Subgoal-based local navigation and obstacle avoidance using a grid-distance field

Anthony S. Maida*, Suresh Golconda, Pablo Mejia, Arun Lakhotia and Charles Cavanaugh

Centre for Advanced Computer Studies,
University of Louisiana at Lafayette,
Lafayette, Louisiana 70504, USA
Fax: (337)482-5791
E-mail: maida@cacs.louisiana.edu
E-mail: suresh_golconda@yahoo.com
E-mail: pablo@cctechnol.com
E-mail: arun@louisiana.edu
E-mail: cdc@cacs.louisiana.edu
*Corresponding author

Abstract: The local path-planning and obstacle-avoidance module used in the CajunBot, six-wheeled, all-terrain, autonomous land rover is described. The module is designed for rapid subgoal extraction in service of a global navigation system that follows GPS-supplied waypoints. The core algorithm is built around a grid-based, linear-activation field (a type of artificial potential field). The local path planner has three novel features: the artificial potential field delivers local waypoints, or navigation subgoals, rather than a gradient; the planner aggressively avoids obstacles; and, the algorithm makes use of a repulsive expansion region to compensate for imperfect manoeuvrability.

Keywords: autonomous navigation; GPS navigation; local path-planning; obstacle avoidance.

Reference to this paper should be made as follows: Maida, A.S., Golconda, S., Mejia, P., Lakhotia, A. and Cavanaugh, C. (2006) 'Subgoal-based local-navigation and obstacle-avoidance using a grid-distance field', *Int. J. Vehicle Autonomous Systems*, Vol. 4, Nos. 2-4, pp.122–142.

Biographical notes: Anthony S. Maida received BA in mathematics, MS in Computer Science and PhD in Psychology at the SUNY at Buffalo. He received postdoctoral training at Brown University and the University of California at Berkeley. He is an Associate Professor at the Centre for Advanced Computer Studies and Institute of Cognitive Science, University of Louisiana at Lafayette. His research is in biologically inspired artificial intelligence and brain simulation.

Suresh Golconda obtained his BS degree from Osmania University and the MS degree from the University of Louisiana at Lafayette, both in Computer Science. He has worked on CajunBot simulation software, navigation control and steering control.

Pablo Mejia, a Member of Team CajunBot, is the main Software Architect for the CajunBot 2004 system and designed and implemented the shared-memory-based blackboard used in CajunBot. He is also a Senior Systems Engineer at C&C Technologies, Inc.

Arun Lakhotia obtained MSc degree from the Birla Institute of Technology and Science. He received PhD from Case Western Reserve University in Computer Science. He is the Project Lead for Team CajunBot and an Associate Professor at the Centre for Advanced Computer Studies, University of Louisiana at Lafayette. He received the 2004 Louisiana Governor's Technology Award. His research interests are in software engineering and computer security.

Charles Cavanaugh received BS and MS degrees from the University of Texas at Tyler and PhD from the University of Texas at Arlington, all in Computer Science. He is an Assistant Professor at the Centre for Advanced Computer Studies, University of Louisiana at Lafayette. He served as Technical Lead of Team CajunBot for the DARPA Grand Challenge, 2004. His research interests are in distributed embedded systems.

1 Introduction

A local-navigation and obstacle-avoidance module, named CajunBot Local Navigation (CBLN), was developed for use in an autonomous, six-wheeled, all-terrain, land-rover known as CajunBot (Cavanaugh, 2004) (see also, http://cajunbot.com). CajunBot is a research testbed equipped with a Global-Positioning System (GPS) that performs long-distance navigation by following GPS waypoints over varied terrain. The CBLN module described herein augments the global GPS-waypoint-following system (G-nav) by enabling the system to plan paths around dynamically detected obstacles. Obstacle detection is based on sensor readings provided by scanning laser range finders. The CBLN module continuously delivers a set of local, nearby waypoints (subgoals) compatible with those used in CajunBot's global, long-distance navigation subsystem. A major design goal was to obtain waypoint-based pluggability with the global GPS-waypoint-following system (G-nav). The CBLN system has the following properties.

- The system supports real-time performance in a vehicle that travels at a rate of about 4 m sec^{-1}.
- The system supports rapid subgoal, or waypoint, extraction.
- The design allows for clear separation of real-time path-planning and real-time steering control (path execution).
- Waypoint-based pluggability allows substituting alternative local path-planning modules for testing purposes.

In the remaining sections of this paper, we describe: our approach; the core obstacle-avoidance path planner; communication between the local obstacle-avoidance path planner and G-nav and a case history in developing and testing the system. Further information about CajunBot hardware and electronics is found in Cavanaugh (2004).

2 Related work and current approach

Feng, Singh and Krogh (1990) and Krogh and Feng (1989) built an early subgoal-based, dynamic obstacle-avoidance system that used range-finder-based obstacle acquisition. Subgoal extraction used the assumption that obstacles were convex polygons (Subgoals were chosen to avoid obstacle vertices.). The system clearly differentiated path-planning from path-execution and steering control (a good design for subgoal-based navigation). They also used the method of expanding obstacles by the bot radius (allows treating the bot as a point, while accounting for its physical radius in obstacle avoidance). This obstacle-expansion technique was also used in Stentz and Hebert (1995) and is also used in our CBLN module.

Barraquand, Langlois and Latombe (1992) described sophisticated Artificial Potential Field (APF), path-planning methods for robot manipulators with many degrees of freedom. These algorithms were not designed to be incremental (they pre-computed many data structures), operate in real-time or cope with situations where the robot deviated from the planned path (as in a heavy, moving vehicle with imperfect steering). However, our CBLN module uses one element of their APF algorithms, namely their W-potential, which computes city-block distance from a goal location on a cell grid.

An APF algorithm uses a charged-particle metaphor, where the robot is (say) positively charged and a desired goal location (or region) is negatively charged. Obstacles are given the same charge as the vehicle. The simulated force vectors can control the steering of the actual robot in the actual world so that the robot approaches the goal while avoiding obstacles. An attractive feature of APF algorithms are the smoothly varying paths that are generated. In the present context, such algorithms have the following drawback pertaining to the form of their output. Global navigation is often subgoal- or waypoint based. An APF algorithm must express its output as waypoints.

Lagoudakis (1998) and Lagoudakis and Maida (1999) was the starting point for the present research. They used a neural-network-based APF (Glasius, Kamoda and Gielen, 1995; Glasius, 1997) in an incremental path-planner. They also used the obstacle expansion technique introduced in Feng, Singh and Krogh (1990).

Stentz and Hebert (1995) describe an integrated local and global navigation system, which is capable of terrain classification. The local navigation system does not do elaborate path-planning, but performs immediate obstacle avoidance. The local navigation provides steering recommendations, rather than waypoints.

The present path-planner uses a Grid-Distance Field (GDF) based on the W-potential in Barraquand, Langlois and Latombe (1992). However, the present work represents a novel synthesis of APF algorithms with subgoal navigation methods. In particular, the methods for extracting subgoals (local waypoints) from APFs are novel. Another novel feature of the path-planner is the feature of *aggressive avoidance*. In general, when a vehicle detects an obstacle, it *could* generate a path that approaches the obstacle and then turns at the last minute. The path is admissible in the sense that it avoids the obstacle. The algorithm described herein generates a path to turn *immediately*. The distinction can be seen by comparing locations (5, 15) in Figure 1b and c. This property offers the advantage of giving a better margin of safety to the steering control system. The GDF has the following features.

