A Gradient Method for Realtime Robot Control

Kurt Konolige
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA
konolige@ai.sri.com

Abstract

Despite many decades of research into mobile robot control, reliable, high-speed motion in complicated, uncertain environments remains an unachieved goal. In this paper we present a solution to realtime motion control that can competently maneuver a robot at optimal speed even as it explores a new region or encounters new obstacles. The method uses a navigation function to generate a *gradient field* that represents the optimal (lowest-cost) path to the goal at every point in the workspace. Additionally, we present an integrated sensor fusion system that allows incremental construction of an unknown or uncertain environment. Under modest assumptions, the robot is guaranteed to get to the goal in an arbitrary static unexplored environment, as long as such a path exists. We present preliminary experiments to show that the gradient method is better than expert human controllers in both known and unknown environments.

1 Introduction

Motion control for a mobile robot can be divided into three general categories.

1. Motion planning. A complete path from robot to goal is generated, and the robot follows this path [5].
2. Local path evaluation. Several paths (e.g., a set of circular arcs of differing radii) in the immediate vicinity of the robot are evaluated to see if they are obstacle-free and lead towards a [9].
3. Reactive. The robot chooses a direction to move, based on current information about obstacles and the goal direction [1,7].

Most current controllers are combinations of these techniques, using motion planning for a global path and the other techniques to deal with uncertain or unknown objects. For example, in the bubble method of Khatib [3], motion planning generates an initial path based on prior knowledge of the environment, and then the path is adjusted as the robot senses obstacles that lie in the way of the path. Similarly, in local path methods, a motion planner generates waypoints, while motion towards the waypoint is determined by checking a small set of paths in the vicinity of the robot for collisions, based on local sensing.

What these hybrid methods sacrifice is optimality. Local path and reactive methods, for example, can get stuck in local minima, and require a recalculation of waypoints by the motion planner. Since the motion planner is computationally expensive, the robot must wait while the new waypoints are found. Even when the local methods work, they do not generate optimal paths for the robot to a waypoint. In fact, they typically do not even have a concept of optimality.

In this paper we present a new method for local navigation, called the *gradient method*, that continuously calculates an optimal path to a waypoint goal, overcoming the limitations of current local methods. The concept of optimality is derived by assigning costs to a path, based on its length and closeness to obstacles, as well as any other criteria that may be chosen. The gradient method computes a navigation function in the local space of the robot, such that the gradient of the navigation function represents the direction of the lowest-cost path at every point in the space. The method is efficient enough to be computed at a 10 Hz rate with modest computational resources.

By itself, the gradient method can generate the lowest-cost path in a static and completely known environment. But in many situations, there are complicated movable obstacles (furniture, doors) whose position may be uncertain or unknown. In the second part of this paper, we develop a sensor fusion algorithm for constructing a Local Perceptual Space [4] of the robot, in which new information is continuously added. In conjunction with the gradient method, it allows the robot to efficiently explore a local area and find a path to the goal.

Finally, we present some experiments that compare the gradient method to expert human controllers.

2 Navigation functions for minimum-cost paths

In this section we consider the generation of minimum-cost paths to a goalset in configuration space. We show how to build a *navigation function* that assigns a potential field value to every point in the space. Traveling along the gradient of the navigation potential yields the minimum cost path to the goalset from any point in the space.

For the rest of this paper, we consider the configuration space to be discretized into a set of points at which the navigation function and path costs will be sampled. Typically the discretization will be uniform, but it need not be. Values for the navigation function and costs at arbitrary non-sampled points can be computed by interpolation between sample points. The goalset is a set of sample points, although again we could relax this assumption without changing the basic formulation.

2.1 Path costs

We would like to find a path with minimum cost to some point in the goalset. Let us represent a path by an ordered set of points in the sample space,

$$P = \{p_1, p_2, \ldots\}, \tag{1}$$

where the set of points are contiguous (either diagonally or rectilinearly), and no point repeats. The last point on the path must be a goalset point, and no other point can be. We abbreviate a path that starts at point *k* by P_k.

In general, the cost of a path is an arbitrary function of the (discretized) path, $F(P)$. But such cost functions are difficult to deal with, and we make the assumption that the path cost is separable into the sum of an *intrinsic cost* of being at a point, along with an *adjacency cost* of moving from one point to the next:

$$F(P) = \sum_i I(p_i) + \sum_i A(p_i, p_{i+1}) \tag{2}$$

Both *I* and *A* can be arbitrary functions. For typical situations, *I* will represent the cost of traversing through the given point, and will be set according to the domain characteristics, e.g., a high cost will be assigned to being near an obstacle. Other possibilities are to have higher costs for unknown regions, slippery regions, etc.

