Obstacle Detection and Avoidance Using TurtleBot Platform and XBox Kinect

Sol Boucher

Research Assistantship Report
Department of Computer Science
Rochester Institute of Technology

Research Supervisor: Dr. Roxanne Canosa
Research Sponsor: RIT Golisano College Honors Committee

20114/August 9, 2012

Roxanne Canosa, Ph.D. ____________________ Date ____________________

Abstract

Any robot that is to drive autonomously must be able to detect and avoid obstacles that it might encounter. Traditionally, this problem has been solved using systems of one or more RGB cameras utilizing complicated and computationally-expensive computer vision algorithms, somewhat unreliable ultrasonic distance sensors, or laser-based depth scanners. However, Microsoft's recent release of the XBox Kinect has opened up new areas of research in the areas of computer vision and image understanding, and this same device can be employed for obstacle detection.

The three-dimensional point cloud provided by the low-cost and commercially-available Kinect platform puts much more information about the surrounding world at the disposal of an autonomous robot. This research investigates the problem of using this data to autonomously detect and avoid obstacles in an unconstrained indoor environment. The algorithm used is a synthesis of the traditional method of choosing turn directions based on the centroid of the detected points and a more novel search of the ground plane for edges and boundaries. Good results are achieved not only for drop-offs and common obstructions, but also when objects are especially short or moving just in front of the robot and perpendicular to it.

Contents

List of Figures

List of Tables

1 Introduction

This research was conducted on Willow Garage's TurtleBot robotics platform, shown in Figure 1. The TurtleBot is an integrated kit backed by Willow Garage's Robot "Operating System" (ROS) robotics suite and the Open Perception Foundation's Point Cloud Library (the PCL), both of which are open-source projects distributed under BSD licenses. The Kinect provides depth information in the form of a three-dimensional point cloud, as shown in Figure 2. The goal was to implement a simple but effective obstacle detection and avoidance system that—using only the data from the Kinect—was able to autonomously roam the hallways on all floors of the building without running into anything. This task presupposed an ability to avoid whatever common hazards, whether stationary or mobile, it might reasonably be expected to encounter during such a journey. Such capability would be desirable for potential use with or integration into future projects developed on the same system or a similar one.

2 Similar Work (Literature Review)

Microsoft launched the Kinect on November 4, 2010, in order to add a new and innovative breed of entertainment to its XBox 360 gaming console. However, the sensor immediately caught the attention of researchers and software developers of all persuasions; as a result, and thanks to the effort of many dedicated hackers, open source drivers were soon available to facilitate its use for more diverse applications. Using these drivers, researchers have since used the Kinect for room mapping, desktop application control, 3-D videoconferencing, surveillance, and even diagnosis and surgery.

When using RGB-D sensors on systems with limited resources, the largest stumbling block tends to be the computational cost of processing each frame of the cloud data. In an attempt to alleviate this burden, Microsoft initially planned to include an onboard embedded microprocessor capable of many common image processing operations, a feature that was cut from the production Kinect. As a result, the full burden of working with the three-dimensional data continues to rest with the main CPU.

The XBox Kinect has an infrared projector and infrared camera separated by

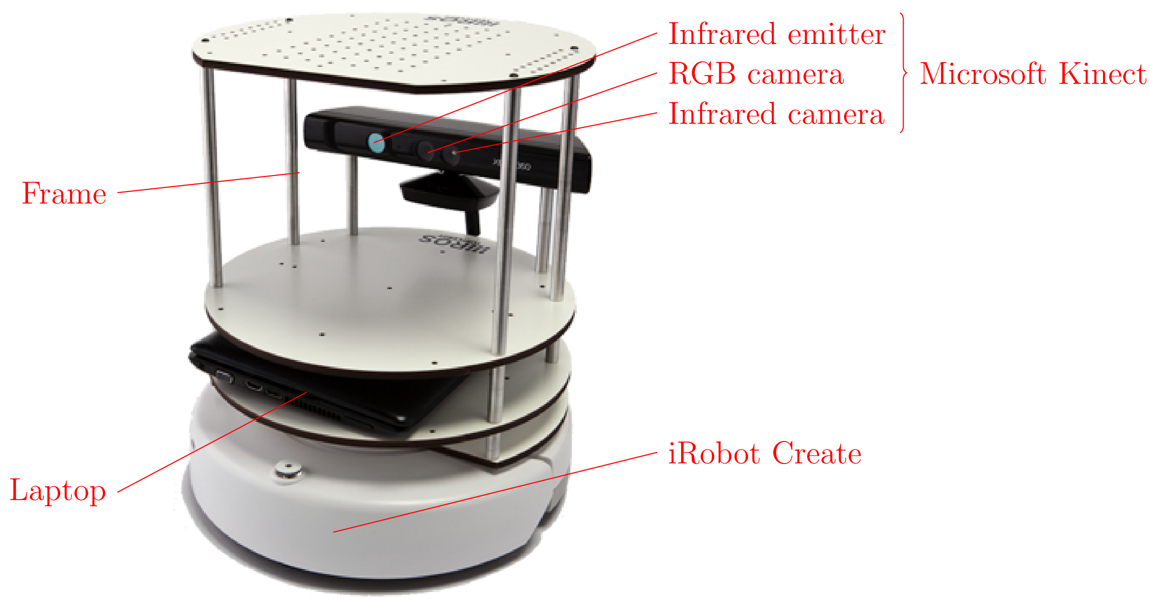
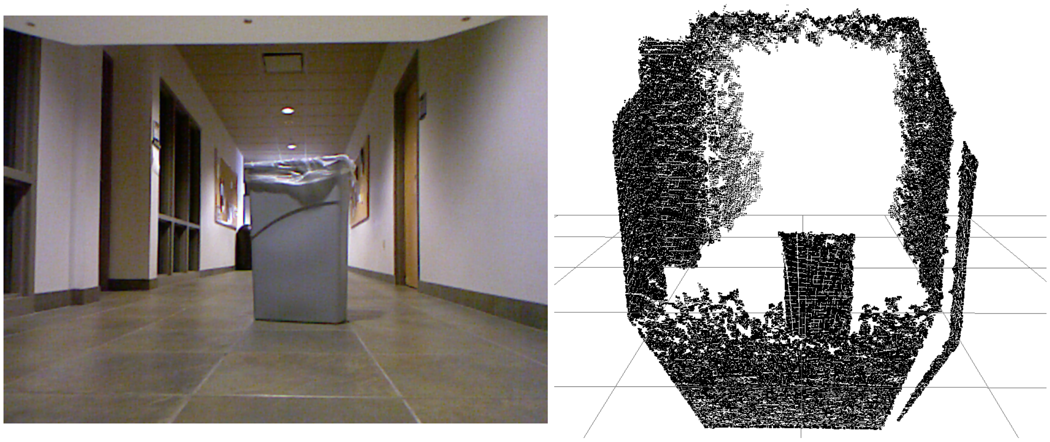


Figure 1: The TurtleBot and its components
(Image credit: `http://ros.org/wiki/TurtleBot`)

about 7.5 cm, and a color camera about 2 cm away from the latter (Nguyen, 2012). The infrared pair is able to assemble a grid of distance measurements triangulated from the lateral displacement of the projected points from the known emitter pattern. Unfortunately, the device is unable to perform any distance measurements closer than about 0.5 m. One method of detecting obstacles is as follows: First, perform a voxel grid downsampling on the point cloud to decrease processing time. Next, apply a pass-through filter to crop out regions of little interest or accuracy. Then, use the RANSAC algorithm to perform plane detection. Finally, Euclidean cluster extraction reveals individual obstacles, and additional analysis of those obstacles is performed in order to determine their sizes. This procedure avoids many difficulties of using a single RGB camera, as well as enjoying faster run times than dual–RGB camera systems.



(a) A picture from the color camera (b) The corresponding 3-D point cloud

Figure 2: Sample imagery from the Kinect

Whenever information from multiple image sensors is integrated, there is a risk that it will not line up appropriately, either due to simple displacement resulting from the sensors' relative positions or because of unique lens distortions created by inconsistencies in the manufacturing process (Herrera *et al.*, 2011). Herrera *et al.* describe their noise-tolerant method for calibrating a color camera and depth camera against each other, enabling the attainment of better results than would ever be possible by calibrating the two cameras individually. They start by computing for the color camera the two-dimensional projection coordinates in the image at which a three-dimensional point in space—the corner of a checkerboard calibration pattern—appears, then perform a distortion correction. Next, they repeat the projection calculation for the depth image, this time using the corners of the plane on which the checkerboard rests—because the board itself isn't visible in this image—and omitting the distortion correction step, as it will be much less effective than for the color imagery. Using the projections and data from several images with different perspectives, it is possible to calculate the rotation and translation necessary to match the two images' reference frames. These first parameters obtained for the color camera are much better than those for the depth sensor, so the former are used to optimize the latter by performing a nonlinear error minimization; then, another minimization is performed across the parameters for both cameras until the results are convergent. Using 35 calibration images, the authors are able to demonstrate comparable accuracy to that achieved by the proprietary calibration algorithm provided with their XBox Kinect test sensor.

A common traditional method of obstacle avoidance is the potential field model, or PFM (Koren and Borenstein, 1999). This model represents targets and obstacles as imaginary attractive and repulsive forces on the robot, respectively. Stored as vectors, such forces are easily summed to find the resultant force vector, which is used directly as the robot's navigation vector. One such implementation—the virtual force field, or VFF—uses a two-dimensional histogram grid populated from ultrasonic range sensors and holding certainty values of how likely it is that an obstacle exists at each location. Objects of interest are assigned corresponding virtual repulsive force vectors with magnitude proportional to their certainty values and inversely proportional to their distance from the vehicle's center. Similarly, the attractive force between the robot and its goal location is proportional to a preassigned force constant and inversely proportional it its distance from the vehicle. After obtaining the resultant force vector, its direction and magnitude are converted into parameters usable by the drive system and issued as movement commands. However, four major problems have been identified that effect all PFM systems, becoming increasingly noticeable as a robot moves faster: The robot may fall into a trap situation when it reaches a dead end, a phenomenon for which workarounds exist. The robot may also be directed in the opposite direction of its target in the case where two close objects stand in front of it with space between, a more difficult problem to handle. Certain environments may also cause the robot to begin oscillating. Finally, more severe oscillations—and even collisions—occur when a robot drives down a narrow hallway with a discontinuity in its side. Together, these factors make the same PFMs that were once seen as simple and elegant much less attractive, especially for applications relying on higher speeds.

One way to detect obstacles using an RGB-D camera is to segment every plane in the point cloud and consider as obstacles both points emerging from the detected planes and planes whose surface orientations differ from that of the ground (Holz *et al.,* 2011). Surface detection may be accomplished computationally cheaply by considering pixel neighborhoods instead of performing distance searches, then computing the normal vector by finding the cross-product of two averaged vectors tangential to the local surface. The coordinates of the points and their corresponding surface normals are transformed to Cartesian coordinates from the robot's perspective, then to spherical coordinates. Only the plane representing the ground is considered navigable, and the RANSAC algorithm is applied to optimize the detected surfaces and compensate for noisy readings. Each plane is then converted to its convex hull, and both horizontal planes other than the ground and planes supported by horizontal planes are considered to be navigational obstacles. This method is able to process plane data at high speed using only sequential processing while remaining relatively accurate: the average deviation is under ten degrees, and objects are properly segmented over 90% of the time. The algorithm is, however, sensitive to very small objects and distant measurements.