1 Field strength changes linearly with (city-block) path-length from the goal.

2 The linear gradient is generated by a simple and fast algorithm (worst-case run-time is proportional to the square of the number of grid cells).

3 Obstacles are avoided by virtue of not falling on the trajectory path created by the gradient. However, because of manoeuvrability uncertainties in the robot, the planned route may not exactly match the realised route. Regions of repulsion from obstacles are used only when the bot deviates from the planned path.

4 The resulting paths do not have smoothly varying trajectories. However, since the global navigation system requires waypoints (rather than a detailed trajectory), this is not a drawback. Further because of the aforementioned manoeuvrability uncertainties, computing a fine-grained exact path is not useful.

Figure 1 Flow maps showing arena from above. Subfigures show direction of travel (before subgoal extraction) towards goal location (30, 20) from every point in arena. Rectangular obstacle is positioned at $x = 23$ and is extended in the y-dimension: (a) Generated by a neural network. Trajectories are smooth but algorithm scales poorly to larger arena sizes; (b) Generated by applying arctan (x, y) to a GDF; (c) Flow-map used the same GDF as in b, but from a rule that uses only gradient sign and not magnitude. Trajectories are straight lines joined by 45° turns. This is good for extracting waypoints. Also, c shows aggressive turning to avoid obstacles. Compare locations (5, 15) in b and c. Finally, flow fields in c do not collide with obstacle but assume point object moves along obstacle

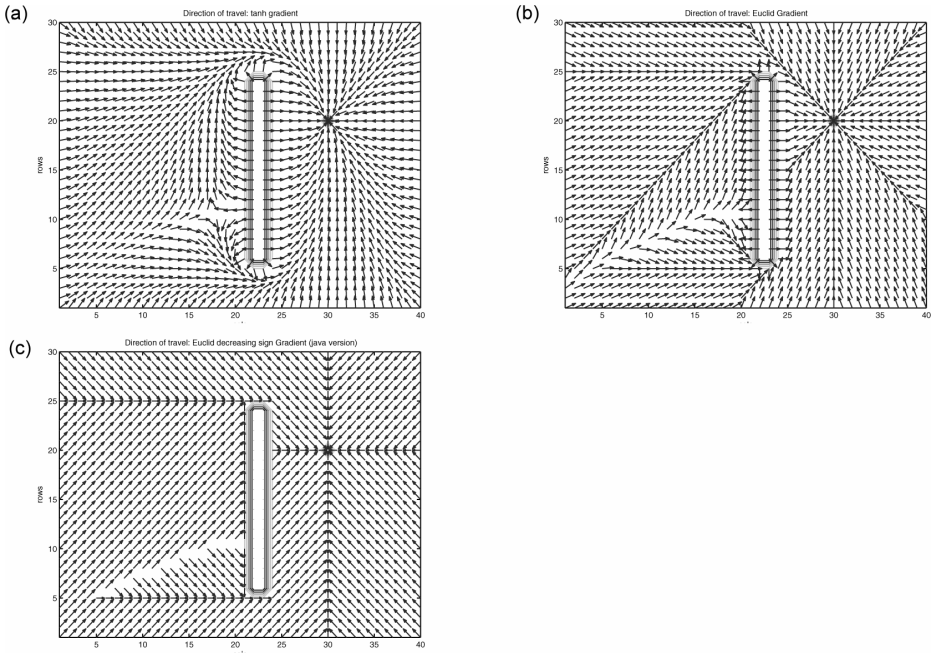


Figure 1 summarises the factors influencing our initial approach to generate and manage an APF. The top-left flow-field was generated by the Neural Network (NN) algorithm in Lagoudakis and Maida (1999). Trajectories are smooth. However, the flow-field robustness is sensitive to parameter choices and, because potential field values are

constrained by a bounded, sublinear activation function, it scales poorly to arenas whose grid size is larger than 50 × 50 (The grids in the figure have dimensions 30 × 40). The top-right figure shows a GDF. Trajectories have sharp turns but performance is stable with respect to parameter adjustments and the algorithm scales well with arena size. Field flows were obtained by applying the negative arctan (x, y) function to the x and y components of the GDF. Further experimentation revealed (bottom left) that if gradient magnitudes were ignored and only the sign (–1, 0 and +1) of the (x, y) gradient components was used, it was easier to extract waypoints from the flow fields (because the paths consisted of line segments joined at 45° angles, see Section 3.5).

The following describes the details and extensions of the CBLN module, its communication with a global GPS-waypoint-following subsystem and its performance in simulation and in the CajunBot vehicle.

3 The local path-generation algorithm

As CajunBot travels, it visits a sequence of static, pre-determined (read from a route description file at system start-up), global GPS waypoints. The specific static waypoint that the bot is approaching is the Active WayPoint (AWP). When this waypoint is reached, the next waypoint in the sequence becomes the new AWP. The CBLN module is responsible for *local path generation* and enables the bot to navigate around obstacles (and undesirable terrain) en route to the AWP (see Figure 2). It dynamically (continuous updates at a rate above 2 Hz) generates a series of subgoal waypoints that lead to the AWP. The module generates paths in a rectangular arena, whose dimensions are about 20–40 m on a side. The arena moves and rotates in unison with the bot because the arena uses a local bot-centred coordinate system (Lagoudakis and Maida, 1999). CBLN communicates with the rest of the system via a real-time blackboard (Corkill, 1991) implemented in shared memory (see Figure 3).

Figure 2 Local arena is constructed and used by the L-nav subsystem to circumvent dynamically detected obstacles. The GDF algorithm operates within a local arena or grid. As the bot moves and rotates, the arena moves and rotates with it as a consequence of using a local, vehicle-centred coordinate system. When the arena is constructed, a subgoal is chosen to be en route to distant AWP. The arena is updated on each L-nav cycle. An arena update can consist of building a new arena or adding terrain descriptors to the current arena

