SUNFEST Technical Report TR-CST01NOV04, Center for Sensor Technologies, Univ. of Pennsylvania, Philadelphia, PA, 2004.

University of Pennsylvania
Center for Sensor Technologies

SUNFEST

NSF REU Program
Summer 2004

THE IMPLEMENTATION OF THE SEGWAY ROBOTIC MOBILITY PLATFORM (RMP) FOR AUTONOMOUS NAVIGATION

NSF Summer Undergraduate Fellowship in Sensor Technologies
Benjamin Bau (Department of Electrical Engineering and Computer Science)-Massachusetts Institute of Technology
Advisor: Professor Vijay Kumar and James Keller

ABSTRACT

A difficult problem in robotics is enabling a robot to navigate autonomously through a previously unexplored indoor environment. Part of this difficulty stems from the stringent requirements on the mobility and the power of a robotic platform. It is necessary that the platform has sufficient maneuverability to operate in a cluttered area and that it has sufficient power to carry the necessary sensors to operate autonomously. The Segway Robotic Mobility Platform (RMP) combines unusually high mobility with the power to carry a sufficient number of sensors to navigate in an indoor environment. This paper discusses how, using a laser range finder, a stereo camera, and the Segway's on-board encoders, the Segway RMP can be enabled to perform tasks of autonomous navigation while navigating through an area with obstacles.

Table of Contents

1. INTRODUCTION

One of the chief goals in robotics is to enable a robotic platform to navigate autonomously through an area with many potential, previously unknown obstacles. Even with the aid of such technologies as the global positioning system (GPS) and a variety of complementary sensors, this problem remains largely unsolved for even outdoor remote applications (for example, none of the competitors in the DARPA grand challenge this past year successfully completed the course). Indoors, the amount of space available for a robot to maneuver is often significantly decreased and the absence of a GPS signal makes it extremely difficult to pinpoint the robot's location. In order to enable a robot to successfully explore and navigate through the floor of an unknown building, one must seek to balance mobility with the ability to carry a sufficient number of sensors for the task.

At the General Robotics Automation, Sensing, and Perception (GRASP) Laboratory at the University of Pennsylvania, we sought to accomplish this task using a new form of vehicle technology, the Segway, which is unusually maneuverable for the amount of weight it can carry. This paper describes the process of equipping a Segway Robotic Mobility Platform (RMP) to perform autonomous navigation by enabling it to locate a colored object indoors and drive up to it while navigating around unspecified obstacles. This lays the groundwork for enabling robots to autonomously explore and operate in unmapped buildings with varying obstacles. Section 2 of this paper describes the Segway RMP, a test product developed based on the Segway HT technology for use as a robotic platform in the MARS2020 DARPA funded research initiative. Additionally, Section 2 provides background on the Remote Objects Control Interface (ROCI)[1] used by the GRASP laboratory to control all of its robots including the Segway RMP. Section 3 recounts the problems encountered in mounting sensors on the Segway due to its peculiar mode of movement and the design of a dynamic sensor platform in order to resolve these issues. Section 4 briefly describes the work done to enable the Segway to follow brightly colored objects autonomously using a stereo camera. Section 5 provides a description of two different methods of obstacle avoidance using a laser range finder. Section 6 describes the results of the implementation and testing of the Segway RMP as an autonomous robot. Finally, Section 7 contains conclusions and recommendations for future work on this project.

2. BACKGROUND

2.1 Segway Robotic Mobility Platform

The GRASP laboratory at the University of Pennsylvania received the Segway RMP as part of the MARS2020 project, which is funded by DARPA. The Segway RMP is configured by the manufacturer to follow external commands for its speed and turn rate. These external commands are transmitted via an onboard laptop computer which communicates with the RMP via a CAN bus. The Segway is special in that it is an automatically balancing, electrically powered, two-wheeled vehicle. Simply put, the Segway remains upright by accelerating in the direction that its platform is falling. Testing has led us to conclude that on certain terrains with low friction surfaces (ice, sand, mud, and dirt) this balancing mechanism becomes a liability because the Segway's attempts to accelerate in order to stay upright result in slippage and even greater loss of control until it eventually falls. This has led us to conclude that the Segway is perfectly suited for indoor operations (rather than outdoors) where its biggest advantages, small foot-print (0.5 meters on a side), zero turning radius, and ability to carry well over one hundred pounds on its platform, make it perfectly suited for operating in a confined space. Whereas most robots equal to or superior to the Segway with respect to maneuverability in confined spaces are not powerful enough to carry the heavy sensors necessary for effective mapping and obstacle avoidance, the Segway's high payload capacity to size ratio makes it suited for the task of indoor autonomous navigation. Additionally, during indoor operation, the Segway rarely encounters the situations that make it most likely to lose traction, which could lead to loss of control and a crash. For example, indoors, the Segway is unlikely to encounter low-friction surfaces or bumpy terrain that interferes with its low-clearance [2].

The Segway RMP provided to the GRASP laboratory came with a simple program for sending commands to it and receiving data (*i.e.* wheel speeds and pitch angle) back from its encoders. A converter (wrapper) for this software was created in order to integrate it with the GRASP lab's software platform (see Section 2.2). Additionally, one of the first things I did upon arriving at the GRASP laboratory this summer was to write a simple user interface that allows the operator to see the commands being sent to the Segway and the readouts from the Segway's internal sensors. This interface (SegwayDashboard) also gave the user the ability to shut down the Segway, scale velocity and turn commands, and prevent commands from being sent to the Segway.

The University of Pennsylvania is not the only university currently working with the Segway RMP. Universities that are using the Segway RMP for similar purposes to the one in the GRASP laboratory include the Georgia Institute of Technology [3], Stanford University [4], and the University of Southern California [5]. In all three cases, these universities have mounted a SICK laser rangefinder on the Segway platform for the purposes of obstacle avoidance and other robotic applications such as terrain characterization. Likewise, this paper discusses how we mount and use a SICK laser rangefinder for obstacle avoidance. One dilemma that we address differently from these universities in mounting and operating the laser rangefinder and other sensors is in how we deal with the effect of the pitching of the Segway's platform on the observations made by the sensors mounted on it. Whereas, Georgia Tech corrects the error in distance measurements caused by this phenomenon by using pitch angle information provided from the Segway's internal encoders to mathematically correct for the Segway's pitching in software [3], we sought to resolve this issue by designing a new platform on which to mount sensors on the Segway (Section 3). Finally, while applications of the laser rangefinder might include terrain characterization, three dimensional area mapping, and supplying information to a remote, human operator, the main focus of the research covered in this paper is to equip the Segway RMP with an obstacle avoider effective enough to allow the Segway RMP to operate independently in a cluttered, unmapped indoor environment while attempting to accomplish a separate autonomous task.

2.2 Remote Objects Control Interface

Like the other robots in the MARS2020 project at the GRASP laboratory, the Segway is run using the Remote Objects Control Interface (ROCI) developed by the GRASP laboratory.

ROCI is a programming platform for robotic applications. Individual processes in ROCI are programmed as modules. For example, there is a module that receives commands from a joystick and a separate module that sends commands to the Segway. This means that modules can be used in a variety of combinations for various tasks. Modules are connected by pins that are responsible for transmitting data. These connections between modules are defined by XML files called tasks. Additionally, pins can connect various tasks allowing connections between processes running on different machines. One of the major advantages of ROCI is the extensibility of applications constructed within it and the reusability of their component modules. In short, since robotic applications are defined by which modules are connected, it is relatively simple to combine different simple robotic applications (such as following a colored object and obstacle avoidance) without writing any additional code. All that is needed is an XML task that connects the component modules in the correct fashion [2].

Fig. 1 shows an example of an assembly of ROCI modules that combines three different tasks: Task Joystick, Task SegwayRemote, and Task SegwayDashboard. When combined in this fashion, these tasks take in joystick commands in the Task Joystick, transmit these commands to a SegwayRemote task where they are changed into drive commands in the Joystick2Velocities module before finally sending them to the Segway module, which relays the commands to the Segway. Simultaneously, the Segway module is attached to a SegwayDashboard module, which provides a user on a different computer with a graphical user interface that displays options for controlling the Segway as well as data on its current behavior. While the Segway Remote task, which includes the Segway module, runs on the computer attached to the Segway, the Joystick and SegwayDashboard tasks can run on one or even two distinct computers.