The path length can be taken into account by assigning *A* to be proportional Euclidean distance the robot travels between the two points; then the sum of *A* gives a cost proportional to the path length.

2.2 Navigation function

A *navigation function N* is the assignment of a potential field value to every element of the configuration space, such that the goalset is always "downhill" no matter where you are in the space [5]. Navigation functions, unlike potential field methods in general, have the characteristic that it is impossible to get stuck in a local minimum, and no search is required to determine a direction to go to arrive at the goalset.

The key idea behind the gradient method is to *assign the navigation function at a point to be the cost of the minimal cost path that starts at that point*. Mathematically,

$$N_k = \min_{P_k} F(P_k), \tag{3}$$

where as before, the path P_k starts at point *k*.

If the intrinsic costs are zero, then the navigation function just represents the distance to the nearest goalset point. Traveling in the direction of the gradient of *N* yields the fastest reduction of path costs, i.e., the minimum distance to a goalset point. In the general case, traveling along the gradient is a minimum cost path to the goalset.

2.3 The LPN algorithm

Computing the values of Equation (3) at every point is difficult, since the size of the configuration space can be large, and finding the minimal cost path in a naïve manner involves iteration over all paths from the point. In this paper, we do not attempt to resolve the problem of the configuration space size, and the gradient method is only appropriate for a small number of dimensions. In our experiments, we assume the robot operates on the floor plane, is circular, and can turn in its own radius, so that a simple XY space is appropriate. The computation we describe here can be done for an arbitrary space, but is not practical for dimensions larger than 3 or 4.

In the literature, simple navigation functions for small-dimension spaces have been computed by a *wavefront algorithm* [1,9]. The goalset points are assigned a value of 0. At each iteration, the points with value *n* are expanded to their nearest (rectilinear) neighbors, giving them a value of *n*+1 if they are not already assigned, and are not obstacles. The process repeats until all points have been assigned. It takes time of the order of the number of points in the space, which is why it can only be used in small spaces. The wavefront algorithm computes the minimal Manhattan distance free-space path to the goalset.

The wavefront algorithm never backtracks, because at each point it advances the navigation function only to those points that have a value one higher than any other assigned point. However, the naïve application of the wavefront algorithm to the navigation function of Equation (3) won't work, because the value of the function can change arbitrarily from one point to the next.

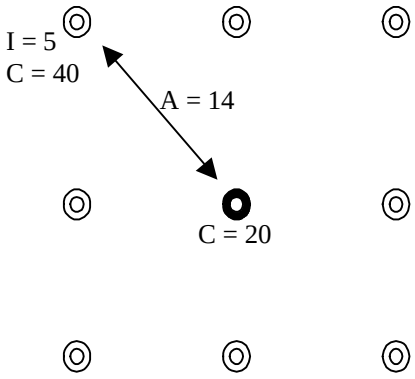

Figure 1. Updating the cost of points in the neighborhood of an active point. See text for explanation.

Instead, we use a linear programming algorithm that is a generalization of the wavefront algorithm, which we call **LPN**. The cost of **LPN** is again the order of the number of points in the space, and it reduces to the wavefront algorithm under the appropriate conditions on the navigation function.

Initially we assign all goalset points a value 0, and every other point an infinite cost. The goalset points are put in an *active list* of points. At each iteration of the algorithm, we operate on each point on the active list, removing it from the list and updating its neighbors. Consider an arbitrary point p that we have just assigned a value. This point is surrounded by 8 neighbors on a regular grid (Figure 1). For any of the neighbors q, we can compute the cost to extend the path from p to this point, using the cost metric of Equation (2). If the new cost at q is smaller than its own cost, we replace it with the new cost, and add q to the new list of active points that constitute the wavefront. For example, in Figure 1, the new cost for the diagonal neighbor is 20+14+5 = 39, which is less than its original cost, so it is added to the wavefront. The process repeats until the active list becomes empty.

It is possible to show that the **LPN** algorithm computes the least-cost path to every point in the workspace.

Proposition 1. At each point in the workspace, the gradient of the navigation function computed by LPN, if it exists, points in the direction of a least-cost path to the goalset.