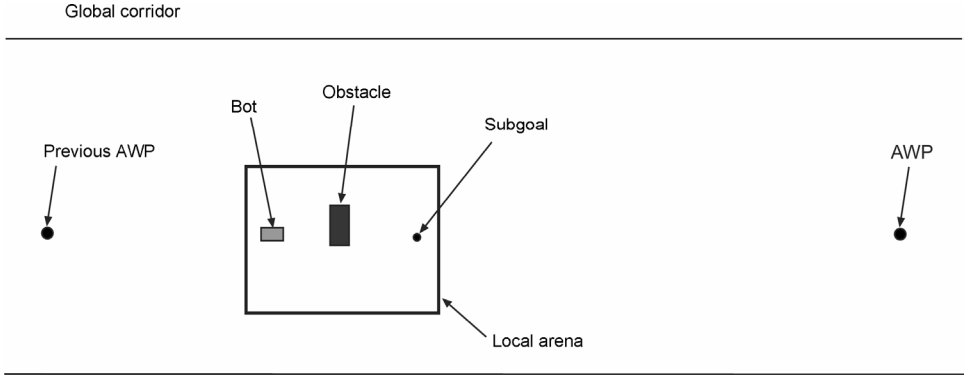
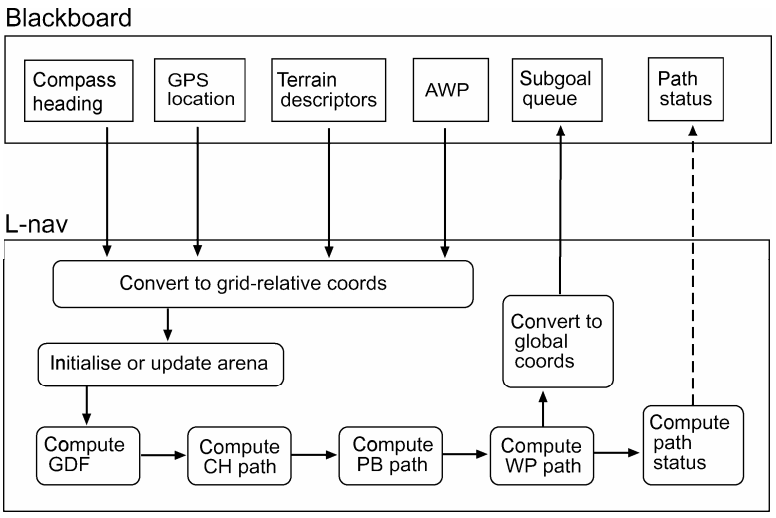


Figure 3 Information flow between the real-time blackboard and the CBLN implementation of L-nav



In the following, we refer to the global GPS-waypoint-following system as G-nav and the local navigation system as L-nav. CBLN is an implementation of L-nav. The L-nav process operates in a read-compute-write cycle, as it reads and writes to the blackboard (Figure 3). Each cycle is an *L-nav cycle*, whose duration is less than 0.5 sec. One L-nav cycle consists of reading path-relevant information, updating the local arena, computing a local-path and then writing the results back to the blackboard as a sequence of local (subgoal) waypoints. Section 4 describes exactly what L-nav reads and writes to the blackboard. Here, we say a few words about how the arena is initialised and then devote the remainder of the section to describe the path-generation computation. When the arena is initialised, a subgoal (or subgoal region) en route to the AWP is generated. The subgoal is chosen to be near enough to the bot's current location so that a local arena can be built to enclose the bot and subgoal, with the bot at one end and the subgoal at the opposite end. The path-generation computation has two components. First, the GDF is computed and then a sequence of local waypoints, or subgoals, is extracted from the GDF. This local sequence leads to the subgoal that was created when the arena was initialised. We now turn our discussion to computing the GDF and then later to local waypoint extraction.

3.1 Grid-distance field

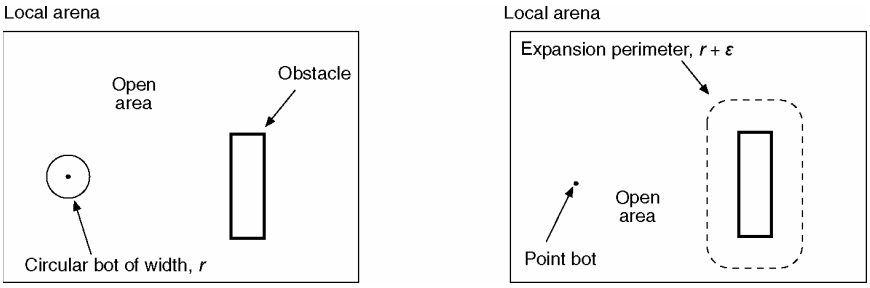
The local arena has dimensions x and y and is represented by a rectilinear grid of C columns and R rows. Grid spacing is about 30 cm. Grid columns represent the x dimension of the continuous arena space and rows represent the y dimension. The grid is used to construct a GDF that generates flow maps and allows a path to be extracted from the bot's current location to the local subgoal region within the arena.

To support path generation, each grid cell falls into one of four types: goal, obstacle, expansion and open (clear). The set of *goal cells* compose the (contiguous) subgoal region that the bot wants to reach. A path is constructed to reach a cell in this set.

Obstacle cells are so named because they represent the space containing known obstacles (as indicated by terrain descriptor input). Within the grid, the bot is modelled as a point object occupying exactly one cell. Since the actual bot has physical extent, *expansion cells* represent grid areas that the 'point bot' should avoid to prevent a collision with an obstacle. The size of the expansion region corresponds to the physical bot radius (assuming a circular bot). Remaining cells are considered *open*.

Figure 4 illustrates the purpose of expansion cells. Consider the case of a circular bot with physical radius, r, that can turn in place (Figure 4 (left)). A collision with obstacle, o, is avoided provided the centre of the bot is always a distance greater than r from o. Since it is easier to model the bot as a point, an equivalent formulation is shown on the right. Here, an expansion radius of size equal to $r + \epsilon$ is created around the obstacle perimeter (where $\epsilon > 0$ is a safety margin). If the bot followed the flow field in the open area of the arena perfectly (see Section 3.5), this would be the end of the story because flow fields in the open area always avoid the expansion region. Unfortunately in a physical world, the bot may enter the expansion region because of imperfect steering or other unanticipated physical event. Therefore, within this expansion region, linear flow fields are generated to direct the bot away from the obstacle. Complications for an elongated bot that does not turn in place are described in Sections 6.1 and 6 and illustrated in Figure 10.

Figure 4 Use of an obstacle expansion region to simplify collision calculations. r is taken to be the radius of a circle circumscribing the rectangular bot. It does not guarantee safety because the bot does not turn in place



A path within the arena from the bot's current location to the subgoal region is constructed from a GDF. Before describing path construction, we describe how the GDF is generated. The grid is an array of cells whose activations evolve over a series of discrete, within-grid time steps. Activation of a given cell changes according to its own type and according to the activations of its four neighbours (north, south, east and west). We first describe how grid activation changes over one time step in the update sweep. The process of propagating stable activations over a grid requires a series of grid-update sweeps. A *grid-update sweep* computes a single activation update to each grid cell and consumes one within-grid time step.

3.2 Grid-update sweep

Let i denote a cell at grid location (c, r). Let act(i, t) denote the 'activation' of i at within-grid time step, t (after t grid sweeps have occurred). The different cell types have different initial activations. cat(i) denotes the type of i and can have values: goal, obst,

open and expand. The symbols: init_goal_act, init_obst_act, init_open_act, init_expand_act denote initial activations for goal, obstacle, open and expansion cells, respectively. The initial activation of each cell (after 0 grid sweeps) is given below and in Table 1. The specific initial values depend on the length of the longest path within the grid. There are $R \times C$ cells. The maximum path length must be less than this, regardless of the obstacle configuration within the grid, because no cell occurs in a path more than once. If there are no obstacles in the grid; then the maximum path length must be less than $R + C$. In practice, the parameter max_pl is heuristically set to a value between these bounds.