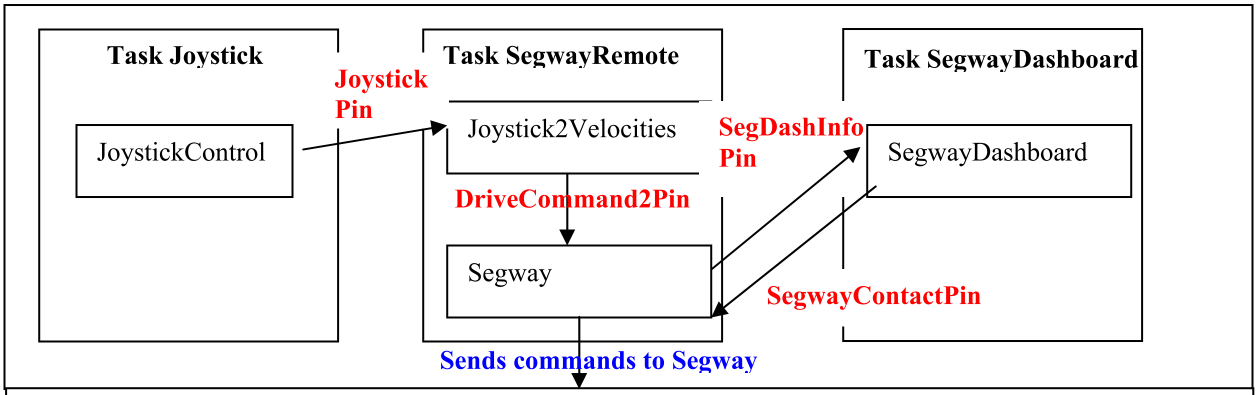


Figure 1: The connections of tasks and modules by pins in order to drive the Segway with the joystick

3. MOUNTING SENSORS ON THE SEGWAY

As described in Section 2.1, the built-in encoders from the Segway provide information on its speed, relative position, and orientation. However, in order to gather information about the space in which the Segway RMP is operating, it is necessary to mount additional sensors such as a camera and a laser range finder on its platform. Due to its balancing mechanism, the Segway's platform can tilt to significant angles when accelerating or having difficulty balancing. This severely inhibits the usefulness of sensors mounted on the Segway's platform in two ways. First of all, it makes locating an object detected by the sensors difficult because the distance measurements will be skewed by the sensor's varying orientation with respect to the ground. Second, the landscape that the sensor is viewing may change entirely. For example, a sensor that is supposed to look in the horizontal direction, which in most cases will be parallel to the ground, may end up observing the ground or the ceiling much of the time rather than the desired landscape. Figure 2 depicts the problem with mounting a sensor flat on the Segway's platform.

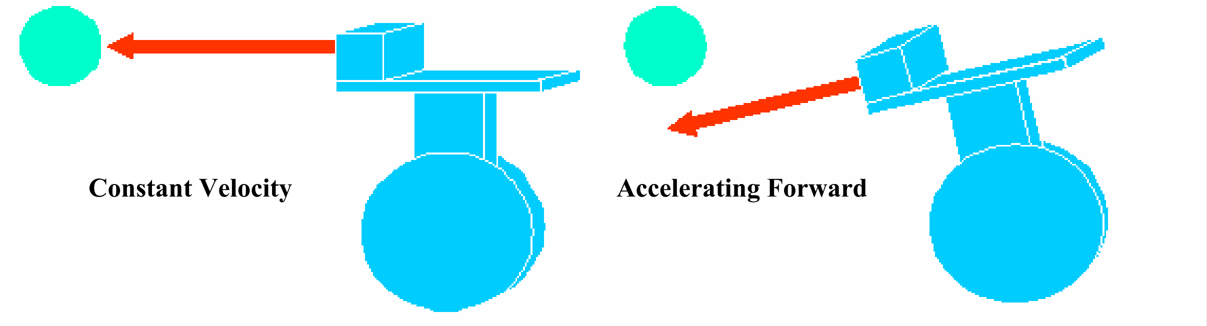


Figure 2: An illustration of the problem of mounting a sensor on a platform that undergoes pitch during motion. The sensor could be, for example, a laser range-finder trying to measure the distance to the ball (the greenish round object on the left).

Although the first problem could be solved in software by recording the Segway platform's pitch angle and using that information to correct for the error in the sensors' measurement of the distance, there is no software solution for the second problem (changing landscape of observation). Therefore, I opted to solve both problems at once by designing a passive, dynamically adjusting sensor platform which would always retain its horizontal orientation.

In designing the sensor platform, I chose to suspend the sensor platform as a pendulum and mount it on the Segway's platform. When the Segway's platform takes on a pitch angle, gravity causes the sensor platform to swing to the horizontal position. Although, it is true that when the platform moves it changes both the sensor's height off the ground and position with respect to the Segway's center, it is easy to compensate for these changes in distance mathematically, and the change in the sensor's frame of observation is insignificant compared to what would occur without the sensor platform. Figure 3 shows how this solution prevents the orientation of the sensor from changing.

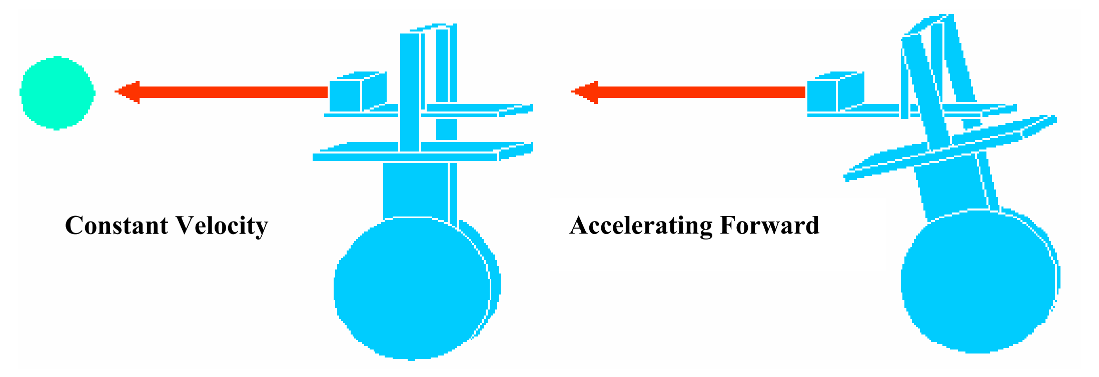


Figure 3: The Segway's dynamic sensor platform largely solves the problems caused by the Segway's pitching.

In order to prevent oscillations as the sensor platform adjusts itself and to prevent the sensor platform from taking on a pitch angle of its own due to inertia when the Segway accelerates or decelerates, the motion of the sensor platform is damped with a commercially bought rotary dashpot (damper). Additionally, the position of the center of mass of the sensors on the sensor platform determines the angle that the platform maintains. For example, if the center of mass is positioned at the platform's center, then the sensor platform will maintain a horizontal orientation. Note that the height of the pivot and the amount of clearance of the sensor platform over the Segway's platform were calculated to allow the sensor platform to correct for pitch angles of over 30 degrees. The distance from pivot to sensor platform (the length of the arm of the pendulum) was chosen to be 19 inches in order to give plenty of room to place sensors on the platform. The minimum clearance necessary for a 20 inch long platform to adjust to a 30 degrees pitch angle (more than we ever expect to see during the course of regular indoor operation) without hitting the Segway platform is given by this formula:

$$height = \frac{2(arm)}{\sqrt{3}} - \frac{arm}{2} + 5\sqrt{3} \rightarrow clearance = height - arm \qquad (1)$$

As can be seen from formula 1, the longer the arm is, the smaller the necessary clearance. For our prototype, the arm length is 19 inches, the height of the pivot above the Segway platform is 23 inches, and the clearance is 4 inches (as can be seen from plugging 19 inches in for the arm length in formula 1, anything over 3 inches of clearance is sufficient to handle a 30 degrees pitch angle without the sensor platform striking the Segway's platform).

Figure 4 shows the dynamic sensor platform mounted on the Segway carrying a LMS-200 laser range finder weighing 4.5 kilograms. In this case, the center of mass of the sensor and platform is in front of the pivot so the resting position of the platform gives the laser a slight angle downwards.

