Avoiding Obstacles: Implementation of an Improved Smooth Navigation Controller

Peter Carbonetto
CPSC 515 Term Project
UBC Department of Computer Science
December 17, 2001

Introduction

José is an autonomous robotic waiter that uses sensory input to help it guide through an obstacle-populated environment. José is equipped with a Triclops stereo vision module that has three identical wide-angle (90-degree field of view) cameras. José also has sonar sensors, but at this point they are not working properly so the robot relies entirely on its visual sense for navigation control. Stereo vision provides rich two-dimensional information about the world, which includes colour, texture and depth. Additionally, José can measure distances from obstacles using stereo triangulation. José uses the range information obtained from stereo triangulation to construct radial maps, which are then used to construct or update a top-down two-dimensional occupancy grid map of its environment. José uses this map to identity free and obstructed regions for path planning.

José's "purpose", as it may be, is to serve food to people located throughout a room. Thus, goal planning is a crucial component to José's control architecture. Given the location of a person and the occupancy grid map, he uses a path-planning algorithm to construct a route to the goal, consisting simply of a sequence of waypoints. The path planner module is decoupled from the navigation controller, which runs onboard the robot and is responsible for following planned path. The navigation controller considers each waypoint along the route as a "mini-goal", and once it reaches the point, it considers how to reach the next point.

This project focuses on implementation of the robot's path navigation module. The current navigation controller is very basic; in order to reach a point in the path, it orients itself towards a waypoint, moves forward in a straight line, and then comes to a full stop when it reaches the waypoint. Only once it reaches a waypoint on the path does it consider the subsequent waypoint. No optimizations have been implemented in the navigation module. The objective of this project is to improve the efficiency of the robot's path navigation without affecting its robustness.

In his CPSC 515 project, Pascal Poupart implemented a smooth navigation controller in order to make José motion appear more human-like and execution of the path more efficient. In this project, the robustness of the smooth trajectory path follower is improved by incorporating obstacle avoidance.

Improving the function of the navigation controller involves coupling it with several modules. It is important to understand the overall functioning of José, so an overview of the control architecture and a preliminary exploration into path planning and navigation control are included in this paper.

As part of the project, I designed a GUI for examining the behaviour of the robot's control modules and for producing concrete results from test runs.

A Brief Tour of José

Figure 1 shows the basic control architecture for José. The robot's motor functions are controlled by *RobotServer* module via communication with the *BSoft* software package. All the modules communicate and share information through the shared memory. For example, the *RadialServer* writes to radial maps region of the shared memory, and concurrently the *Mapper* module polls the shared memory for new radial map information.

José's communication architecture proved to be a significant obstacle in this project. One reason is because posting messages in the shared memory requires significant overhead. To get access to the shared memory, a module must first create an object with privileged shared memory access to a specified region. To write information to shared memory, the module has to lock and unlock the region in shared memory. To receive a posted message, the module