Table 1 Initial values for grid square activations. Goal square activations are clamped to 0. Obstacle activations are clamped to twice the city-block maximum path length ($C + R <$ max_pl $< C \times R$). Other activations are initialised, but not clamped, to values midway between goal and obstacle activation

Category	*Initial value*	*Clamped?*
init_goal_act	0	Yes
init_obst_act	2 × max_pl	Yes
init_open_act	max_pl	No
init_expand_act	max_pl	No

$$\text{act}(i, 0) = \begin{cases} \text{init\_goal\_act} & \text{if cat}(i) = \text{goal} \\ \text{init\_obst\_act} & \text{if cat}(i) = \text{obst} \\ \text{init\_open\_act} & \text{if cat}(i) = \text{open} \\ \text{init\_expand\_act} & \text{if cat}(i) = \text{expand} \end{cases} \tag{1}$$

Let nb(i) be the set of City-Block (CB) neighbours of i. These are the cells at locations: ($c + 1$, r), ($c - 1$, r), (c, $r + 1$) and (c, $r - 1$). Let nb_act(i, t) denote the set of the activations of the neighbours of i. That is,

$$\text{nb\_act}(i, t) = \{\text{act}(j, t) \mid j \in \text{nb}(i)\} \tag{2}$$

Let max and min respectively denote the maximum and minimum of a set of scalars. For each grid sweep, each cell updates its activation according to the following rule.

$$\text{act}(i, t+1) = \begin{cases} \text{init\_goal\_act} & \text{if cat}(i) = \text{goal} \\ \text{init\_obst\_act} & \text{if cat}(i) = \text{obst} \\ \min(\text{nb\_act}(i, t)) + 1 & \text{if cat}(i) = \text{open} \\ \max(\text{nb\_act}(i, t)) - 1 & \text{if cat}(i) = \text{expand} \end{cases} \tag{3}$$

The interesting cases are the open cells and expansion cells. For an open cell, the cell reads the activations of its four neighbours and remembers the minimum value. It then computes its own activation by adding one to that minimum value. An expansion cell also reads the activations of its four neighbours but remembers the maximum value instead of the minimum. It subtracts one from this maximum. Section 3.4 extends Equation (3) to include cell cost.

The equation is used with the propagation algorithm given in Section 3.3 to propagate an activation field until it is stable. To a first approximation, direction of the bot's travel is determined by travelling down the stablised activation gradient. When the field stablises, the open squares code the city-block distance to the goal for a path that avoids

the obstacle expansion region. For expansion squares, when the value of the parameter max_pl is subtracted from the stablised activation of an expansion square, the quantity codes the city-block distance to the boundary between the open region and the expansion region. While in the expansion region, the activations provide a direct path to the open/expansion boundary. When in the open area, the activations provide a path to the goal.

3.3 Propagation of a stable grid-distance field

The GDF algorithm performs grid sweeps until the GDF is stable. The GDF is guaranteed to be stable if the number of grid-update sweeps is at least max_pl. The pseudo-code below uses Equation (3) to propagate a stable GDF.

do_times *max_pl*
  begin
  actBuff = **compute** all new activations according to Formula (3);
  store *actBuff* **into** *grid*;
  end;

In the above, the update sweep is synchronous: new activations are computed for all of the cells (and saved into a temporary grid called *actbuff*) before any values are changed. Each grid-update sweep updates the activations for each of the $C \times R$ cells once. Thus a stable GDF is obtained after max_pl $\times$ C $\times$ R cell-activation updates (within-grid time steps). This must occur in less than one real-time second. The following theorems describe the convergence properties of the activations that emerge as the GDF propagates.

The preliminary definitions formalise the notion of grid city-block distance. Given a cell, *i*, the *City-Block (CB) neighbours* of *i* are the four cells touching *i* to the north, south, east and west. A *path* is a sequence of cells without duplicates. A *CB path* from cell *i* to *j* is a sequence of cells where the first element of the sequence is *i*, the last is *j* and any two adjacent cells in the sequence are CB neighbours. The *CB distance* from cell *i* to *j* is the length of the shortest CB path from *i* to *j*. Path length is defined as the number of elements in the sequence minus one. The next theorems (stated without proof) describe the number of grid update sweeps required for the GDF to converge, where convergence means that subsequent update sweeps will not change activation values (activation reaches a fixed point). The path extraction algorithms in Section 3.5 assume that the GDF has become stable (converged).

Theorem 1. For a grid consisting of only goal and open cells, after $C + R$ update sweeps, the activation of each open cell is stable and codes its city-block distance from the closest goal cell.

Theorem 2. Let G be a grid consisting only of goal, open and obstacle cells. If the activations of the obstacle cells are clamped to 2 $\times$ max_pl, then after max_pl update sweeps, the activation of each open cell codes its city-block distance to the nearest goal cell.

Theorem 3. During GDF propagation, once an open cell reaches an activation that codes its CB path distance to the nearest goal cell, its activation reaches a fixed point. That is,

$$\text{act}(i;\, t+1) = \text{act}(i;\, t).$$

3.4 Incorporating a cost function

The update Equation (3) can be extended to include cell cost. Let cost(i) ≥ 1 denote the cost of cell i, which is some measure of how difficult it is to travel through the region of the arena represented by that cell. Default cost is one. The revised activation-update formula is given below.

$$\text{act}(i, t+1) = \begin{cases} \text{init\_goal\_act} & \text{if cat}(i) = \text{goal} \\ \text{init\_obst\_act} & \text{if cat}(i) = \text{obst} \\ \min(\text{nb\_act}(i, t)) + \text{cost}(i) & \text{if cat}(i) = \text{open} \\ \max(\text{nb\_act}(i, t)) - 1 & \text{if cat}(i) = \text{expand} \end{cases} \tag{4}$$

Let max cost be the largest cost value used in the grid. Maximum path length is now max_cost × C × R. Intuitively, giving a cell a cost greater than one can be interpreted as magnifying the size of the cell, thereby making it seem farther from the goal. There is no concept of cost for the expansion area because the bot cannot penetrate deeply into an expansion area without causing a collision (safety margin of ϵ). Therefore, when in the expansion area, the bot never needs to travel far to leave it. In practice, the most common scenario is that the bot sideswipes the expansion area and must veer away from the obstacle, while continuing on its path.

3.5 Path extraction and local waypoint generation

The stable GDF supports the extraction of flow maps, paths and subgoal waypoints. For a stable GDF, the activation value of an open cell, i, codes the city-block distance to the nearest subgoal cell from i's location. From this information, it is possible to extract a path from i to the subgoal, circumventing arena obstacles. Internal to the GDF, a path is represented as a non-duplicating cell sequence, where the last cell in the sequence is a subgoal cell.

In general, trajectories are extracted from gradients Lagoudakis and Maida (1999) by applying the arctan (x, y) function (or its negative, depending on whether the gradient is ascending or descending) to the x and y gradient components of each arena location. If smoothly varying trajectories are not required (e.g. as is the case for waypoint generation), one can ignore the magnitude of the gradient components (x and y) and use only their sign (–1, 0, +1). Hence in the GDF path-extraction algorithm, only the signs of the x and y gradient components are computed and assigned to each cell. From these values, local headings are computed for each cell. Rules for mapping gradient sign into local compass heading are given in Table 2. The headings are quantised into eight values known as the local, grid-relative compass headings. These quantised headings are used for path extraction (explained later). Drawing the local compass heading at each cell location generates the quantised flow map shown in Figure 1c. The definitions below help characterise the paths.

