Autonomous robot software development using simple software components

Thomas M. Burke *ab, Chan-Jin Chung a

a Dept. of Math and Computer Science, Lawrence Technological University
Southfield, MI 48075, USA
b Autodesk – Manufacturing Solutions Division, Novi, MI 48375, USA

ABSTRACT
Developing software to control a sophisticated lane-following, obstacle-avoiding, autonomous robot can be demanding and beyond the capabilities of novice programmers - but it doesn’t have to be. A creative software design utilizing only basic image processing and a little algebra, has been employed to control the LTU-AISSIG autonomous robot - a contestant in the 2004 Intelligent Ground Vehicle Competition (IGVC). This paper presents a software design equivalent to that used during the IGVC, but with much of the complexity removed. The result is an autonomous robot software design, that is robust, reliable, and can be implemented by programmers with a limited understanding of image processing. This design provides a solid basis for further work in autonomous robot software, as well as an interesting and achievable robotics project for students.

Keywords: Robotics, Autonomous Robot, Computer Science, Education, Simple, Software, KISS, Image Processing

1. INTRODUCTION
An autonomous robot is a robot that can sense the surrounding environment, navigate safely through it, and perform specific tasks, all without human intervention. Autonomous robot software design is an emerging field and the subject of considerable research. In the not-so-distant future, autonomous robots are expected to reduce risk by replacing humans by operating in hostile environments such as burning buildings, bomb searches, and war zones.

Since 1993, the U.S. Department of Defense, S.A.E, General Motors, and others have sponsored the Intelligent Ground Vehicle Competition (IGVC). The IGVC is a friendly, 3-day competition that encourages research of autonomous robots and provides a common metric to compare different robot designs.

The Lawrence Technological University - AISSIG robot competed in the autonomous challenge portion of the 2004 IGVC. In the Autonomous Challenge, robots have 9 attempts to navigate a course consisting of lane lines and obstacles. A representative section of the course appears in Fig. 1. If the robot crosses a lane line or collides with an obstacle, the robot must terminate that attempt. Teams are ranked based upon how far along the course their robots successfully travel.

![Image of course](image_url)
Since the operating environment of the autonomous robot is constantly changing, the software used to control these robots must be robust and reliable. Adherents to the KISS principle of software design argue that in order to achieve this goal, autonomous robot software designers should “Keep It Simple, Stupid”. The LTU-AISSIG team of Vachon, Burke and Nair did not initially take this approach [5]. The first iteration of the AISSIG software design made use of artificial intelligence and computer vision techniques including: Evolutionary Computation using ES (1+1) with 1/5th rule [2], Artificial Neural Net [2], Image Texture Classification [1], and Fuzzy Logic Controller [2].

While these techniques worked well during testing of the AISSIG robot, conditions on the IGVC course were different and more challenging than were seen on the test course at Lawrence Tech. In addition, the weather forecast called for periods of sun, clouds, and rain on the day of the Autonomous Challenge – so there was no way to know what the lighting and course conditions would be when the AISSIG robot made its nine attempts. What was needed was a nimble software design that could easily be tweaked between attempts without a lot of time spent training neural nets and testing major changes. This required simple components, well understood by the designers, which would respond intuitively to minor changes between course attempts with little or no testing.

Working to the AISSIG team’s advantage was that the existing software design was modular, and had components that communicated with each other exclusively through well-defined interfaces. As needed, the complex algorithms in software components were able to be replaced by simpler, but functionally equivalent, algorithms. The result was a more robust and reliable software design than before simplification.

2. HARDWARE DESIGN

The software design presented here is not hardware dependent, but it is beneficial to provide context by briefly explaining the AISSIG robot hardware.

The LTU-AISSIG robot is based on the Tedder [4] design, but modified to improve stability [5]. The robot has three wheels - one passive caster wheel in the front and two drive wheels in the rear. Two motors power the drive wheels independently in response to instructions from an onboard laptop computer to a motor controller connected via a parallel port. If both wheels are powered at the same level, the robot will move in a straight line. If the power levels are different, the robot will turn, providing a steering ability similar to an army tank.

