.

4.4 Implementation

With controller and device abstractions, the Dashboard system is a framework for modeling of control systems. For the autonomous guidance problem, Subramanian [46] had developed drivers for all the I/O devices and algorithms for alleyway navigation and basic headland turning. The device programs were appropriately wrapped into Device plug-ins and the alleyway navigation algorithm was modeled as a controller. A GUI capable of displaying sensor outputs was also created.

![Dashboard Implementation Schematic](image)

**Figure 4-4. Dashboard Implementation Schematic**

4.4.1 GUI

The dashboard GUI was implemented using the open-source Qt GUI toolkit. The Qt toolkit provides and easy object-oriented interface to GUI and it is platform
independent. Since platform independence has been one of the goals of the system, using Qt is essential. The GUI allows the user to selectively add and remove devices from instantiation list. This feature proved helpful in isolating hardware issues, and the system's resilience against partial failures. The GUI shows readings from all devices and controllers that are active at any point. In addition to monitoring, the user can also manually start and stop controllers.

Figure 4-5. Dashboard Device Selection

Figure 4-6. Main Window
4.4.2 Other Features

Other features of the system include a record and play system capable of recording the vehicle’s environment including images and LADAR readings and playing back. The record and play functions are implemented as controller capable of producing a data file when the user selects record. The file can later be used to view and test various controllers. This feature was mainly used in development of control algorithms described in later chapters.

The auto-config feature allows the user to set system configuration parameters and choose which controllers to start at run-time. Detailed configuration information like the turning radius, actuator limits, dimensions of the vehicle, position of the camera and other system variables are fed in the form of a text file during program startup. The program then parses this configuration information and sets the appropriate parameters. This feature was particularly useful in iteratively testing different system configurations. An external script could change the configuration file for each iteration and run the program and record results. An integrated GUI based logging system, provides a visual display of vital errors, warnings and debug information during run time and records them to disk for diagnostics.

4.5 Discussion

4.5.1 Advantages

To sum up the advantages of having the Dashboard system,

- It provides an object-oriented framework for incremental development, and improved scalability.

- Using C++, as opposed to other high-level languages, provides maximum performance and portability

- Since, all the components are designed to run within the same address space, memory can be shared and thus only minimal replication of data is required.

The software framework developed provides construction of concurrently running controllers and also allows them to hierarchically arrange them as the user requires
them to be, without imposing any specific hierarchy. The abstractions provided in the framework will help the user to model any arbitrary device or controller and the object will be ready to inter-operate with the existing system with minimal code for communication. Also, since the resource access has been standardized by means of the DashBoard, code changes to a certain component do not have a ripple effect across the system. As the object-oriented model sufficiently abstracts the components, a system designer could build the system from its components without requiring to understand the internals of each component. These qualities are desirable for incremental development and maintenance.

In addition to enabling communication, the framework also takes care of the life-times of the individual components. Once the programmer has specified an individual component, creation of the component object, selective instantiation of the component and finally destruction are taken care of by the framework. Also the framework allows isolation of components, i.e., a component could be stopped and restarted again without interrupting the other independent components. This feature greatly helps in dealing with hardware failures in the system.

4.5.2 Limitations

**Resource access overhead.** Though, the framework is designed for maximum performance, every access to a measured value or control function generates some overhead. In the present system, this overhead is a function of the number of active resources currently in the system \(O(n)\)). This could be a scalability bottleneck when the system is extended to use resources in the order of several hundreds. Using efficient searching algorithms for arbitration of resource access calls, this could be brought down to be independent of number of resources \(O(1)\)).

**Synchronization.** The Dashboard system models each controller (control loop) to be a separate thread of execution. When multiple controllers access a single chunk of memory simultaneously, inconsistencies could occur. For instance, the camera updates
a chunk of memory of size \(300 \times 400 \times 3\) bytes, every \(\frac{1}{20}\)th of a second. When other components request for the image, the Dashboard system simply returns a pointer to the image memory, for reading. As, reading and writing happens in parallel, it is possible that the reader threads get inconsistent data. The framework introduces special set of functions that serialize the read and write of data. Synchronization also comes with an increased runtime overhead, because of the waiting time.

Also, it is possible that a single actuator might receive conflicting commands from two different controllers. This could be a serious issue when it comes to mechanical parts. Such an error can cause wear of mechanical parts or pose safety concerns. It is possible to associate or 'lock-in' actuators with a particular controller and deny access to others. Unfortunately, it is difficult to generalize this idea across all components. Sometimes the application itself might need to associate multiple controllers. For example, consider vehicle speed control - A software speed controller might be associated with the actuators. Another controller, possibly an obstacle detector might require an emergency stop. Denying access to the emergency stop mechanism could be disastrous. Further, associating a set of actuators with a controller, translates to remembering the set associations for a certain period of time. So, special functions for associating and dissociating control have to be introduced. This will considerably complicate the user interface of the framework. An easy solution to this issue is to allow the user build special control arbitration components. For example, in the vehicle speed control problem posed above, the user has to create a special control arbitration component that receives inputs from both the speed controller and obstacle detector and in turn commands the actuator.

**Type safety.** The framework’s role is limited to resolving data accesses and does not do any processing on the data itself. From the framework’s point of view, a producer produces a chunk of bits and a consumer requires it; no information about the data type is involved in the exchange. Thus, when reading the data, the consumer is not
aware of the data type and has to assume that the producer produced in accordance with a set standard. Thus, data type changes in the producer would require changes in the consumer code. Storing type information along with the data would help, but will result in increased complexity and runtime overhead. The trade-off here is increased dependence between components, which leads to an avalanche of changes when a single component is changed.

**Distribution issues.** Since the system is designed to take advantage of shared memory to export and import data, distribution of components across a network will require a significant amount of effort. When components from other computers request for information, the framework has to be able to replicate the data and pass along the network. Also, when multiple nodes are involved, routing information for each node has to be stored and made available to every other node. Network management, i.e., maintaining the routing information for participating nodes and handling joining/leaving nodes becomes a responsibility of the framework. Since the system envisioned for agricultural guidance did not require extensive distribution, at least in the immediate future, in realization of short term goals, the dashboard system was designed inherently stand-alone. Nevertheless, future distribution extensions are possible on top of the existing framework.
CHAPTER 5
ODOMETRY

Odometry is the estimation of position by accumulating displacement and direction of displacement over time. Before the introduction of GPS for civilian use, odometry methods were widely used for navigation. Inertial navigation systems, using accelerometers and orientation sensors, based on odometry are used for position estimation in aircrafts and ships. Due to the mechanical nature of such systems and higher costs, simple and less accurate methods are popular in short distance and indoor navigation. The effectiveness of GPS in agricultural applications is dependent on signal reception in the field [51], which in turn is dependent on landscape and surrounding vegetation. Odometry based on wheel encoders and direction sensors have been a popular method among robot builders. By simply integrating all displacements and heading angles over a period of time, the current position of the vehicle can be estimated. Due to the accumulative nature of the calculation, errors produced in the measurement tend to accumulate. Thus dead reckoning demands the use of precise sensors.

In [46], the vehicle was guided by observing the tree line at the path boundaries and trying to stick to the path-median. Headland turning maneuver was based on yaw feedback given by the IMU; without feedback on the position of the vehicle, the vehicle was commanded to turn-around. In both cases, a precise odometry technique was not required. However, if precise position feedback was made available to the system, robust and efficient navigation algorithms might be able to take advantage of it. For example, the old system was guided based on tree positions at a fixed look-ahead point on the path way. Thus the path generated by the guidance system is as jagged as the tree line. If the system considered a section of path ahead, with position feedback, a smooth path could be generated. In the case of turning, with position feedback in $(x, y)$ co-ordinates and a control algorithm, a smooth turn path can be generated, allowing
for uncertainties like uneven row endings and unexpected artifacts at the end-of-row. Thus position feedback makes a strong case for the overall robustness of the guidance system. The scope of this research effort covers the development of a robust position feedback system and development of a turning algorithm that takes advantage of it. This robust estimation method could open a door of possibilities for advanced turning behaviors such as obstacle avoidance, optimal path planning etc.

After finding the use of magnetometer and wheel encoder based tracking unsuitable for robust guidance, a visual odometry based method of tracking, developed in [59], was implemented. For the purpose of rough-terrain, open-field navigation, this method was further refined by using a Kalman filter. In this chapter, a detailed account on visual odometry and how it can be used for robust vehicle guidance will be presented with supporting experimental results.

5.1 Methods

Traditionally, agricultural guidance was enabled by following crop rows by mechanical or optical means. Mechanical contact methods using special crop-feeler attachments were used to sense the crop row for guidance [23, 40, 58]. Mechanical guidance methods limited the operational speed and accuracy of the system. Non-contact methods based on laser ranging [18], ultrasound [43] and dead-reckoning [1] were developed to improve accuracy and speed. With the introduction of GPS in the 90s, O'Connor et al. [32] demonstrated the use of GPS for agricultural purposes. Several researchers have found the inertial navigation system useful in conjunction with a GPS. [24, 30] used Fiber optic Gyroscope (FOG) in addition to an RTK-GPS for accuracies of up to 5 cm in curved paths. Later, Guo & Feng [16] introduced a low-cost sensor system based on a GPS (with position accuracy of 3 m and update rate 1 Hz) and solid-state inertial measurement unit. A sensor fusion algorithm fused the sensor outputs to achieve an accuracy of 0.3 m at a 50 Hz update rate. Also, the system was showed to tolerate GPS signal outages lasting up to 30 seconds, with a maximum positioning
error of 0.5 m. Showing that the GPS may be unsuitable in agricultural environments due to poor satellite reception, [9] took advantage of computational power, to build vision and LADAR based guidance systems for greenhouse and citrus grove guidance applications [51]. Since, the role of GPS is limited to localization, a robust autonomous guidance platform would need some kind of environment sensing addition to make local adjustments to a global path, such as maneuvering around obstacles.