Another method of making use of depth information for the purpose of detecting obstacles is to examine the 3-D slopes between detected points (Talukder, 2002). The points may be considered to compose a single obstacle if this slope—measured with respect to the horizontal—is steeper than a set slope and if their height difference falls within a predetermined range. Such obstacles may be found by searching the image from the bottom row and finding for each obstacle pixel in that row all the other pixels that meet the aforementioned criteria with respect to that pixel. The resulting points may also be classified as obstacle points, and the process repeated to find all such associated points. Finally, individual objects may be picked out by applying the transitive property of the above obstacle composition criteria. This works very well if the terrain and robot are both flat, but becomes a more difficult task as the terrain becomes rough or if the robot is expected to climb ramps.

Although the latter few approaches offer robust, proven functionality and are highly applicable to the type of sensor used, this project sought a simpler solution and didn't require segmentation or identification of individual objects. Thus, it began instead with the development of what would evolve into an implementation of one of the most common simple obstacle avoidance algorithms, simply turning away from the centroid of the detected offending points. However, this venerable approach was extended to consider not only obstacles themselves but also edges on the ground plane, an addition that enabled the detection of several additional danger scenarios that could not be handled by the traditional method alone.

3 Background

The Point Cloud Library includes many data types and numerous algorithms that make working with point clouds extraordinarily easy. The first of the algorithms used in this research was the 3-D voxel grid filter, which downsamples point cloud data by modeling the input dataset with a three-dimensional grid having cubic cells of user-supplied dimensions (Rusu, 2011). Each cell containing at least one point in the original image is then populated with a single voxel placed at the centroid of the points within that part of the input.

The research made extensive use of the plane edge detection algorithms simultaneously developed by Changhyun Choi, a Ph.D. student at the Georgia Institute of Technology (Choi, 2012). One of the utilized algorithms simply finds points bordering on those whose coordinates are set to NaN values, thereby computing the absolute boundaries of a plane. Particularly useful was his high curvature edge detection algorithm, which locates the points making up the boundaries between the floor and those objects that rest on it using integral images and Canny edge detection.

Integral images are a common technique in modern computer vision, and are used to detect distinctive image features (Viola and Jones, 2001). They are essentially tables storing for each coordinate in the corresponding image the sum of the pixel values lying in the box bounded by that coordinate and the upper-left corner of the image. Features from an integral image can then be used for a wide variety of purposes, including estimation of a 3-D image's surface normals.

The Canny edge detector starts by smoothing the input image to reduce noise (Canny, 1986). Next, the spatial gradients of the resulting image are measured in order to expose the edges, each of which is assigned a strength based on the distinctiveness of its gradient. The directions of the edges are determined in two dimensions using these gradients, then the directions are used to trace the edges. Those edges with strengths above a certain threshold are kept, while those with strengths between that value and a lower constant are kept only if they are connected to one or more edges from the former group.

PCL also provides a radius outlier removal, which accepts from the user a search radius and a minimum number of neighbors (O'Leary, 2011). It then searches the neighborhood surrounding each point in the image and removes that point if it has fewer than the specified number of neighbors.

As the project progressed, it became necessary to discern information about the ground plane directly in front of the robot. In order to determine which points were part of this plane, a linear model was calculated from the y- and z-coordinates of two known floor points, one—(z_1, y_1)—very near to the robot and the other—(z_2, y_2)—farther away. First, the plane's slope m was computed, as in Equation 1:

$$m = \frac{y_2 - y_1}{z_2 - z_1} \tag{1}$$

Next, the y-intercept y_0 was calculated using the average of the coordinates substituted into the point-slope form of a linear equation (Equation 2):

$$y_0 = \frac{y_1 + y_2}{2} - m\frac{z_1 + z_2}{2} \tag{2}$$

The resulting slope and intercept were both stored; thus, the y-coordinate corresponding to a given z-coordinate could be calculated using Equation 3's simple linear equation:

$$y = mz + y_0 \tag{3}$$

Sufficient deviation of the z-coordinate from its expected value allowed the conclusion that the point was not, in fact, part of the ground. Another—slightly less strict—threshold was used to broaden consideration to points that were very near the ground plane, as well as those actually composing it.

4 Approach

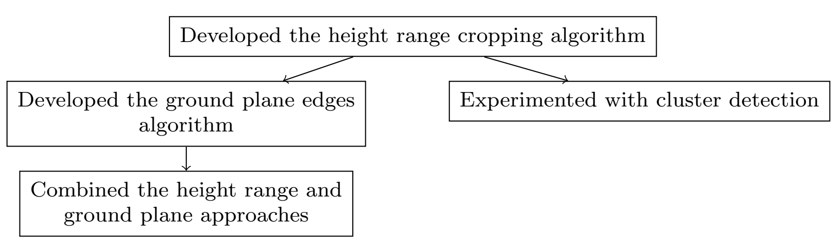


Figure 3: An overview of the progression of code development

This section of the report describes the development process of the project's algorithms and code. A brief visual overview covering the major stages of development appears as Figure 3, and a sub-sections providing a corresponding narrative of each stage follow. The included Appendix provides practical information about using the robot, setting up a development environment, and upgrading the PCL installation, as well as a glossary of ROS-related terminology, lists of useful ROS commands and documentation resources, and a complete copy of the final version of the code for this project.

4.1 The height range cropping algorithm

The point cloud coming off the Kinect exhibited noticeable noise, was extremely dense, and was consequently slow to transmit, display, and process. Thus, the first action taken was the application of a voxel grid filter to downsample the data and eradicate most of the noise while achieving better update speeds and faster processing time. Noticing that both Holz *et al.* and Nguyen used surface detection algorithms, while Koren and Borenstein simply didn't train sensors on the floor, a decision was made to crop the y-dimension so as to discard all points falling outside the robot's height range. This step—which was possible because the robot was going to be used chiefly in indoor environments possessing smooth terrain—made it possible to ignore the floor and focus exclusively on those points that represented actual obstacles. However, it also meant sacrificing the ability to climb ramps and traverse highly uneven floors.

The initial revision of the obstacle avoidance algorithm simply split the view into three parts: The center region was used to determine whether to proceed forward or turn, the latter of which was triggered whenever the number of points in this region exceeded a set noise threshold. Once the robot had entered a turning mode, it ceased forward motion and decided on a direction by choosing the peripheral vision field with fewer points in it. The entire field of view was cropped in the z-dimension in order to prevent the robot from being distracted by objects well ahead of its current position.

The biggest problem with this first version was that the robot was prone to becoming stuck oscillating in place between a left and right turn when faced with a sufficiently large obstruction. To work around this problem, the machine was only allowed to choose a direction of rotation as long as it wasn't already turning. In this way, it was forced to pick a direction whenever it first encountered an obstacle, then continue turning in that direction until it was able to drive forward again. As a side effect, it would now rotate *ad infinitum* when enclosed on all sides.

As testing continued, it became clear that the noise threshold was preventing the detection of many small—but still significant—obstacles. Decreasing this constant, however, caused the robot to turn spuriously in order to avoid offending points that were, in fact, nothing but noise. To solve this problem, the noise threshold was eliminated altogether by instead averaging the number of points in the forward regions of the last several images taken.

Next, a relatively minor but undeniable problem was discovered: given a scene where the only obstacle was located mainly within one half of the center region and didn't extend into either periphery, the robot might just as easily turn toward the object as away from it, thereby forcing itself to turn farther. Replacing the consideration of the peripheral regions with a simple turn away from the centroid of all points detected in the center region solved this issue.

4.2 Experiments with cluster detection

In an effort to allow the traversal of more complicated, maze-like situations, work began on a track that would eventually lead to a dead end. The idea was that, in severely confined spaces, the robot will attempt to turn long before reaching a wall, missing the side passageway because it turns all the way around before it ever gets to the point where it could have entered it. In order to solve this problem, an attempt was made at implementing the ability to distinguish between individual objects using the Point Cloud Library's built-in implementation of the simple Euclidean cluster detection algorithm. An iterative algorithm to determine the perpendicular distances between objects' edges was developed and implemented, and the new measurements were used to determine whether the robot could fit through a given gap. Next, the areas in front of the gaps were checked for blockages, then the candidate openings were ranked based on their distances from the center of view. Unfortunately, it soon became clear that although this approach did a better job of planning logical paths in confined spaces, it was largely unsuitable for use with the Kinect because of the sensor's inability to detect sufficiently-close obstacles. This meant that, before even getting through a gap, the bot would lose sight of it. In order to work around this hardware limitation, a state machine could have been implemented and the ability to measure driving distance could have been added. Unfortunately, such steps would have resulted in complete blindness during the time the robot was traversing the gap, and consequently a vulnerability to any unexpected environmental changes during that time. As such, the work was abandoned in search of a more general and universally-applicable solution.

4.3 The ground plane edges algorithm

Toward the end of development of the gap detection algorithm, another severe problem surfaced; it was discovered that, due to a combination of noise and distortions in the robot's coordinate system, both of the algorithms developed thus far were unable to detect objects as much as a couple of inches high. Noticing that all objects resting on or otherwise obscuring the ground created prominent occlusions on it, an effort was made toward detecting these discontinuities in the ground plane. First, a section of the ground corresponding to the region immediately in front of the robot—and hence in its path—was selected from the rest of the point cloud by tight cropping. Then, the slope of the floor was modeled to account for the Kinect's coordinate distortion, and all points falling outside a given height tolerance of this plane were filtered out. By examining the surface normals of this isolated sample, the edge points could be estimated. Next, a radius-driven minimum neighbors filter was applied to eliminate false positives. The results were promising when tested on a smooth carpet: after some fine-tuning, no false positives were being detected and a good number of edge points arose when any given obstruction was placed on the ground in front of the sensor. Unfortunately, speed had become a problem, as estimating the edge points was taking several seconds per sample.

It was in order to solve the speed issues that Choi's work was used; by making use of the organization of the Kinect's point cloud data instead of constructing an entire search tree for each frame, his algorithms were able to function at least an order of magnitude faster than the main PCL edge detection routines. At first, his absolute plane boundaries detector was used, but this was not ideal for two main reasons: First, it was unable to pick up objects in the middle of the portion of the plane which we were examining. Additionally, it was vulnerable to poor-quality floor samples far ahead, which would appear as rounded patches cutting into the distant edge of the floor plane measurably. Consequently, Choi's class was patched to enable greater control over its high curvature edge detection, which—similarly to the earlier approach—makes use of the plane's normals rather than its boundaries, and is therefore less vulnerable to noise once one has filtered out all but the closest points to the floor plane. A careful tuning of the edge detection and outlier removal parameters succeeded in eliminating almost all false positives while quite effectively capturing the footprints of those objects that intruded on the focal area of the ground plane. The robot was then programmed to turn away from the centroid of the detected edge points.