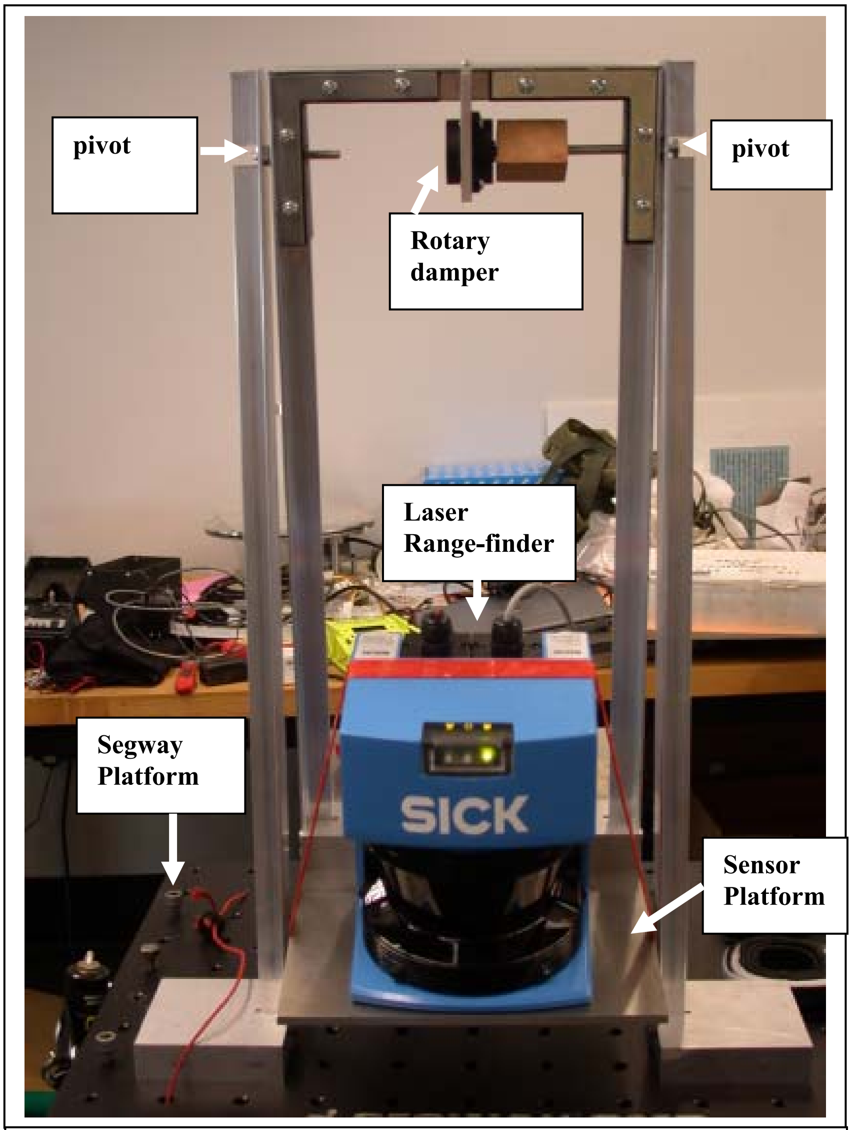


Figure 4: A photograph of the sensor platform mounted on the platform of the Segway with the laser rangefinder mounted on it.

4. SEGWAY BLOB FOLLOWER

One of the sensors that can be mounted on the sensor platform is a stereo camera. This sensor is used to detect and locate brightly colored (orange, green, and pink) objects. Taking advantage of ROCI's (see Section 2.2) flexibility, we were able to use the same modules for identifying and calculating an object's distance from and angle to the camera as were used on another robotic platform-the Clodbuster (a small autonomous truck) [2]. Initially, I also used the same module to send drive commands to the Segway as was used to send commands to the Clodbuster in order to follow the object. Although this proved to work fairly well, the differences between the Segway and the Clodbuster eventually led me to redesign the way in which the drive commands were sent to the Segway based on the information provided by the camera about the object's location.

In the case of the Clodbuster, which is much smaller than the Segway, the costs of occasional erratic commands is less severe, both in terms of distance covered in the wrong direction and the types of crashes that may occur from a sudden wrong command, than they are for the Segway, which weighs approximately 130 lbs.

In the original system, upon receiving the estimated location of the object, a drive command was sent out which was proportional to the distance from the object. For example, when the object was five meters away from the Segway, the drive command would be larger than when it was two meters away. If the robot traveled to within a certain distance of the object (for example one meter) then it would be ordered to back up. Although typically estimates of the location of the object were on the mark, a few wrong estimates of an object's location could cause a wild response in the Segway's driving. These wrong estimations could be caused by the object not being quite the expected color or a simple change in lighting.

In order to smooth over these erratic measurements, I wrote a SegwayBlobFollower module that filtered the data before sending out drive commands. The module accomplished this task by collecting a number (specified by the user) of readings of the locations of points on the object (given by distance and angle) and comparing these readings with projected (extrapolated) distance and angle based on the Segway's velocity. Only measurements that were consistent with the projections were used to control the Segway. Moreover, by comparing the new measurements with the previous point chosen, an approximation of the rate of change of the distance and angle of the object with respect to the Segway is made and used to project the object's future location so that the next set of points can be filtered in the same manner.

In this way, data points that are egregiously wrong are likely to be ignored since only the one considered to be most "likely" is chosen from any given group. There are two major flaws in this approach, however. The first is that since multiple data observations have to be made before a drive command is determined the reaction time of the Segway is decreased in proportion to the number of data points that are collected in each group. Although unfortunate, this lag is dealt with by decreasing the speeds specified by the drive commands sent to the Segway. The second and more serious problem is that there are multiple sources of error in this method. These sources of error take three forms. The first is that the first data point cannot be checked against any projected position. Instead, it is simply the average of the first group of data points. A single outlier in this group would skew the original starting point, thus skewing future projections. The second source of error is that the estimates of rate of change are constant with respect to the moving Segway - not to any absolute point. In short, if either the Segway or the object is accelerating, then the projection will be incorrect. The third source of error is that if none of the points in the group are near the correct point, the point chosen from that group will then be wrong, as will be the rate of change estimates, causing future projections of the object's location to be wrong.

Although all of these are valid sources for concern, this algorithm tends to be self correcting. Operating under the premise that errors in the object's location are random or at least temporary, one can assume that even if the system begins to make incorrect projections of the object's position, it will eventually correct itself. In other words, if a projected location for the object is wrong, then it is unlikely that any of the locations in the group collected will match it. The point chosen will therefore enforce a correction on future projections. For example, in the case where the object is accelerating away from the Segway, all the points in the group of measurements collected should be farther away than the projection would indicate. However, since the period between groups is not all that significant, the module will most likely choose a measurement that is close to correct (if one exists), and the resulting estimate of velocity will be greater than the previous one thus taking into account the acceleration of the object. For example, if the current distance to the blob is 5 meters and the estimated rate of change in distance is 3 m/s, then the projected distance to the blob after a second will be 8 meters. However, since the blob is accelerating with respect to the Segway, rather than any of the points taken in the next group being 8 meters, the module may get measurements of 11, 9, 13… meters. The blob follower will then choose 9 meters as the blob's location. Even if this is an underestimate of the distance to the blob, the new estimated speed will have increased taking into account the acceleration and forcing the projected value to be more in line with reality. Over the long term, any error should converge to zero as the general trend of the data forces projections to correspond with reality.

Although this blob follower has been written in software and preliminary tests show that it tends to perform at least as well as the original blob follower in locating and giving commands to follow the object, extensive testing has not been done on this module to see whether it actually is an improvement over the software previously used for Segway navigation.

5. SEGWAY OBSTACLE AVOIDER

The Segway is equipped with a SICK laser rangefinder mounted on its sensor platform for the purposes of obstacle avoidance. The laser is capable of making a 180 degree scan of the area around it. In mounting the laser, I chose to direct it downwards so that on a completely clear plane, all of its readings except for the first and last one will hit the ground. Figures 5 and 6 show the necessary calculations for finding the expected distance to the ground for any angle in the laser's scan.