Figure 2 shows the **LPN** algorithm at several different stages, starting from a single goal point. The rectangles represent the intrinsic cost of the sample points, with a large rectangle being infinite cost. These cost are computed from the obstacles in the workspace, as described in the next section.

The navigation function is implicitly described by the gradient direction at each point, here indicated by the short line segments. Where the navigation function is uniform, as it is in the vicinity of obstacles, the gradient is undefined. Note how the gradient points strongly away from the interior of the obstacles. Another interesting feature of the navigation function is the presence of *ridges*, points at which the gradient is equal in several directions. Ridges represent choice points in finding a minimal-cost path: any direction can be chosen, they lead to equal cost paths. In the last frame, there is a ridge that extends upward from the obstacle just above the goal point, where it is equally costly to go either left or right around the obstacle.

In the sequence of Figure 2, the wavefront is com-

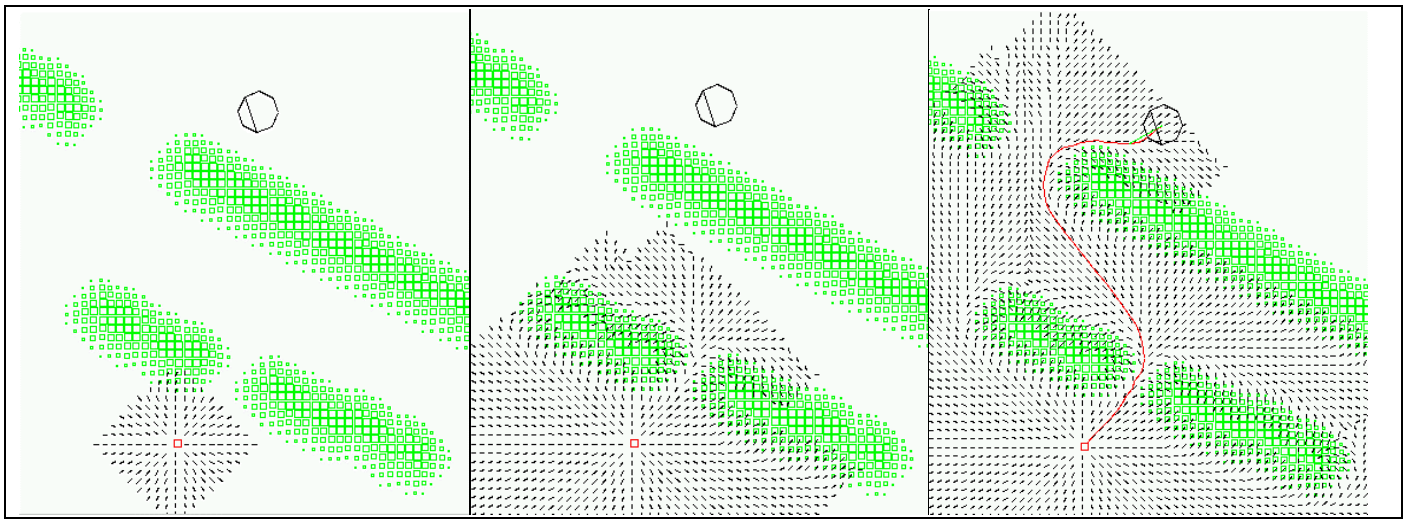

Figure 2. Three stages of the LPN algorithm, starting from a single goal point. The rectangles indicate the intrinsic cost of a point. The gradient direction at each point is shown as a short black line. The images shown are at 10, 30, and 70 iterations. The interpolated path from the robot to the goal is shown in the last image.

posed mostly of the set of points on the periphery of the gradient field. However, there can also be wavefront points on the interior. For example, in the second image, wavefronts from two sides of the obstacle have converged, and interior points will be added to the active list as long as they can update their neighbors with lower costs.

Finally, the minimal-cost path from the robot to the goal is computed and shown in the last frame. This path is easily found once the navigation function is known: starting from the robot, move a short distance along the gradient. At the new point, find the gradient by interpolation, move a short distance, and repeat. Note that although we show the gradient at all points in the space, we need only compute along the path from the robot to the goalset.

2.4 Obstacle costs

One of the problems with the original wavefront algorithm was that it produced paths that grazed obstacles. Other variations place the path as far as possible from obstacles. The **LPN** algorithm computes a generalized form of the navigation function that minimizes the cost of the path, and so the path behavior around obstacles can be controlled by the assignment of intrinsic costs, in the following way.