Table 2 Rules for extracting grid-relative compass headings from the sign of the *x* and *y* gradient components. Heading is local to a cell

Sign of x-gradient	*Sign of y-gradient*	*Local heading*	*Sign of x-gradient*	*Sign of y-gradient*	*Local heading*	*Sign of x-gradient*	*Sign of y-gradient*	*Local heading*
–1	–1	SW	–1	0	W	–1	+1	NW
0	1	S	0	0	N/A	0	+1	N
+1	1	SE	+1	0	E	+1	+1	NE

An *annotated grid* is one in which each cell has been assigned a quantised local compass heading. Let cell *i* have an associated local compass heading. The heading points to exactly one of the eight adjacent cells, say *j*. Cell *i* is said to *target j*.

Because local headings are quantised, the GDF supports extraction of paths composed of line segments joined at 45° and 90° angles. Such a path is shown in Figure 5. The points where these lines are joined are candidate local waypoints. Note that even though cell activation codes the city-block distance, the rules of Table 2 allow the extraction of a CH path, defined as follows. Let *P* be a path of *N* annotated cells. *P* is a *CH path* if the cell at position *i* in the sequence ($1 \leq i < N$) targets the cell at position $i + 1$.

Figure 5 The CH path is composed of straight lines joined together at 45° angles. Background shows field flow map. This grid does not use expansion cells. Figure 6 shows a grid and associated CH path that uses expansion cells

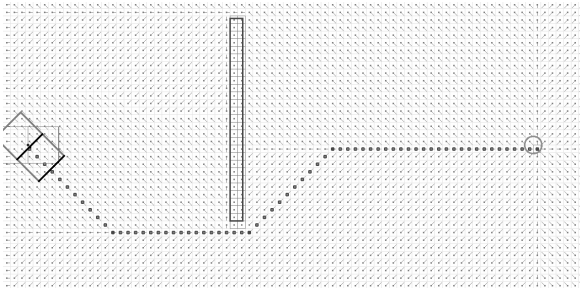
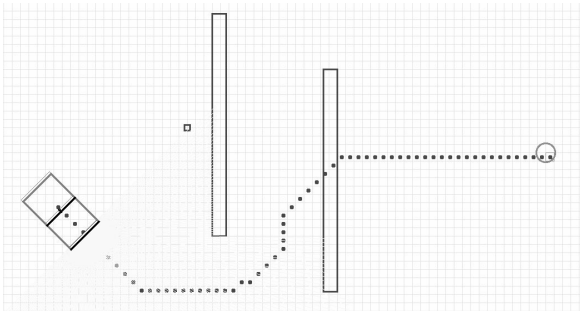


Figure 6 Incremental obstacle detection. Laser range finder (yellow) does not detect the full extent of the distant obstacle. Path is constructed on the basis of the information available. The arena must be continuously updated (2 Hz or greater) and a new path generated as the robot travels. Path bends in the middle of the path are too close together to serve as waypoints. They must be filtered



3.5.1 Procedure to construct a CH path from a stable GDF

Let G be an annotated grid and i be a cell. The objective is to construct a CH path from i to a goal cell. Initialise path P to the empty sequence. Put cell i at the end of P. While i is not a goal cell do the following. Set i to the target of i and then add the new value of i to the end of P.

For a CH path, all adjacent cells in the path are neighbours in the grid (each cell has eight neighbours). We do not want each adjacent cell in a CH path to be mapped into a separate waypoint because the waypoint-spacing would be too small (~30 cm apart). To provide better support for waypoint extraction, we use a new path data structure that makes explicit bends in the path (changes in direction). The next type of path – the PB path – saves only those sequence elements that represent changes of direction. Below, we formally define *path bend* and *PB path*.

Definition 1. Let P be a CH path. Each path cell has an associated local compass heading. The bends in the path correspond to positions in the sequence where the CH changes. Consider two adjacent cells at positions i and $i + 1$ in the path sequence. If i and $i + 1$ have different compass headings, then the cell at position i+1 represents a path bend.

Definition 2. A path-bend (PB) path is a subsequence of a CH path, P, that contains exactly the cells (with order preserved) in P that are path bends.

3.5.2 Procedure to extract path bends from a CH path

The path is a sequence of cells, where the sequence has N elements. Start with the cell at position $i = 2$ in the sequence. If the heading at position i differs from that at position $i - 1$, then position i represents a path bend. Add this to the end of the sequence. Increment i and repeat the process until cell i is a goal cell. Add the goal to the sequence.

In a world where the bot did not have physical steering limitations, the cells in a PB path would yield acceptable waypoints. Because of these physical limitations, one more type of path is needed to represent the final waypoint path.

Definition 3. A WP path is a subsequence of a PB path whose cells satisfy bot-specific, waypoint acceptability criteria.

To extract waypoints from a smoothly varying trajectory, one must identify path locations of high curvature. Using a linear GDF to create paths composed of line segments joined by discrete turns simplifies this problem greatly. The *path bends*, as defined earlier, are obvious and serve as candidate local waypoints. Waypoint filtering heuristics are used to ensure that the waypoints are reachable by the bot. The heuristics take into account the factors of waypoint separation, bot manoeuvrability and bot heading.

To summarise, three path-sequence types are used (see Table 3). These are CH, PB and WP, where WP $\subseteq$ PB $\subseteq$ CH. That is, each sequence is a subsequence of the previous. In particular, the CH path is filtered to obtain the PB path. This in turn is filtered to obtain the WP path. The WP path consists of cells. The locations of each of these cells must be converted from grid-relative coordinates into global coordinates and then published to the blackboard as subgoals.

Table 3 Summary of path types used to obtain waypoints

Path type	*Function*
CB	Cell activations code shortest CB distance to subgoal.
CH	Local headings extracted from CB activations using gradient sign.
PB	Filtered CH path retains cells where direction changes.
WP	Filtered PB path retains cells matching spacing and reachability criteria.

CB: city-block.

CH: compass heading.

PB: path bends.

WP: waypoints.

4 Communication between L-nav and the blackboard

L-nav communicates with the rest of the CajunBot system by reading and writing to the real-time blackboard implemented in shared memory (Figure 3). L-nav is one of the many processes that interact with the blackboard. Each type of data residing in the blackboard is owned by exactly one process that has write privileges for that data type. Other processes only have read privileges. L-nav owns the subgoal waypoints that it writes to the blackboard for use by G-nav. However, as seen in Figure 3, it reads compass heading data, GPS location data, various terrain descriptors (e.g. obstacle locations) from the digital terrain map and a reference to the AWP. Data on the blackboard is expressed in global coordinates. The L-nav read-and-write processes must translate between the blackboard global coordinates and the grid-relative arena coordinates. The list given below summarises the complete L-nav cycle (from Section 3) with its read-and-write components included.