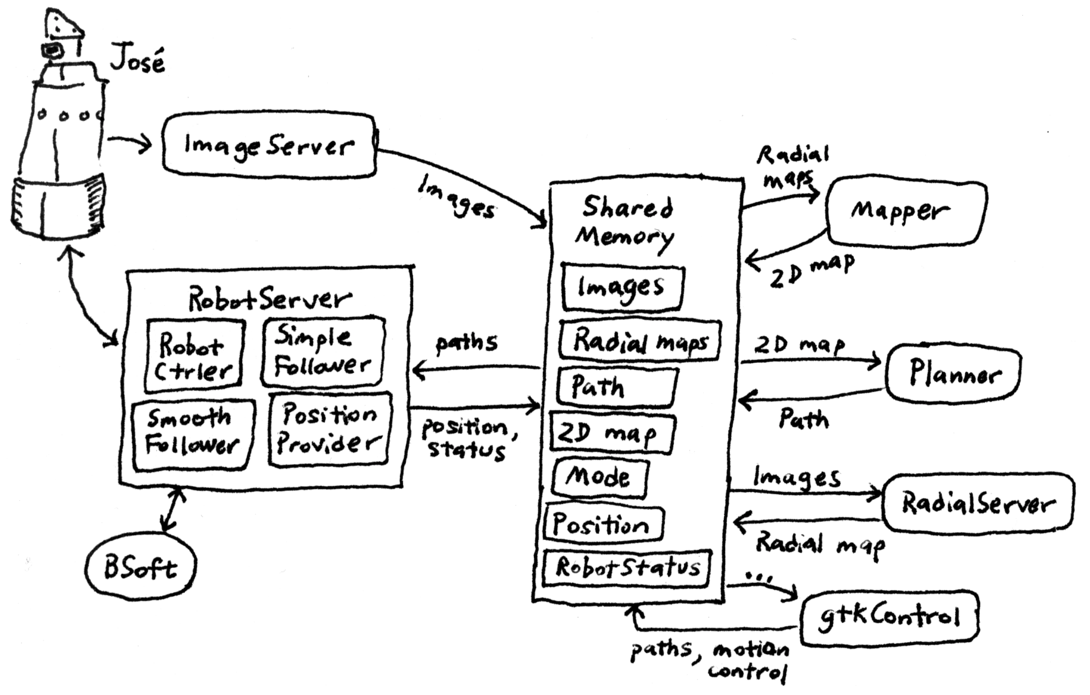


Figure 1. *Overview of José's control architecture.*

repeatedly queries the shared memory for possible updates.

Due to the high communication overhead, the modules tend to be highly decoupled and they maintain their own local state. Over the course of development of the software, each module evolved separately without regard to the functioning of the other modules. As a result, local state information is inconsistent from one module to another, which can cause communication problems.

Additionally, functioning of some of the modules is highly sensitive to the current state of the environment, and often crashes if the proper conditions are not met. Balancing the requirements of a few modules at one time proved to be a difficult task.

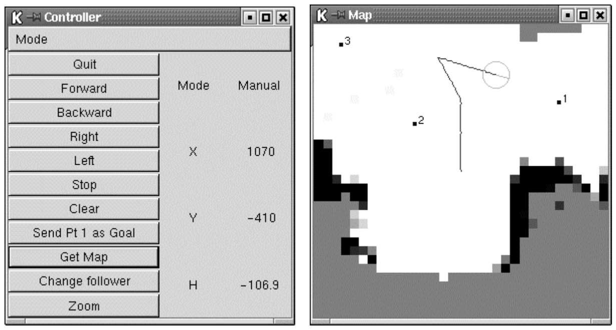
The problems I discovered throughout the progress of this project could be overcome with the establishment of simpler communication standards, an integrated approach to the development of module functionality, and a rigourous characterization of the behaviour of each module. Better state consistency and less redundant functionality would follow from these guidelines. Clearly, these are not easy objectives to meet, but adhering to them would greatly ease future development – in particular, adding mechanisms for learning.

gtkControl: The Graphical User Interface

José already has several GUIs to control and examine the behaviour of the robot, but they lack many useful features. The first objective of this project was to develop a flexible, multifunctional and expandable GUI to serve as a kind of development and experimentation laboratory. It was important that the software was reasonably flexible, to allow the incremental addition of features as needed and to leave room for future expansion since it was impossible to include all useful features for testing.

This step in the project took the longest to complete because I incorporated a large number of aspects of the robot into this interface. It took a while to reach a comfortable level of understanding where I could use the modules competently. The benefit is that I am now very familiar with the robot's control architecture. Don Murray implemented the core graphical interface and manual robot controls, and I added the more advanced features.

The *Forward*, *Backward*, *Right*, *Left* and *Stop* buttons are used for manual control of the robot. The map window updates the location of the robot in real time and draws the path history as a red line. The map window automatically zooms out when the robot moves off the edge of the drawing region. The



Mode Manual

X 1070

Y −410

H −106.9

Figure 2. *The* gtkControl *interface. The window on the left is the controller and the window on the right is the map display.*

Clear button erases the path history and readjusts the zoom level.

gtkControl allows the user to manually create paths. Clicking on the left mouse button adds a waypoint to the path. The waypoints are shown as numbered dots in *Figure 2*. Clicking on the middle mouse button erases the path. Clicking on the right mouse button sends the path to the RobotServer to be executed by the navigation controller. The user can select different navigation controller behaviours by clicking on the *Change Follower* button. When pressed, the *Send Pt 1 as Goal* button activates the planner module, which plans a path from the current location of the robot to the goal point (the first waypoint). The plan created by the planner module is then displayed as numbered points in the map window. The *Zoom* button zooms out the map by one level.