Dead-reckoning combined with vision and LADAR based guidance system could produce more robust results. However, since dead-reckoning produces only a position estimate, the effectiveness of the system will depend on the accuracy of the estimate. Subramainian [46] used a wheel encoder and a 3-axis orientation sensor. Wheel slip causes the encoder to record excess movement, than the actual distance covered. In agricultural environments, wheel slip due to uneven, marshy terrain is very common. On the other hand, the orientation sensor uses a magnetometer to detect earth’s magnetic field, and is easily interfered by external magnetic fields. Benson et al. [3] successfully used magnetometers, in conjunction with GPS, for odometry; but had to mount the magnetometer in a special aluminum mount, holding the sensor away from the vehicle’s body. The sensor requires calibration whenever it comes in contact with a metallic surface, since the residual magnetic fields\(^1\) could affect the sensor’s accuracy. Also, tests conducted in typical outdoor environments revealed that the sensor is disturbed by other vehicles and large mechanical structures in the vicinity.

An alternative to using the orientation sensor is to use the vehicle’s steering angle to estimate the orientation analytically. Given the steering angle \(\gamma\) radians, the

\[^1\] Magnetic fields left on metallic objects due to sustained contact with magnets, typically from manufacturing processes
Orientation $\theta$ of the vehicle after a time interval $t$ is given by,

$$\theta = \frac{d}{r_\gamma}$$  \hspace{1cm} (5–1)

Where, $d$ is the distance covered in time $t$.

and, $r_\gamma$ is the radius of curvature possible with $\gamma$.

It can be seen that, the heading estimate $\theta$ in the above method, is again dependent on the distance estimate from the encoder, making it only as accurate as the encoder readings. However, it can still be used to produce a valid estimate of the vehicle position, and will be used later.

Machine vision has been a major area of exploration for the agricultural guidance community. Over the past few decades, researchers mainly exploited the visual demarcation between crop rows and soil (cut-uncut crop or tilled-untiled soil), as a cue for guidance. Typically, a camera mounted in front of the vehicle, observes the field ahead. The images are segmented in terms of vegetation and soil using image processing techniques. A straight line representing the crop row is fitted using the crop segments, and is used for guidance. Searcy [38] used the infrared spectrum to differentiate between soil and vegetation. A Bayesian classifier was used to segment the image in terms of crop rows and soil. Gerrish et al. [15] controlled a tractor along near straight line paths, using computer vision alone. The system was developed to be able track any crop rows by allowing the user to initially select the row color from the image; the system would segment the rows then on.

The vision methods discussed above used vision as a means of identifying crop row position, and did not track the movement of the vehicle itself. Pinto & Reid [34], using a set of parallel crop rows in the field of view (FOV), the alignment of the crop rows with respect to the vehicle’s (observer’s) heading was extracted. It is also possible to detect and track the motion of the observer, using visual odometry techniques. Wang et al. [56] uses a stereo vision based heading estimation system for tracking a moving...
vehicle in an open field environment. In this method, a pair of images were acquired and stereo-matched to obtain the 3-D locations of the points in space. Using subsequent images from the moving observer, the movement of the features were tracked in 3-D. The motion vectors were then processed to determine the magnitude and direction of observer motion. In [59], visual odometry was used to guide an indoor vehicle platform, proposed for greenhouses. Assuming an even terrain ahead of the vehicle, this method successfully tracks the vehicle using a single camera. Visual odometry seems promising for vehicle tracking because it is not susceptible to interference as in case of magnetometers and provides a reasonably accurate measure of direction and heading. In the following sections, details of the work done in [59] and its adoption to citrus grove guidance is discussed.

5.2 Visual Odometry

The advantages that visual odometry offers against the other odometry techniques discussed above are,

- Measurement is based on the movement of objects around the vehicle, and hence is a more direct form of measurement than measuring wheel rotations.
- Not affected by slippage and interference from other sources
- The only sensor needed is a camera; and in most guidance systems, camera is already a part of the system.
- Measures motion in terms of 3-D vectors.

Visual odometry is very popular among robotics community and has been widely used in land vehicle navigation [29]. NASA’s mars rover used this technique to track its way through the sandy martian terrain [11]. In most navigation applications, a stereo vision system is used to extract features in three dimensions, and by tracking those features in the subsequent frames, 3D motion vectors were derived. The technique discussed in [59] simplifies the problem of finding the 3D location of features by
assuming that the feature point is on the ground plane. The need for stereo vision and the associated computation is avoided by making this simplifying assumption.

![Sprayer vehicle](image1) ![Ground plane assumption](image2)

Figure 5-1. Visual Odometry in [59]

### 5.2.1 Technique Overview

The determination of distance traveled by the observer involves four main steps - Feature detection, feature tracking, ground plane projection and motion detection. A set of prominent features from the image are first detected using a feature detection algorithm such as Harris corner detection. In the next frame, if the camera has moved, the features will be displaced. A feature tracking algorithm such as the KLT tracker, conducts a search around the previous feature positions. The displacement of the feature points from the previous frame to the current frame are returned as motion vectors. These motion vectors are in the image co-ordinates and have to go through a set of coordinate transformations to determine how much the features have moved in the real world. The real-world motion vectors of the set of features are filtered for outliers and averaged to produce a single vector, the reverse of which represents the estimated motion of the observer. The tracked features are retained in memory and feature tracking, transformation, etc are repeated for the subsequent frames. As features continuously move out of the FOV and are lost. When the number of features falls below
a set threshold, new features are added to the tracking list by invoking the detection algorithm.

The existing guidance system already had a camera for the tree row detection; the same set up can be used for visual odometry with the addition of software.

![Image](image_url)

Figure 5-2. Visual odometry overview

### 5.2.2 Coordinate Systems

There are three co-ordinate systems involved in going from feature points in a 2-D image to real-world co-ordinates. As shown in the figure 5-3, the camera mount’s geometry gives rise to three co-ordinate systems - the camera co-ordinate system (C) with origin $O_C$, vehicle co-ordinate system (V) with origin $O_V$ and the global system (G) with origin $O_G$. In the following sections, the co-ordinate systems will be defined and the transformations between them derived.

#### 5.2.2.1 Camera model

The intrinsic $3 \times 3$ camera calibration matrix $C$ provides transformation between a point in the image, to a vector in Euclidean 3-D space. $C$ is constituted by camera specific parameters like principal point, focal length, skew and distortion.
Figure 5-3. Coordinate systems

\[
\begin{bmatrix}
\alpha \\
\alpha \\
\alpha \\
\end{bmatrix}
= \begin{bmatrix}
\alpha_u & 0 & u_0 \\
0 & \alpha_v & v_0 \\
0 & 0 & 1
\end{bmatrix}
\]

(5–3)

and \( \begin{bmatrix}
x_r \\
y_r \\
f
\end{bmatrix} \) is the vector passing through the direction of the point of interest in Euclidean 3-D space.

\( \alpha_u, \alpha_v, u_0, v_0 \) are called the intrinsic parameters of the camera, and can be determined by camera calibration. The MATLAB camera calibration toolbox was used to determine the intrinsic parameters of the vehicle's camera.

The extrinsic camera calibration matrix (X) is a \( 3 \times 4 \) rotation-translation matrix that translates points from camera co-ordinates to vehicle co-ordinates. The vehicle co-ordinate system is based on the vehicle’s center, and the camera co-ordinate system
is based on the camera origin. \( \mathbf{X} \) is determined by physical measurements taken from the set-up, like camera elevation, camera orientation etc.,

\[
\begin{bmatrix}
x_v \\
y_v \\
z_v \\
1
\end{bmatrix} = \mathbf{X} \cdot \begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix}
\]

(5–4)

where, \( \mathbf{X} = [\mathbf{R}|\mathbf{T}] = 
\begin{bmatrix}
1 & 0 & 0 & L_x \\
0 & \cos \alpha & -\sin \alpha & L_y \\
0 & \sin \alpha & \cos \alpha & L_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

(5–5)

\([x_v y_v z_v]^T\) represents a point in vehicle co-ordinate system and \([x_c y_c z_c]^T\) represents a point in camera co-ordinate system.

The translation parameters – \( L_x, L_y \) and \( L_z \) represent the translation of camera’s origin w.r.t the vehicle co-ordinates and \( \alpha \) is the camera’s orientation. The transformation matrix,

\[
\mathbf{V} = \mathbf{X} \cdot \mathbf{C}^{-1}
\]

converts 2-D points from the image to 3-D points with respect to the vehicle’s center. All feature points from the images need to undergo this transformation to find the actual distance traveled in vehicle co-ordinates.

**5.2.2.2 Global coordinates**

In addition to the above two co-ordinate systems, a global co-ordinate system, based on the vehicle’s starting position, is required to track the vehicle’s position. Initially the global axes and the vehicle’s co-ordinate axes intersect, and when the vehicle moves, they begin to separate. Though this transformation is not required for visual odometry it will be used for mapping of field features such as tree rows and
planning turns. This transformation matrix is incrementally updated by new position and orientation of the vehicle, as the vehicle moves.

5.2.3 Feature Point Tracking

Figure 5-4. Visual odometry— control flow

5.2.3.1 Feature detection

A feature point is defined as an isolated local area of maximum or minimum intensity in an image. It is desirable to have a feature point that is stable under a variety of geometric deformations, for tracking them into future frames. [39] proposes a feature selection method that selects features that are optimal for tracking, based on texture and feature dissimilarity. The goodFeaturesToTrack algorithm as described in [39] is available as a method in the OpenCV library. This algorithm is used for initializing the visual odometer with an initial set of features ($N_{max}$). goodFeaturesToTrack can be configured with various parameters such as, number of features ($N_{max}$), minimum feature
quality \((Q_{min})\), distance between features \((d_f)\) and feature size \((s_f)\). These parameters were selected based on experiments conducted with the visual odometer and will be discussed toward the end. Since, feature point on the ground plane are the only features of interest, we restrict our search area to a rectangular region of interest (ROI) in the image, that represents a trapezoidal piece of ground, ahead of the vehicle. The ROI is empirically selected to have its top left vertex at \((\frac{h}{2}, \frac{w}{4})\) and bottom right vertex at \((\frac{4h}{5}, \frac{3w}{4})\), where \(h\) and \(w\) are the height and width of the image. Note that the ROI is selected for a fixed camera angle \((12.2^\circ)\) and might have to change with changes in the camera angle.