Suppose the obstacles in the workspace are given by a set of obstacle points. Let *d(p)* be the distance of the point *p* to the nearest obstacle point. Then assign:

$$I(p) = Q(d(p)), \qquad (4)$$

where *Q* is a decreasing function of its argument. Figure 3 shows two examples of a *Q* function. Both have a very high cost within 20 cm of any obstacle, which is the robot radius. In the function *Q1*, the cost falls rapidly with distance after this point, while in *Q2*, it falls much more slowly. Hence *Q1* will allow the robot to approach obstacles more closely than *Q2*. The intrinsic costs in Figure 2 were computed with the function *Q1*.

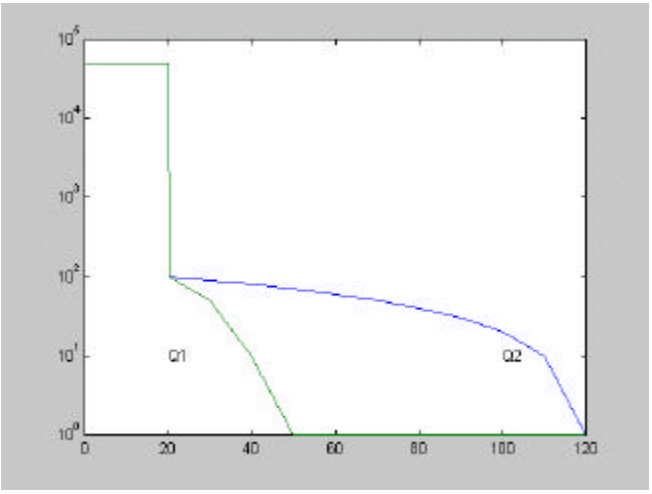

Figure 3. Two examples of an intrinsic cost function for obstacles. Cost is on the vertical axis, distance (in cm) on the horizontal axis.

Note that an intrinsic cost function like *Q2* will not prevent the robot from going through tight spots, but it will make it more likely that a longer path around the constriction will be less costly. For example, in Figure 2 the function *Q2* would put the path to the left of the obstacle above the goal point.

In practice the intrinsic costs can be assigned in the following way. Given a set of obstacle points, each point is labeled with the distance to the nearest obstacle point using the LPN algorithm, with the goalset being the obstacle points, all intrinsic costs at zero, and the adjacency cost set to Euclidean distance. Then, the *Q* function converts the distance at each point to an intrinsic cost.

2.5 LPN timing

The LPN algorithm runs in time proportional to the number of points in the workspace. Two LPN calculations must be done: one for the obstacle cost computation, and one for the navigation function. In most cases, the obstacle cost calculation can be stopped after a few iterations, and the navigation function dominates the computation time.

On a modest PC (266 MHz Pentium), a C implementation averages about 1 μs per workspace point. For a 10 m by 10 m workspace, with a grid size of 10 cm, it will take 10 ms to calculate the navigation function. Thus, the LPN algorithm is suitable for realtime control. Typically we run it at a 10 Hz rate, which is fast enough to run the robot efficiently at speeds up to 1 m/s.

3 Environment modeling

To be able to move efficiently, a robot must have a good model of its environment. The most interesting domains are those in which there is only partial information about the location of permanent objects (walls, doorways), there are movable static objects (chairs, tables, doors), and there are dynamic objects (people, other robots).

Range sensors on the robot give only partial and uncertain information about objects. For example, a robot may know that the room it has entered has a doorway on the other side, but may not be able to see it because it is blocked by other objects in the room (Figure 4). As the robot starts to search for the doorway, it may have a limited field of view, and so only a small portion of the room is under active observation at any given time. If

the robot forgets what it has just seen, then it may think that there is a doorway in an area it has just explored.

3.1 Local Perceptual Space

One solution to the problem of limited field-of-view is to fuse sensor information as the robot moves into a coherent geometric space. Such a Local Perceptual Space (LPS) gives a more comprehensive view of the immediate environment of the robot [4]. If the environment is static, and if the robot can track its position perfectly, then the LPS will build up a perfect composite picture of the environment, within the limits of sensor error. In the typical case, both these assumptions are violated, so the LPS is valid only over a short spatio-temporal interval.