gtkControl runs locally on José's computer, so computational load of the GUI was an issue. Redrawing the map window is a computationally intensive task and could hamper native control of the robot. To reduce the time devoted to redrawing the map – without sacrificing the timeliness of the results – the program keeps track of the robot's position history in real-time but only redraws the map at an interval of 0.3 seconds. The window on the right in *Figure 2* shows the updated occupancy grid map; black regions represent sensed obstacles, white regions are unoccupied and grey regions are unknown to the robot. Displaying this map in real-time slows down the robot significantly, so it is only displayed once after the user clicks on the button *Get Map*.

Collision Avoidance

José receives 2D images in real-time from its Triclops trinocular stereo vision camera module. Using the stereo vision model developed by Don Murray and Jim Little, the *RadialServer* module constructs an occupancy grid, specified as a single row of distance values. Each column represents the distance from the robot's coordinate frame origin to the nearest obstacle. For a more thorough description of how the occupancy grid map is computed, see [3].

Stereo vision is an advantageous scheme for obstacle detection because it is robust, flexible and relatively simple. It also presents a significant challenge since the data is particularly unreliable. Obstacle mapping using the technique in [3] is very sensitive to noise. For example, stereo mismatch errors often cause "spikes" in the radial map. Errors can be partly reduced by projecting the 3D stereo information onto the plane. The radial map server uses several techniques, including image median filtering and spike removal, to improve the reliability of the sensor data, as outlined in [3]. The collision detection algorithm for this project only considers valid column values and not those removed by filtering.

Each pixel in the radial map has a region of uncertainty determined by

$$X = \frac{(x \pm 0.5)Z}{f}$$

where d is the disparity from stereo matching, $Z = d \pm 0.5$ is the depth, x is the image plane coordinate and f is the focal length. The uncertainty measurement therefore directly depends on the depth of the pixel. Happily, for the purposes of collision avoidance we are only interested in the closest obstacles so high column values in the radial map can be ignored. For the implementation in this paper, the only obstacles considered are those no more than 800 mm from the robot's origin.

Note that collision detection is limited by the 90-degree field of view of the stereo vision sensors and by the focal point location of the Triclops stereo head (approximately 0.5 m).

The collision avoidance algorithm for the navigation controllers further increases the reliability of the radial data by projecting the 2-dimensional radial data onto a line and, further more, over time. The collision detection algorithm looks at all column values in the radial map but only uses the smallest value for the purposes of detecting a collision.

Additionally, the algorithm considers several distance values over time since a necessary condition for obstacle detection is that it is getting closer to the robot, and not further away.

Here is a description of the general collision detection algorithm, which is used by the navigation controller at every time step. Initially, the *gClosest* = ∞ (smallest distance value) and $m = 0$. The constant k is the lower bound for the number of times the algorithm detects the obstacle moving closer before it is certain of this fact. The algorithm does not detect a collision unless the closest obstacle is no further than distance $d = 800$ *mm*.

1 Look at all valid columns in the radial map except those on the edge of the field of view, and set the variable *closest* to the smallest column value.

2 If *closest* is smaller than the previous smallest value, *gClosest*, then the robot is moving closer to an obstacle and set *gClosest* = *closest* and increase the value of m by 1. Otherwise, if *closest* and greater than the value of *gClosest*, set $m = 0$ and gClosest = ∞.

3 If $m > k$ and *closest* $< d$, report an imminent collision.

The avoidance strategy upon detection of an obstacle in the robot's path depends on the navigation controller and is discussed in the two subsequent sections.

A Simple Navigation Controller

Before tackling obstacle avoidance for the navigation controller, I implemented it first for the simple navigation controller designed by Don Murray.

This navigation controller employs no optimization techniques. To execute a path, it moves forward on a straight line from one waypoint to another. The robot directs its control towards the next waypoint by repeatedly comparing its current heading to the angle formed by the line between its current position and the next waypoint, and adjusts its heading if the difference in angle is greater than 5 degrees.