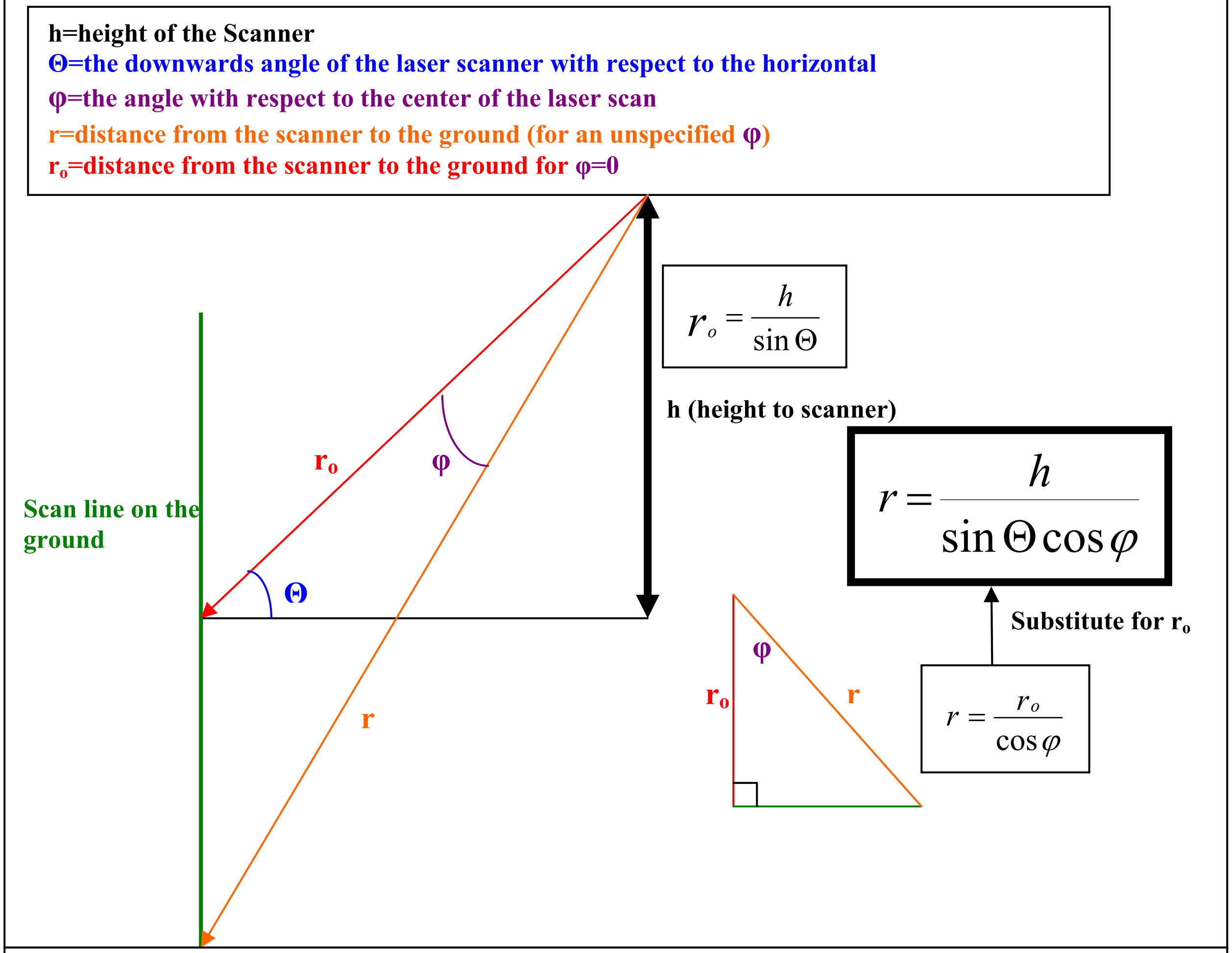

h=height of the Scanner
Θ=the downwards angle of the laser scanner with respect to the horizontal
φ=the angle with respect to the center of the laser scan
r=distance from the scanner to the ground (for an unspecified φ)
r_o=distance from the scanner to the ground for φ=0
$r_o = \frac{h}{\sin\Theta}$
h (height to scanner)
$r = \frac{h}{\sin\Theta\cos\varphi}$
Substitute for r_o
$r = \frac{r_o}{\cos\varphi}$
Scan line on the ground
r_o
φ
Θ
r
φ
r_o
r

Figure 5: The plane between r_o and r is covered by the SICK laser. Shown are the calculations to determine the distance to the ground (assumes the ground is flat) as measured by the laser scanner in the absence of obstacles and holes as a function of the scan angle phi.

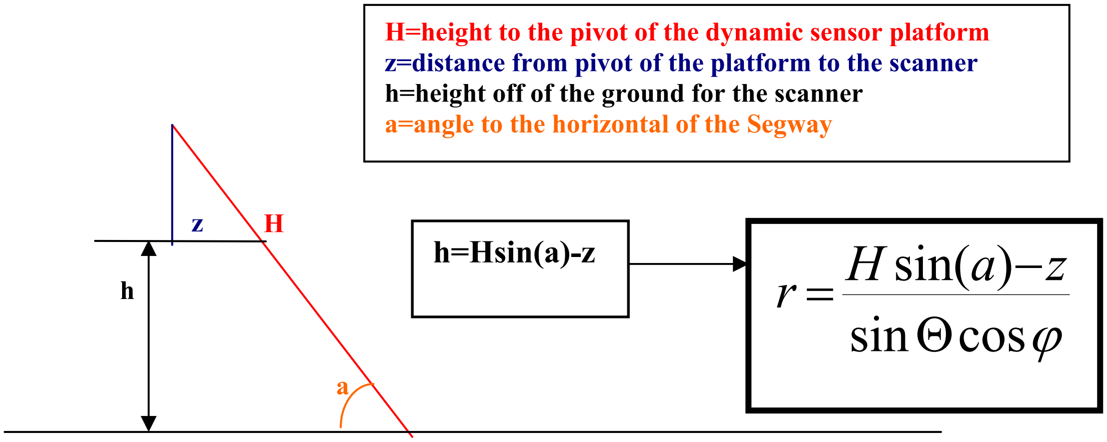

H=height to the pivot of the dynamic sensor platform
z=distance from pivot of the platform to the scanner
h=height off of the ground for the scanner
a=angle to the horizontal of the Segway
z
H
h
a
h=Hsin(a)-z
$r = \frac{H\sin(a) - z}{\sin\Theta\cos\varphi}$

Figure 6: Refined calculation of the distance to the ground factoring in the pitch. This calculation accounts for a scanner mounted on the dynamic sensor platform

As the figures show, the expected distance to the floor for every scan is determined by the equation:

$$r = \frac{h}{\sin\Theta\cos\varphi}, \tag{2}$$

where h is the height of the laser from the ground, Θ is the angle of the laser with respect to the horizontal, and φ is the angle in the laser's scan. Taking into account the Segway's pitch and the change in position of the laser on the dynamic sensor platform, the equation is modified to:

$$r = \frac{H\sin(a) - z}{\sin\Theta\cos\varphi}, \tag{3}$$

where *a* is the Segway's angle with respect to the horizontal, z is the distance to the pivot of the sensor platform from the laser, and H is the distance to the pivot from the bottom of the Segway.

Armed with this equation, for every data point given by the laser scan (angle and distance to object), we know whether the laser is detecting the ground, an obstacle, or a hole. If the scan returns the expected distance to the ground for that angle, then no obstacles were detected in that direction. On the other hand, should the laser's scan return a shorter distance than expected that means that there is an obstacle in that direction. Moreover, if the distance returned is longer than expected, then there is a hole in the ground in that direction. To allow for slight slopes and small obstacles that the Segway could easily roll over there is a +/- 0.1 meter margin of error before a reading is considered to be distinct from the ground. This precludes slopes larger than 10 degrees (which is near the maximum that the Segway RMP can handle). Since this obstacle avoider is intended primarily for indoor use, this limitation is not a serious one.

The algorithm above was used by both types of obstacle avoiders considered for this project. Section 5.1 describes the main obstacle avoider for this project. In section 5.2 are the outlines of an obstacle avoider that may eventually replace the design described in section 5.1. Although the laser range finder is an extremely accurate, active sensor, the fact that it only provides data on a part of a single plane leaves it vulnerable to several scenarios, listed in section 5.3.

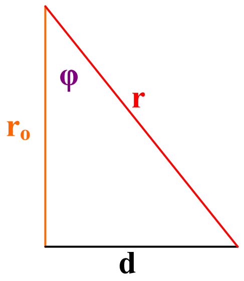
5.1 A Preliminary Obstacle Avoider

This obstacle avoider is passive as long as no obstacles are detected. In the absence of obstacles, it passes on the unmodified drive commands it receives from other modules. Upon detecting an obstacle, it ignores the drive commands it receives and issues its own commands until the obstacle is cleared.

The obstacle avoider detects an obstacle by checking whether the ground has been found for the range of angles that allow the Segway clearance to continue in a straight line from its present position. All readings for other angles are ignored because they are presumed not to interfere with the Segway's current motion. Figure 7, shows how to calculate the range of angles in the laser scan to check for obstacles.