For our experiments, we represent the LPS as a set of points in the workspace, with floating-point precision. Each point corresponds to a narrow-beam laser range-finder reading. Unlike evidence grid methods [6,8], there is no probabilistic representation of uncertainty. The algorithm we use to update the LPS is simple, but has proven to be practical, even in the presence of moving objects. The main steps to integrate a new range reading are the following.

1. [New points] Add all the new range readings into the LPS at a position indicated by the robot's dead-reckoned motion, and index them by their angle with respect to the robot.
2. [Visibility] For each old LPS point in the FOV of the range sensor, delete that point if it is closer than a new range reading at that angle.
3. [Staleness] If a point is further than a threshold distance from the robot, or has existed for more than a certain time, delete it.

The threshold distance determines the size of the LPS surrounding the robot. In general, a good value depends on the accuracy of the dead reckoning of the robot, and the uncertainty in sensor readings. The rate of decay of the old sensor readings depends on how many moving objects are detected, and how fast they move.

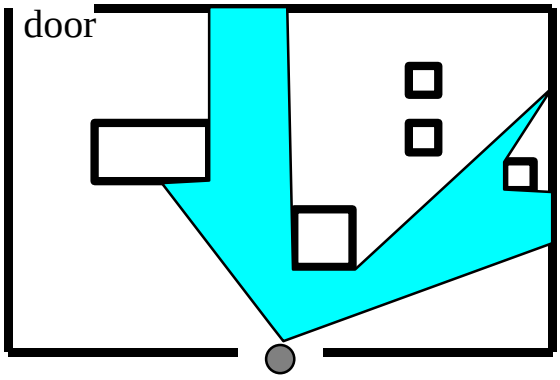

Figure 4. The robot's field of view and an occluding object block the observation of the exit door.

Figure 5 shows a typical LPS sequence as the robot tries to find a path to the goal point. The area is a cluttered room, with tables, chairs, boxes, and other obstacles, but no moving objects. Initially the robot has almost no information about where the exit door is located. Every 100 ms, a new set of range readings is added to the LPS, and the gradient path is completely recalculated. In fact, the qualitative nature of the path changes four times in the course of a small motion of the robot, as the gradient method discovers that the most promising areas are blocked. In addition to finding the correct topological exit from the room, the gradient method constantly fine-tunes the path of the robot on the basis of current information, e.g., as it discovers small occluded obstacles near the doorway. Finally, having arrived at the goal position in the hallway, the robot is now able to correctly construct a return path, even though the doorway and room interior are no longer in view.

3.2 Exploration

The gradient method and LPS construction were designed to circumvent the problem of getting stuck in local minima. We can show that, in static environments, these methods will always get the robot to the goal.

Proposition 2. If the sensors and dead reckoning of the robot are error-free, the environment is static, and there exists a path to the goal within the range of the LPS, then the robot will eventually arrive at the goal, if it follows the gradient path.

Obviously any real robot is not perfect, and it is possible to not find a path to the goal, for example if the sensors falsely indicate that an opening is closed. Still, robot performance continues to improve, with the sensors becoming more accurate, and the dead reckoning remarkably good when the turn error is corrected by solid-state rate gyros. While we have not yet done a comprehensive error analysis for the uncertain case, our experiments (Section 4) indicate that the gradient method does better than human operators in typical indoor environments.

3.3 Dynamic objects

Since the gradient path can be recomputed every 100 ms, it can correctly account for moving objects, generating paths to the goal that avoid these objects. In many simple situations it suffices to have the robot follow the gradient path as it changes to avoid the object. However, this simple idea does not account for the dynamics

of the robot and the object; for example, if the robot is on a collision course with an object, the gradient path will not reflect this until they are close. Although we have made some progress in this area, it is a research issue that cannot be adequately addressed here.

4 Experimental results

We have performed a small number of experiments to validate the gradient method and its associated LPS construction. These experiments do not constitute a statistically significant sample at this point, and we still need to address complicated human-robot interface issues. Nonetheless, we provide them as evidence that the gradient method is indeed successful at efficient high-speed control of a mobile robot in an uncertain environment.