In actual fact, this path following algorithm does not need collision detection since the planner module guarantees that the path is obstacle-free on the straight lines between each pair of waypoints. In reality, this is definitely a bad assumption since the planner does not consider moving obstacles; the global occupancy grid map maintained by the map module does not get updated once it is initially created. In this implementation, the collision detection algorithm can spot an obstacle when the occupancy grid map has no knowledge of such an obstacle. Clearly, consistency of state would be improved if the occupancy grid map was continuously updated, but changing this is not in the scope of the project. To circumvent problems associated conflicts in world representation, the simple navigation controller assumes that the occupancy grid map is always accurate but the path handed to it may not be safe from collisions in the case where it was not generated by the planner module.

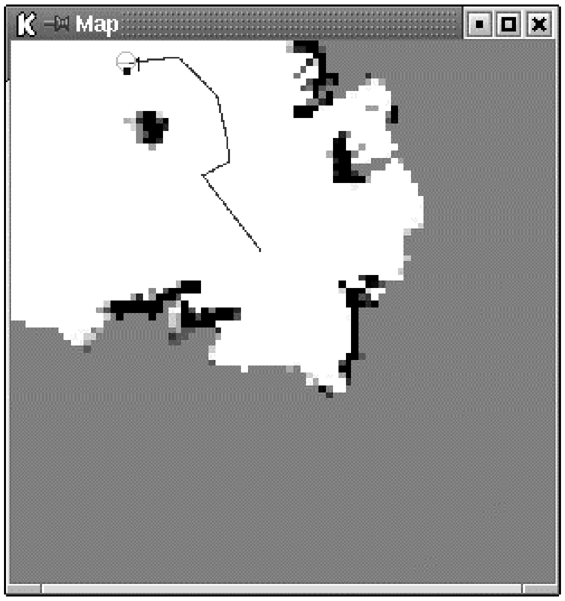


Figure 3. *A path followed using the simple navigation controller. The robot initially moves toward waypoint 1, and then uses the Planner module to redirect the path when the obstacle is detected.*

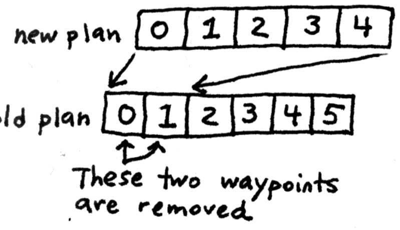
To test the algorithm, I created paths with obstacles between pairs of waypoints. *Figure 3* shows a straight-line path executed by the path follower. José went around the obstacle by requesting a new path when it reached the obstacle.

If the robot is moving forward (the translation speed > 0.1 mm/s), the simple navigation controller gets the most recent radial map information shared memory and runs the algorithm described above to test for obstacles. If the algorithm detects an obstacle in the current path, the controller does the following:

1 Activate the planner module and have it construct a new path where the goal point is the

next waypoint in the current path and the first waypoint is the current position of the robot.

2 The newly created plan is then inserted into the old plan.



3 Obstacle detection is deactivated until the next waypoint in the new path is reached. This measure is taken to ensure that the path follower will stop new paths for the same obstacle.

4 The follower resumes normal execution.

The Smooth Navigation Controller

Pascal Poupart, in his project for CPSC 515, used the Pure Pursuit technique to implement a smooth controller [4]. The main objective of my project was to improve the robustness of his implementation. Unfortunately, a great deal of time was spent getting the original smooth trajectory code to work (not to mention the fact that José experienced some very unpleasant high-speed collisions in the process!). This was partly due to the changes made to the control system since Poupart's code was written, rendering it incompatible with the current system. This task was further exacerbated by a complete paucity of comments. Once Poupart's implementation functioned adequately, I added code for collision avoidance.

What follows is a description of the implemented algorithm inspired by the Pure Pursuit technique with the added functionality for collision detection, some constructive criticism drawn from observation and suggestions for future improvement. *Figure 3* shows an example of a path executed by the new smooth navigation controller.