Unfortunately, this approach alone was unable to detect obstacles falling completely in front of the area of interest on the ground or expansive holes at any distance. In anticipation of such situations, the total number of detected ground points was compared to a set threshold; if it fell under this value, the robot would back up in order to get a broader view of the obstruction. This turned out to be a poor way to handle the situation, however, as the number of ground points varied significantly depending on the type of flooring, and backing up blindly often resulted in crashing into some invisible obstruction. As such, absolute plane boundaries were merged back in, this time in addition to curvature detection, and with the added restriction of ignoring expected border regions for the former in order to solve the problem of distant noise. Now, if the edge of the ground moved into the area where the plane was expected to be fully intact, it was assumed that there was either a hole encroaching upon the robot's position or an object between the Kinect and the close edge of the portion of the ground visible to it, and the detected edge points were pooled with the curvature keypoints in order to determine which direction to turn.

Together, the curvature points and outstanding plane boundary points were able to keep the Kinect from getting close enough to most obstacles to become completely blind. However, to further ensure the robot's safety, a third check was added: As the robot drove forward or turned, it constantly remembered the direction in which it would have turned—whether or not it had actually done so—given the data from the previous frame. In the case where no ground points were visible, and thus something was completely obscuring the Kinect's view, it would then begin to turn in the stored direction. This step proved effective against high-speed situations where moving objects' trajectories, when combined with the processing delay, brought obstructions out of the robot's view before it had yet evaluated them, as well as scenarios where a large obstruction was suddenly placed very close to the robot's front.

4.4 Combining the height range and ground plane approaches

While the floor occlusion detection approach worked very well for just about everything, it had a somewhat significant disadvantage that was not shared by the earlier height range–cropping approach: When confronted with a long, deep object having a region without floor contact—a bench or vending machine, for instance—the system was unable to detect it because of its lack of interactions with the floor plane. In order to solve this shortcoming, the two approaches were combined into a single program; each was placed in a separate thread, with a third thread to integrate the steering advice of each. This approach solved the problem of suspended objects and enabled faster response to objects detectable by the less computationally-intensive height region approach while preserving the robust detection capabilities of the surface analysis.

A detailed visual summary of the code's progression along with the time frames of feature additions is given in Figure 4. The test cases noteworthy enough to have prompted implementation changes are collected in Table 1. Additionally, the pseudo code for the final project is discussed in Figure 5.

5 Results and Discussion

While Section 4 discussed the changes that were made to the code as a result of the challenges cataloged in Table 1, we will now discuss the behavior of the final implementation when faced with these same scenarios. In order to quantify the system's success rate, intensive testing of these cases was performed.

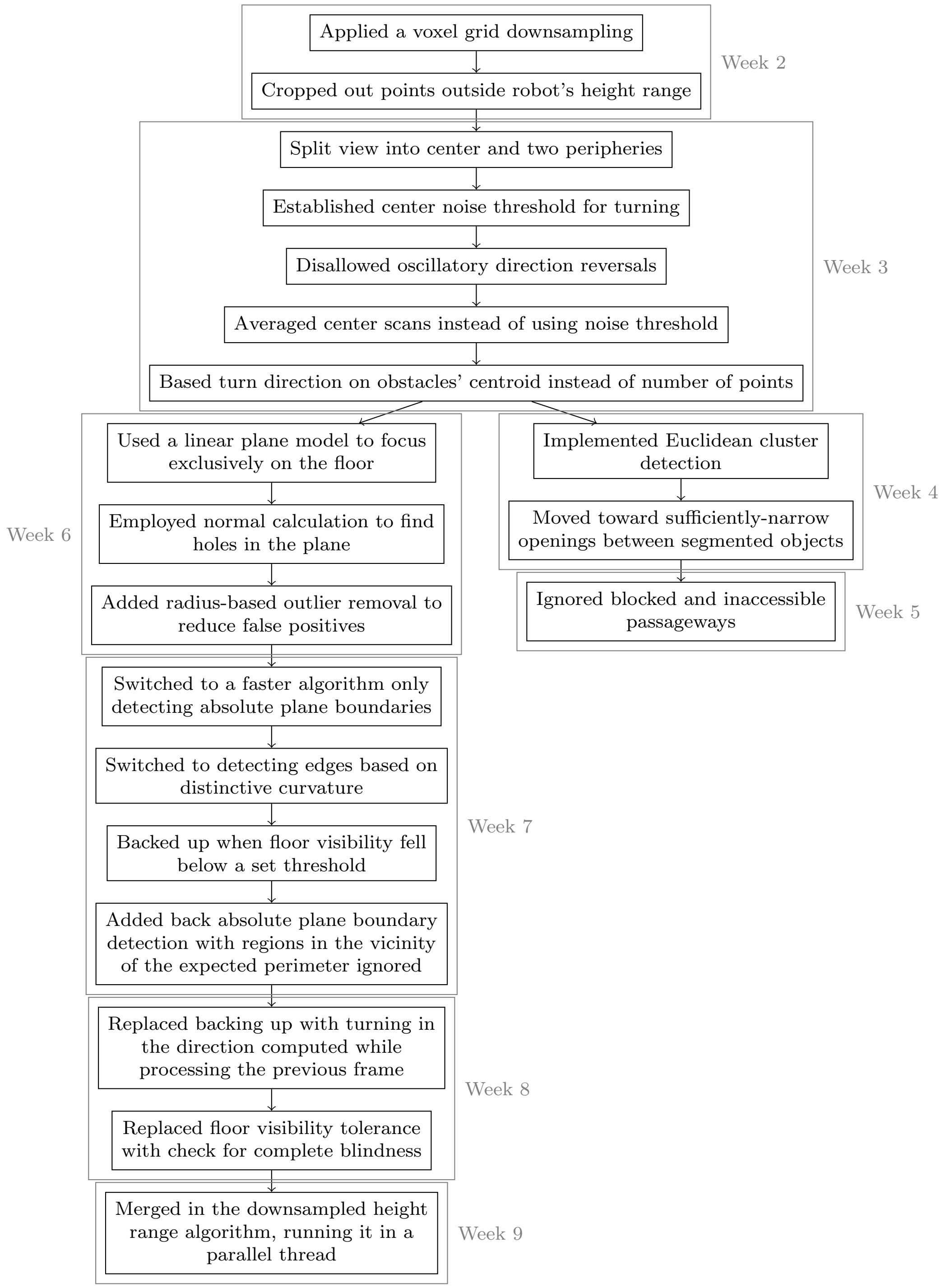
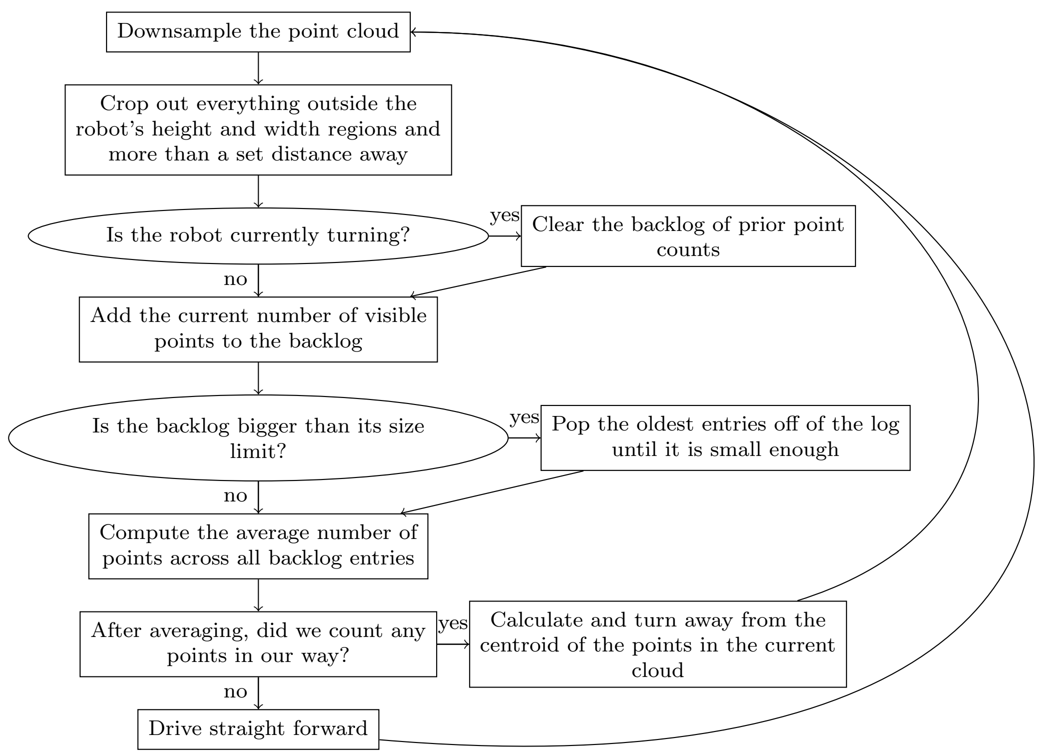
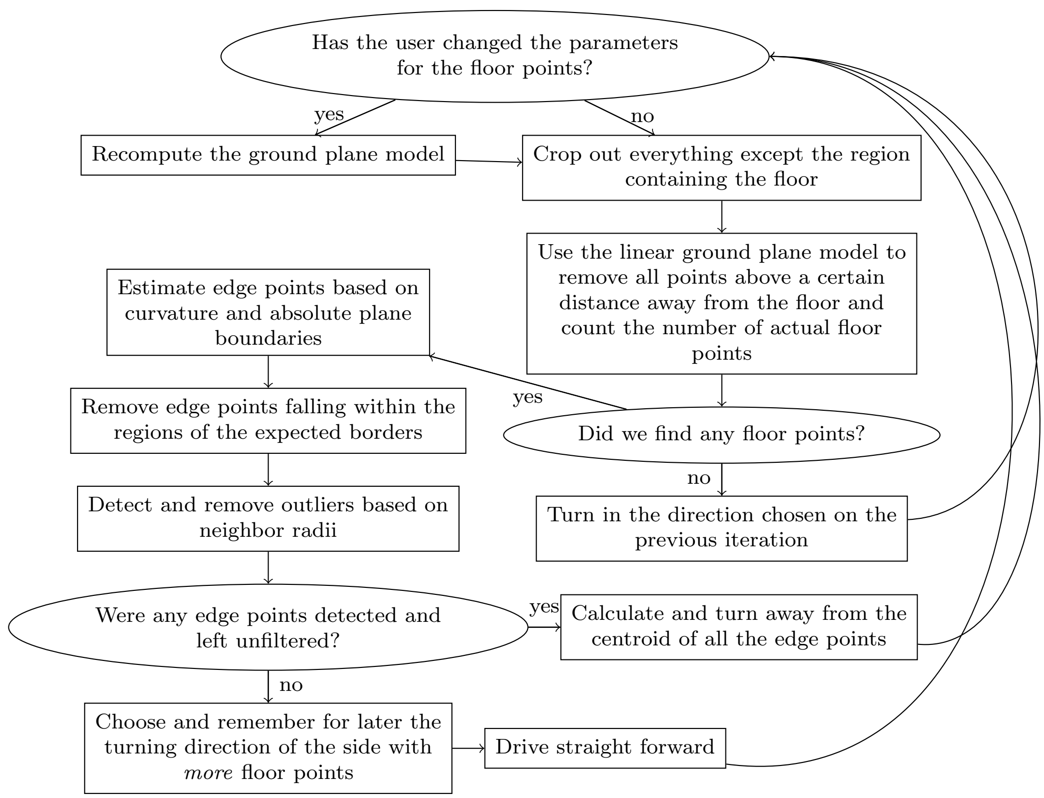