The two main questions we wanted to address were:

1. How well did the gradient method perform against a human operator in an unknown envi-

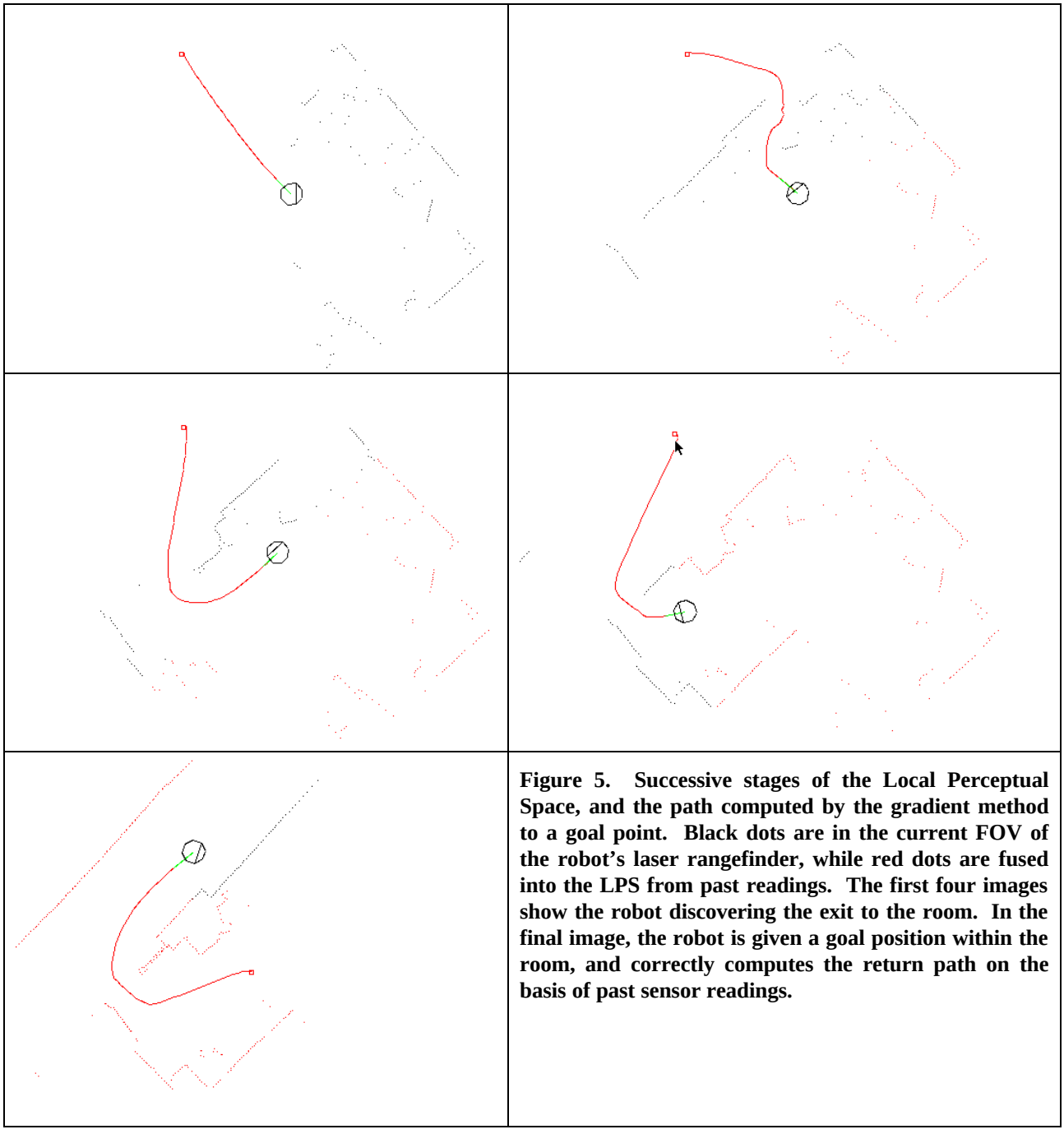

Figure 5. Successive stages of the Local Perceptual Space, and the path computed by the gradient method to a goal point. Black dots are in the current FOV of the robot's laser rangefinder, while red dots are fused into the LPS from past readings. The first four images show the robot discovering the exit to the room. In the final image, the robot is given a goal position within the room, and correctly computes the return path on the basis of past sensor readings.