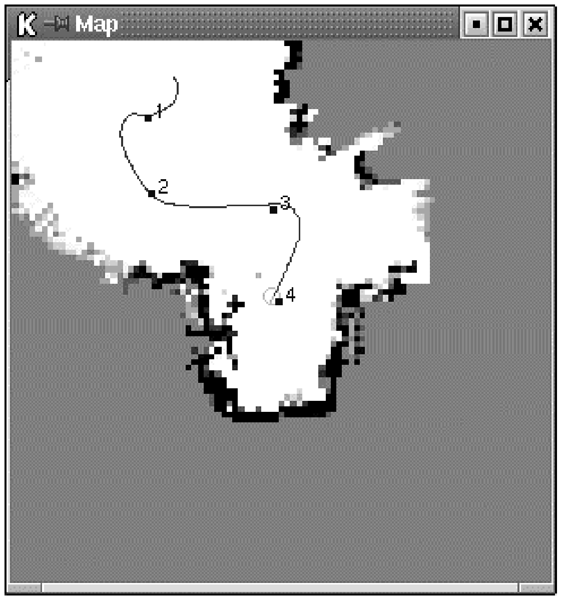


Figure 3. *Path followed using the smooth path navigator. The numbered dots are the waypoints of the path.*

The Algorithm. To incorporate collision avoidance, the navigation controller is either in one of two states: "normal mode" or "safe mode". The smooth controller enters "safe mode" when an obstacle is detected. Otherwise it remains in "normal mode". In safe mode the robot is very cautious and does not take chances. It does not deviate from the "beaten path", therefore reducing the risk of colliding into obstacles. The beaten path in this case is the set of straight lines connecting pairs of waypoints in the path. These straight lines are free from obstacles by virtue of the fact that the path-planning module generated them, so they are called "safe lines". In safe mode, the robot guides itself back to the straight-line path. It is not allowed to cut corners or make wide turns that steer it further away from the straight-line path. Once it has resumed navigation on the beaten path, the robot goes back to normal mode.

Here is a description of the path navigation algorithm:

1 Get the current position and heading of the robot in global coordinates, and store this information in a variable called *whereRobot*.

2 Find out if the robot has reached the current target waypoint. Calculate the Euclidean distance between the coordinates of the target waypoint and *whereRobot*. If the distance is less than a minimum bound *D > 0*, the robot has reached the target and so the target is updated to the next waypoint in the path. If the point robot reaches the last waypoint, tell the RobotServer the path is complete.

3 If the robot is moving forward (i.e. it has a translation speed great than 0.1 mm/s) and the robot is in normal mode, update the radial map and check for obstacles using the obstacle detection algorithm described above. If the algorithm detects an obstacle, turn on safe mode.

4 Determine the goal point of the robot for this iteration and store it in the variable *goalPoint*. The goal point is a waypoint on the path at least a Euclidean distance of *d* on a straight-line path from the current location of the robot. Another way to state it: the goal point is the same as the target waypoint on the path unless the it is less than distance *d* from the current location of the robot. For the purposes of the implementation, *d* is very small so it only considers the subsequent point when it was fairly close to the target waypoint.

5 Define the velocity space *V* of all pairs (v, ω) where *v* in the set of possible translation velocities and *w* is in the set of possible angular velocities. We can consider *V* as the space of possible screw motions. *V* is bound by the non-holonomic constraints of the robot. The robot has a maximum translational and a maximum rotational acceleration, v'_{max} and ω'_{max} respectively. Additionally, we specify a maximum translation velocity, v_{max}, and a maximum angular velocity, ω_{max}. This is primarily because the algorithm is not perfect and we want to avoid collisions. As mentioned in Poupart's paper, it has the added benefit of decreasing the size of the search space window. Moreover, setting an upper bound on the translational velocity increases the reliability of the algorithm (the reason for this is discussed in detail below). Negative velocities are not considered because we want the robot to move with the camera facing forward so it can detect potential obstacles. With all this in consideration, *v* and ω are limited to the following ranges:

v_t: $[max(0, v_{t-1} - v'_{max}), min(v_{max}, v_{t-1} + v'_{max})]$
ω_t: $[min(-\omega_{max}, \omega_{t-1} - \omega'_{max}), min(\omega_{max}, \omega_{t-1} + \omega'_{max})]$

The current implementation has the following parameter values:

v_{max}	= 0.2 m/s
ω_{max}	= 10 degrees/s
v'_{max}	= 0.2 m/s^2
ω'_{max}	= 10 degrees/s^2

To make searching the space *V* practical, it is approximated with a discretization. The finer the discretization, the more accurate the execution of the path will be; the coarser the discretization, the faster the algorithm will run. For this implementation, the search space is set to a 5 x 5 grid in the 2-dimensional space *V*.