The lone sensor used in the autonomous challenge is a DV camcorder that delivers real time video to the laptop via firewire. The laptop computer uses the Windows 2000 operating system and runs the robot control software in a Java Virtual Machine (JVM).
3. SOFTWARE DESIGN

3.1 Overview
The goals of the autonomous vehicle to stay within the lane lines and avoid obstacles, cannot be accomplished without knowing where the lane lines and obstacles are located in relation to the robot. The video provided by the DV camcorder is the source of this information and our image processing is designed to discover the region of the image that represents all areas of the image where the vehicle could potentially move without crossing a lane line or colliding with an obstacle. We call this region the “clear path”.

The strategy for finding the clear path is straightforward. We split the image into rectangular multi-pixel segments for processing. Since we know that the lane lines are white and the obstacles are white and orange, we use color information in the image to determine which segments represent lane lines, obstacles, or neither. Also, we know that our vehicle is inside of all lane lines and short of all obstacles. We use this knowledge to eliminate segments outside of the lane lines and past the obstacles from consideration. The contiguous segments that remain constitute the clear path.

Once the clear path has been determined, a heuristic algorithm is applied to determine the optimal direction and speed for our vehicle to travel to stay within the lane lines and to avoid the obstacles. That direction and speed are then converted to power values which are applied to the wheels causing the vehicle to move.

Occasionally, conditions are such that one or more of the modules encounter inconsistencies or errors that prevent them from functioning as designed. When this situation occurs, control of the vehicle is sent to a module designed to deal with unusual situations. The vehicle stays under this module’s control until such time that the condition no longer exists. These steps are performed in a loop until some stop condition is reached. This program flow is outlined in Fig. 3, and a description of each component follows.
3.2 Image Capture
The video frames used as input are acquired using a standard DV camcorder and transferred to the computer in real time thru a fire-wire (IEEE 1394) port. In order to capture the images and present them to our Java environment for processing, the JavaDV [3] library is used. JavaDV allows Java programs to capture images from a number of video devices. Although similar functionality is available via Java Media Framework, JavaDV supports all DirectShow devices including DV camcorders.
3.3 Noise Removal

The first processing step that is performed on the image is to apply a low-pass filter [1] to remove noise as well as to make the color change from one pixel to the next less pronounced. This is important when trying to discern between a white line painted on green grass by color. The paint that is used does not evenly cover the blades of grass, and at camcorder resolutions, some of the pixels that should clearly be classified as white, turn up as green. The low-pass filter goes a long way toward eliminating this problem by averaging a pixel's color with that of its neighbors, thereby making all the “white” pixels easier to detect.

3.4 Pixel Classification

The goal of the pixel classification is to determine, by color, whether each pixel represents a location that the vehicle is allowed to go, such as grass, or a location where the vehicle is forbidden to go, such as a lane line or obstacle. Since we know that the locations that are forbidden are white or orange, and the locations that are allowed are not white or orange, pixel classification simplifies to determining whether a pixel is white, orange, or some other color.

While problem of pixel classification appears to be simple enough to solve, getting it to work reliably is quite difficult. It is difficult because under different lighting conditions, the RGB value of a “white” pixel (or orange for that matter) does not remain constant - camcorders automatically adjust the brightness, contrast, and color balance of an image before making it available thru the fire-wire port. In addition, the RGB values of white in the sunshine are different than the RGB values of white in the shade. Therefore, white is not a single color, but a range of colors. The main challenge of pixel classification is to describe this range of colors in a way that allows pixels to easily be tested as to whether or not they are white. This is done by segmenting the color space and determining which segment contains the pixel color.
Segmenting the RGB color space into white, orange and other is straightforward. If you think of RGB as a Cartesian space then all of the colors can be thought to be contained by a cube with black at the origin (R = G = B = 0) and white at the opposite vertex (R = G = B = 1.0) This RGB color space is shown in Fig 6. The region of this color cube that is considered white is represented by a cylinder whose axis lies on the line R = G = B, and has a height and radius discovered through experimentation. This cylinder encloses the classic white along with many shades of gray. In our operating environment, these gray shades indicate a white line or obstacle in one of a variety of lighting conditions, including shade.