Figure 4: The complete progression of code development by week

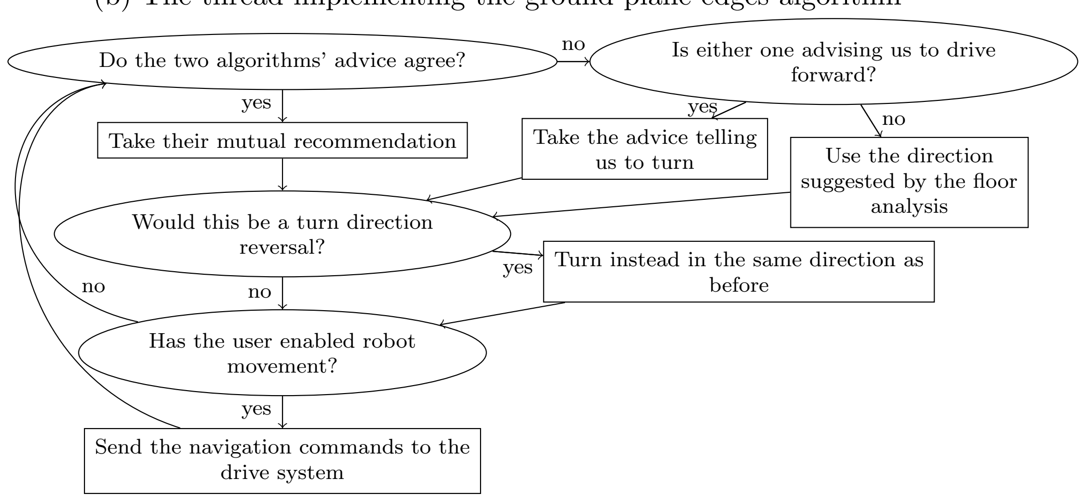


(a) The thread implementing the height range cropping algorithm

Figure 5: The final program's flow of control (continued on page 20)



(b) The thread implementing the ground plane edges algorithm



(c) The thread integrating the decisions of the two separate algorithms and issuing appropriate drive commands

Figure 5: The final program's flow of control *(continued)*

Table 1: Test cases to which the implementation was subjected

Scenario	Examples tested	True positives	False negatives	% correct
Wide obstacle	Wall, recycle bin, round trashcan	55	0	100%
Tall, thin obstacle	Chair leg, table leg	36	0	100%
Short object on the ground	Pad of paper, marker, serial console cable	29	25	54%
Hole in the ground	Staircase	18	0	100%
Suspended object	Vending machine	18	0	100%
	Overall:	156	25	86%

5.1 Intensive testing by structured test cases

For each row of Table 1, the specific cases given in the second column were tested at three different distances. First, each object was tested at long range; it would be placed ahead of the robot's area of interest so that the robot would first encounter it as it was in the process of approaching. Next, the object was placed at medium distance away, within the area of interest, before the algorithm was started, so that it was visible from the very beginning of the trial. Finally, it was placed extremely close to the robot so that it fell in front of the region visible to the Kinect and was not directly visible. At each distance, the item would be tested six times, three on each side of the robot, and each trial would be recorded as either a true positive or false negative, with the former also classified by expected or unexpected turn direction.

When the autonomous robot encounters a wide obstacle, both algorithms are able to sense it. Additionally, the robot is not allowed to reverse the direction of its turn, so it cannot enter an oscillatory state. A corollary to this behavior is that, if the robot is surrounded on all sides, it will continue spinning until freed. During testing, the system was able to detect the sample object 100% of the time, as well as choose the appropriate turn direction in all cases except when the wall was in the closest configuration. In such a case, the robot's view is completely blocked, and it is forced to decide on a direction arbitrarily unless it has already remembered one while processing a previous frame.

Tall, thin obstacles with a limited base footprint such as a chair or table are best detected by the height range algorithm. It is only thanks to the noise reduction attained by averaging several samples that it is able to spot such small objects. Testing of this functionality revealed 100$^{\%}$ accuracy for both detection and choice of direction.

Short objects resting on the floor, including markers and pads of paper, are typically invisible to the cropping algorithm. However, the curvature of the contact points between their edges and the ground plane is usually detectable to the other approach, largely depending upon the distinctiveness of their edges and amount of contact area. While the experimental results show only a 54$^{\%}$ success rate for this type of challenge, it should be noted that 18 of the 25 false negatives occurred at the closest distance. Detection was actually impossible in these cases because the objects had completely disappeared from the Kinect's field of view and weren't tall enough to occlude the visible region of the floor. If this portion of the test is to be discounted, one instead finds an 81$^{\%}$ accuracy, with the console cable detected every time. Given that the pad of paper mimics the ground plane and the marker has very little floor contact, this detection rate seems reasonable.

When a hole in the ground such as a descending staircase comes into view, the ground plane algorithm detects that the ground's absolute plane boundary has receded into an unexpected region. All plane boundary points falling outside of the expected regions are treated identically to curvature points, and the direction recommendation is accordingly based on their centroid. Such situations were detected without error and resulted in the direction of the shorter turn the majority of the time.

Upon approaching a suspended or apparently-suspended object such as a bench or vending machine, the plane detection algorithm sees no change in the floor plane. The cropping method, however, is able to see that an object is infringing upon the robot's height range, and directs it to turn away from the obstruction's centroid. These threats were always detected, and test unit was low enough to the floor to obscure the Kinect's view of the ground by the time it was very close, so that even the close trials discovered its presence. Of course, this ability would vary with the elevation of the bottom ledge of the object in question, as well as its depth if positioned in front of a wall.

On the whole, the algorithms suffer from almost no problems with false positives. However, intense external infrared radiation is capable of masking the Kinect's infrared grid. If, for instance, the robot is approaching a patch of direct sunlight shining through a window, it will appear from the Kinect's point cloud as though there is a hole in the ground: the absolute plane boundary will begin to recede into the expected ground region. Consequently, the robot will avoid such regions, treating them as actual danger.

When face-to-face with an entirely transparent glass wall, the Kinect's infrared grid passes straight through the window. Therefore, the barrier isn't reflected in the returned point cloud, and neither algorithm is able to see it at all.

5.2 Extensive testing by environmental observation

With the intensive test cases evaluated, the robot was released on all three floors of the building for unstructured test drives in order to determine how likely it was to encounter the already-tested situations. Each time the robot turned, either a true positive or a false one was recorded, depending on whether there was actually an object in its way. Additionally, false negatives were to be noted every time the robot actually ran into anything; however, this never occurred in the test environment. The results of such observation are noted in Table 2.

Each false positive uncovered by the first two courses occurred during the robot's transition from a tile surface to a carpet, or vice versa, and resulted when the slight height changes between the surfaces triggered the ground plane curvature detection, and could likely be solved simply by fine-tuning the ground plane curvature detection parameters. Such cases never resulted in more than a few degrees of turning before the robot resumed driving forward. In the third and fourth trials, the robot drove across a floor with larger, less regular tiles; here, it would turn away from particularly uneven edges. As before, it also picked up a few false positives when transitioning between floorings. However the majority of its unprompted turns in this case stemmed from sunlight: While on the tile floor, it encountered several patches and spent some time wandering between them before being freed.

The addition of the ground plane edge detection to the traditional obstacle avoidance solution brings several key advantages: First, examining the curvature of the plane enables the detection of almost any obstacle that makes contact with the floor, including those that are very short. Next, looking for absolute plane edges means hazards that have no corresponding obstacle within the robot's height range—such as holes in the ground—can be easily avoided. Finally, since objects passing in front of the infrared emitter occlude the floor in front of the sensor, examining plane edges also reveals the presence of objects suddenly appearing very close to the sensor, including animate objects whose motion is perpendicular to the robot's; in this way, the algorithm is able to infer the presence of objects that are closer than the Kinect's hardware-limited minimum range. The latter principle makes the plane analysis approach especially beneficial for sensors such as the Kinect that are unable to see objects closer than a certain distance away.

As noted earlier, the complete implementation fails to perform in two specific cases: It is vulnerable to false positives in patches of direct sunlight and to false negatives in the case of transparent walls. Such problems stem from the limitations of the infrared-based Kinect point cloud; however, they could likely be solved by examining the Kinect's RGB data in tandem with its depth values.

Table 2: Observations from unstructured test drives

Location	**True positives**	**False positives**	**False negatives**	**Success**
First floor	29	1	0	97%
Second floor	55	2	0	96%
Third floor	66	4	0	94%
Atrium, side wing	105	17	0	85%
Overall:	255	24	0	91%

6 Table of Hours Worked

The time spent working on the project is addressed in Table 3. The time frame of the code's evolution is described in Figure 4.

7 Conclusion

The combination of the two methods of achieving obstacle avoidance was highly successful because they complemented each other so well: The crop-

Table 3: Hours spent working on the project

Week	Research	Implementation	Testing	Documentation	Administration	Subtotal
1	14:00			6:20	18:00	38:20
2	22:00	9:00	3:00	3:20	1:30	38:50
3		12:00	4:00	16:00	12:40	44:40
4	3:00	8:00	5:30	4:00	5:00	25:30
5	3:00	13:00	3:00	10:00	3:00	32:00
6		11:10	9:00	4:00	20:00	44:10
7		18:00	8:00	5:00	14:00	45:00
8		8:00	5:00	21:00	8:10	42:10
9		4:00	3:20	28:00	5:00	40:20
10		3:00	10:00	6:30	14:30	34:00
					Total:	385:00

ping approach, which was by nature incapable of detecting very short objects or floor discontinuities, was able to rely on the less common plane surface analysis for these tasks. The plane analysis, on the other hand, was poor at detecting the truly- or apparently-suspended objects that were readily detected by the other. As might be expected, then, the synthesis of the two algorithms was able to autonomously navigate in almost all tested indoor situations without a problem. Thus, the project's goals were realized.

References

Canny, John. (1986). A computational approach to edge detection. In *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI),* Vol. 8, no. 6, November 1986. July 26, 2012. ⟨`http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4767851`⟩.