![Figure 5-5. Feature detection ROI and features in a typical FOV](image)

5.2.3.2 Tracking

The Lucas-Kanade method for optical flow estimation is used for tracking in [59]. A pyramidal implementation of the same algorithm, that optimizes the runtime has been implemented in [5]. The pyramidal searching algorithm, first computes pixel motion in a low resolution image, and uses it as a motion estimate. The accuracy of the estimate is improved by repeatedly searching on higher resolution images. This helps tracking large
pixel motions in fewer search steps. A set of features from a frame $F_t$ from time $t$ and their positions $P_t$ are given to the algorithm, and the algorithm computes new positions $P_{t+\tau}$ using $F_{t+\tau}$. Due to occlusions and finite image size, the algorithm declares some features as 'lost', using a feature status indicator for each feature. Also, a tracking quality figure ($Q_\tau$) based on the dissimilarity between the feature points in the two frames is produced at the end of the tracking computation. $Q_\tau$ can be used as a metric for prioritizing the motion estimates when calculating the resultant motion of the observer. While feature detection is limited to the ROI, tracking is not limited and is allowed to proceed till the edge of the image.

5.2.3.3 Displacement estimation

The relative displacement of the vehicle from time instant $n-1$ to $n$ can be determined by finding the relative displacement of the feature points on the ground plane.

Using the camera matrix $C^{-1}$ from equation 5–6, a vector pointing in the direction of the feature point in $C$ can be found. By further applying the transformation $V$, the vector can be represented in terms of co-ordinate system $V$.

Thus, the vector w.r.t $V$ is

\[
\begin{bmatrix}
    x_r \\
    y_r \\
    z_r \\
    1
\end{bmatrix} = \mathbf{X} \cdot \mathbf{C}^{-1} \cdot \begin{bmatrix}
    u_i \\
    v_i \\
    1
\end{bmatrix}
\]

The intersection of the vector in the above equation, with the ground plane, given by $y_V = 0$ represents the corresponding point on the ground plane.

Solving the equation of the line and the plane, the point of ground plane is,
The ground projections \((g^n_i)\) of feature points from frame \(F_{n-1}\) and \(F_n\) are thus determined.

The difference vector, \(v^n_i = g^n_i - g^{n-1}_i\), represents the motion vectors of each feature point \(i\), on the ground plane. A weighted numerical average based on the quality metric \(Q_i\) from the tracking algorithm, is applied on the vectors to arrive at a single resultant vector \(-v^n\), that describes the displacement of the observer during time instant \(n\).

5.2.3.4 Orientation estimation

The motion vector obtained in the previous section describes the motion in both the \(X\) and \(Z\) directions w.r.t \(G\). The angle between the lines connecting the points \(g^{n-1}_i\) and \(g^n_i\) with the vehicle’s origin \(O_v\) can be used as a reasonable estimate of the change in vehicle heading.

The change in heading \(\gamma_i\) can be found using the vector dot product,

\[
\gamma_i^n = \cos^{-1}\left(\frac{g^{n-1}_i \cdot g^n_i}{|g^{n-1}_i||g^n_i|}\right)
\]

Figure 5-6. Ground plane motion vector
A weighted average over all the feature points produces the final heading change $\gamma^n$ at instant $n$.

### 5.2.3.5 Vision system calibration

To validate the vision system developed, the following experiment was performed.

- The ground ahead of the vehicle was graduated in increments of $1 \, m$, to form a 2-D grid of points
- An image of the graduated ground plane was captured using the vision camera
- These markings on the ground plane, with known 3-D locations — $(x_i, y_i, 0)$, were located in the image as pixel values — $(u_i, v_i)$
- The estimated 3-D positions, generated by the camera matrix, were compared with ground truth values, and the error was characterized.

Upon comparison, it was found that the error in the estimation showed an approximately quadratic trend, with increase in linear distance from the camera. The reasons for the error could be because of the accuracy of the transformation matrices. Since, some of the matrix elements are physical measurements, the accuracy of such measurements depends on the measurement method. Even though a millimeter accurate laser range finder was used in the measurements, due to physical limitations accuracy could be limited. For instance, the camera is sealed in a metal casing, and there is no accurate means of locating the camera’s origin or the exact orientation of the camera. Thus a final calibration step becomes necessary to correct for such errors.

The error in the estimation was then modeled using the MATLAB curve-fitting tool, and a relation between the estimation error Vs. the linear distance of the point was determined. This relation was used to produce an accurate 3-D position estimate of an arbitrary point of interest. The calibration procedure improved the accuracy of the vision system on flat terrain.

Though the 3-D positioning of features is accurate, some features are abruptly lost in subsequent frames. When the tracking algorithm loses track of a feature, the feature is declared 'lost'. But in case of relatively flat, featureless or fine-textured terrain like a
concrete floor, the disparity between feature patterns is very small and lost features are often mistaken for nearby similar patterns. While such errors are hard to eliminate, such errors can be minimized by using a weighted average of tracking quality.

Sometimes loss of features could be due to jerky motion or occlusion. To minimize tracking error in such scenarios, the tracking algorithm is allowed to drop certain frames that are considered untraceable. When a frame is dropped, the motion vector estimated for the previous time instant is considered for the current time instant. After a drop event has occurred, the tracker is reset with a new set of features.

5.2.4 Experiments and Discussion

The greater the number of feature points tracked by the system, greater is the robustness of the system. With more points to track, the computational load on the system is also high. From the tracking algorithm, it can be seen that the tracker needs a certain minimum number of features to behave reliably. The system initially starts with a maximum number of features and starts losing features as the vehicle moves. The tracker calls the feature detector when the number of features falls below a set minimum. The more often the feature detector is called, the higher the computational cost. Assuming that the features are lost at a constant rate, allowing the tracker to operate at a wider band of feature count, should help minimize the computational costs. On the other hand operating on a lesser number of features can affect the accuracy of measurement. To find the effect of the operating band on the computational cost and accuracy, trials with different operating bands on the same terrain were conducted. The vehicle was driven on a straight line path of 75 ft (22.8 m)$^2$ and the tracking results were noted down.

$^2$ The typical turn circumference is about 15 m–20 m
Table 5-1. Effect of feature count on accuracy and computational efficiency

<table>
<thead>
<tr>
<th>$N_{max}$</th>
<th>$N_{min}$</th>
<th>Error %</th>
<th>frames dropped</th>
<th>Processing time (ms/frame)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>13.77</td>
<td>5</td>
<td>32.43</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>-1.71</td>
<td>3</td>
<td>33.30</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>5.93</td>
<td>5</td>
<td>49.69</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>21.40</td>
<td>5</td>
<td>34.64</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1.57</td>
<td>5</td>
<td>47.02</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>6.41</td>
<td>5</td>
<td>46.56</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>6.90</td>
<td>6</td>
<td>37.44</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>9.67</td>
<td>5</td>
<td>51.29</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>6.22</td>
<td>5</td>
<td>35.54</td>
</tr>
</tbody>
</table>

From the results in table 5-1, it can be seen that when the algorithm is allowed to operate within a wider band of feature counts, the accuracy is higher. At the same time, more features contribute to a higher processing time.

Effect of terrain. The accuracy of visual odometry greatly depends on the quality of features and the quality of tracking. In surfaces where the disparity between features is rich, like patchy and grassy landscape, the technique could be expected to perform better than on fine-textured surfaces. The results shown in table 5-2 demonstrate the relative applicability of this technique on terrains.

From the results in table 5-1, it can be seen that when the algorithm is allowed to operate within a wider band of feature counts, the accuracy is higher. At the same time, more features contribute to a higher processing time.

Effect of terrain. The accuracy of visual odometry greatly depends on the quality of features and the quality of tracking. In surfaces where the disparity between features is rich, like patchy and grassy landscape, the technique could be expected to perform better than on fine-textured surfaces. The results shown in table 5-2 demonstrate the relative applicability of this technique on terrains.

Figure 5-7. Sample images of road and turf used
Table 5-2. Effect of terrain on accuracy: Error in the distance traveled

<table>
<thead>
<tr>
<th>Terrain</th>
<th>Average Error (%)</th>
<th>Max Error (%)</th>
<th>Min Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Road</td>
<td>35.50</td>
<td>41.14</td>
<td>29.56</td>
</tr>
<tr>
<td>Grass</td>
<td>10.42</td>
<td>14.40</td>
<td>5.04</td>
</tr>
</tbody>
</table>

5.2.5 Limitations

Apart from the above factors, other factors like operating speed, camera mount angle and terrain roughness affect the accuracy. For experimental purposes, the operating speed of the vehicle was maintained around an average of $1.2 \text{ m/s}$.

The errors observed in table 5-2, while seemingly acceptable for the purposes of controlling the vehicle over short turns, requires a relatively smooth and feature-rich terrain. When the terrain is rough, or if the vehicle experiences substantial vibration due to acceleration, accuracy is lost considerably. By observing the behavior of the visual odometer on field recorded sequences, the effect of vibration on feature tracking was observed. As feature tracking is disrupted due to a jerk in the sequence, the odometer is forced to drop the current frame and search for new features in the next frame. Thus tracking is disrupted for two consecutive frames. With every jerk in the motion, accuracy is compromised and the errors accumulate in the final estimate.

5.3 Kalman Filtering

Unlike common digital filters that use only a range of measurements to estimate the accurate value of a variable, the Kalman filter takes the underlying process, that produces the measurements, into consideration to produce an estimate. Thus it is possible to have noisy sensor measurements and still have a reasonably accurate idea of the environment, using a Kalman Filter. Benson et al. [3] used the same to fuse GPS and INS measurements, thus combining the accuracy of GPS with the high update rate of the INS. With computational cost becoming cheaper every day, the filter allows for the use of cheap, imprecise sensors with added computational effort instead of expensive and accurate sensors. In our application, the Kalman filter was used to refine visual odometry to become a viable technique for guidance.
5.3.1 Kalman Filter Theory

Kalman filter fuses information from an internal system model and the actual measurement. By recursively correcting the internal model with external measurements, keeps the model ‘in-track’ with the environment. For this reason, the filter needs a mathematical model of the physical system that produces the measurement.