Another way to construct a color space is to use cylindrical coordinates with hue corresponding to an angle of revolution, saturation as radius, and brightness as height. Since the minimum value of brightness generates black for all hue and saturation values, the HSB space is usually represented as a cone instead of a cylinder – as in Fig 7.
The region of the HSB space that is considered white is a cylinder along the main axis containing all brightness values greater than a brightness threshold, and with a saturation value less than the maximum saturation value. The values for brightness threshold and maximum saturation are determined through experimentation. This cylinder encloses the same gray values that the rotated cylinder did in RGB color space, but in a way that is much easier to code and to modify when overall lighting conditions change.

For the purposes of the IGVC, we are also interested in whether or not a pixel is orange. This classification is also easy to perform in HSB space. Orange pixels are those which have a hue value in the range of -30 to 30 degrees.

The color classification for each pixel is recorded in a two-dimensional array of booleans called the pixel classification array. Elements of the pixel classification array are set to true if the corresponding pixel is classified as white or orange, and false otherwise. A rendering of a typical pixel classification array is shown in Fig 8.

![Fig 8 – Frame before and after pixel classification. All pixels classified as white or orange appear white. All other pixels appear black.](image)

### 3.5 Region Classification

Depending on the conditions, there may be a substantial amount of noise in the pixel classification array. This is because when a blade of grass is painted, it is not always covered entirely by the paint. This results in a whitish green color being detected by the camcorder. This same color will be detected if the blade of grass is angled to the sunlight in such a way that much of the white light from the sun is reflected into the camera. Therefore, pixel classification alone is not enough to distinguish lane lines and obstacles from the grass in an image.

If you look at Fig 8, you are able to see an example of this noise near the right and left edges of the pixel classification array. As humans, we do not consider this noise part of the lane line because the density of white-classified-pixels is low. Likewise, it is the white pixel density that the region classification component uses to classify regions of the pixel classification array.

The 160x120 pixel classification array is segmented into 16x12 regions of 10 pixels square. The number of white pixels contained in each region, are counted. If that count exceeds some threshold, the entire region is considered white. The threshold value is determined through experimentation in expected lighting conditions. A threshold value of 30% was used with good success on the IGVC course.

The output of the Region Classification module is a 16x12 array of booleans. A true value indicates that the corresponding region of the input image contains a lane line or obstacle. A false value indicates that the region may be part of the clear path. A rendering of the region classification array is shown in Fig. 9.
3.6 Clear Path Determination
As stated earlier, the clear path is the region of an input bitmap where the vehicle can safely move without crossing a lane line or colliding with an obstacle. The clear path module attempts to extract the clear path from the region classification array.

It is important that we have a concise method of describing the clear path. For this purpose we use a data set called the “clear path vector”. The clear path vector is an array of 16 floats, each with a magnitude of between 0.0 and 1.0, and each representing the amount of clear path in the corresponding column in the Region Classification Array.

There a couple of important bits of a priori knowledge that will help us do construct the clear path vector from the region classification array. First, we know that the robot is currently between the lane lines. If this were not the case, the robot would already have failed and the attempt terminated. Next we know that we can eliminate from the clear path all regions in our field of view that are further away from the robot than a lane line or obstacle. This is true since the robot would first have to cross a lane line or strike an obstacle to reach that region, and that is not allowed.

With this knowledge, an element of the clear path vector can be calculated by a simple algorithm. For a given column in the region classification array, count the number of clear regions between the robot and the closest non-clear region. Divide this count by the total number of rows, and you will get a normalized representation of how much clear path exists in that column. Repeat for all columns. The clear path vector for the region classification array in Fig. 9 is shown below:

Clear Path Vector = [1.0 ,1.0, 1.0, 1.0, .75, .75, .75, .5, .41, .25, .17, .17, .08, 0.0]

Viewing a rendering of the clear path vector is useful in understanding just what the clear path vector represents. Fig. 10 shows a bar-graph type rendering of the clear path vector along with a rendering of the region classification array that it was generated from. You will notice that if you superimpose the rendering of the clear path vector over the region classification array, all regions in which the robot is allowed to travel are painted by the clear path vector rendering. This exercise illustrates how the clear path vector efficiently represents the clear path.