- Read the path-relevant information (vehicle location, terrain descriptors) from the blackboard and convert it to grid-relative coordinates.
- Initialise or update the arena using the information from the previous step (plus cached terrain descriptor information from the previous four L-nav cycles).
- Propagate a stable GDF and extract subgoal waypoints as described in Section 3.
- Write the waypoints (expressed in global coordinates) to the blackboard.

The following presents more detail information on exchange between the blackboard and the arena. We describe arena construction heuristics, conversions between global and grid-relative coordinates, terrain descriptor input and local waypoint (subgoal) output.

4.1 Arena construction heuristics

As the bot travels, if there are no detected obstacles, an empty arena moves along with the bot; and L-nav publishes one subgoal waypoint to the blackboard, which is the arena subgoal en route to the AWP (defined at beginning of Section 3).

When the arena is first constructed, the arena length is chosen to be the distance of the bot to the subgoal plus three bot lengths. Initial distance between the bot and subgoal

is 1,500 cm. If this subgoal lies in the vicinity of a known obstacle then the arena is enlarged to accommodate a more distant subgoal (distance increment is 100 cm). This process repeats until a subgoal is generated that appears to be in open space. It is possible that the AWP will be closer than the arena subgoal (more discussion in Section 4.3). The path planner can also determine whether the bot is trapped or whether, given the obstacle configuration, the arena is too small to properly plan a path.

When an arena is built, a coordinate transformation is defined between the gobal and grid-relative coordinates. The transformation is based on the following information and is used in Steps 1 and 4 of the aforementioned L-nav cycle.

- The global position of the bot is known (from GPS) and the initial position of the bot in the arena is known (by stipulation). This establishes the translation component.
- The heading from the global bot location to the AWP is known. The arena subgoal is stipulated to be due east in grid-relative coordinates. This establishes the rotation component.
- The transformation does not have a scaling component.

4.2 Blackboard interaction

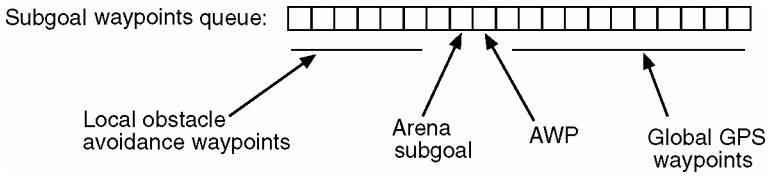
L-nav understands two kinds of terrain descriptors. These are no-go and cost descriptors. The no-goes indicate points in space that have hard obstacles. When a grid cell becomes an obstacle cell, then any open cells within the bot radius of the new obstacle cell are designated as expansion cells. Cost descriptors establish particular points in space as having higher cost (greater than the default of one)[1]. The terrain descriptors on the blackboard do not have persistence. Consequently, within L-nav a terrain-descriptor buffer with a history of five (previous four and current) L-nav cycles is used.

When there are obstacles or terrain cost descriptors, then the arena is non-empty. In this case, its position remains fixed in global space and the bot moves within the arena (in contrast to the circumstance of an empty arena). The arena is allowed to exist for up to 20 L-nav cycles. If the bot does not reach the arena subgoal within this period, the arena is destroyed and rebuilt afresh starting at the bot's new location. The cached terrain descriptors are added to the new arena. This design was chosen experimentally and offers a compromise between reactive performance and persistence of memory.

4.3 Waypoint queue

As G-nav publishes the identity of the current AWP to the blackboard, L-nav publishes subgoal waypoints to the blackboard. The subgoal waypoint queue holds up to 20 waypoints as shown in Figure 7. The queue has two parts delimited by the location of the AWP. Preceeding the AWP are the local obstacle-avoidance waypoints that were generated in the arena. Immediately in front of the AWP is the arena subgoal. Following the AWP, are the next GPS waypoints in the bot's preset itinerary. A queue size of 20 was chosen to have enough capacity to hold the local obstacle-avoidance path plus a few global waypoints. In practice, local obstacle-avoidance paths never have more than eight waypoints. When the bot approaches the end of its journey, the queue has fewer waypoints according to the number of waypoints remaining in the preset itinerary. At the end of the journey, the AWP is the last waypoint in the queue.

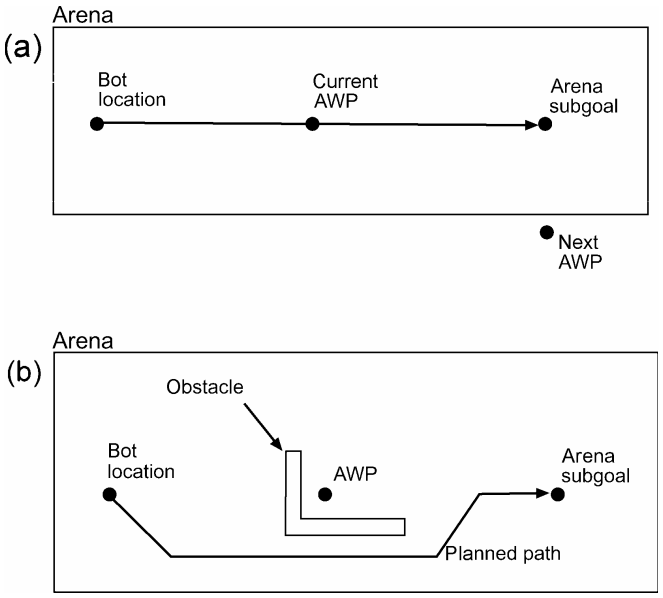
Figure 7 Subgoal waypoint queue. The bot achieves these waypoints in sequence. The queue is continuously updated



As mentioned in Section 4.1, the subgoal waypoint may be beyond the AWP. The reason for allowing this is as follows. If there are no obstacles, then if the bot travels to the subgoal, it will also travel through the AWP, thereby achieving its objective. There is the complication that the bot will backtrack when it tries to achieve the AWP after reaching the subgoal. G-nav detects this situation and discards that AWP to prevent backtracking.

There are two possibilities that this can fail. First, if the arena goal is considerably beyond the AWP, then reaching it may take the bot considerably off course as it travels to the subsequent AWP (see Figure 8a). This problem is mitigated by having the arena persist for a maximum of 20 L-nav cycles. Thus, the bot is guaranteed to head towards the subsequent AWP within 20 L-nav cycles of reaching the current AWP. The second problem is that the AWP could be missed if there are obstacles en route to the more distant subgoal as shown in Figure 8b.