Consider the following noisy linear system,

\[ x_{k+1} = Ax_k + Bu_k + w_k \quad \text{(State equation)} \]
\[ y_k = Cx_k + z_k \quad \text{(Output equation)} \]

(5–8)

Where, \( x_k \) is the state vector, \( u_k \) is the input vector, \( w_k \) is the process noise vector and \( z_k \) is the measurement noise. \( A, B \) and \( C \) are matrices that form the state-space description of the system.

Noise is introduced in the system in two forms — during the transition of states from \( k \) to \( k+1 \) and during measurement. Assuming that \( w_k \) and \( z_k \) are independent random variables representing Gaussian white noise, the covariances \( Q \) and \( R \) can be expressed as,

\[ Q = E[w_k w_k^T] \quad R = E[z_k z_k^T] \]

(5–9)

Where the operator \( E[\cdot] \) denotes the expected value.

The Kalman filter recursively operates in two distinct steps — Predict and Update. First, the filter makes an effort to accurately determine the estimate of \( x_{k+1} \) using the state model. Along with the state estimate, a process covariance estimate \( P_k \) is also produced. In the update step, the state estimate produced is refined using the measurement obtained at time \( k \). The amount of impact the measurement \( y_k \) will have on final estimate of the state is dependent on the process error covariance and the measurement error covariance. A ‘weighting’ matrix called the Kalman Gain \( K \) is produced in the update step that helps blend the predicted state and the measurement.
### Predict Steps

\[
\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \\
P_k^- = AP_{k-1}A^T + Q
\]

(5–10)

### Update Steps

\[
K_k = P_k^C (CP_k C^T + R)^{-1} \\
\hat{x}_k = \hat{x}_k^- + K_k(z_k - C\hat{x}_k^-) \\
P_k = (I - K_k C)P_k^-
\]

(5–11)

Where,
- \(\hat{x}_k\) - Kalman filter estimate of state
- \(P_k\) - Kalman estimation error covariance
- \(\hat{x}_k^-, P_k^-\) - apriori estimates of the above
- \(K_k\) - Kalman Gain
- \(Q\) - Process noise covariance
- \(R\) - Measurement noise covariance
- \(I\) - Identity matrix

Though \(Q\) and \(R\) are shown as constants in the above equations, they can vary with time. In equations 5–11, \(K_k\) is used to correct the apriori state estimate using a blend of \(Q\) and \(R\). Thus by tuning these covariance matrices the impact of measurement or internal estimate on the final state estimate can be controlled. The updated state estimate \(\hat{x}_k\), called the aposteriori estimate, is produced with a minimized error covariance, and is used to calculate the apriori estimate of the next time step [60].

### 5.3.2 Enhanced Visual Odometry

#### 5.3.2.1 System model

To apply Kalman filtering to any problem, the physical process that produces the measurement has to be modeled first. In odometry, the state variables that uniquely
define the state of the system at any time instant are the position and heading. Thus, the state of the system can be represented as,

\[
\begin{bmatrix}
  x_k \\
  y_k \\
  \theta_k
\end{bmatrix}
\]

Where,

- \textit{x}_k, \textit{y}_k \quad \text{- Vehicle Position in global co-ordinates}
- \textit{\theta}_k \quad \text{- Vehicle heading in global co-ordinates}

The inputs to the vehicle are acceleration and steering. Wheel rotations are measurable through the encoder connected to the vehicle’s drive. The speed command given to the vehicle, through the manual pedal or through the driving computer, can be directly measured as displacement command using this encoder. Thus the input vector \textit{u}_k can be written as,

\[
\begin{bmatrix}
  n_k \\
  n_k \alpha_k
\end{bmatrix}
\]

Where,

- \textit{n}_k \quad \text{- Wheel rotations}
- \textit{\alpha}_k \quad \text{- Steering command}
- \textit{n}_k \alpha_k \quad \text{- Wrench Effort}

The transition matrix \textit{A} from 5–8, models the dynamics in the system and is also the unforced response of the system. By making a simplifying assumption that the system does not exhibit dynamics, \textit{A} can be assumed to be a 3 \times 3 identity matrix. This means that the system is lossless and continues to be in motion once set in motion. Also, there is no interaction between the individual state variables. The control matrix \textit{B} maps the inputs to state variables, and involves constant transformations. Measurement from the sensor — in this case visual odometer— is directly in the form of the state variables and
thus the measurement matrix can be assumed to be a $3 \times 3$ identity matrix. Thus, a simple model for the physical system as below has been arrived at.

\[
\begin{bmatrix}
    x_{k+1} \\
    y_{k+1} \\
    \theta_{k+1}
\end{bmatrix}
= \begin{bmatrix}
    1 & 0 & 0 \\
    0 & 1 & 0 \\
    0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x_k \\
    y_k \\
    \theta_k
\end{bmatrix}
+ \begin{bmatrix}
    K_w \cos \theta_k & 0 \\
    K_w \sin \theta_k & 0 \\
    0 & K_s
\end{bmatrix}
\begin{bmatrix}
    n_k \\
    n_k \alpha_k
\end{bmatrix}
\]

\[y_k = \begin{bmatrix}
    1 & 0 & 0 \\
    0 & 1 & 0 \\
    0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x_k \\
    y_k \\
    \theta_k
\end{bmatrix}\]  \hspace{1cm} (5–12)

Where,

- \(K_w\) - Wheel constant
- \(K_s\) - Steering constant

### 5.3.2.2 Determination of constants

\(K_w\) relates measured wheel rotations to distance covered in global coordinates. Derivation of \(K_w\) is done by experimental means; the vehicle was driven to set distances and the encoder readings measured. The constant was determined by the following relation,

\[K_w = \frac{\text{Distance covered}}{n_k}\]

The experiments were repeated over different terrain and path trajectories and an average \(K_w\) was determined.

\(K_s\) relates the displacement, steering angle and heading angle. Let \(\alpha\) be the steering angle required to describe a circle of radius \(r\). Then,

\[\alpha_k \cdot r = K\]  \hspace{1cm} (5–13)
Given a circular trajectory of radius $r$, the heading angle $(\theta_k)^3$ when in trajectory can be found from the distance the vehicle moved along the circumference of the circle.

![Figure 5-8. Illustration of $K_s$ determination experiment](image)

From equation 5–13,

$$\theta_k = \frac{K_w \cdot n_k \cdot \alpha_k}{K} = K_s \cdot n_k \alpha_k$$

Where, $K_s = \frac{K_w}{K}$.

The constant $K_s$ was determined by experimentation. The vehicle was driven in circles of varying radii, and the steering angle required in each case was noted down.

---

3 Note that $\theta_k$ and $\alpha_k$ are in radians
With enough number of readings, the constant $K$ in each case was determined and averaged.

### 5.3.2.3 Kalman filter design

![Diagram of Kalman filter design](image)

Figure 5-9. Kalman filtered visual odometry—control flow

In order to completely implement the Kalman filter described in equations 5–10 and 5–11, the error covariances for the process and measurement have to be filled in the Kalman filter. According to the application requirements, we have a measurement
system, whose accuracy is dependent on the quality of the surface, and certain discontinuous events like rough terrain and sudden start or stop. It is desirable to have a robust filter that is responsive to such discontinuous events in the environment. In case of surface quality, it is possible to model the measurement error covariance as a continuous function of surface quality. However, for discontinuous events, a continuous function may not help.

The measurement error covariance of $5–11$, $R$ is a $3 \times 3$ matrix. The visual odometer provides the tracking error (uncertainty in tracking) as a quality measure for every successfully tracked feature. When the ground surface is feature rich this error is minimal and vice versa. Thus, the variance of tracking error ($V_f$) can be thought of as a direct measure of visual odometer’s reliability. However, the tracking quality is one-dimensional unlike the state that is three-dimensional. Assuming that the variables are not correlated, and applying the same variance to all the state variables, we get,

$$ R = V_f \cdot I^3 $$

Where, $V_f = \sum_{f=1}^{n} e_f^2$ and $e_f$ is the tracking error for an individual feature $f$.

However, in face of uncertainties, like bumps in the terrain, the tracker loses all tracked features and has to drop the frame. In such an event, there are no features to consider and the relationship described above breaks. The algorithm handles this case by feeding infinity (high numerical value) for $R$. Consequently, the Kalman filter, in the next state estimation step, just ignores the visual odometer’s state estimate.

The process noise covariance is insignificant in this case, as the process model is relatively simple. Thus a constant empirical value is fed into the $Q$ matrix of the update equation $5–11$.

$$ Q = C \cdot I^3 $$
5.4 Experiments

To find out how effective the Kalman filtered odometer was a set of experiments were done on different conditions, and the performance of visual odometer with and without Kalman filtering was measured and compared.

5.4.1 Distance Measurement

The vehicle was run for a set distance (17.7 m) on grassy terrain, and the distance recorded by the visual odometer with and without Kalman filter was recorded. The main challenge in grassy terrain is dropped frames due to jerky motion.

Table 5-3. Distance measurement performance

<table>
<thead>
<tr>
<th>Trial</th>
<th>Unfiltered (mm)</th>
<th>Error%</th>
<th>Kalman filtered (mm)</th>
<th>Error %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15741</td>
<td>12.445</td>
<td>17056</td>
<td>3.638</td>
</tr>
<tr>
<td>2</td>
<td>15520</td>
<td>14.046</td>
<td>17409</td>
<td>1.644</td>
</tr>
<tr>
<td>3</td>
<td>16850</td>
<td>5.044</td>
<td>17447</td>
<td>1.429</td>
</tr>
<tr>
<td>4</td>
<td>15471</td>
<td>14.407</td>
<td>17250</td>
<td>2.542</td>
</tr>
<tr>
<td>5</td>
<td>16669</td>
<td>6.185</td>
<td>17063</td>
<td>3.598</td>
</tr>
</tbody>
</table>

The above experiment was repeated on a fairly featureless road surface for a distance of 22 m. The main challenge here is the quality of features.