Want to keep a width w clear-> angles of interest are:

2*d=w
r_o=h/sin(Θ)



w/2=$r_o$sin(φ)
substitute for r_o: hsin(φ)/(sin(Θ)cos(φ))

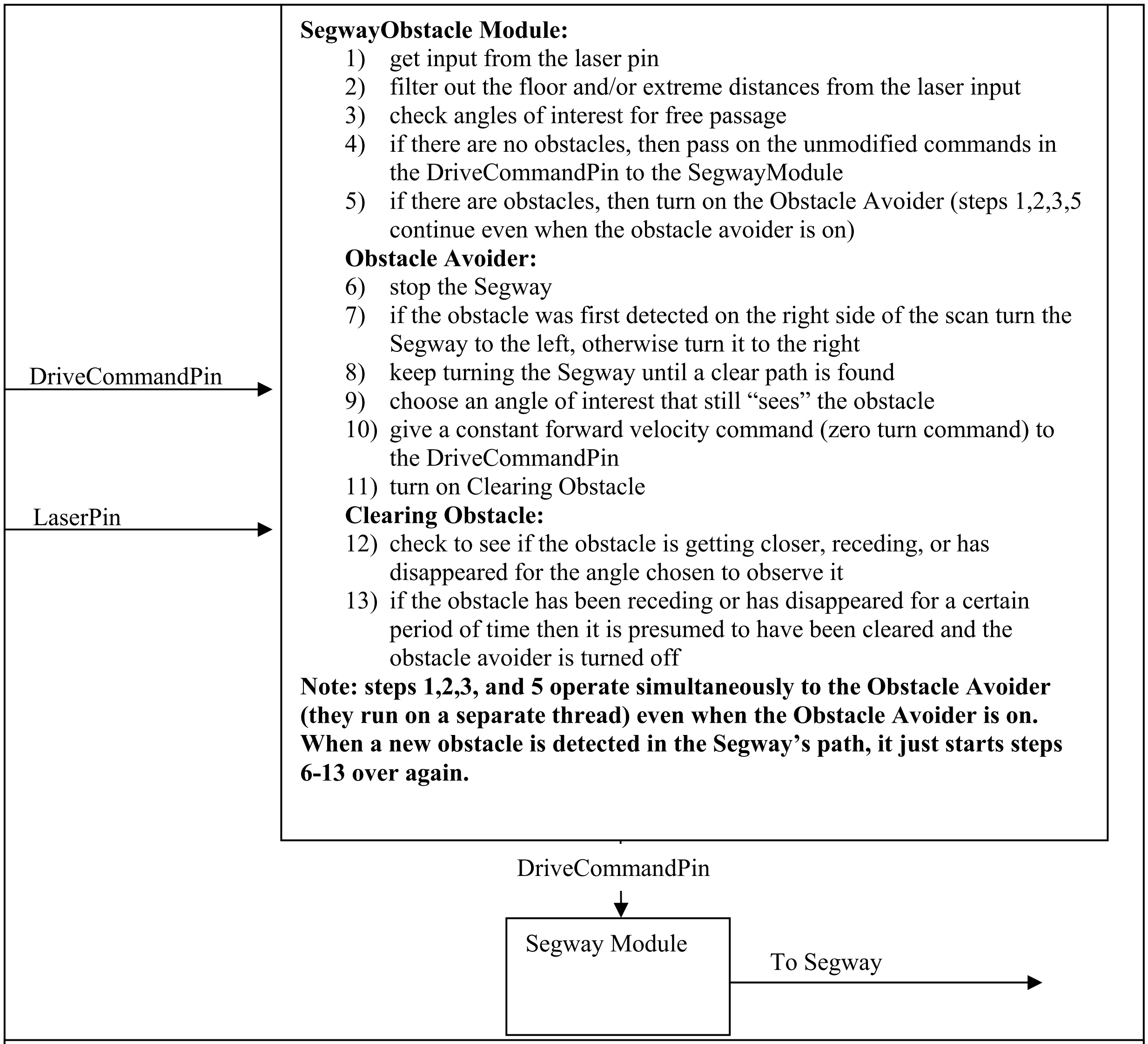
$$\varphi_1 = \tan^{-1}\frac{w\sin\Theta}{2h}$$

Figure 7: The calculation of the range of angles (-φ_1 to φ_1) needed to be checked to assure the presence of a gap of width w or greater that allows free motion of the Segway

The equation,

$$\varphi_1 = \tan^{-1}\frac{w\sin\Theta}{2h} \text{ (from -}\varphi_1\text{ to }\varphi_1\text{),} \qquad (4)$$

describes which angles should be checked to assure that a free passage of width w is available for the Segway to go through. Figure 8 describes the algorithm used by this module:



SegwayObstacle Module:

1) get input from the laser pin
2) filter out the floor and/or extreme distances from the laser input
3) check angles of interest for free passage
4) if there are no obstacles, then pass on the unmodified commands in the DriveCommandPin to the SegwayModule
5) if there are obstacles, then turn on the Obstacle Avoider (steps 1,2,3,5 continue even when the obstacle avoider is on)

Obstacle Avoider:

6) stop the Segway
7) if the obstacle was first detected on the right side of the scan turn the Segway to the left, otherwise turn it to the right
8) keep turning the Segway until a clear path is found
9) choose an angle of interest that still "sees" the obstacle
10) give a constant forward velocity command (zero turn command) to the DriveCommandPin
11) turn on Clearing Obstacle

Clearing Obstacle:

12) check to see if the obstacle is getting closer, receding, or has disappeared for the angle chosen to observe it
13) if the obstacle has been receding or has disappeared for a certain period of time then it is presumed to have been cleared and the obstacle avoider is turned off

Note: steps 1,2,3, and 5 operate simultaneously to the Obstacle Avoider (they run on a separate thread) even when the Obstacle Avoider is on. When a new obstacle is detected in the Segway's path, it just starts steps 6-13 over again.

Figure 8: Shows the inputs and outputs for main obstacle avoider module of the project and provides a step by step description of the algorithm used within the module.

The obstacle avoiding algorithm follows these steps: (1) checks the laser data for an obstacle; (2) if there is no obstacle then the module passes on the unmodified drive commands to the Segway module (if the obstacle avoider is already on, then it sends the drive commands that it specifies) or if there is an obstacle turns on (restarts) the obstacle avoider. The following steps take place when the obstacle avoider has just been turned on: (3) stops the Segway; (4) turns the Segway at a constant speed until the laser data declares that the Segway can proceed in a straight line freely; (5) chooses an angle in the laser scan that still "sees" the obstacle; (6) the Segway is ordered to move forward with a constant velocity until the obstacle avoider is turned off or restarted; (7) the chosen angle is checked to see if the obstacle is getting closer, receding, or has disappeared; (8) if the obstacle has been receding or is no longer observed for a certain period of time then the obstacle avoider is turned off. Note that steps 1 and 2 operate concurrently to steps 5-8 so if the robot encounters an obstacle while obstacle avoiding, it simply restarts the obstacle avoider and forgets about the prior obstacle. In this way, the algorithm deals with encountering additional obstacles while trying to avoid one. Also note that the reasoning behind observing the obstacle's distance and turning off the obstacle avoider when the obstacle has been receding or is no longer visible is that this observation indicates that the robot is about to pass a curve or corner in the obstacle or to leave it behind entirely. Therefore, the robot can either turn back towards its destination, having passed the obstacle, or at least choose a vector closer to the one desired if it has passed a corner or curve in the obstacle. The reason that the Segway continues in its straight line path for a period of time after the obstacle has been observed to have been cleared is that the laser rangefinder is observing the obstacle at a point ahead of the actual robot, so it observes that the obstacle will be cleared before the robot has actually cleared it. The additional time driving forward allows the robot to reach and pass the point at which it observes the obstacle to have been cleared. Figure 9 has a depiction of how the robot would act in encountering a single obstacle under this algorithm.