Figure 8 Arena subgoal issues. (a) Bot deviates from next AWP if it tries to achieve the arena subgoal. (b) Bot entirely misses AWP if it avoids obstacle en route to arena subgoal



5 Method of study and performance

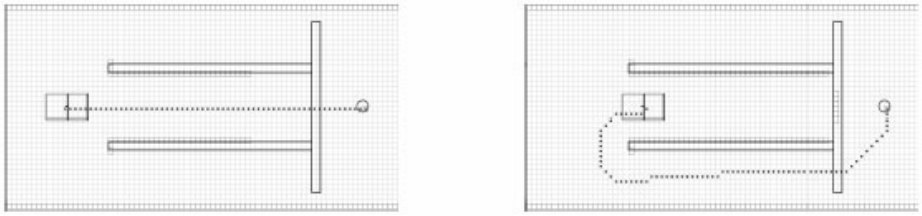
CajunBot was tested in the 2004 United States Defense Advanced Research Projects Agency (DARPA) Grand Challenge Qualification and Inspection Demonstration (QID) obstacle course at the California Speedway in March 2004 (see http://www.darpa.mil/grandchallenge). The length of this course is ~2.4 km. On its best run, CajunBot completed 75% of the course avoiding obstacles in or surrounding the path, before coming too close to a wall and being stopped manually. The failure was not attributable to the CBLN L-nav module, but to the detection and correction of errors in the INS sensor. CajunBot also participated in the 2005. Grand Challenge finals, travelling ~27 km in the final race. The vehicle used an improved path planner briefly described in the summary section of this paper.

Because CajunBot was built in 4 months, nearly all of the L-nav testing was done in simulation. This section describes how the simulations were developed, what they revealed and some anecdotes about how CajunBot performed in the QID. There were six phases of simulation development before the final L-nav module was deployed on CajunBot. These were:

Potential field visualisation. APFs were initially generated in a Java prototype and visualised using MATLAB. Examples appear in Figure 1.

Initial embedded environment simulations. In this phase, we built a Java-based agent/environment simulation of a bot navigating a local arena with obstacles. Examples are shown in Figures 5, 6 and 9. As the simulator developed, we gradually added more realism to the bot and environment such as including manoeuvrability limitations, skid steering and steering delays. Internals of the core algorithm were aggressively visualised to reveal anomalies and bugs. We also systematically added increasingly complex obstacle configurations, such as that shown in Figure 9. In these simulations, the bot performed better with the gradient-sign flow maps than with the arctan flow maps. When steering delays were introduced, even at low speeds, the bot's direction of travel would oscillate severely and the vehicle would crash. In discussion, it was pointed out that G-nav was waypoint-based and perhaps the best way for L-nav to communicate with G-nav was by supplying local subgoal waypoints to G-nav. It also seemed likely that using waypoints would simplify the steering control issues. Once the decision to use waypoint-based interaction was made, the arctan flow fields were abandoned and we committed to using the gradient-sign flow fields. It was at this point in the simulation work, that the waypoint-extraction heuristics described in Section 3.5 were developed. When waypoint-based control was added to the local arena, the steering-oscillation issues improved.

Figure 9 Dead end canyon. (Left) Bot enters canyon to approach subgoal. The canyon has a dead end, which is beyond sensor range. (Right) When dead end is detected, L-nav generates a new path to subgoal. If bot follows this path, it will hit canyon wall while turning



Initial C++ prototype. At this point, we had a local navigation prototype named CBLN, which was written in Java. However, the CajunBot software was written in C++. We needed to translate the L-nav system into C++ without introducing bugs. The low-level potential-field software was translated and then tested by visualising it in MATLAB using the methods similar to those used to generate Figure 1. However, to test the other components we needed either to link it with the existing Java testbed and visualisation environment (perhaps via JNI) or to translate the environment into C++. Since such a large percentage of the core algorithm was visualised, it was decided that translating the Java testbed into C++ and OpenGl/Glut was the best approach.

G-nav control of L-nav. Although it was envisioned that G-nav and L-nav should easily work together if their mode of interaction was waypoint-based, we had not yet developed an algorithm to realise this. The previous C++ testbed was targeted to model L-nav only. We decided to enhance it to include a G-nav testbed component. This component modelled the bot operating in a global environment supplied with obstacles and operating under GPS-waypoint control. When an obstacle came within the bot's sensor range, G-nav would invoke L-nav to obtain a sequence of subgoal waypoints that bypassed the obstacle(s). The arena construction heuristics and coordinate transformations described in Section 4.1 were developed using this testbed. This testbed also enabled testing of elements of the terrain descriptor interface. Most importantly, this testbed allowed for simultaneous observation and visualisation of the internals of G-nav and L-nav working together.

Blackboard interaction between L-nav and G-nav. In the previous simulator, G-nav invoked L-nav when an obstacle was detected. In the physical CajunBot, L-nav and G-nav were to be concurrent processes interacting via the blackboard in shared memory. The objective of this simulation was to transform L-nav into a concurrent process reading and writing to shared memory. Thus the previous prototype needed to be refactored to reflect this design. However, it still needed to retain the ability to simultaneously visualise G-nav and L-nav working together. This stage of development yielded the designs shown in Figures 3 and 7, except for the terrain descriptor component in Figure 3. In this version of the simulation, we simulated obstacle acquisition internally. However, in the physical CajunBot the terrain descriptors were extracted by another concurrent process.

Comprehensive CajunBot simulator. At this point, the code for the L-nav concurrent process was transferred to the comprehensive Cajunbot simulator to address as many system integration issues as possible. Output from this simulator is shown in Figure 11. In particular, this allowed testing of the blackboard communication between the sensor systems, L-nav and the complete G-nav. In this phase, a few coordinate transformation bugs were revealed, such as failing to translate between meters and centimeters when reading from the blackboard (the G-nav module of the previous phase did not correctly model the CajunBot G-nav coordinate system). Also some waypoint extraction bugs were uncovered. Perhaps, most revealing from a software engineering viewpoint is that the L-nav software gave uninformative error messages when it received unanticipated types of data provided by the richer simulation environment. Thus, it was difficult to distinguish bugs in L-nav from receiving bad data from other processes. Earlier simulations should have used a richer spectrum of testing data.

6 Summary, limitations and future work

The local path-planning and obstacle-avoidance module used in the CajunBot autonomous ground vehicle was described. A design requirement of the module was that it supports rapid subgoal extraction in service of a global navigation system that follows GPS waypoints. The core algorithm is built around a grid-based, linear-activation field (a type of artificial potential field). The local path planner has four distinct features:

1 Field strength changes linearly with (city-block) path length from the goal and the city-block gradient is generated by an algorithm whose worst case run time is proportional to the square of the number of grid squares.

2 The linear-activation field delivers local waypoints, or navigation subgoals, rather than a gradient.

3 The planner aggressively avoids obstacles

4 The algorithm makes use of a repulsive expansion region to compensate for imperfect manoeuvrability.

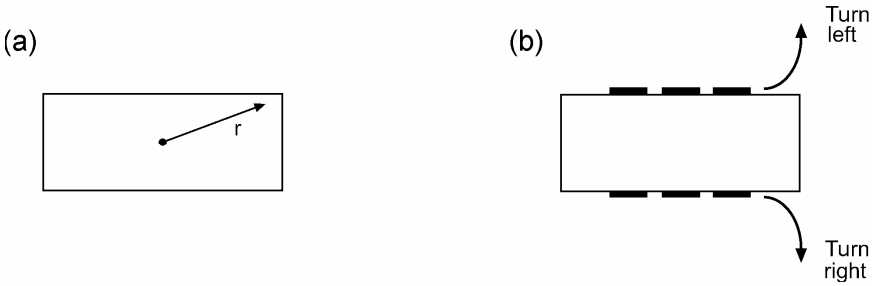
6.1 Limitations

The path planner described herein has limitations that derive from its limited model of vehicle steering limitations. The following flaw in the L-nav system was revealed in simulation experiments when the core algorithm was being developed (Java-based agent/environment simulation). Figure 9 (left) shows the bot travelling towards a goal in a canyon, which has a dead end beyond its sensor range. The most direct route to the goal appears to be through the (unknown) dead end. When the sensors detect the dead end, a new local path is generated (right). However, the bot heading is misaligned with the new local path. If the bot tries to follow the path, it will turn into the canyon wall. However, if the bot were aligned with the path, there would be no problem. Hence a test must be made whenever the bot turns more than 90°.