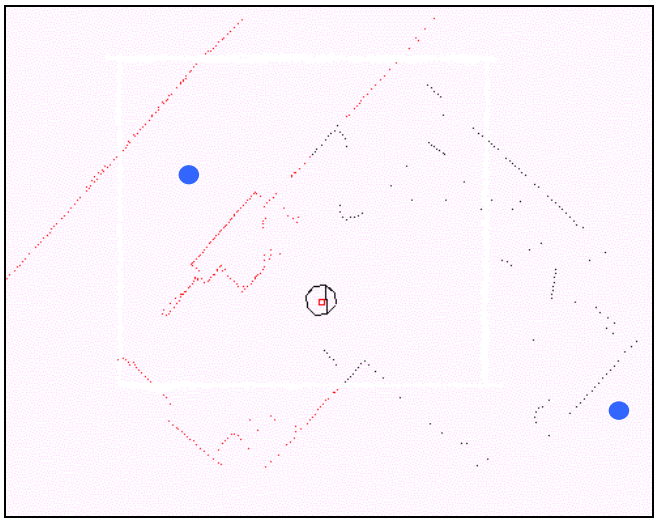

Figure 6. Test Environment. The image shows the robot's view of the large room in the middle of a run, with black dots representing the robot's current scan, and red dots the previous scan information. The task is to go from one of the blue goal positions to the other.

ronment?

2. The same, in a known environment.

We expect the results to be different, because both humans and robots can optimize their control when the environment is known beforehand. The first experiment stresses the ability to explore efficiently, while the second places more emphasis on fast control of the robot.

The basic task is to move the robot from an initial position to a goal position about 10 meters away, as fast as possible (see Figure 6). The optimal route is through a large room with offset doors. The room contains numerous confusing obstacles for the sensors, including chairs, tables and other furniture, and at times moving people. Both humans and robot were presented with the same information, namely the LPS displayed from a plan view.

In the first experiment, the robot was started at one of the goal positions, and was turned to face away from the room, so no portion of the environment between the goal positions was visible. Both goals were always present, so the operator and robot could see where the robot had to go. It is very difficult to get unbiased and comparable data for this experiment, because the robot must be put into a new environment each time.

In the second experiment, the robot was again positioned in a similar way, but the sensor scans were left from a previous run, so it was easy to see where the robot had to go. We recorded a number of runs for both operators and the robot in this scenario.

The human operator controlled the robot by pressing "joystick keys" that changed the robot's velocity, and turned the robot. We used two operators: one had a moderate amount of skill at controlling the robot from previous work, and the other was highly experienced. In all experiments, the robot's angular and translational accelerations were fixed, and the maximum velocity was set at 1 m/s.

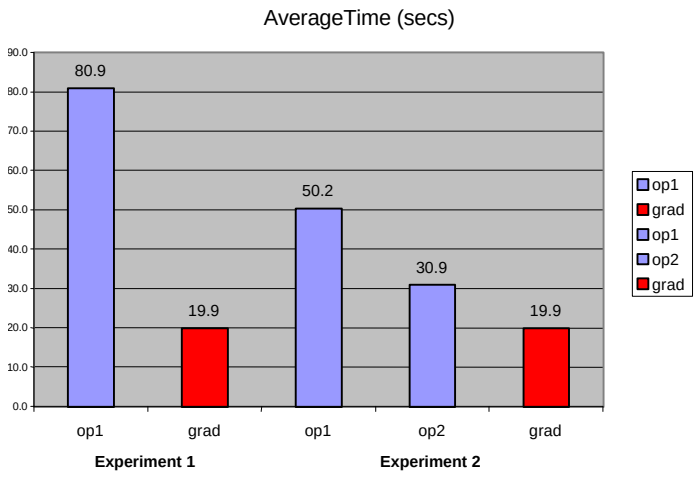

Table 1. Average time-to-goal for both experiments.

4.1 Results

For the first experiment, it was easy to run the robot multiple times to get statistics, but impossible to erase the operator's memory. The results below include only one run by the moderately-skilled operator in this case.

Time is the basic performance category (Table 1). In Experiment 1, the operator had some difficulty figuring out which direction the robot should go, and also crashed at one point trying to maneuver through a doorway. The time is 4 times worse than the automatic gradient algorithm. In the second experiment, the two operators did significantly better, but were still outclassed by the robot's algorithm. In all experiments the results were consistent, with variances less than 2 seconds.

Not crashing the robot is an important goal. In the first experiment the operator ran into a doorway while trying to get through. In the second experiment, he averaged one crash every three runs. The robot was completely safe.