Table 5-4. Distance measurement performance on featureless surface (Road)

<table>
<thead>
<tr>
<th>Trial</th>
<th>Unfiltered (mm)</th>
<th>Error%</th>
<th>Kalman filtered (mm)</th>
<th>Error %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16002</td>
<td>37.482</td>
<td>21929</td>
<td>0.322</td>
</tr>
<tr>
<td>2</td>
<td>15587</td>
<td>41.143</td>
<td>17464</td>
<td>20.618</td>
</tr>
<tr>
<td>3</td>
<td>16085</td>
<td>36.773</td>
<td>21033</td>
<td>4.395</td>
</tr>
<tr>
<td>4</td>
<td>16980</td>
<td>29.564</td>
<td>19239</td>
<td>12.550</td>
</tr>
<tr>
<td>5</td>
<td>16598</td>
<td>32.546</td>
<td>19048</td>
<td>13.418</td>
</tr>
</tbody>
</table>

When the visual odometer is run on a fine-textured surface, the feature tracking algorithm was mislead due to less dissimilarity between adjacent features. The reliability figures reported by the visual odometry indicated reliable tracking, whereas in reality, features were mis-tracked. This again mislead the Kalman filter in accepting the visual odometer’s measurements. However, we observe a marked improvement in the turf, where the Kalman filter successfully rejected dropped frames.
5.4.2 Heading Measurement

To validate the orientation measurement mechanism, a set of tests were conducted on the machine. With a constant steering command, the vehicle was made to describe circles and was made to turn till a fixed angle. Before starting the test, the vehicle's current orientation was marked on the ground. Then with a constant steering angle of $30 \text{ deg}$, the vehicle was driven in a circle until the start position was reached again. The actual angle described by the circular motion ($2\pi \text{ rad}$), was compared against the measured angles.

Table 5-5. Heading measurement performance

<table>
<thead>
<tr>
<th>Trial</th>
<th>Actual Angle (rad)</th>
<th>Unfiltered Error (rad)</th>
<th>Error (%)</th>
<th>Kalman Filtered Error (rad)</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.28</td>
<td>6.50</td>
<td>-3.50</td>
<td>6.11</td>
<td>2.71</td>
</tr>
<tr>
<td>2</td>
<td>6.28</td>
<td>6.29</td>
<td>-0.15</td>
<td>6.07</td>
<td>3.34</td>
</tr>
<tr>
<td>3</td>
<td>6.28</td>
<td>5.99</td>
<td>4.62</td>
<td>6.02</td>
<td>4.14</td>
</tr>
<tr>
<td>4</td>
<td>6.28</td>
<td>6.19</td>
<td>1.43</td>
<td>6.11</td>
<td>2.71</td>
</tr>
<tr>
<td>5</td>
<td>6.28</td>
<td>6.25</td>
<td>0.48</td>
<td>6.08</td>
<td>3.18</td>
</tr>
</tbody>
</table>

The results above show that the Kalman filter did not significantly improve heading measurement, except for the fact that the range of error values have reduced from 8.12% to 1.43%. The filter’s inaccuracy can be attributed to inadequate vehicle model in equation 5–12. The interactions between the individual wheels and the body of the vehicle, and the dynamics involved in steering a moving vehicle could improve the kalman filter’s estimate of position.
CHAPTER 6
CITRUS GROVE NAVIGATION

In the previous chapters a software architecture that is capable of hosting discrete controller components and a kalman filtered visual odometry technique were developed. Using the software architecture of chapter 4, it is possible to construct a control system with multiple simultaneous controllers exchanging data among themselves. Using the odometry technique discussed in chapter 5 as a feedback, it is possible to develop advanced navigation controllers. In this chapter, a simple navigation control system that demonstrates the usability of the two concepts above has been presented. The control system developed will then be employed to guide the vehicle to citrus grove requirements — alleyway and end-of-row guidance.

6.1 Software Architecture

The DashBoard framework developed in chapter 4 was used as a container of all the software controller components. The Turn controller, Alley-way controller and Odometer were developed independently and integrated into the framework. The framework takes care of creating and maintaining the communication between these components. Figure 6-1 shows all the software components that constitute the final navigation application. The Alley-way controller component contains software that was developed previously and was wrapped with the DashBoard-specific wrapper code to integrate into this present software architecture. From the figure, it can be seen that the vehicles steering and speed are controlled by three different components — User Interface (GUI), Turn control and Alley-way control. The Alley-way control utilizes vision and Lara information to segment the path and issues steering and speed commands. On the other hand, the Turn controller utilizes the newly developed Odometer to measure position and issues speed and steering commands. The vehicle is also directly controllable by the user through the GUI. (Refer chapter 4 for details on concurrency and communication issues).
6.2 Navigation Functions

6.2.1 Alleyway Navigation

The software controllers for alleyway navigation developed in [46] was re-used in the application. A color-image segmentation algorithm segments the path from the vegetation and produces lines representing the path boundary. For reliable path segmentation, LADAR measurements from the path indicating the location of vegetation is also used. Information from the two sources is fused using a Fuzzy-Kalman filter and is used for steering the vehicle through the alley way. Note that this way of guidance is only possible while the vehicle is in the alleyway. Thus, an end-of-row detection check is performed in the controller as a part of vision processing. When the vehicle detects
an end-of-row, alleyway control is temporarily suspended and the turn controller is activated.

![Diagram of Alley-way Navigation — Control Flow]

Figure 6-2. Alley-way Navigation — Control Flow

### 6.2.2 End-of-Row Turn Control

In [46], a simple point-to-point driving controller using the GPS for position feedback was implemented and tested. The resulting system was able to reach a discrete set of input points with a position accuracy of 1 m [46]. The distance between the current position and the required position was calculated and the vehicle was driven in the direction of the target point until the target distance reduced to zero. This same idea was extended in the current research, except that the position feedback used here is Visual Odometry.

A PID controller was implemented to regulate the speed to a set cruise speed. With controlled speed, the vehicle was driven in the direction of the target until the target was reached. Since the vehicle platform used is non-holonomic, the vehicle will not necessarily be able to reach targets in straight-line paths. Also the steering mechanism is mechanically limited to a certain maximum angle, so directing the vehicle in the target’s direction is not possible in a single step. With continuous position and heading feedback from the odometer, the system continuously adjusts the steering until the
target is achieved. In each step, the system incrementally gets close to the target and finally reaches the target point.

In the next step, a discrete set of points representing a loop-turn was given to the driving controller and the controller guided the vehicle to the points, one after the other thus describing a loop turn. The way-points representing the turn were recorded from a manual loop turn done in the headland. During the trials, the vehicle was found to closely follow the manual turns. Figure 6-3 is an illustration of the commanded points for the loop-turn and vehicle trajectory at the end-of-row.

Figure 6-3. End-of-row Turning illustration

6.2.2.1 Point-to-point driving controller

Assuming that the user supplies the control points for the required maneuver, the system has to be capable of steering itself to the next via-point. A Point-to-Point driving controller was built for this purpose. The controller controls both the steering and speed of the vehicle in-order to achieve the via-points in the order they were supplied to the controller. Internally, a FIFO queue of via-points is maintained in the controller’s memory, which the user and other controllers can access through the DashBoard. The via-point
queue is appended with points at the end, and the controller constantly tries to reach
the point at the front of the queue. When the oldest via-point is reached, it is dequeued,
and the controller pursues the next in queue. To determine required steering and speed,
the driving controller internally relies on two distinct controllers. The interconnection of
various component controllers is shown in figure 6-4.

A simple on-off control strategy has been adopted for point-to-point driving. The
controller gives out steering and speed signals to the vehicle, until the time a via-point is
reached. 'Reaching' a point here can be defined as placing the vehicle within $\delta_c$ m from
the target point. Where $\delta_c$ is the controller distance tolerance. This simple, on-off control
strategy works well when the via-points are more or less aligned with the vehicles
current heading. As the vehicle platform is mechanically limited to a maximum steering
angle, targets that require sharp turning in a short distance might be unable to reach.
With this simple strategy, the vehicle resorts to making circles around the target in-order
to reach the target. To enable to controller to recognize when targets are missed, the
target angle $\alpha_z$ is constantly monitored as the target approaches. As $\alpha_z$ diverges to more

Figure 6-4. Point-to-Point driving controller block diagram
than ±90 deg, the tolerance $\delta_C$ is compromised by a constant factor. This helps prevent the vehicle move around the target points and straying from intended trajectory.

### 6.2.2.2 Steering and speed control

Using position and heading feedback from the visual odometer the steering controller calculates the heading angle required to reach the target using the following relation.

$$\theta_{req} = \tan^{-1} \left( \frac{x_c - x_t}{y_c - y_t} \right)$$
Where, \( x_c \) and \( x_t \) represent the current vehicle position and target position.

Quite often the required steering angle is more than what is mechanically possible by the vehicle. This steering value is limited before sending to the actuators.

By differentiating position information over time, the instantaneous velocity of the vehicle is derived and is used for speed control. Since, the Visual Odometer is a relatively slow component and speed control requires to be run at a much higher frequency to achieve precise control - position information from the encoder was preferred rather than the visual odometer. A PID controller was used to arrive at the required velocity. The PID controller was then tuned for smooth performance, on plain and uneven terrain. To limit the noise in the control loop from entering the controller and hence amplified, an averaging filter was employed the speed controller feedback path.

6.3 Results

6.3.1 Evaluation of Point-to-point Driving Controller

In order to evaluate the performance of the turn controller, the accuracy with which the vehicle can reach a point in space is evaluated. Command points on space at different distances and directions were given to the vehicle and the vehicle was let to navigate to the commanded point. Upon stopping, the path of the vehicle was traced and difference between the commanded position and the actual final position of the vehicle were measured and plotted.