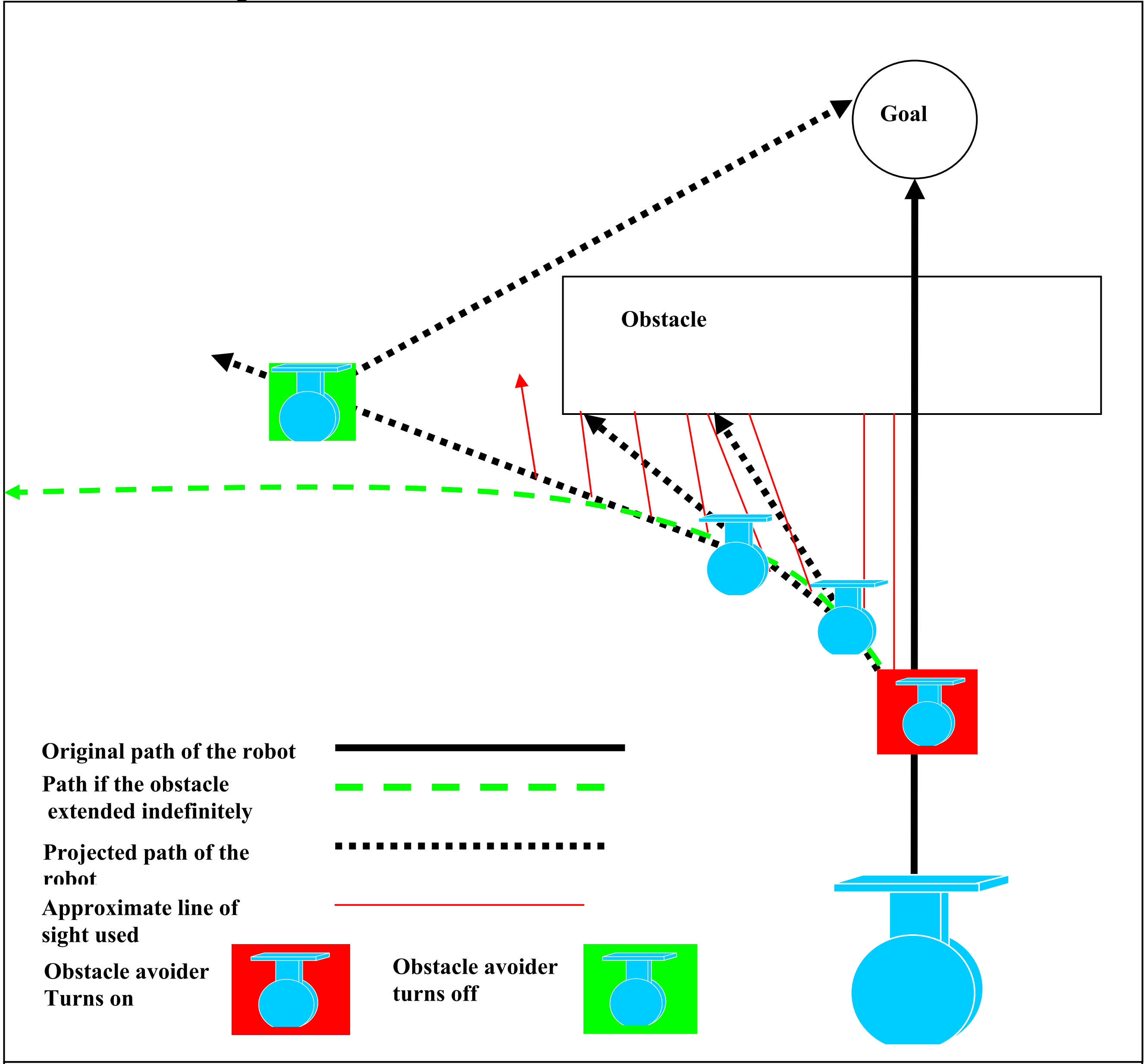

Goal
Obstacle
Original path of the robot
Path if the obstacle extended indefinitely
Projected path of the robot
Approximate line of sight used
Obstacle avoider Turns on
Obstacle avoider turns off

Figure 9: Depiction of how the Segway would behave when encountering a cornered obstacle (not to scale)

One of the flaws in this approach is that there is no awareness of the area in which the Segway is operating so there is no attempt by the module to chart an optimal course through all the obstacles recorded towards the goal. In other words, when the Segway is avoiding an obstacle, there is no awareness of the ultimate goal; it just wants to get around the obstacle at hand. In fact, the Segway may end up moving in a closed loop without ever reaching its target. Another problem is that there are scenarios where the robot could be caught in an infinite loop. Although a simple infinite loop can be ruled out by keeping the robot from choosing a trajectory 180 degrees trajectory from its current one when encountering an obstacle until that is determined to be the only available route, more complex infinite loops (for example, traveling in a circle around four points) can only be ruled out by implementing an obstacle avoider with mapping.

A flaw that should be corrected is that the Segway may already be turning away from the obstacle it views directly ahead under its regular drive commands but will still be sent into obstacle avoidance because the module always presumes that the robot is going straight ahead. Furthermore, currently, the module does not take into account the fact that the range of angles of interest (that could disrupt the Segway's path) increases the closer the object is to the machine. In the current configuration, the module only checks the range for a clear path at the distance that the laser scan reaches the floor. However, this is a much smaller range of angles than it would need to check for closer obstacles because the "rays" from the laser spread less for shorter distances. So a previously undetected obstacle that is very close to the Segway may be seen by the scanner but ignored because it is not in range

There are several ways to eliminate the problem of prematurely turning on the obstacle avoider. The first, and simplest, is to take advantage of the Segway's zero-turning radius to always have the Segway turn in place and then travel forward in a line. In that case, whenever the Segway is at risk of running into an obstacle, it is going straight. Alternatively, this problem could be solved by taking into account the current turn commands and speed in determining the range of angles from the scan to check for obstacles (if the Segway is turning to the right then check the same number of angles but in a range that is shifted to the right, of the center of the scan). The problem of missing close by obstacles can be eliminated by checking a wider range of angles for obstacles a short distance away and a decreasing range of angles from the scan for obstacles farther away. Although none of these solutions were implemented in the current obstacle avoidance module, it is likely that these improvements can be readily made. Unfortunately, there is no way to eliminate the rest of these problems without resorting to mapping (see Section 5.2).

As outlined earlier when encountering an obstacle, the Segway stops, then turns, then goes straight. At no point is it undergoing rotational and translational motion at the same time. This method was chosen because it provided the safest and simplest design. By stopping the Segway, we ensure that it has as much time as it needs to find an open vector in which to travel. Moreover, the magnitude of both the velocity and turn commands can be left as constant and need not be calculated based on sensor input (*i.e.* if the Segway is quickly approaching the obstacle it must decrease its translational velocity and increase the rate of its rotation). Attempting to have the Segway safely undergo dynamic navigation would require several additional layers of complexity in determining velocity and turn commands. At best, such a system could only match the current one in safety (preventing collisions). Given that the processing speeds of the onboard computer rapidly decline with the addition of more sensors, such as the stereo-camera, the dynamic approach may currently be either infeasible or at least unsafe given our current computing power.

Tests indicate that the Obstacle Avoider described here works as intended. However, it has never been calibrated (the necessary measurements made) or tested for cases when the avoider is supposed to maneuver around holes in the nominal ground plane in addition to obstacles.

Note that this obstacle avoider deals fairly well with moving obstacles-it turns to avoid them just as it would avoid static obstacles and since it has no memory of obstacle location, it is not permanently affected by the obstacle's motion.

5.2 Obstacle Avoidance With Mapping

This obstacle avoider differs from the one in section 5.1 in that it plans its course around multiple obstacles rather than trying to explore one obstacle at a time. In essence, this obstacle avoider is made up of two parts: a mapper module and a navigator module. The area in which the Segway operates is mapped with a Cartesian grid. Upon starting, the Segway's location is set to be some point (currently the origin of the grid). The x and y coordinates of its goal are also set. From that point on, the Segway's position is calculated using its encoders, which provide integrated fore/aft distance and integrated yaw position. Drift in the information provided by these encoders, due to wheel slippage, is the major source of error in the mapping. Just as for the obstacle avoider described in section 5.1, the laser makes a reading and measurements that coincide with the ground are ignored. However, this time, all angles are checked and the ground distance to the obstacle from the Segway is determined using this equation:

$$d = \sqrt{r^2 - h^2}\ , \qquad (5)$$

where the determination of r and h is described in Fig. 5. Using this distance and the angle in the laser scan and already knowing the Segway's x and y coordinates and its orientation, the x and y coordinate of the obstacle can be found and placed in the grid.

The map exists as a two-dimensional array with the addresses corresponding to the integer x and y coordinates. When an obstacle is detected it is placed in the container with the closest x and y coordinates. For example, an obstacle at (2.7, 3.1) goes in the [2, 3] container. Note that this method is inefficient in that each laser reading of an obstacle is treated separately. In short, every time the laser scan "sees" an obstacle in close to or the same location it will drop another recording of an obstacle into the container for that area rather than combining groups of readings into single obstacles. This could cause memory issues. Future improvements to this module should group obstacle readings for different locations that are close together into the single large obstacle that they probably form (or at least certainly would be as far as the robot is concerned). Ultimately, such a solution would be more efficient and elegant than the current one.