5 Get the cross product

$$(whereRobot - start) \times (end - start)$$

where *start* is the previous waypoint in the path, *end* is the next waypoint in the path, and *whereRobot* is the current location of the robot (as defined in step 1). Since these are 2D points, a third component is added to compute the cross product. The z-component of the dot product is stored in the variable *c*. This cross product is used in safe mode to prune screw motions that move the robot further from the safe line.

6 Get the cross product

$$(end - start) \times [cos(h), sin(h)]^T$$

where *h* is the current heading of the robot. Store the z-component of the cross product in c'.

7 When *c* and c' both have the same sign or when $c' = 0$, it means that the robot is no longer moving away from the safe line and safe mode is turned off.

8 Search the discretized space *V* and select the optimal screw motion. For each point in the velocity space V, do the following:

a Given (v, ω), find the position and heading of the robot at some small timestep Δt in the future.

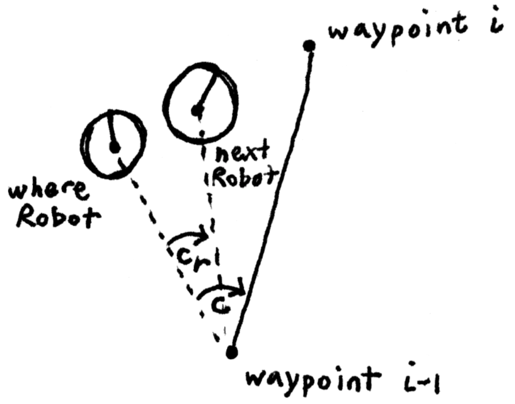
b Find the Euclidean distance between *goalPoint* and the future location of the robot. Store this value in Δd.

c Find the difference in angle between the heading at the future location and the direction the robot should be heading (the angle formed by the line from the future location to the goal point). Normalize the value and store it in $\Delta\theta$.

d If the robot is in safe mode, find out if the screw motion (v, ω) moves the robot closer or further away from the safe line. Compute the cross product

$$(nextRobot - start) \times (whereRobot - start)$$

where *nextRobot* is the computed future location of the robot computed in step a. Store the cross product in c_r. If *c* and c_r have different signs, it implies the robot moving away from the safe line and not closer to it, and return ∞ for the value of the objective function. Otherwise, compute the objective function in step e. For further explanation, see the diagram just below.



e Find the objective value of the *(v,ω)* pair by computing *F:*

$$F(v,w) = DF \times \Delta d + AF \times \Delta\theta$$

where *DF = ½* is the distance factor and *AF = 1* is the angle factor. These values are not necessarily optimal, but they worked adequately in the test cases.

9 Set the new translational and angular velocity of the robot to the objectively optimal *(v,ω)* pair.

In simple test cases this algorithm worked well. The paths were smooth and José avoided obstacles in all of the situations he faced. See *Figure 4* for an example. However, there are still several problems with the algorithm.

Discussion. The algorithm could be improved if the look-ahead distance *d* for finding the goal point were increased. Optimally, *d* can be set to infinity – this greatly improves the efficiency of the path navigation without incurring a large additional computational cost. The problem is that the robot is not guaranteed to pass by every waypoint on the path. As an example, if *d* is set to infinity and the last waypoint is the same as the first, the robot will not move because it is already at the end of the path, but at the same time the controller will not report having completed the path because it has not moved past waypoint 1. We could get around this problem by selecting only the screw motions that pass within a distance of *D > 0* of the subsequent waypoint in the path.