A 30cm accurate DGPS was used for providing ground truth estimates for the experiment. When a via-point, 9m in front of the vehicle is issued to the controller, the vehicle used the visual odometer feedback to reach the via-point. The figure 6-6, shows a plot of actual positions recorded by the GPS and estimated positions tracked by the visual odometer. It can be observed that, the position error in the estimation accumulates over time and hence results in divergence from actual path. The experiment was conducted on fairly smooth terrain and the velocity was set at 0.5 m/s.
Since the Visual odometer runs at a lower frequency than the GPS, the readings obtained had to be interpolated for comparison. The following statistics resulted from the comparison.

Table 6-1. Straight line tracking performance

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Error (m)</td>
<td>0.746</td>
</tr>
<tr>
<td>Max. Error (m)</td>
<td>1.629</td>
</tr>
<tr>
<td>RMS Error (m)</td>
<td>0.456</td>
</tr>
</tbody>
</table>

The error statistics for different trial varied by different amounts indicating randomness in the tracking process. This randomness could be a result of vision as discussed previously in the development of the visual odometer. Also better vehicle modeling could help reduce divergence over time.

6.3.2 Evaluation of Loop-turn Behavior

As explained in the previous sections, a set of command points representing a loop-turn are given to the vehicle and the turning performance during each turn was
measured in terms of path tracking accuracy. The commanded trajectory and the actual trajectory were recorded and compared.

The vehicle was first commanded to turn into the next row using a loop-turn and the turn path taken was recorded using a DGPS. As in the previous experiment, the intended path and the actual path diverge, resulting in growing position error.

![Path tracking accuracy for a loop turn](image)

**Figure 6-7. Path tracking accuracy for a loop turn**

| Mean Error (m) | 0.774 |
| Max. Error (m) | 1.532 |
| RMS Error (m)  | 0.419 |
| End-pt Error(m) | 0.506 |

**Table 6-2. Loop turn tracking performance**

The repeatability of the above results suffer badly due to randomness of environmental factors. There is no guarantee that the same features will be selected in each trial. Noise in the odometer readings primarily induced due to the vibration of camera mount, amounts to randomness in behavior. Another systemic error that is evident in figure 6-7 is that the GPS used was mounted on the vehicle's front, whereas the odometer tracked
the vehicle’s center point. The distance between the two is at least 0.5 m, and this has amounted to a constant offset in the turn.

Another major factor that affects visual odometry is the fact that this method uses monocular vision. The features collected from the ROI, are assumed to lie on a flat ground plane. In reality, the terrain is uneven and vegetation at the end-of-row could contribute to estimation error.

However, even with path tracking error, this approach is able to make an end-or-row turn in an unsupervised setting which is an improvement over the previous open-loop technique employed in [46]. With end-point error of 0.5 m, the method warrants placement of the vehicle in the next row.

**Modeling errors.** The kalman filter developed in the previous chapter is based on a simple model. Models for four wheeled, car-like vehicles have been widely studied and used in the research community [41]. Also, the vehicle steering is not instantaneous in nature; the vehicle takes a cycloidal path when turning rather than the assumed circular path. These modeling improvements could help the kalman filter estimate the state even more accurately. In future, the accuracy of the present system could be improved by taking the above factors into consideration.
CHAPTER 7
CONCLUSION

A generic software control architecture and an autonomous guidance system based on the software framework, were developed during the course of this research. Agricultural guidance systems have been developed by researchers around the world over the decades. For these systems to be scalable and inter-operable with other systems, a common software framework is necessary. The DashBoard framework is a first step toward such organization of software. As effectiveness metrics for the software system are impossible to arrive at, in the current state of software technology, by means of first hand experience in developing the guidance system based on the framework proved easy. The core components and usability of the programming interfaces will be put to test by future developers trying to scale the system.

The odometry technique developed is based on the work done by [59] for indoor robot guidance. When using visual odometry in outdoor environments, vibration played a disrupting role in the tracking. By trying to selectively include the data from alternative sensors, the effect of vibrations were reduced. This adaptation was then successfully used for tracking the vehicle while guiding through end-of-row turns.

The previous guidance system suffered from uncertainties in the natural environment. Development of intelligent guidance algorithms, were impeded due to limited tracking ability in the vehicle. Also, expansion of capabilities meant significant rework of software in the system. The very complexity of the initial architecture, presented a hurdle for further development and maintenance of the system. An effort to address both these issues has been made through this research.

The DashBoard system can be expected to play a major role in scalability of this system. Expansion of the system's capabilities has been organized by means of devices and controllers. Also, as the existing software has been organized the same way,
maintenance becomes easier. As a result, the foundation for a robust guidance system has been arrived at.

An odometer component that presents open interfaces for development of intelligent algorithms has been made available for future development. This new tracking ability might prove crucial for improving robustness and efficiency of the system. With no special sensors needed, visual odometry is also a cost-effective technique.

7.1 Limitations

7.1.1 Software Architecture

Custom frameworks like the dashboard and the one developed for agricultural navigation in [17] could be useful for simple stand alone systems. But when these systems scale, or become a part of a bigger control system, properties like distribution, protection and CBSE become important. Large systems tend to be highly distributed and be developed and maintained by a group of people.

Currently the dashboard system does not have a distributed extension. Thus, any interface to another computer should be programmed by the user. Even in the navigation control system developed, the sensory processor (Pentium III/ Windows XP) communicates with the vehicle controller (Pentium II/ Linux) via serial interface. The main sensory processor sees the vehicle controller as a ‘device’ and the device class includes code for serial communication. A simple request-reply protocol has been developed for enabling communication between the systems. Scaling problems might arise in two scenarios — addition of components, addition of nodes.

The driving controller controls the vehicle actuators and reads the encoder. If we suppose a requirement that requires an additional parameter to be communicated between the two nodes arises, the communication protocol has to be extended or at least, programming effort in both the sender’s side and the receiver’s side has to be put in. When a new node, for instance, a hand-held monitoring node has to be connected to
the system, again the communication between the new node and the old nodes has to be programmed. Thus, networking might become a bottleneck for scalability.

Given that the dashboard framework is object-oriented, the existing framework could be extended to support distribution of component controllers. One major step toward distribution will be to figure a transparent and efficient method to transfer data from one component to another. By being transparent, the individual components offer a uniform programming interface, independent of which node they are on. Currently, communication between components is enabled by exposing pointers to a single memory location, where data from sensors is continuously filled in. This shared-memory approach offers simplicity of programming and run-time efficiency, but does not extend to a distributed environment. When components are distributed on a network they are no more part of the same address space.

7.1.2 Odometry

The Kalman filtered visual odometry technique developed has been shown to be effective for distances less than 10 m, such as the end-of-row turning. From the results in table 5-5, it can be seen that the technique can produce errors up to 20% on featureless surfaces. Also, the same technique cannot be expected to measure accurately over longer distances, due to the accumulative nature of error.

7.2 Future Directions

Pursuing the vision of developing a robust guidance platform for citrus groves, further research on the subject can take multiple directions. Some of them are listed below:

- Distributed software framework
- Advanced Navigation
- Advanced Motion control

The utility of distribution has been discussed enough in the previous sections. Distribution will allow the guidance platform to inter-operate with implements and other
intelligent components in the system. Various distributed frameworks for robotics and automation are available as of today. Each framework can be unique in terms of core architecture, usability, portability and efficiency. The framework that is most suited for the application has to be selected after studying the advantages and disadvantages of individual frameworks. System developers also resort to developing custom frameworks, due to reasons of usability and reliability.

Being able to plan paths around obstacles and for situations other than just the alleyway and end-of-row, might help extend the guidance platform for a wide variety of applications. Such robust navigation methods can also reduce the amount of human intervention needed in guidance.

The current motion controller is a simple PID controller that doesn’t consider loop delay and saturation in the control loop. As a result, the actual path followed deviates from the generated path. This can be improved by using more faithful vehicle models, that include characteristics like delay and saturation. In [10], the vehicles, lateral and longitudinal slip is considered for the vehicle’s motion model. These factors could come into play when extending autonomous guidance to heavy off-road equipment.
APPENDIX A
USING THE DASHBOARD

This appendix is devoted to documenting the semantic details of the DashBoard framework described in Chapter 4. The requirements, design and applicability of the framework were discussed in detail in Chapter 4. This section of appendix, can be treated as a user manual for constructing components in the framework.

The DashBoard is an abstract interface separating the control logic from the underlying implementation. It can be viewed as a switching device similar to a telephone switching network and does not manipulate the data that it handles.

When the using the DashBoard, the user has to define the functions and data members of a component(or a device) and specify what resources are accessible for other components. The DashBoard will export all resources(data and control functions) that are marked exportable, and will make them accessible to other components in the system. The user’s role is limited to creating a device and marking which resources are exportable, rather than the user creating a group of globally accessible devices and presenting a unique interface for each device for the control logic to use. This creates two distinct problems,

1. Since devices are free to have their own unique interface, the development of control functions using these devices will need knowledge of all the device interfaces.

2. Since each device is statically connected to the control logic, simple tasks like enabling/disabling a particular device at runtime, or switching one device for another require considerably effort.

The DashBoard framework handles the first issue by abstracting all the devices available in the system and exposing an uniform interface for all devices. i.e., the device programmers are allowed to export data and services from a device and the DashBoard takes care of indexing these resources with unique user defined IDs. Thus accessing a device is a matter of reading or writing to a resource ID, and all the IO operations could be simplified to just a read or write in the control logic. The user describing the
control logic has to either read or write a resource using its ID, instead of having to have unique access functions for each type of device. The *DashBoard* takes care of routing and translating the read/write calls to the appropriate devices. The control logic simply accesses resources and all other IO-specific code like reading hardware ports, waiting, synchronization etc., are included in the device class. This isolation between the control logic and the underlying implementation makes the control logic portable and maintainable.

With the *DashBoard* control logic is not statically connected to resources in devices. Every request from the control logic is associated with a resource-ID and is resolved by the *DashBoard* during runtime. Thus, it is possible to dynamically enable/disable and even switch resources by re-associating the devices under the *DashBoard*, with no changes to the control logic.