3.7 Clear Path Post Processing
Due to the noisy nature of our input, it is likely that a region will occasionally be incorrectly classified. When this happens, it will result in a single bad value in the clear path vector. The negative effects of such a bad value can be
minimized by applying a low-pass filter to the clear path vector. This is done by replacing the clear path vector elements with a 5 element rolling average with the value of elements 0, 1, 14, and 15 being set to 0. Fig. 11 shows a clear path vector before and after post processing.

3.8 Robot Action Determination

Once the clear path has been represented by the clear path vector, we must determine the direction in which to steer the robot in order to minimize the likelihood of leaving the lane or hitting an obstacle. A quick inspection of the post-processed clear path vector shows that the element with the greatest magnitude “points” in the optimal direction. This is almost always the case. Exceptions occur when more than one element is equal to the greatest magnitude. In this case, choosing the element with the greatest magnitude and closest to index 7 or 8 (straight ahead) results in a direction that steers clear of danger while minimizes the turning radius.

3.9 Robot Action Execution

The described method of determining the robot action produces an index into the clear path array to represent the desired robot direction called the “direction index”. We must then convert the direction index into commands that will make the robot travel in that direction. Since the AISSIG robot uses tank-type differential steering, the control mechanism is to send power levels to the motors attached to the right and left wheel. The same power level sent to both wheels will cause the robot to go straight, while different power levels sent to the left and right wheel will cause it to turn.

The trick is to send the right power level to each wheel in order to turn it an amount that corresponds to the supplied Direction Index. An array of motor power levels is used to perform this mapping, with the level values determined thru experimentation. An example mapping array for the AISSIG robot appears in Fig 13.
<table>
<thead>
<tr>
<th>Direction Index</th>
<th>% Power Left Wheel</th>
<th>% Power Right Wheel</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>90</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>70</td>
<td>90</td>
</tr>
<tr>
<td>7</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
<td>70</td>
</tr>
<tr>
<td>10</td>
<td>90</td>
<td>55</td>
</tr>
<tr>
<td>11</td>
<td>90</td>
<td>40</td>
</tr>
<tr>
<td>12</td>
<td>90</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>90</td>
<td>20</td>
</tr>
</tbody>
</table>

Fig 13 - Contents of array used to map Direction Index to motor power levels

For example, if the direction index was calculated to be 4, the power level for the left wheel is set to 40 and the power level for the right wheel is set to 90 – causing the robot to turn to the left.

To increase the likelihood of staying within the clear path, the robot should reduce its speed when it is in close proximity to lane lines or obstacles. This proximity can be gauged by the magnitude of the clear path vector element that represents our desired direction - the smaller the magnitude, the closer the lane line or obstacle. Since the clear path vector elements are in the range of 0.0 thru 1.0, multiplying the motor power level by the magnitude before sending it to the motor controller has the desired result.

In the event that one or more of the described modules fails to generate meaningful output, the control software will want to have a fallback behavior for the robot. One desirable behavior, is for the robot to not move for a second or two. It takes a camcorder some time to adjust to sudden changes in lighting conditions such as glare, camera flash, or a cloud moving in. In these cases, the robot will be ready to go again after a slight pause.

Sometimes an error condition is caused by a lane line or obstacle directly in front of the robot. This could be the result of the robot moving into a blind corner, or not executing a sharp enough turn to clear an obstacle. Whatever the cause, when this condition is detected the robot will need to backup, turn slightly and then proceed normally. Maintaining a history of recent robot actions can be useful in determining which direction to turn.

4. CONCLUSION

Complicated problems do not always require complicated solutions. The control software for the LTU-AISSIG robot was able to travel nearly 250 feet in the IGVC autonomous challenge before encountering standing water and leaving the course. This distance placed the LTU-AISSIG robot, 6th amongst the 28 robots that entered the competition. The relative success of this simple software design, in what is a decidedly non-simple application, provides validation of the KISS approach and provides a foundation upon which to build for future IGVC events.
REFERENCES

5. Vachon, Burke, Nair, AISSIG – Intelligent Ground Vehicle Competition Design Report, 2004

* tom@tomburke.net