Choi, Changhyun. (2012). Organized Edge Detection. PCL Google Summer of Code Developers Blog. July 26, 2012. ⟨`http://pointclouds.org/blog/gsoc12/cchoi`

Herrera, Daniel *et al.* (2011). Accurate and practical calibration of a depth and color camera pair. In *Computer Vision, Image Analysis, and Processing (CAIP 2011).* June 8, 2012. ⟨`http://www.ee.oulu.fi/~dherrera/papers/2011-depth_calibration.pdf`⟩.

Holz, Dirk *et al.* (2011). Real-time plane segmentation using RGB-D cameras. In *Proceedings of the 15th RoboCup International Symposium,* Istanbul, July 2011. June 11, 2012. ⟨`ais.uni-bonn.de/papers/robocup2011_holz.pdf`⟩.

Koren, Y. and Borenstein, J. (1991). Potential field methods and their inherent limitations for mobile robot navigation. In *Proceedings of the IEEE Conference on Robotics and Automation,* Sacramento, April 7-12, 1991, 1398-1404. June 8, 2012. ⟨`http://www-personal.umich.edu/~johannb/Papers/paper27.pdf`⟩.

Nguyen, Van-Duc. (2012). Obstacle avoidance using the Kinect. June 15, 2012 ⟨`http://scribd.com/doc/80464002/Obstacle-Avoidance-Using-the-Kinect`⟩.

O'Leary, Gabe. (2011). Removing Outliers Using a Conditional or RadiusOutler Removal. Point Cloud Library Tutorials. July 26, 2012. ⟨`http://pointclouds.org/documentation/tutorials/remove_outliers.php`⟩.

Rusu, Radu B. (2011). Downsampling a PointCloud using a VoxelGrid filter. Point Cloud Library Tutorials. July 26, 2012. ⟨`http://pointclouds.org/documentation/tutorials/voxel_grid.php`⟩.

Talukder, A. *et al.* (2002). Fast and reliable obstacle detection and segmentation for cross-country navigation. In *Proceedings of the IEEE Intelligent Vehicle Symposium (IVS 02),* 610-618. June 25, 2012. ⟨`http:`

//users.soe.ucsc.edu/~manduchi/papers/IV02.pdf⟩.

Viola, Paul and Jones, Michael. (2001). Robost real-time object detection. In *Second International Workshop on Statistical and Computational Theories of Vision (IJCV 2001),* Vancouver, July 13, 2001. July 26, 2012. ⟨http://research.microsoft.com/~viola/Pubs/Detect/violaJones_IJCV.pdf⟩.

Appendices

1 Starting the TurtleBot

1. Disconnect both chargers from the robot, if applicable.
2. Turn on the iRobot Create by pressing the power button on its back; the power light should turn green.
3. Unplug and remove the laptop from the TurtleBot.
4. Open the laptop's lid and press the power button.
5. Close the laptop, replace it in the chassis, and reconnect the cables.
6. Wait until the Ubuntu startup noise sounds; at this point, the robot is ready to accept connections.
7. From another machine, enter: `$ ssh turtlebot@turtlebot.rit.edu`
8. Once authenticated, ensure that the robot service is running: `$ sudo service turtlebot start`
9. The iRobot Create should beep and its power light should go out. The robot is now ready for use.
10. Enter the following command to enable the Kinect: `$ nohup roslaunch turtlebot_bringup kinect.launch &`
11. Enter the following command to enable the Interactive tab in RViz and allow GUI-driven teleoperation: `$ nohup rosrun turtlebot_interactive_markers turtlebot_marker_server &`
12. You may now safely close your robot shell connection: `$ exit`

2 Stopping the TurtleBot

1. Connect to the robot.
2. Stop the Interactive Markers server: `` $ kill `ps -ef | grep marker_server | tr -s " " | cut -d " " -f 2` ``

3. Stop the Kinect driver with: `$ kill ‘ps -ef | grep kinect.launch | grep -v grep | tr -s " " | cut -d " " -f 2‘`

4. Release the iRobot Create with: `$ rosservice call /turtlebot_node/set_operation_mode 1`

5. At this point, it is safe to plug the charger into the iRobot Create. If you want to turn off the laptop as well, continue with the below steps instead.

6. Shut down the robot laptop: `$ sudo halt`

7. Turn off the Create by pressing its power button.

8. Plug in the chargers for the iRobot Create and the laptop.

3 Setting up a Development Workstation

1. Ready a machine for your use. (We'll assume you're using Ubuntu 10.04 through 11.10.)

2. Ensure that your system has either a hostname or a static IP that is visible from the robot.

3. Download the ROS package signing key: `$ wget http://packages.ros.org/ros.key`

4. Add the signing key to your system: `$ sudo apt-key add ros.key`

5. Add the ROS repository to your system: `$ sudo apt-add-repository http://packages.ros.org/ros/ubuntu`

6. Update your repository cache: `$ sudo apt-get update`

7. Install the TurtleBot desktop suite: `$ sudo apt-get install ros-electric-turtlebot-desktop`

8. Edit your bash configuration(`$ $EDITOR ~/.bashrc`), adding the following lines to the end:

- `source /opt/ros/electric/setup.bash`
- `export ROS_MASTER_URI=http://turtlebot.rit.edu:11311`
- `export ROS_PACKAGE_PATH=<directory where you'll store your-source code>:$ROS_PACKAGE_PATH`

9. Write and close the file, then enter the following command in each of your open terminals: `$ source ~/.bashrc`
10. Install the Chrony NTP daemon: `$ sudo apt-get install chrony`
11. Synchronize the clock: `$ sudo ntpdate ntp.rit.edu`
12. If the robot and the workstation both have hostnames, but they are in different domains, perform the following steps. (In this example, the robot is at `turtlebot.rit.edu` and the workstation is at `turtlecmd.wireless.rit.edu`.)
    - (a) On each machine, right-click the Network Manager applet in the notification area, choose *Edit Connections...*, and open the properties for the specific connection that is being used.
    - (b) On the IPv4 Settings tab, change the *Method* dropdown to `Automatic (DHCP) address only.`
    - (c) In the *DNS servers* field, enter the same DNS servers that were being used, with commas in between (e.g. `129.21.3.17, 129.21.4.18`).
    - (d) In the *Search domains* field, enter the local machine's domain first, followed by the remote machine's. For instance, in our example, one might enter `rit.edu., wireless.rit.edu.` on the robot and `wireless.rit.edu., rit.edu.` on the workstation.
    - (e) Save all your changes and exit the Network Connections dialog.
    - (f) Force a reconnection by clicking on the Network Manager applet, then selecting the network to which you are already connected.
13. If the workstation has no qualified hostname and is to be reached via a static IP, make the following changes on the robot instead:
    - (a) Edit the robot's hosts file: `$ sudo $EDITOR /etc/hosts`

(b) For each static host, add a line such as: ⟨`IP address`⟩ ⟨`hostname`⟩. It is important to note that the *hostname* you use must exactly match the output of `$ hostname` on the development workstation.

(c) Save the file and quit; the changes should take effect immediately and automatically.

4 Upgrading to the Latest Version of PCL

The version of the Point Cloud Library shipped with ROS lags significantly behind that available directly from the community. These instructions show how to install the latest version of PCL on top of an existing ROS Electric installation.

1. Create a folder to contain the build files and a replacement copy of the `perception_pcl` stack: `$ mkdir ~/ros`
2. Install the Python package management utilities: `$ sudo apt-get install python-setuptools`
3. Install the dependencies for the `rosinstall` utility: `$ sudo easy_install -U rosinstall`
4. Create a new ROS overlay in the current directory: `$ rosinstall . /opt/ros/electric`
5. Install any missing build dependencies: `$ rosdep install perception_pcl`
6. Obtain a rosinstall file describing the repository for the perception stack: `$ roslocate info perception_pcl` ⟩ `perception_pcl.rosinstall`
7. Edit the rosinstall file to point at the correct repository for Electric: `$ sed -i s/unstable/electric_unstable/ perception_pcl.rosinstall`
8. Fetch the makefiles for the stack: `$ rosinstall . perception_pcl.rosinstall`
9. Inform your shell of the overlay's location: `$ source setup.bash`

10. Move into the `cminpack` directory: `$ cd perception_pcl/cminpack`
11. Build the package: `$ make`
12. Move into the `flann` directory: `$ cd ../flann`
13. Build the package: `$ make`
14. Move into the `pcl` directory: `$ cd ../pcl`
15. Select the most recent tagged version of the code, for instance: `$ sed -i s/\\/trunk/\\/tags\\/pcl-1.6.0/ Makefile`
16. Build the PCL codebase: `$ make`
17. Move into the `pcl_ros` directory: `$ cd ../pcl_ros`
18. Build the ROS PCL bindings: `$ make`
19. Move back out into the stack: `$ cd ..`
20. Build the stack's particulars: `$ make`
21. Edit your `bashrc` file to add the following line after the line that sources the system-wide ROS `setup.bash`: `source ~/ros/setup.bash`
22. If you intend on continuing to use your current terminals, enter the following in each after saving the file: `$ source ~/.bashrc`

5 Backporting in a Class from the PCL Trunk

Often, the trunk version of the Point Cloud Library will fail to compile; therefore, it may be desirable to backport a specific class from trunk into a released copy of the library. For instance, the code written for this project relies on the OrganizedEdgeDetection class, which—at the time of writing—is only available from trunk. These steps present an example of how to backport revision 6467 of this specific class and the new subsystem on which it relies into the 1.6.0 release of PCL. We'll assume that the steps from Section 4 of the Appendix have already been completed.

1. Change to the source directory of your newly-compiled copy of the Point Cloud Library: `$ roscd pcl/build/pcl_trunk`

2. Download the required 2d subsystem: `$ svn checkout http://svn.pointclouds.org/pcl/trunk/2d@r6467`

3. Move into the directory that is to contain the OrganizedEdgeDetection header: `$ cd features/include/pcl/features`

4. Download the header: `$ svn export http://svn.pointclouds.org/pcl/trunk/features/include/pcl/features/organized_edge_detection.h@r6467`

5. Move into the directory that is to contain the templated code: `$ cd impl`

6. Download the templated source: `$ svn export http://svn.pointclouds.org/pcl/trunk/features/include/pcl/features/impl/organized_edge_detection.hpp@r6467`

7. Move into the directory that is to contain the instantiations: `$ cd ../../../../src`

8. Download the instantiations list: `$ svn export http://svn.pointclouds.org/pcl/trunk/features/src/organized_edge_detection.cpp@r6467`

9. Move back into the root of the code directory: `$ cd ../..`

10. Edit the `features` package's build configuration: `$ $EDITOR features/CMakeLists.txt`

    (a) At the end of the SUBSYS_DEPS list, add: `2d`

    (b) Under `set(incs`, add: `include/pcl/${SUBSYS_NAME}/organized_edge_detection.h`

    (c) Under `set(impl_incs`, add: `include/pcl/${SUBSYS_NAME}/impl/organized_edge_detection.hpp`

    (d) Under `set(srcs`, add: `src/organized_edge_detection.cpp`

11. Apply the necessary patches, as described in the `included` directory that comes with my code.

12. Return to the package root: `$ roscd pcl`

13. Build in the changes and relink: `$ make`

6 ROS Glossary

- **message.** A ROS communication packet that carries information between *nodes* in a single direction. New message types may be declared by creating text files in a package's `msg` directory and enabling the `rosbuild_genmsg()` directive in its `CMakeLists.txt` file. The composition of existing message types may be found using the `rosmsg show` command. ROS automatically generates a C++ struct for each message type; these struct types are declared in the ⟨`package`⟩`/`⟨`messagetype`⟩`.h` headers; these must be included before they may be used. Two examples of useful message types are `std_msgs/Int32`—an `int`—and `geometry_msgs/Twist`—used for driving the Create around.