Instead of allowing the user to place code in any fashion, the framework regulates the placement of device specific code and control logic. The user is first expected to program all the devices and link them to the dashBoard object. The exported resources from these devices will be accessible for reading/writing through the dashboard’s get/set interface. The control logic code has to be placed inside specialized objects called controllers and has to be linked to the dashboard. The controller objects created this way also gain the ability to access all resources under the framework using the same get/set interface. The interaction between a user who intends to create a multi-controller software control system, and the framework is reduced to the following three situations.

- Add I/O devices
- Add controllers
- Access resources

Usage information on each of the above functions are detailed in the sections below.
Adding an I/O device

Procedure

This section assumes that the user already has the software driver required for communicating with the hardware device. For instance, say a GPS has to be integrated into the system. The user is responsible for programming the communication primitives for the the GPS. For the purpose of this illustration, let us assume that a certain GPS offers the following three readings — latitude, longitude, velocity and offers a service for setting the device configuration flags. Let the functions be wrapped into a simple C++ object CGPS that models the actual GPS device.

**GPS Class Declaration:**

```cpp
class CGPS{
public: //GPS services
    double lat;
    double lon;
    double velocity;
    int setConfig(long flags);

private: //Device Internals
    double getLatitude();
    double getLongitude();
};
```

The member variables lat, lon always contain the latest GPS readings and the setConfig() function allows the user to configure the hardware device, by sending in setting bits. The software refers to data members of a device as Parameters and the services as Control functions.

To integrate the above class into the DashBoard system the following steps have to be followed.

**Step 1** The device class (CGPS) has to inherit the CDbfDevicePlugin base class.

**Step 2** Each input function that performs a certain control action on the device must be declared using the DECLARE_CONTROL_FN declaration macro inside the class’ scope.
Step 3 In the class' constructor, the member-variables that hold the readings have to be registered for export using any of the following functions - passiveParamExport() or activeParamExport(). To associate a member function as an updater function for a certain active parameter, the updater function must be a public member of the same class, and has to be declared using the DECLARE_UPDATER_FN macro.

Step 4 The CDbfDevicePlugin defines virtual functions for device initialization and de-initialization. The sub-class can optionally implement these functions. These functions will be called when the user explicitly requests the dashboard to attach or detach the device. So, it is desirable to add one time device initialization operations like port settings, open/close comm ports etc. Placing this code in the constructor means, the ability to initialize system hardware on request is lost, since all device constructors will be called irrespective of whether the device is attached or not.

Step 5 In a final plugging-in step, the class has to be instantiated into the DashBoard system in the DbfPluginHolder.cpp file, using the PLUG_IN macro. This macro contains code for creation of the device class and export and listing of member data and functions in the DashBoard.

Syntax

1. DECLARE_CONTROL_FN(class, function, returntype, argtype): Declare a control function.

   class Name of the class in which the function is declared in.

   function Name of the control function that is being declared.

   returntype Control function return type.

   argtype Control function argument type [The number of arguments passable to a control function is limited to one].

2. void passiveParamExport(DbfParamID ID, string name, string desc, void* ptr)

   : Allows export of passive member data. Passive members are member variables that do not require explicit update calls from the DashBoard.
ID    A unique parameter ID using which, clients can address this parameter. [Parameter IDs have to be created in DBFParams.h, for each new parameter exported]

name  A text name representing the parameter, for display purposes

desc  A text description of the parameter like purpose, units etc., for display purposes

ptr   Pointer to the member data that is exported

3.   void activeParamExport(DbParamID ID, string name, string desc, void* ptr, updater fn): Allows export of active member data. Active members are member variables that require update calls from the DashBoard. The update call is triggered before each time the parameter is accessed.

ID    A unique parameter ID using which, clients can address this parameter. [Parameter IDs have to be created in DBFParams.h, for each new parameter exported]

name  A text name representing the parameter, for display purposes

desc  A text description of the parameter like purpose, units etc., for display purposes

ptr   Pointer to the member data that is exported

fn    An updater function that has to be called before reading the parameter

4.   DECLARE_UPDATER_FN(class, function, returntype, argtype): Declare a control function.

class Name of the class in which the function is declared in.

function Name of the control function that is being declared.

returntype Control function return type.

argtype Control function argument type [The number of arguments passable to a control function is limited to one].

Example

The CGPS class declared as a simple class containing device IO functions in the previous code snippet, to be integrated into the DashBoard system has to have following code changes. Let's assume that the members lat and lon are updated by calling the appropriate update functions and the member velocity is being periodically updated by
an interrupt mechanism within the object CGPS. Thus in *DashBoard* terms, lat and lon are 'active' parameters and velocity is 'passive'.

**GPS IODevice class declaration:**

```cpp
#include "DbfDevicePlugin.h"

class CGPS: public CDbfDevicePlugin {

public: // GPS services
    double lat;
    double lon;
    double velocity;

    DECLARE_CONTROL_FN(CGPS, setConfig, int, long);
    DECLARE_UPDATER_FN(CGPS, getLatitude, double, dbfVoidPtr);
    DECLARE_UPDATER_FN(CGPS, getLongitude, double, dbfVoidPtr);

private: // Device Internals
};

**Constructor definition:**

```cpp
CGPS::CGPS(void) {
    // Export parameters
    activeParamExport(DBF_PID_GPS_LATITUDE, "Latitude",
                      "Latitude from GPS (deg)",
                      &lat, getLatitude);

    activeParamExport(DBF_PID_GPS_LONGITUDE, "Longitude",
                      "Longitude from GPS (deg)",
                      &lon, getLongitude);

    passiveParamExport(DBF_PID_GPS_VELOCITY, "Velocity",
                       "Ground velocity from GPS (m/s)",
                       &velocity);

    // Export control
    controlExport(DBFC_PID_GPS_CONFIG, "Config bits",
                  "Set GPS device configuration",
                  setConfig);
```
// Set device name and description strings
// This will be used for display purposes
m_DeviceName = "Sample GPS";
m_DeviceDescription = "Measures the current position in \
   Lat(deg) and Lon(deg)";

//
//
}

dbfErr CGPS::initialize(void)
{
   // Open GPS port
   .
   .
   // Initial settings
   .
   .
}

dbfErr CGPS::deinitialize(void)
{
   // Close GPS port
   .
   .
   // Reset hardware to original state
   .
   .
}

The user also has to provide definitions for all the updater and controller functions declared in the class. Also, the user has to make sure the parameter and control IDs, used in the export functions are defined in the "DBParams.h" header, under the appropriate enums for parameters and control functions. Finally, to attach this class to the Dashboard system, the class has to be instantiated using the PLUG_IN(<class name>) macro in the DbfPluginHolder.cpp placeholder file. For the GPS, adding PLUG_IN(CGPS); in the appropriate section for IO devices is enough for attaching to the system. Upon
invoking the DashBoard, all classes thus attached to the dashboard will be instantiated and then the appropriate devices could be selectively initialized upon request.

**Adding a Controller**

In the *DashBoard* system, a controller is a software component, that has the following properties.

- Loops infinitely at a set frequency
- Accesses IO devices periodically
- Can export data and control functions themselves

**Procedure**

Any controller class must be a sub-class of `CDbfControllerPlugin` class. Using this class, the user has to write the main control loop code under a specific function and the framework makes sure that the code is called repeatedly with a desired frequency, for the controller’s lifetime. The base-class defines the following set of pure-virtual functions which every controller is required to implement.

1. `dbfErr controllerMain(dbfVoidPtr)`
2. `dbfErr controllerInit(dbfVoidPtr)`
3. `dbfErr controllerDeInit(dbfVoidPtr)`

The *controllerMain* function will be called repeatedly by the DashBoard system, thus the user has to implement the main controller loop under this function. The *controllerInit* and *controllerDeInit* functions are called only once during the start and end of the controller’s life-time. These functions could be implemented for setting up the controller, initialization of variables etc.

In the controller’s member functions, all parameters and control functions exported IO devices plugged into the system, are accessible. The controller programmer has to use simple function calls to access the exports by their respective IDs. The semantics of the resource access calls with be discussed in the next section.
Since controllers might also have to be able to export internal measurements and support control functions, the CDbfControllerPlugin class extends all functionality from the CDbfDevicePlugin class discussed in the previous section. This allows the controller to be treated like an IO device as well. Thus all the data export and control export semantics described above is applicable to the controllers too. i.e., controllers can produce readings and be controlled by a controller at a higher-level. This property warrants the ability to build multi-level controllers using the DashBoard.

Example

Consider a simple example of a PID velocity controller, using velocity readings from the GPS class created above as a input and controlling the vehicle’s speed.

Controller Declaration:

```cpp
#include "DbfControllerPlugin.h"

class CruiseController : public CDbfControllerPlugin
{

public:
    CruiseController();
    ~CruiseController();

    DECLARE_CONTROL_FN(CruiseController,setCruiseSpeed, dbfErr,
                        dbfVoidPtr);

    dbfErr controllerMain(dbfVoidPtr);
    dbfErr controllerInit(dbfVoidPtr);
    dbfErr controllerDeInit(dbfVoidPtr);

private:
    double Kp;
    double Ki;
    double Kd;
    CPIDController pid;

};

Controller Definition:

CruiseController::CruiseController(void) {

controlExport(DBFCID_CRUISE_SPEED,"Cruise Speed",
    "Set Cruise controller set-point",
    setCruiseSpeed);

CruiseController::~CruiseController(void)
{
}

dbfErr CruiseController::setCruiseSpeed(dbfVoidPtr speed)
{
    pid.setPoint(double (*speed));
}

dbfErr CruiseController::controllerInit(dbfVoidPtr p)
{
    pid.setConsts(Kp,Ki,Kd);
    pid.setPoint(0);

    return DBFERR_SUCCESS;
}

dbfErr CruiseController::controllerDeInit(dbfVoidPtr p)
{
    return DBFERR_SUCCESS;
}

dbfErr CruiseController::controllerMain(dbfVoidPtr p)
{
    double measurement;
    double command;
    dbfErr err;

    //read velocity from GPS
    measurement = getDouble(DBFPID_GPS_VELOCITY);

    command = pid.getCommand(measurement);

    //Assume a hypothetical vehicle speed controller with
    //ID DBFCID_VEHICLE_SPEED
    err = set(DBFCID_VEHICLE_SPEED, command);

    return err;
As a final step the cruise controller class has to be instantiated using the macro PLUG_IN_CLR within the DbfPluginHolder.cpp file. The syntax for the macro is as below:

\texttt{PLUG\_IN\_CLR(CNAME, CLRID, FREQ)} - Plug in a controller object into the \textit{DashBoard} system.