The obstacle avoider module receives an updated map and the Segway's location from the mapper module. It then attempts to plot a straight line between the Segway's current location and its goal. If there is an obstacle in the way on the map, the module determines whether it is more obstructive in the y or x direction. It then chooses a secondary goal with the same x or y coordinate as the obstacle that appears to be reachable by a straight line on the map. For simplicity's sake, the Segway is either undergoing rotational or translational motion at any given time, not both. Once a safe line is determined the Segway turns onto a straight line to the new destination. If the destination is reached, then the Segway once more tries to make a straight line to its ultimate goal. If on the way to this secondary destination another obstacle appears on the updated map, then the Segway chooses another secondary destination and the prior one is thrown out. In this case, once the secondary destination is reached, the module again tries to plot a straight line route to the ultimate goal. In time, this method of choosing a route through the map should be changed to one where a full course is plotted with multiple way-points around all the known obstacles, not just the one of most immediate concern. Furthermore, search algorithms should be implemented to choose the most optimal trajectory based on a pre-defined cost function such as distance or time of travel.

To check whether any given line is safe, the contents of the grid spaces near the line are checked to ensure that no obstacles in them are within a certain margin of the Segway. In this way, there is some efficiency gain because only containers with contents that might possibly interfere with the current path are checked.

The most glaring problem with this obstacle avoider is that in its current implementation, an obstacle that is moving is recorded permanently on the map as blocking a certain area. In a busy environment this would render this obstacle avoider useless. An immediate, but somewhat complex, solution is to remove obstacles from the map as well as add them. However, this is not a viable alternative until the mapping mechanism is improved to the extent that the mapping groups whole areas together as obstacles rather than just individual points.

Although, this obstacle avoider has been implemented in software, it has never been tested with the Segway.

Figure 10, shows how the Segway would respond to an obstacle using this obstacle avoider.

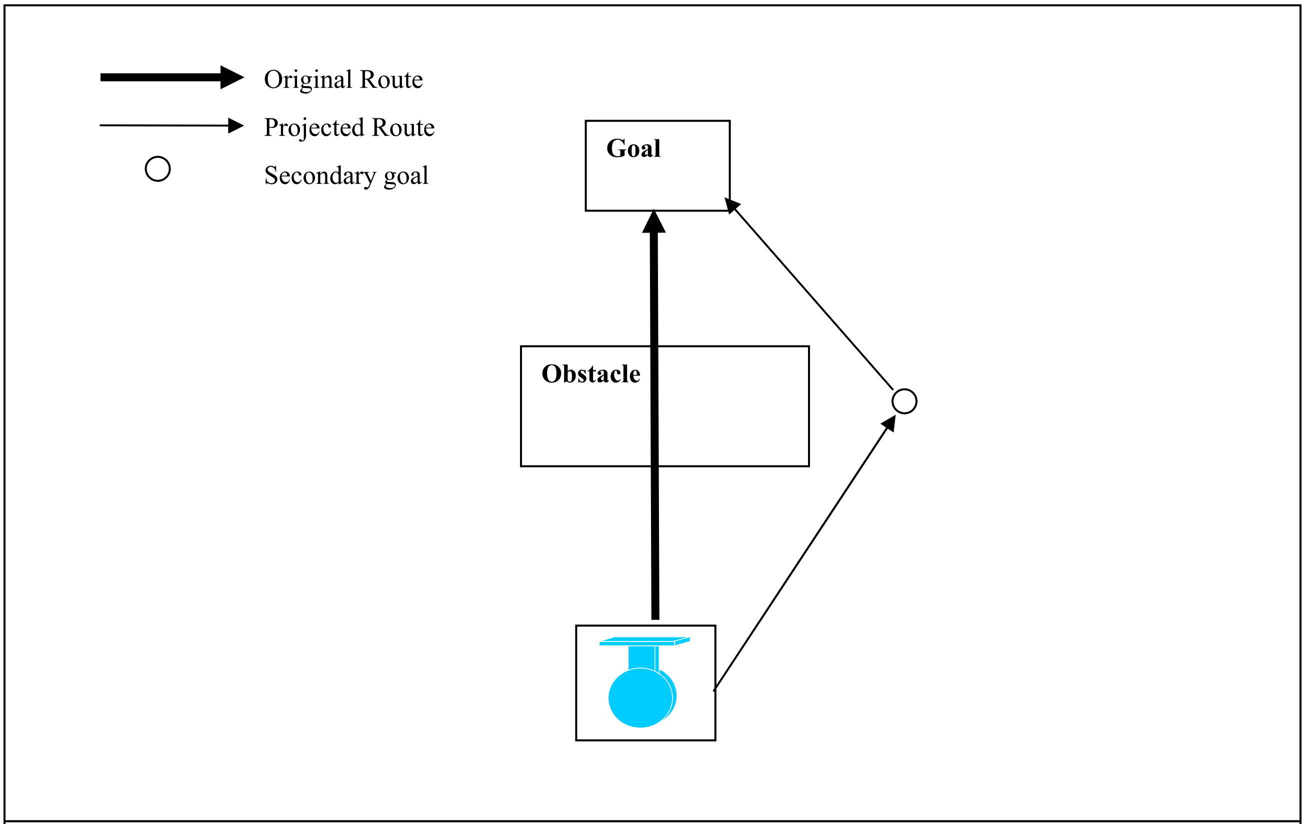
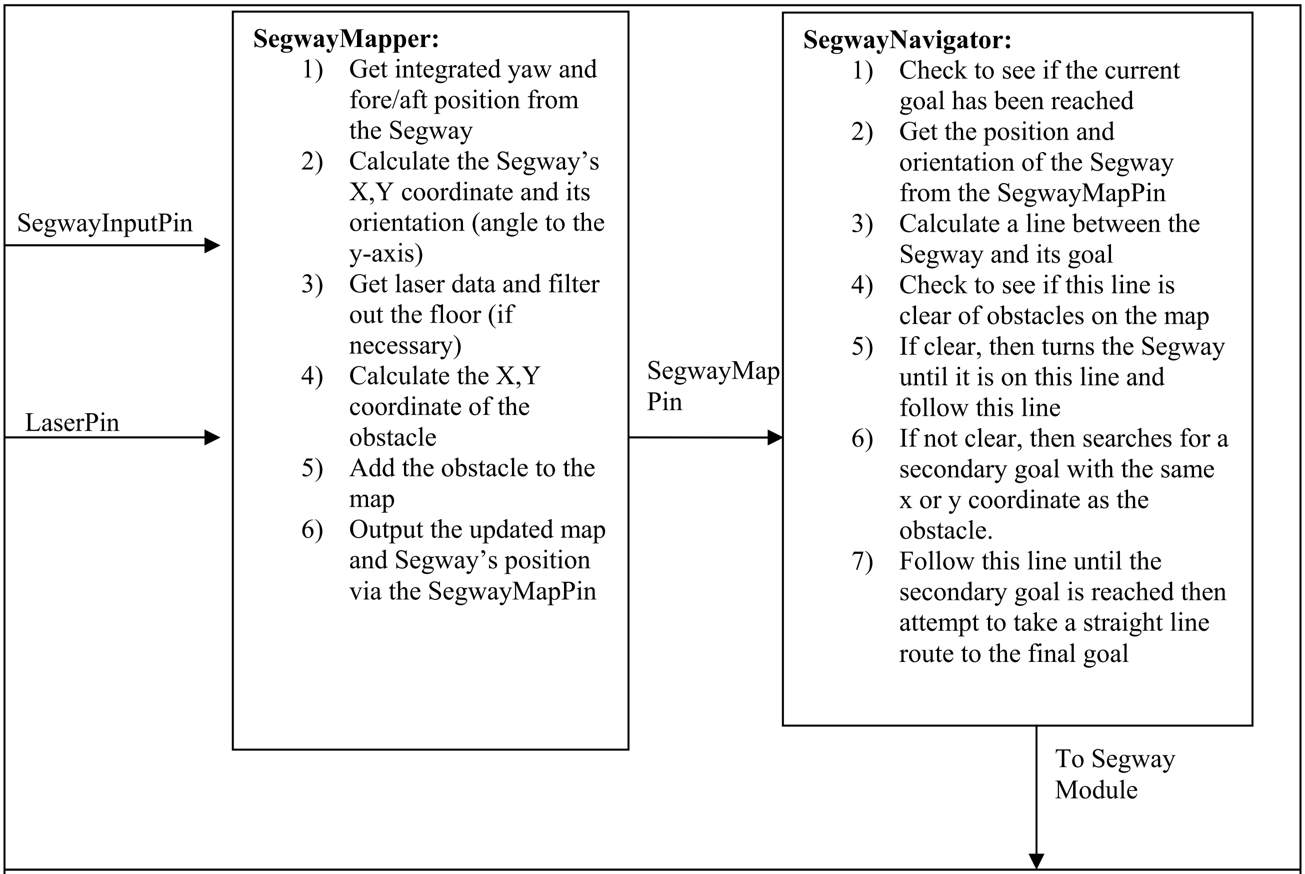


Figure 10: Demonstrates how the obstacle avoider described in 5.2 would go around an obstacle between it and its goal

Figure 11 provides an outline of how this obstacle avoider works.