- **node.** A single ROS executable, which may be added to a *package* by appending a `rosbuild_add_executable` directive to the `CMakeLists.txt` file of the latter. Once the package has been compiled using GNU Make, each of its nodes may be run using the `rosrun` command.

- **package.** A "project" containing executables and/or libraries; new packages may be created with the `roscreate-pkg` command, and existing ones may be imported into a dependent one by adding `depend` tags to its `manifest.xml` file. A couple of important packages are `roscpp`, which contains the `ros/ros.h` header that allows one to interface with ROS, and `pcl_ros`, which depends on the `pcl` package to provide the Point Cloud Library bindings.

- **parameter.** A variable hosted on the ROS parameter server; it is persistent across multiple runs of a node, provided that the ROS master is not restarted. Depending upon the node's implementation, changing one of its parameters while it is running may also affect its continued behavior. The user interface to the parameter server is provided by the `rosparam` command, while the C++ API supports the analogous `setParam`, `getParam`, `deleteParam`, and other methods located in the `ros::NodeHandle` class.

- **service.** A link between ROS *nodes* allowing two-way communication carried in the form of service types from a client to a server.

The user may call an existing service using the `rosservice` command, while C++ programs may create and call services via the `ros::ServiceServer` and `ros::ServiceClient` classes, which may be built by means of the `advertiseService` and `serviceClient` methods of `ros::NodeHandle`. Service types—the analog of *messages* from the world of *topics*—may be declared in text files within a package's `srv` directory after enabling its `CMakeLists.txt` file's `rosbuild_gensrv()` call. Service types' components may be seen with the `rosservice show` invocation, and C++ service structs are generated and used similarly to those for *messages*. One example of a service used on the TurtleBot is `/turtlebot_node/set_operation_mode`, which takes an integer—usually 1, 2, or 3—responds whether it is valid, and brings the iRobot Create into either Passive, Safety, or Full mode, respectively.

- **topic.** A link between ROS *nodes* that allows one-way communication of information carried in the form of *messages* from a publisher to one or more subscribers. The user may publish or subscribe to a topic by means of the `rostopic` command, while C++ programs may do so by creating a `ros::Publisher` or `ros::Subscriber` object using the `ros::NodeHandle` class's `advertise` or `subscribe` method. Examples of topics on the TurtleBot are `/cmd_vel`—modified in order to to control the Create's drive and steering—and `/cloud_throttled`—which provides the point cloud from the Kinect.

7 ROS Commands

This section aims to list the commands needed to interface with ROS and briefly address their commonly-used arguments. For the sake of clarity, the following conventions are used: unless otherwise noted, arguments in ⟨angled brackets⟩ are required, while those in [square brackets] are optional.

- **roscore** brings up a ROS master, which is useful for experimenting with ROS on one's workstation when the TurtleBot is not online. However, in order to actually use this master instead of the TurtleBot's, one must do the following in each pertinent shell: `$ export ROS_MASTER_URI=http://localhost:11311`

- **roscd** ⟨**package**⟩ is a convenience script that allows one to immediately move into the root directory of the specified *package*.
- **roscreate-pkg** ⟨**package**⟩ [**dependencies**] initializes package directories to contain the source code for one or more modules. The package directory structure will be created in a new subdirectory called *package* within the current folder, which must appear in `$ROS_PACKAGE_PATH`. Typically, the *dependencies* should include the `roscpp` package—which contains the ROS C++ bindings—as well as any other ROS packages that will be used, such as `pcl_ros`. The dependencies may be modified later by editing the `manifest.xml` file in the root directory of the package to add additional `depend` tags. ROS nodes may be added to a project by adding a `rosbuild_add_executable` directive to the `CMakeLists.txt` file, also located in the package root.
- **rosmsg** ⟨**verb**⟩ ⟨**arguments**⟩ shows information about currently-defined message types that may be passed over topics. When *verb* is `show` and the *arguments* are ⟨`package`⟩`/`⟨`messagetype`⟩, for instance, an "API reference" of the types and names of the variables in the message's corresponding struct hierarchy is displayed.
- **rosparam** ⟨**verb**⟩ ⟨**parameterpath**⟩ supports *verb*s such as: `list`, `set`, `get`, and `delete`. In the case of the former, the *parameterpath* may be omitted if a complete listing is desired. The `set` invocation expects an additional argument containing the new value to be appended.
- **rosrun** ⟨**package**⟩ ⟨**node**⟩ is simply used to execute a *node* once the *package* has been compiled with `make`.
- **rosservice** ⟨**verb**⟩ ⟨**servicepath**⟩ allows interfacing with the presently-available services over which service types may be sent. When *verb* is `list`, *servicepath* may optionally be omitted, in which case all services will be shown. With `call`, the user may call a service by passing arguments and receive a response as supported by the service type. The `type` *verb* is important, as it returns the package and service type corresponding to the service at the specified path.
- **rossvc** ⟨**verb**⟩ ⟨**arguments**⟩ allows querying currently-defined service types for passing over services. When *verb* is `show` and the *arguments* are of the form ⟨`package`⟩`/`⟨`servicetype`⟩, the command outputs an "API reference"–style listing of the types and names of the variables in

the struct type representing the service type.

- **rostopic ⟨verb⟩ ⟨topicpath⟩** provides a bridge to currently-advertised topics over which messages may be passed. When *verb* is `list`, *topicpath* may optionally be omitted to list all available topics. With `echo`, the user may subscribe to a topic and view the data that is being subscribed to it. Conversely, invocation with `pub` allows publishing to the topic, which will influence the nodes that are presently subscribed to it. The `type` *verb* is particularly useful: it prints the package and message type of a given registered topic.

8 Useful Links

Unfortunately, much of the ROS documentation is rather terse and unfriendly. Here, I've made an effort to catalog the documentation that I found most helpful. I've also included documentation from the PCL website, which is perhaps better organized and certainly more comprehensive than that available on the ROS Wiki.

- ROS TurtleBot wiki: `http://ros.org/wiki/TurtleBot`
- ROS tutorials: `http://ros.org/wiki/ROS/Tutorials`
- ROS C++ tutorials: `http://ros.org/wiki/roscpp/Tutorials`
- ROS C++ overview: `http://ros.org/wiki/roscpp/Overview`
- ROS C++ API reference: `http://ros.org/doc/electric/api/roscpp/html`
- ROS Kinect calibration tutorials: `http://ros.org/wiki/openni_launch/Tutorials`
- ROS PCL data type integration examples: `http://ros.org/wiki/pcl_ros`
- PCL tutorials: `http://pointclouds.org/documentation/tutorials`
- PCL API reference (1.1.0): `http://docs.pointclouds.org/1.1.0`
- PCL API reference (1.6.0): `http://docs.pointclouds.org/1.6.0`

- PCL API reference (trunk): `http://docs.pointclouds.org/trunk`

9 Code Listing

```
#include "ros/ros.h"
#include "pcl_ros/point_cloud.h"
#include "pcl/point_types.h"
#include "pcl/filters/passthrough.h"
#include "pcl/filters/voxel_grid.h"
#include "pcl/features/organized_edge_detection.h"
#include "pcl/filters/radius_outlier_removal.h"
#include "geometry_msgs/Twist.h"

/**
Represents a request for a particular drive action, which may be to go straight, turn left, or turn right
*/
enum DriveAction
{
    FORWARD, LEFT, RIGHT
};

/**
Performs obstacle detection and avoidance using two algorithms simultaneously
*/
class TandemObstacleAvoidance
{
    private:
        ros::NodeHandle node;
        ros::Publisher velocity;
        ros::Publisher panorama; //downsampled cloud
        ros::Publisher height; //heightRange's region of interest
        ros::Publisher ground; //groundEdges's region of interest
        ros::Publisher occlusions; //ground-level occlusions
```

```
        DriveAction currentMOTION; //pilot's account of what was last done: detection algorithms should not modify!
        DriveAction directionsPrimary; //the height range algorithm's suggestion
        DriveAction directionsSecondary; //the ground edges algorithm's suggestion
        std::list<int> heightRangeFrontSamples;
        double last_GROUND_CLOSEY, last_GROUND_CLOSEZ, last_GROUND_FARY, last_GROUND_FARZ; //only recalculate the below when necessary
        double GROUND_SLOPE, GROUND_YINTERCEPT; //model the ground's location
        DriveAction groundLastForcedTurn; //which way we would have turned: should never be set to FORWARD

        const char* directionRepresentation(DriveAction plan)
        {
            switch(plan)
            {
                case LEFT:
                    return "LEFT";
                case RIGHT:
                    return "RIGHT";
                default:
                    return "FORWARD";
            }
        }

    public:
        /**
        Constructs the object, starts the algorithms, and blocks until the node is asked to shut down. By default, all calculations are performed, but no commands are actually sent to the drive system unless the user sets the <tt>drive_move</tt> parameter to <tt>true</tt>, using the <tt>rosparam</tt> command, for instance.
```

```
    @param handle a <tt>NodeHandle</tt> defined with the nodespace containing the runtime parameters, including <tt>drive_move</tt>
    */
    TandemObstacleAvoidance(ros::NodeHandle& handle):
        node(handle), velocity(node.advertise<geometry_msgs::Twist>("/cmd_vel", 1)), panorama(node.advertise<pcl::PointCloud<pcl::PointXYZ> >("panorama", 1)), height(node.advertise<pcl::PointCloud<pcl::PointXYZ> >("height", 1)), ground(node.advertise<pcl::PointCloud<pcl::PointXYZ> >("ground", 1)), occlusions(node.advertise<pcl::PointCloud<pcl::PointXYZ> >("occlusions", 1)), currentMOTION(FORWARD), directionsPrimary(FORWARD), directionsSecondary(FORWARD), last_GROUND_CLOSEY(0), last_GROUND_CLOSEZ(0), last_GROUND_FARY(0), last_GROUND_FARZ(0), groundLastForcedTurn(LEFT)
    {
        ros::MultiThreadedSpinner threads(3);
        ros::Subscriber heightRange=node.subscribe("/cloud_throttled", 1, &TandemObstacleAvoidance::heightRange, this);
        ros::Subscriber groundEdges=node.subscribe("/cloud_throttled", 1, &TandemObstacleAvoidance::groundEdges, this);
        ros::Timer pilot=node.createTimer(ros::Duration(0.1), &TandemObstacleAvoidance::pilot, this);

        threads.spin(); //blocks until the node is interrupted
    }

    /**
    Performs the primary obstacle detection and motion planning by downsampling the tunnel-like region in front of the robot and matching its approximate height and width
```

```
    @param cloud a Boost pointer to the <tt>PointCloud</tt> from the sensor
    */
    void heightRange(const pcl::PointCloud<pcl::PointXYZ>::Ptr& cloud)
    {
        //declare "constants," generated as described in pilot
        double CROP_XRADIUS, CROP_YMIN, CROP_YMAX, CROP_ZMIN, CROP_ZMAX, HEIGHT_DOWNSAMPLING;
        int HEIGHT_SAMPLES;
        bool HEIGHT_VERBOSE;

        //populate "constants," generated as described in pilot
        node.getParamCached("crop_xradius", CROP_XRADIUS);
        node.getParamCached("crop_ymin", CROP_YMIN);
        node.getParamCached("crop_ymax", CROP_YMAX);
        node.getParamCached("crop_zmin", CROP_ZMIN);
        node.getParamCached("crop_zmax", CROP_ZMAX);
        node.getParamCached("height_downsampling", HEIGHT_DOWNSAMPLING);
        node.getParamCached("height_samples", HEIGHT_SAMPLES);
        node.getParamCached("height_verbose", HEIGHT_VERBOSE);

        //variable declarations/initializations
        pcl::PassThrough<pcl::PointXYZ> crop;
        pcl::VoxelGrid<pcl::PointXYZ> downsample;
        pcl::PointCloud<pcl::PointXYZ>::Ptr downsampled(new pcl::PointCloud<pcl::PointXYZ>);
        pcl::PointCloud<pcl::PointXYZ>::Ptr front(new pcl::PointCloud<pcl::PointXYZ>);
        int averageObstacles=0; //number of points in our way after averaging our readings

        //downsample cloud
        downsample.setInputCloud(cloud);
```

```
if(HEIGHT_DOWNSAMPLING>=0) downsample.setLeafSize((float)HEIGHT_DOWNSAMPLING, (float)HEIGHT_DOWNSAMPLING, (float)HEIGHT_DOWNSAMPLING);
downsample.filter(*downsampled);

//crop the cloud
crop.setInputCloud(downsampled);
crop.setFilterFieldName("x");
crop.setFilterLimits(-CROP_XRADIUS, CROP_XRADIUS);
crop.filter(*front);

crop.setInputCloud(front);
crop.setFilterFieldName("y");
crop.setFilterLimits(CROP_YMIN, CROP_YMAX);
crop.filter(*front);

crop.setInputCloud(front);
crop.setFilterFieldName("z");
crop.setFilterLimits(CROP_ZMIN, CROP_ZMAX);
crop.filter(*front);

if(currentMOTION!=FORWARD) heightRangeFrontSamples.clear(); //use straight snapshots while turning
heightRangeFrontSamples.push_front(front->size());
while(heightRangeFrontSamples.size()>(unsigned)HEIGHT_SAMPLES) heightRangeFrontSamples.pop_back(); //constrain our backlog

//compute average number of points
for(std::list<int>::iterator location=heightRangeFrontSamples.begin(); location!=heightRangeFrontSamples.end(); location++)
    averageObstacles+=*location;
averageObstacles/=heightRangeFrontSamples.size();