4.2 Analysis

Under all conditions, and in all categories, the gradient algorithm did significantly better than the human operators, even when they had a high degree of skill controlling the robot, and had trained up on the environment. To analyze why this is the case, we look at a typical situation in the room (Figure 7). Here the robot

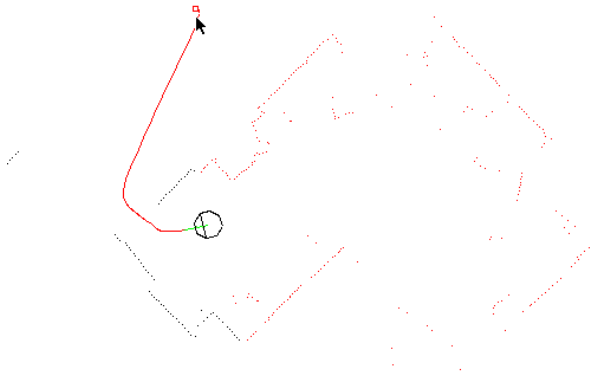

Figure 7. The robot's path in Experiment 1, going through a doorway.

has planned a path around the walls of the room to the goal position. Because it has a complete path, it can look ahead and calculate exactly how fast it can go at each point on the path, based on its dynamic characteristics. It can also precisely control how it moves along the path, updating its speed every 100 ms. Thus, the robot can move very quickly until just before it gets near the doorway, then slow down to maneuver through.

In contrast, even though a human operator has the same perceptual information, he or she has difficulty estimating the exact velocity the robot can have during difficult maneuvering, such as going around a corner. Thus, a human operator is at a disadvantage in controlling the robot's trajectory to such a precise degree. The choice is to slow down and play it safe, or move quickly and risk a crash. In fact, we recorded a large number of keystrokes at these critical points, showing that the operators were trying very hard to vary the robot's trajectory in an optimal way; they just were not able to respond as quickly or accurately as the gradient algorithm.

Another area in which the gradient algorithm excels is in discovering the best path on a continuing basis, as new information is added from the sensors. In the first experiment, the operator had trouble deciding how to exit the room, and spent some time exploring the upper right portion of the room (see Figure 7) before deciding that it was not a good way to go. The gradient algorithm also initially selected this route in Experiment 1; but by the time the robot had reached the center of the room, it had seen enough of the environment to reject this path in favor of one through the correct doorway, and proceeded to follow it with little lost time. In fact, there was no difference between the robot's times in the first and second experiment.

5 Conclusion

We have presented a new method for local navigation, the *gradient method*, that computes optimal paths to waypoint goals. The method is efficient enough to be used for realtime control, and is now the main control strategy for all the robots in our lab. It overcomes the limitations of other local control paradigms because it computes a complete set of optimal paths to every point in the workspace, avoiding local minima and other control problems. The method has been shown to work well on real robots equipped with laser range finders, operating in complicated indoor environments.

Our future work will concentrate on extending the gradient method to dynamic situations.

References

[1] Arkin, R. C. Integrating behavioral, perceptual and world knowledge in reactive navigation. *Robotics and Autonomous Systems* 6, pp. 105-122, 1990.

[2] Barraquand, J. and J. C. Latombe. Robot motion planning: a distributed representation approach. *IJRR*, 1990.

[3] Khatib, O. Real-time obstacle avoidance for manipulators and mobile robots, *IJRR* 5 (1) pp. 90-98, 1996.

[4] Konolige, K. and K. Myers. The SAPHIRA architecture: a design for autonomy. In *Artificial Intelligence Based Mobile Robots: Case Studies of Successful Robot Systems*, D. Kortenkamp, R. P. Bonasso, and R. Murphy, eds., MIT Press, 1998.

[5] Latombe, J. C. *Ro bot Motion Planning.* Kluwer Academic Publishers, Boston, 1991.

[6] Moravec, H. and A. Elfes. High resolution maps from wide angle sonar. In *Proc. ICRA*, pages 116–121, 1985.

[7] Saffiotti, A., K. Konolige, and E. Ruspini. *A Multivalued Logic Approach to Integrating Planning and Control.* Artificial Intelligence **76** (1-2) 1995.

[8] Schultz, A. and W. Adams. Continuous Localization using Evidence Grids. *Proc. ICRA*, Leuven, Belgium,1998.

[9] S. Thrun, A. B¨ucken, W. Burgard, D. Fox, T. Fr¨ohlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schimdt. Map learning and high-speed navigation in RHINO. In D. Kortenkamp, R. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, Cambridge, MA, 1998.