The Pure Pursuit technique depends on a large number of parameters set somewhat arbitrarily by the programmer – there is no obvious way to find near-optimal parameters for the algorithm. This may be a good opportunity to employ a learning algorithm to train the robot on the optimal set of parameters. Similarly, in an artificial fish simulation,

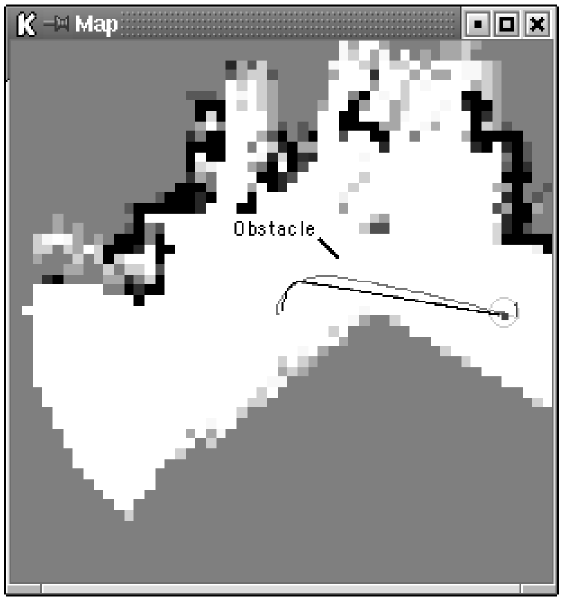


Figure 4. *This image shows two executions of a path with the smooth controller. In the first case (shown in grey) there is no obstacle. In the second case (shown in black) there is an obstacle preventing the robot from following a completely smooth path. Note that the obstacle does not show up in the map display because the occupancy grid map did not get updated with the new obstacle information, and was only added for this report.*

a learning algorithm was used to optimize the efficiency of motion patterns of autonomous agents [5]. Perhaps a simple training set could be used to explore the space of parameters through trial and error, where the optimizing function is the time taken to complete the specified path.

Another fundamental limitation to the smooth controller algorithm is that the goalPoint is chosen in space but prediction of the future location of the robot occurs over time. Under bounded conditions and if the parameters are chosen correctly, this is not a big problem. However, in the general case this could render the algorithm ineffective. If the speed of the robot at a point in time is relatively high, the objective function will grant preference to slower velocities (resulting in a potentially less optimal solution) and will essentially ignore the heading term. It is difficult to find a suitable look-ahead value *d* given a range of possible translation speeds. One solution is to select a large value for *d*, so the distance error is high for all translational velocities, but this will cause the objective function to be highly sensitive to modification of the parameters *DF* and *AF.* Also with a large *d* the algorithm runs into the path-completing problem discussed above. Under the current algorithm there appears to be a narrow region of acceptable values for *d*.

A detailed discussion on selecting optimal screw motions for path controllers can be found in [7]. The translational and angular velocities can be decoupled with a few assumptions about motion constraints. The paper proposes the use of fuzzy controllers to find the optimal screw motion. The drawback is that this technique requires expert knowledge of the control system.

Conclusion

This project was an excellent opportunity to become intimately acquainted with the design of José's control architecture and to discover its associated problems. I implemented a GUI that enables users to easily create paths and test their execution using different navigation control behaviours.

Despite the numerous complications, I managed to implement a robust smooth navigation controller with obstacle avoidance. It completed the paths more efficient than the simple navigation controller in the test suite, and maintained a reasonably smooth path even when obstacles were in the way. I outlined several reasons why the current implementation may not work in the general case and I offer some suggestions for future improvement of the algorithm.

References

[1] Rodney A. Brooks. "A Robust Layered Control System for a Mobile Robot." *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, March 1986, pp. 14-23.

[2] Pantelis Elinas, Jesse Hoey, Darrel Lahey, Jefferson D. Montgomery, Don Murray, Stephen Se and James J. Little. "Waiting with Jose, a vision-based mobile robot."

[3] Don Murray and Jim Little. "Using real-time stereo vision for mobile robot navigation." *Workshop on Perception for Mobile Agents* at CVPR 1998.

[4] Pascal Poupart. "A Smooth Navigation Controller." *CPSC 515 Project Report*, Department of Computer Science, University of British Columbia.

[5] Demetri Terzopoulos, Xiaoyuan Tu and Radek Grzeszczuk. "Artificial Fishes: Autonomous Locomotion, Perception, Behaviour and Learning in a Simulated Physical World." *Artificial Life*, Vol. 1, No. 4, 1994.

[6] Scott A. Wallace, John E. Laird and Karen J. Coulter. "Examining the Resource Requirements of Artificial Intelligence Architectures." CGF-BR-00, May 2000.

[7] Jeffrey S. Wit. "Vector Pursuit Path Tracking for Autonomous Ground Vehicles." *PhD Thesis*, University of Florida, 2000.