```

```
        //let's DRIVE!
        if(averageObstacles>0) //something is in our way!
        {
            float centroidX=0;

            //compute the centroid of the detected points
            for(pcl::PointCloud<pcl::PointXYZ>::iterator point=front->begin(); point<front->end(); point++)
                centroidX+=point->x;
            centroidX/=front->size();

            if(HEIGHT_VERBOSE)
                ROS_INFO("HEIGHT_RANGE :: Seeing %4d points in our way\n -> Centroid is at %.3f i", averageObstacles, centroidX);

            if(centroidX<0) //obstacle(s)'[s] centroid is off to left
                directionsPrimary=RIGHT;
            else //centroidX>=0
                directionsPrimary=LEFT;
        }
        else //nothing to see here
            directionsPrimary=FORWARD;

        //send our imagery to any connected visualizer
        panorama.publish(*downsampled);
        height.publish(*front);
    }

    /**
    Performs secondary obstacle detection and motion planning by detecting curvature changes on, boundaries of, and absense of the ground plane
    @param cloud a Boost pointer to the (organized) <tt>PointCloud</tt> from the sensor
```

```
*/
void groundEdges(const pcl::PointCloud<pcl::PointXYZRGB>::Ptr& cloud)
{
    //declare "constants," generated as described in pilot
    double CROP_XRADIUS, CROP_YMIN, CROP_YMAX, CROP_ZMIN, CROP_ZMAX, GROUND_BUMPERFRONTAL, GROUND_BUMPERLATERAL, GROUND_CLOSEY, GROUND_CLOSEZ, GROUND_FARY, GROUND_FARZ, GROUND_TOLERANCEFINE, GROUND_TOLERANCEROUGH, GROUND_NORMALSMOOTHING, GROUND_THRESHOLDLOWER, GROUND_THRESHOLDHIGHER, GROUND_OUTLIERRADIUS;
    int GROUND_NORMALESTIMATION, GROUND_OUTLIERNEIGHBORS;
    bool GROUND_VERBOSE;

    //populate "constants," generated as described in pilot
    node.getParamCached("crop_xradius", CROP_XRADIUS);
    node.getParamCached("crop_ymin", CROP_YMIN);
    node.getParamCached("crop_ymax", CROP_YMAX);
    node.getParamCached("crop_zmin", CROP_ZMIN);
    node.getParamCached("crop_zmax", CROP_ZMAX);
    node.getParamCached("ground_bumperfrontal", GROUND_BUMPERFRONTAL);
    node.getParamCached("ground_bumperlateral", GROUND_BUMPERLATERAL);
    node.getParamCached("ground_closey", GROUND_CLOSEY);
    node.getParamCached("ground_closez", GROUND_CLOSEZ);
    node.getParamCached("ground_fary", GROUND_FARY);
    node.getParamCached("ground_farz", GROUND_FARZ);
```

```
            node.getParamCached("ground_tolerancefine", GROUND_TOLERANCEFINE);
            node.getParamCached("ground_tolerancerough", GROUND_TOLERANCEROUGH);
            node.getParamCached("ground_normalsmoothing", GROUND_NORMALSMOOTHING);
            node.getParamCached("ground_thresholdlower", GROUND_THRESHOLDLOWER);
            node.getParamCached("ground_thresholdhigher", GROUND_THRESHOLDHIGHER);
            node.getParamCached("ground_outlierradius", GROUND_OUTLIERRADIUS);
            node.getParamCached("ground_normalestimation", GROUND_NORMALESTIMATION);
            node.getParamCached("ground_outlierneighbors", GROUND_OUTLIERNEIGHBORS);
            node.getParamCached("ground_verbose", GROUND_VERBOSE);

            //model the plane of the ground iff the user changed its keypoints
            if(GROUND_CLOSEY!=last_GROUND_CLOSEY || GROUND_CLOSEZ!=last_GROUND_CLOSEZ || GROUND_FARY!=last_GROUND_FARY || GROUND_FARZ!=last_GROUND_FARZ)
            {
                GROUND_SLOPE=(GROUND_FARY-GROUND_CLOSEY)/(GROUND_FARZ-GROUND_CLOSEZ);
                GROUND_YINTERCEPT=(GROUND_CLOSEY+GROUND_FARY)/2-GROUND_SLOPE*(GROUND_CLOSEZ+GROUND_FARZ)/2;
                last_GROUND_CLOSEY=GROUND_CLOSEY;
                last_GROUND_FARY=GROUND_FARY;
                last_GROUND_CLOSEZ=GROUND_CLOSEZ;
                last_GROUND_FARZ=GROUND_FARZ;
            }

            //variable declarations/initializations
            pcl::PassThrough<pcl::PointXYZRGB> crop;
```

```
        pcl::OrganizedEdgeDetection<pcl::PointXYZRGB, pcl::Label> detect;
        pcl::RadiusOutlierRemoval<pcl::PointXYZRGB> remove;
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr points(new pcl::PointCloud<pcl::PointXYZRGB>);
        pcl::PointCloud<pcl::Label> edgePoints;
        std::vector<pcl::PointIndices> edges;
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr navigation(new pcl::PointCloud<pcl::PointXYZRGB>);
        int trueGroundPoints=0; //size of the ground itself, not including any obstacles
        double trueGroundXTotal=0; //total of all the ground's x-coordinates

        //crop to focus exclusively on the approximate range of ground points
        crop.setInputCloud(cloud);
        crop.setFilterFieldName("x");
        crop.setFilterLimits(-CROP_XRADIUS-GROUND_BUMPERLATERAL, CROP_XRADIUS+GROUND_BUMPERLATERAL);
        crop.setKeepOrganized(true);
        crop.filter(*points);

        crop.setInputCloud(points);
        crop.setFilterFieldName("y");
        crop.setFilterLimits(CROP_YMAX, 1);
        crop.setKeepOrganized(true);
        crop.filter(*points);

        crop.setInputCloud(points);
        crop.setFilterFieldName("z");
        crop.setFilterLimits(CROP_ZMIN, CROP_ZMAX+GROUND_BUMPERFRONTAL);
        crop.setKeepOrganized(true);
        crop.filter(*points);

        //ignore everything that is not the ground
```

```
for(pcl::PointCloud<pcl::PointXYZRGB>::iterator location=points->begin(); location<points->end(); location++)
{
    double distanceFromGroundPlane=fabs(location->y/*point's actual y-coordinate*/ - (GROUND_SLOPE*location->z+GROUND_YINTERCEPT)/*ground's expected y-coordinate*/);

    if(distanceFromGroundPlane>GROUND_TOLERANCEROUGH) //this point isn't anywhere near the ground
    { //these aren't the points we're looking for
        location->x=std::numeric_limits<float>::quiet_NaN();
        location->y=std::numeric_limits<float>::quiet_NaN();
        location->z=std::numeric_limits<float>::quiet_NaN();
    }
    else if(distanceFromGroundPlane<=GROUND_TOLERANCEFINE && fabs(location->x)<CROP_XRADIUS-GROUND_BUMPERLATERAL && location->z>GROUND_CLOSEZ+GROUND_BUMPERFRONTAL && location->z<CROP_ZMAX-GROUND_BUMPERFRONTAL) //actually part of the ground and in the subregion where we do not tolerate intruding plane edges
    {
        trueGroundPoints++;
        trueGroundXTotal+=location->x;
    }
    //else part of the ground border or a contacting object: just keep it
}