- \textbf{CNAME} : Name of the controller class
- \textbf{CLRID} : A unique ID which the dashboard users can use to address the controller
- \textbf{FREQ} : The frequency with which the controller’s controllerMain function will be called, in Hz

Using the above semantic, the instruction for instantiation for the CruiseController example would be : \texttt{PLUG\_IN\_CLR(CruiseController, DBFCLRID\_CRUISECTRL, 20)}. This statement makes sure that a cruise controller object, defined in the previous code segments, will be instantiated and will be run at a frequency of 20 Hz.

\textbf{Resource Access}

During runtime a single \texttt{CDashBoard} object manages all the devices, controllers and their associated resources. Users who create the object can use its services to access the underlying controllers and devices. The following primary services are offered by the object

- Attach detach devices and controllers
- Get and Set parameters and control functions

\textbf{Syntax}

The detailed list of services offered by the \texttt{CDashBoard} class are as below:

1. \texttt{dbfErr getParamPtr(DbfParamID paramID, DbfParamPtr* paramPtr)} - Get an imported parameter. Dashboard users can call this function to access any sensor exported parameter. This function doesn’t call updaters, as it is intended to be plain.
   \begin{itemize}
   \item \texttt{paramID} - The parameter ID(the key for searching)
   \item \texttt{paramPtr} - Pointer to a location where pointer to the requested resource is stored
   \end{itemize}
   \begin{itemize}
   \item return - Errors incurred while fetching the requested parameter
   \end{itemize}
2. `dbfSInt32 getInt(DbfParamID paramID, dbfVoidPtr updaterData = NULL)` -
Get an imported integer parameter
Dashboard users can call this function to access any sensor exported parameter. The call is cascaded to an updater function if defined by the driver
- **paramID** - The parameter ID (the key for searching)
- **updaterData** - A means of passing data to the updater function
- **return** - Integer parameter value

3. `dbfFloat getFloat(DbfParamID paramID, dbfVoidPtr updaterData = NULL)` -
Get an imported floating point parameter
Dashboard users can call this function to access any sensor exported parameter. The call is cascaded to an updater function if defined by the driver
- **paramID** - The parameter ID (the key for searching)
- **updaterData** - A means of passing data to the updater function
- **return** - Floating point parameter value

4. `dbfDouble getDouble(DbfParamID paramID, dbfVoidPtr updaterData = NULL)` -
Get an imported floating point parameter
Dashboard users can call this function to access any sensor exported parameter. The call is cascaded to an updater function if defined by the driver
- **paramID** - The parameter ID (the key for searching)
- **updaterData** - A means of passing data to the updater function
- **return** - Floating point parameter value

5. `dbfErr updateParam(DbfParamID paramID, dbfVoidPtr updaterData = NULL)` -
Update a particular parameter (doesn't return parameter)
Updates a listed parameter by calling the defined 'updater' fn.
- **paramID** - The parameter ID (the key for searching)
- **updaterData** - A means of passing data to the updater function
- **return** - Errors incurred while fetching the requested parameter

6. `dbfErr updateParam(DbfParamID paramID)` - [Overloaded] Update a particular parameter (doesn't return parameter)
Updates a listed parameter by calling the defined 'updater' fn.
- **paramID** - The parameter ID (the key for searching)
- **return** - Errors incurred while fetching the requested parameter

7. `dbfErr set(DbfControlID controlID, dbfVoidPtr controlArg = NULL)` -
Execute a control function
Dashboard users can call this function to access any actuator-exported control function.
- **controlID** - The control ID (the key for searching)
controlArg - Pointer to a structure that can be passed to the control function (the dbfVoidPtr argument is a means of passing parameters to a variety of functions with a unified interface)

return - Errors incurred while executing a requested control function

8. dbfErr set(DbfControlID controlID, dbfSInt32 integerArg) - Execute a control function [integer arg overload]
Dashboard users can call this function to access any actuator-exported control function.
controlID - The control ID (the key for searching)
integerArg - Pointer to a structure that can be passed to the control function [makes user programs look better]
return - Errors incurred while executing a requested control function

9. dbfErr set(DbfControlID controlID, dbfFloat floatArg) - Execute a control function [float arg overload]
Dashboard users can call this function to access any actuator-exported control function.
controlID - The control ID (the key for searching)
floatArg - Pointer to a structure that can be passed to the control function [makes user programs look better]
return - Errors incurred while executing a requested control function

10. dbfErr start(DbfControllerID controllerID) - start an automatic controller
Dashboard users can call this function to start any plugged-in control function.
controllerID - The controller ID (the key for searching)
return - Errors incurred while executing a requested control function

11. dbfErr stop(DbfControllerID controllerID) - stop a running automatic controller
Dashboard users can call this function to stop any plugged-in control function currently running.
controllerID - The controller ID (the key for searching)
return - Errors incurred while executing a requested control function

12. dbfUInt32 getFreq(DbfControllerID controllerID) - find the frequency with which a controller is running
Returns the frequency with which a controller is set to run.
controllerID - The controller ID (the key for searching)
return - Frequency

13. dbfBool isRunning(DbfControllerID controllerID) - find controller status
Checks whether a controller is currently running or not.
controllerID - The controller ID (the key for searching)
return - true if running, false if stopped
14. const std::vector<InputCapability * >* getInputCapabilities(void)
15. const std::vector<OutputCapability * >* getOutputCapabilities(void)
16. const std::vector<ControlCapability * >* getControlCapabilities(void)
   - Check the resources available from the dashboard. Each function above returns an array of 'capability' structures. This function is helpful for dashboard's clients for listing details of resources available in the system. Note that device resources are listed no matter they are attached or not.

**Capability Definition:**

//Measurement Capability
typedef struct Icapabilities_struct
{
   DbfParamID paramID;
   string paramName;
   string paramDesc;
   string deviceName;
   dbfDeviceInstID deviceInstID;
}"InputCapability;

//Control capability
typedef struct Ocapabilities_struct
{
   DbfControlID controlID;
   string controlName;
   string controlDesc;
   string deviceName;
   dbfDeviceInstID deviceInstID;
}"OutputCapability;

//Controller capability
typedef struct Ccapabilities_struct
{
   DbfControllerID controllerID;
   string deviceName;
   dbfDeviceInstID deviceInstID;
}"ControlCapability;

return - List of all capabilities of the requested category

17. dbfErr attach(dbfDeviceInstID deviceID) - Attach a device to the dashboard.
Attaching a device means the device's `initialize` function will be called and the resources (parameters and control functions) contained within the device will be available for access upon successful completion of attach.

deviceID - unique ID of the device that has to be attached.

return - Errors encountered during device initialization.

18. `dbfErr detach(dbfDeviceInstID deviceID)` - Detach a device from the dashboard

Detaching a device means, the device's `deinitialize` function will be called and the resources exported from the device will be unavailable.

**Example**

The services offered by the dashboard have been designed for constructing applications on top of it. In this example, a velocity control application is developed on top of the *Dashboard*. The components of the *DashBoard* — CGPS and CCruiseController have already been developed in the previous sections. Though, the application will not have to know about these components classes, the resource IDs declared in the previous examples will be used in the main application.

**Application Code:**

```cpp
#include "Dashboard.h"
#include <iostream>
#include <vector>

namespace std;

int main(void)
{
    CDashboard db;

    const std::vector<InputCapability*> * pInputsList;
    const std::vector<OutputCapability*> * pOutputsList;

    //Populating the capabilities list
    pInputsList = db.getInputCapabilities();
    pOutputsList = db.getOutputCapabilities();

    // Application Code...
```
inputCapsReader inReader = pInputsList->begin();
outputCapsReader opReader = pOutputsList->begin();

//Listing and attaching the resources one by one.
//The devices can be attached selectively too.
cout<<"List ofMeasured params:";
for( ; inReader != pInputsList->end(); inReader++)
{
    cout<<(*inReader)->paramName<<"["<<(*inReader)->paramID<<"],";
    db.attach((*inReader)->deviceInstID);
}

cout<<"List of Controls:";
for( ; opReader != pOutputsList->end(); opReader++)
{
    cout<<(*opReader)->controlName<<"["<<(*opReader)->controlID<<"],";
    db.attach(*opReader)->deviceInstID);
}

//Start the the PID speed controller
db.start(DBFCLRID_CRUISECTRL);

cout<<"Initial Velocity = "<<db.getDouble(DBFPID_GPS_VELOCITY)<<endl;

//set vehicle speed
db.set(DBFCID_CRUISE_SPEED, 10);

//Wait till the vehicle picks up
wait(2000);

//Print increased velocity
cout<<"Current Velocity = "<<db.getDouble(DBFPID_GPS_VELOCITY)<<endl;

db.stop(DBFCLRID_CRUISECTRL);
return 0;
Figure B-1. UML Representation of the Guidance application Using Dashboard
Figure B-2. UML Representation of DashBoard components
REFERENCES


BIOGRAPHICAL SKETCH

Sundar Subbiah graduated from Bharatidasan University with a Bachelor of Engineering degree in 2003. He received a Master of Science degree in Computer and Information Science Department from the University of Florida in Spring 2010. Before graduate studies, he has been working in the software industry for three years. His research interests include Robotics and Distributed Real-time Embedded Systems. He aspires to develop newer and innovative autonomous systems for the growing robotics industry. He also enjoys camping, gardening and other outdoor activities.