An expansion region for obstacle avoidance, as shown in Figure 4, works perfectly with a circular bot that can turn in place. Figure 10a shows an elongated bot that can turn in place. The distance r shows the radius of a circle circumscribing this bot. CajunBot uses an expansion radius of this size. For an elongated bot that can turn in place, a region of this size gives complete safety although it does classify some passages as impassable, which are in fact passable.

However, CajunBot only approximates the above assumptions. CajunBot is a skid steer vehicle (Figure 10b). Thus, it rotates about the left side when turning left and the right side when turning right. Simulations showed that an expansion region of $r + \epsilon$ is adequate in practice as long as the bot is aligned with the planned path and is following a path consisting of 45° turns.

Figure 10 Elongated bot and limited manoeuvrability. (a) Radius of a circle circumscribing an elongated bot that can turn in place. (b) When turning left (right), skid steer bot rotates about left (right) side



6.2 Current work

Our current work circumvents these limitations by incorporating a richer model of steering constraints at the path-planning level. In recent work, we have extended the GDF with an *options-set* consisting of virtual steerable paths that emanate from the front of the bot. The options-set encodes knowledge of the vehicle's steering constraints. Figure 11 displays new similation results that show the canyon with the dead end beyond sensor range again. However, the figure now shows the set of virtual steerable paths (options-set) that emerge from the front of the bot. All of the paths in this set must satisfy three constraints:

1 They must be steerable, given the bot's current position, heading and speed.

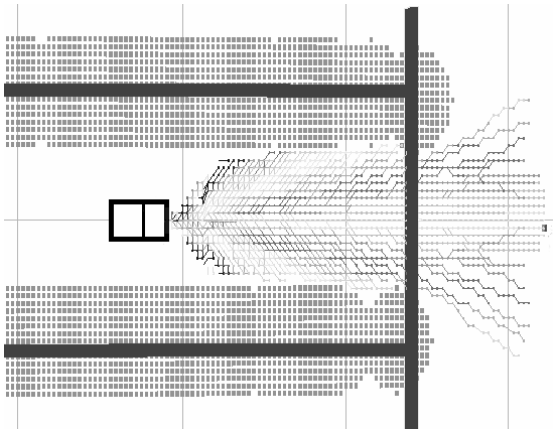
2 Their end-points must bring the bot closer to the goal, as indicated by the stablised GDF

3 They must not pass through the expansion region of any obstacles.

The path-planner must choose the best path from this set that makes the most progress towards the goal. Once chosen, the path planner extracts local waypoints from that path and then publishes them to the subgoal queue of the blackboard. Since all paths in the options-set are steerable, given the bot's current heading and speed, the bot is guaranteed to choose a path compatible with its steering limitations. When the bot's sensors detect the dead end, the options-set will become empty. This is because the stablised GDF gradient will point 180° opposite to the bot's current heading. Given the bot's current heading, there is no steerable path that makes progress towards the goal as indicated by the new stablised GDF (paths end in front of the bot but GDF gradient points to the back of bot). With an empty options set, the bot comes to a stop. This represents a principled method to handle situations like that in Figure 9, yet it is still based on the stablised GDF already implemented. The subgoal extraction rules are, however, somewhat more complicated.

Further, this virtual steerable path approach has been tested in the field in preparation for the 2005 DARPA Grand Challenge. In the actual challenge, the method was used and the vehicle travelled successfully for ~27 km in the Grand Challenge Finals.

Figure 11 A canyon with dead end beyond sensor range. Sides of canyon have been detected and expansion region computed. Lines emanating from front of bot are potential paths in the options set. They are constrained to be consistent with the steering limitations of the vehicle. When the dead end is detected the options set becomes empty and the bot stops



Acknowledgements

Nitin Jyoti and Arun Pratap Indigula helped develop the comprehensive 2004 CajunBot simulator. Firas Bouz developed the obstacle-detection capabilities in the 2004 CajunBot and Amit Putambekar and Guna Seetharaman developed them in the 2005 CajunBot. The authors also wish to acknowledge the 2004 CajunBot team and the 2005 CajunBot team.

The project was supported in part by Louisiana Governor's Information Technology Initiative and benefited from the sponsorship of C&C Technologies, MedExpress Ambulance Service, Lafayette Motors, Firely Digital, Oxford Technical Solutions and SICK USA, Inc.

References

Barraquand, J., Langlois, B. and Latombe, J.C. (1992) 'Numerical potential field techniques for robot path planning', *IEEE Transactions on Systems*, *Man and Cybernetics*, Vol. 22, pp.224–241.

Cavanaugh, C.D. (2004) 'Design and integration of the sensing and control subsystems of CajunBot', Paper presented at the *7th International IEEE Conference on Intelligent Transportation Systems*, pp.41–46. In proceedings.

Corkill, D.D. (1991) 'Blackboard systems', *AI Expert*, Vol. 6, pp.40–47.

Feng, D., Singh, S. and Krogh, B.H. (1990) 'Implementation of dynamic obstacle avoidance on the CMU Navlab', Paper presented at the *1990 IEEE Conference on Systems Engineering*, pp.208–211. In proceedings.

Glasius, R. (1997) *Trajectory Formation and Population Coding with Topographical Neural Networks*. PhD thesis, Katholieke Universiteit Nijmegen.

Glasius, R., Kamoda, A. and Gielen, C. (1995) 'Neural network dynamics for path planning and obstacle avoidance', *Neural Networks*, Vol. 8, pp.125–133.

Krogh, B.H. and Feng, D. (1989) 'Dynamic generation of subgoals for autonomous mobile robots using local feedback information' *IEEE Transactions on Automatic Control*, Vol. 34, pp.483–493.

Lagoudakis, M.G. (1998) 'Mobile robot local navigation with a polar neural map', Master's thesis, University of Louisiana, Lafayette.

Lagoudakis, M.G. and Maida, A.S. (1999) 'Neural maps for modile robot navigation', Paper presented at the *1999 IEEE International Joint Conference on Neural Networks*. In proceedings.

Maida, A. (2004) 'The post-QID CajunBot local-navigation modules', Documentation for CBLN module, May.

Stentz, A. and Hebert, M. (1995) 'A complete navigation system for goal acquisition in unknown environments', *Autonomous Robots*, Vol. 2, pp.127–145.

Notes

[1]The L-nav primitives to add terrain descriptors to the arena are add_obstacles and add_location_cost_info. They take global coordinates. get path retrieves the local waypoints from the arena. More documentation is available in Maida (2004).