```

```
if(trueGroundPoints>0) //don't waste time if we're blind
{
    //detect edges
    detect.setInputCloud(points);
    detect.setEdgeType(detect.EDGELABEL_HIGH_CURVATURE+detect.EDGELABEL_NAN_BOUNDARY);
    if(GROUND_NORMALESTIMATION>=0) detect.setHighCurvatureNormalEstimationMethod((pcl::IntegralImageNormalEstimation<pcl::PointXYZRGB, pcl::Normal>::NormalEstimationMethod)GROUND_NORMALESTIMATION);
    if(GROUND_NORMALSMOOTHING>=0) detect.setHighCurvatureNormalSmoothingSize((float)GROUND_NORMALSMOOTHING);
    if(GROUND_THRESHOLDLOWER>=0) detect.setHighCurvatureEdgeThresholdLower((float)GROUND_THRESHOLDLOWER);
    if(GROUND_THRESHOLDHIGHER>=0) detect.setHighCurvatureEdgeThresholdHigher((float)GROUND_THRESHOLDHIGHER);
    detect.compute(edgePoints, edges);

    if(GROUND_VERBOSE)
        ROS_INFO("GROUND_EDGES :: Saw raw %4lu curves and %4lu borders", edges[3].indices.size(), edges[0].indices.size());

    //assemble the detected points
    navigation->header=points->header;
    for(std::vector<pcl::PointIndices>::iterator edge=edges.begin(); edge<edges.end(); edge++)
        for(std::vector<int>::iterator pointIndex=edge->indices.begin(); pointIndex<edge->indices.end(); pointIndex++)
```

```
                    if(fabs((*points)[*pointIndex].x)<CROP_XRADIUS-GROUND_BUMPERLATERAL && (*points)[*pointIndex].z>GROUND_CLOSEZ+GROUND_BUMPERFRONTAL && (*points)[*pointIndex].z<CROP_ZMAX-GROUND_BUMPERFRONTAL) //point is far enough from the edge
                        navigation->push_back((*points)[*pointIndex]);

            //eliminate outliers
            if(GROUND_OUTLIERRADIUS>=0 && navigation->size()>0)
            {
                remove.setInputCloud(navigation);
                remove.setRadiusSearch((float)GROUND_OUTLIERRADIUS);
                if(GROUND_OUTLIERNEIGHBORS>=0) remove.setMinNeighborsInRadius(GROUND_OUTLIERNEIGHBORS);
                remove.filter(*navigation);
            }
        }
        else if(GROUND_VERBOSE) ROS_INFO("GROUND EDGES :: Lost sight of the ground!");

        //plan our next move
        if(navigation->size()>0) //curve or plane boundary in our way
        {
            float centroidX=0;

            //where are our obstructions centered?
            for(pcl::PointCloud<pcl::PointXYZRGB>::iterator point=navigation->begin(); point<navigation->end(); point++)
                centroidX+=point->x;
            centroidX/=navigation->size();
```

```
                if(GROUND_VERBOSE) ROS_INFO("GROUND_EDGES :: Seeing %3lu offending points centered at %.3f i", navigation->size(), centroidX);

                //choose a course of action
                if(centroidX<0) //offenders mostly to our left
                    directionsSecondary=RIGHT;
                else //centroidX>=0
                    directionsSecondary=LEFT;
                groundLastForcedTurn=directionsSecondary; //continue the same turn even if we lose sight of the ground in the next frame
            }
            else if(trueGroundPoints==0) //where'd the ground go?
            {
                if(GROUND_VERBOSE) ROS_INFO("GROUND_EDGES :: Ground has vanished; calling for emergency evasive maneuvers!");
                directionsSecondary=groundLastForcedTurn;
            }
            else //we're all clear
            {
                directionsSecondary=FORWARD;
                groundLastForcedTurn=trueGroundXTotal/trueGroundPoints>0 ? RIGHT : LEFT; //in case we lose sight of the ground in the next frame, we'll turn toward the direction where more of it is visible
            }

            ground.publish(*points);
            occlusions.publish(*navigation);
        }

        /**
        Integrates the decisions of the two obstacle detection methods and sends an appropriate drive
```

```
	command only if the <tt>drive_move</tt> parameter is set
	@param time the <tt>TimerEvent</tt> that triggered our schedule
	*/
	void pilot(const ros::TimerEvent& time)
	{
		//declare "constants," plus Vim macros to generate them from "populate 'constants'"
		#if 0
			:inoremap <cr> <esc>
Jldf,d/\a
f)C,
			$r;j
		#endif
		double DRIVE_LINEARSPEED, DRIVE_ANGULARSPEED;
		bool DRIVE_MOVE, DRIVE_VERBOSE;

		//populate "constants," plus a Vim macro to generate them from "clean up parameters"
		#if 0
			:inoremap <cr> <esc>
>>>>^f.l6sget
eaCached
f"l"yyt"f)i,
"ypvT,l~
j
		#endif
		node.getParamCached("drive_linearspeed", DRIVE_LINEARSPEED);
		node.getParamCached("drive_angularspeed", DRIVE_ANGULARSPEED);
		node.getParamCached("drive_move", DRIVE_MOVE);
		node.getParamCached("drive_verbose", DRIVE_VERBOSE);

		//variable declarations
		DriveAction newMotion;
		geometry_msgs::Twist decision;

```

```
//decide what to do, given the advice we've received
if(directionsPrimary!=directionsSecondary) //algorithms are at odds
{
    if(DRIVE_VERBOSE)
        ROS_INFO("PILOT :: One recommendation says %5s and the other counters %5s", directionRepresentation(directionsPrimary), directionRepresentation(directionsSecondary));

    if(directionsPrimary==FORWARD) newMotion=directionsSecondary;
    else if(directionsSecondary==FORWARD) newMotion=directionsPrimary;
    else newMotion=directionsSecondary; //it thought about this harder
}
else //we're agreed!
    newMotion=directionsPrimary;

//don't reverse the direction of a turn
if(newMotion!=FORWARD && currentMOTION!=FORWARD && newMotion!=currentMOTION)
{
    if(DRIVE_VERBOSE) ROS_INFO("PILOT :: Overrode recommended oscillation");

    newMotion=currentMOTION; //keep rotating in the same direction we were
}

//make our move
switch(newMotion)
{
    case LEFT:
        if(DRIVE_VERBOSE) ROS_INFO("PILOT :: Turning %5s", "LEFT");
```

```
                        decision.angular.z=DRIVE_ANGULARSPEED;
                        break;
                    case RIGHT:
                        if(DRIVE_VERBOSE) ROS_INFO("PILOT :: Turning %5s", "RIGHT");
                        decision.angular.z=-DRIVE_ANGULARSPEED;
                        break;
                    default:
                        decision.linear.x=DRIVE_LINEARSPEED;
                }
                if(DRIVE_MOVE) velocity.publish(decision);

                //tell the obstacle detectors what we've done
                currentMOTION=newMotion;
            }
};

int main(int argc, char** argv)
{
    ros::init(argc, argv, "xbot_surface"); //string here is the node name
    ros::NodeHandle node("surface"); //string here is the namespace for parameters

    //initial parameter values
    node.setParam("crop_xradius", 0.2); //should be slightly greater than robot's radius
    node.setParam("crop_ymin", -0.07); //should be slightly above robot's height
    node.setParam("crop_ymax", 0.35); //should be slightly above the ground's highest point
    node.setParam("crop_zmin", 0.0); //greater than zero excludes points close to robot
    node.setParam("crop_zmax", 1.5); //farthest to search for obstacles: lower for tighter maneuvering, higher for greater safety
    node.setParam("height_downsampling", 0.04); //less is more: should be low enough to eliminate noise from the region of interest (negative for [really bad] default)
```

```
node.setParam("height_samples", 5); //number of samples to average: should be low enough to prevent false positives
node.setParam("height_verbose", false);
node.setParam("ground_bumperfrontal", 0.1); //extra uncropped space on the front and back edges of the plane whose edges, borders, and presence are disregarded; note that for the front edge only, this is used with ground_closez to tolerate the gap between the robot and plane
node.setParam("ground_bumperlateral", 0.02); //extra uncropped space on the left and right edges of the plane whose edges, borders, and presence are disregarded
node.setParam("ground_closey", 0.3525); //y-coordinate of the closest point on the ground
node.setParam("ground_closez", 0.8); //corresponding z-coordinate for bumper border and modeling the plane
node.setParam("ground_fary", 0.47); //y-coordinate of a far point on the ground
node.setParam("ground_farz", 2.5); //corresponding z-coordinate for modeling the plane
node.setParam("ground_tolerancefine", 0.03); //maximum y-coordinate deviation of points that are still considered part of the ground itself
node.setParam("ground_tolerancerough", 0.1); //maximum y-coordinate deviation of points that are evaluated at all
node.setParam("ground_normalsmoothing", -1.0); //smoothing size for normal estimation (negative for default)
node.setParam("ground_thresholdlower", 1.0); //for curvature-based edge detection: cutoff for consideration as possible edges (negative for default)
node.setParam("ground_thresholdhigher", 1.7); //for curvature-based edge detection: cutoff for definite classification as edges (negative for default)
node.setParam("ground_outlierradius", 0.05); //radius used for neighbor search to filter out outliers (
```

```
    negative to disable outlier removal)
    node.setParam("ground_normalestimation", -1); //normal estimation method: as defined in IntegralImageNormalEstimation (negative for default)
    node.setParam("ground_outlierneighbors", 6); //minimum neighbors to be spared by outlier persecution (negative for default)
    node.setParam("ground_verbose", false);
    node.setParam("drive_linearspeed", 0.5);
    node.setParam("drive_angularspeed", 0.3);
    node.setParam("drive_move", false);
    node.setParam("drive_verbose", true);

    TandemObstacleAvoidance workhorse(node); //block to do obstacle avoidance

    //clean up parameters, plus a Vim macro to generate them from "default parameter values"
    #if 0
        :inoremap <cr> <esc>
^f.l3sdelete
f,dt)f;C;
j
    #endif
    node.deleteParam("crop_xradius");
    node.deleteParam("crop_ymin");
    node.deleteParam("crop_ymax");
    node.deleteParam("crop_zmin");
    node.deleteParam("crop_zmax");
    node.deleteParam("height_downsampling");
    node.deleteParam("height_samples");
    node.deleteParam("height_verbose");
    node.deleteParam("ground_bumperfrontal");
    node.deleteParam("ground_bumperlateral");
    node.deleteParam("ground_closey");
    node.deleteParam("ground_closez");
    node.deleteParam("ground_fary");
    node.deleteParam("ground_farz");
    node.deleteParam("ground_tolerancefine");
    node.deleteParam("ground_tolerancerough");
```

```
    node.deleteParam("ground_normalsmoothing");
    node.deleteParam("ground_thresholdlower");
    node.deleteParam("ground_thresholdhigher");
    node.deleteParam("ground_outlierradius");
    node.deleteParam("ground_normalestimation");
    node.deleteParam("ground_outlierneighbors");
    node.deleteParam("ground_verbose");
    node.deleteParam("drive_linearspeed");
    node.deleteParam("drive_angularspeed");
    node.deleteParam("drive_move");
    node.deleteParam("drive_verbose");
}
```