SegwayMapper:

1) Get integrated yaw and fore/aft position from the Segway
2) Calculate the Segway's X,Y coordinate and its orientation (angle to the y-axis)
3) Get laser data and filter out the floor (if necessary)
4) Calculate the X,Y coordinate of the obstacle
5) Add the obstacle to the map
6) Output the updated map and Segway's position via the SegwayMapPin

SegwayNavigator:

1) Check to see if the current goal has been reached
2) Get the position and orientation of the Segway from the SegwayMapPin
3) Calculate a line between the Segway and its goal
4) Check to see if this line is clear of obstacles on the map
5) If clear, then turns the Segway until it is on this line and follow this line
6) If not clear, then searches for a secondary goal with the same x or y coordinate as the obstacle.
7) Follow this line until the secondary goal is reached then attempt to take a straight line route to the final goal

Figure 11: Shows the process for running the obstacle avoider

5.3 Problems With The Obstacle Avoider

Unfortunately, the first method of obstacle avoidance is fundamentally flawed. The simple fact is that with a two dimensional scan where the laser is mounted high above the ground, there is a great chance of the laser scan simply shooting over obstacles. With the passive obstacle avoider, the robot will see an obstacle to its side but since it is memory-less will not remember later that it should not turn in toward that obstacle. In the case of the second obstacle avoider, the robot will not hit an obstacle it has previously detected as long as the robot's coordinates are determined with a fair degree of accuracy. Therefore, when using an obstacle avoider with mapping, this problem is not so severe. Using the range-finder's 180 degrees scan, the robot will discover and map all obstacles it could possibly collide with before the range-finder starts shooting over them. In this case, the only requirement is that the robot starts out in an area free of obstacles that its laser might scan over.

At first, one might think that this problem could be easily solved simply by mounting the laser close enough to the ground so that the only obstacles it shoots over are those that the robot can roll over without trouble. Unfortunately, this introduces a whole new class of problems: the obstacles that it would shoot under. Particularly, indoors, the environment is not that of a plane world; the area in which the robot must operate is populated with tables, chairs, and benches which a laser close to the ground will miss entirely. Ultimately, the use of a 3-d laser range-finder will completely eliminate this problem.

6. CURRENT STATE OF THE PROJECT

During the course of this project, I successfully designed and mounted a dynamic sensor platform to compensate for potential difficulties in collecting data from sensors. Preliminary testing of the platform with a SICK laser range-finder and a stereo camera mounted on it shows that it does, in fact, provide a stable platform for mounting sensors on the Segway.

Additionally, there exists both a working prototype of a blob follower and an obstacle avoider that have undergone preliminary testing with the Segway. Currently the obstacle avoider has only been tested while receiving drive commands from a joystick. By replacing the joystick module with the modules used to develop drive commands for blob following in the XML file that defines an application in ROCI, the resulting application would allow the Segway to perform completely autonomous navigation by allowing it to attempt to follow a blob and avoid obstacles simultaneously. Early trials with putting this application together have indicated that the stereo camera consumes a significant portion of the Segway-based laptop's processing power, causing the computer to freeze when used in conjunction with the obstacle avoider. Cutting down on the camera's resolution seems to mitigate this problem, but it is not yet clear whether this is sufficient for autonomous operation given the throughput constraints of our current laptop computers.

Another problem involving the proposed autonomous navigation (obstacle avoidance while attempting to approach a blob of a certain color) is that currently the camera is flat-mounted on the sensor platform. This means that it faces in the direction that the Segway is facing. In many cases when trying to maneuver around obstacles, the Segway will have to turn away from the object it is attempting to approach. Just as a human would turn his head to look at the blob even while not walking directly towards it, the camera should have a pan mount so that it can turn with respect to the platform on which it is mounted in order to keep the object of interest in view. This would require some slight modifications in the modules related to locating the blob with respect to the Segway in order to correct for the camera's orientation.

As mentioned earlier, although the concept of a second, more advanced obstacle avoider with more potential for future improvement has been considered and I have begun implementing it, it has not yet been tested on the Segway and may require some modifications before it is a workable alternative to the current obstacle avoider. Also, since I do not know how quickly the encoders on the Segway will accumulate error this concept may not actually be a viable alternative at all. However, if the Segway's position can be tracked accurately, such an obstacle avoider would be a great improvement over the one currently employed.

Finally, as mentioned at the end of section 5.1, although the current obstacle avoider has been shown to work fairly well with respect to multiple obstacles, it has not undergone extensive testing in order to calibrate it most effectively. First, the avoider has not been calibrated to recognize and deal with holes and drop-offs since the mathematics require exact measurements of how the laser is mounted (in order to identify the location of the ground), and a final mounting has not been developed yet. Second, extensive testing would be useful in helping to identify such variables as how far ahead obstacles should be detected and responded to. Also, testing might show that it is necessary to add more sophistication to interpreting the laser data than just filtering out the floor. It may be necessary to use the laser reading not only to determine the obstacle but to find out exactly how close it is in order to determine the response. At the end of section 5.1, I mentioned several problems with this obstacle avoider and sketched out potential solutions. Given time and testing all of these could be used to vastly improve its performance.

7. CONCLUSIONS AND RECOMMENDATIONS

There are several steps that can be taken in the near future. First, the current obstacle avoider should undergo extensive testing to identify its weaknesses, to determine how exactly to mount the laser on the sensor platform, and to complete its calibration. Then, an attempt should be made to operate the blob follower in conjunction with the obstacle avoider and to streamline this process. If preliminary efforts in this direction are successful, then the camera should be mounted with its own motor so that it can rotate freely to track the target even when the Segway is not heading towards it. Additionally, the more complex obstacle avoider, which uses mapping, should be completed. For both concepts of obstacle avoiders, obstacles are assumed to be static. Future research into dealing with moving obstacles would also be beneficial.

In the long run, more mechanical improvements will be necessary to enable the Segway to be fully autonomous. The most important of these is to replace the current laser range-finder with one with 3-dimensional scanning ability, or to optimize an obstacle avoider that uses mapping.

The combination of maneuverability and onboard power gives the Segway great potential for autonomously navigated, indoor applications. One of the most significant drawbacks in mounting sensors directly fixed to the Segway, the pitching of its platform, can easily be resolved by constructing a sensor platform similar to the one outlined in this paper. Moreover, early preparations for the Segway to perform autonomous tasks such as blob following and obstacle avoidance are meeting with success and laying the groundwork for more advanced robotic applications.

8. ACKNOWLEDGMENTS

I would like to thank my advisors Professor Vijay Kumar and James Keller at the GRASP laboratory at the University of Pennsylvania. Additionally, I would like to thank the rest of the GRASP lab staff and students. Finally, I would like to thank the National Science Foundation and the SUNFEST staff for giving me this research opportunity.

9. REFERENCES

1. Luiz Chaimowicz, Anthony Cowley, Vito Sabella, Camillo J. Taylor, ROCI: A Distributed Framework for Multi-Robot Perception and Control, Proc. 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, Las Vegas, Nevada, USA, Oct. 2003, pp. 266-271.

2. Benjamin Bau, A Report on the Segway's Performance as a Robotic Mobility Platform, http://www.cis.upenn.edu/marsteams/Segway/Segway%20site.htm, 2003.

3. Tom Collins, Segway RMP Experiments at Georgia Tech, DARPA MARS Segway Workshop, http://www.cc.gatech.edu/ai/robot-lab/segway/, September 23, 2003.

4. The Segbot, http://www.segbot.com.

5. Andrew Howard, Denis F. Wolf and Gaurav S. Sukhatme, Towards Autonomous 3D Mapping in Urban Environments, http://www.cres.usc.edu/pubdb_html/files_upload/393.pdf.