Subgoal-based Local-navigation and Obstacle-avoidance using a Grid-distance Field*

Anthony S. Maida Suresh Golconda Pablo Mejia Arun Lakhotia

Tech report: 2nd working draft
June 15, 2004

Abstract

The local-path-planning and obstacle-avoidance module used in the CajunBot autonomous land rover is described. The module is designed for rapid subgoal extraction in service of a global navigation system. The core algorithm is built around a grid-based, linear-activation field which supports real-time performance. The subgoal extraction methods used in conjunction with an artificial potential field are novel. Another novel feature of the path-planner is that of *aggressive avoidance*: the planned path turns to avoid an obstacle as soon as it is detected. The system has been tested in simulation and in the CajunBot autonomous, six-wheeled, all-terrain, land rover.

1 Introduction

A local-navigation and obstacle-avoidance module named CBLN (CajunBot Local Navigation) was developed for use in an autonomous, six-wheeled, all-terrain, land-rover known as CajunBot [2]. CajunBot is a research testbed equipped with a global-positioning system (GPS) that conducts long-distance navigation by following GPS waypoints over varied terrain. The CBLN module described herein augments CajunBot's abilities by enabling her to plan paths around dynamically detected obstacles and undesirable terrain. The module continuously, in real time, delivers a set of local, nearby waypoints (subgoals) compatible with those used in CajunBot's global, long-distance navigation subsystem. The CBLN system meets the following design requirements.

1. The system supports real-time performance in a vehicle that travels at a rate of about 10 mph (4.47m/s).
2. The system supports rapid subgoal, or way-point, extraction.
3. The design allows for clear separation of realtime path-planning and realtime steering control (path execution).
4. Waypoint-based pluggability allows substituting alternative local path-planning modules for testing purposes.
5. The algorithm considers terrain cost. For instance, some terrain types are navigable but should be avoided. The algorithm provides waypoints to avoid such terrain unless there is no alternative.

*Long list of acknowledgements.

1.1 Related work and current approach

Feng & Krogh [4, 7] 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 [13].

Barraquand et al. [1] 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 precomputed many data structures), operate in real-time, or cope with situations where the robot deviates from the planned path (as in a fast-moving vehicle with imperfect steering). However, our CBLN module does use 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. However, in the present context such algorithms have the following drawback pertaining to the form of their output. Global navigation is often subgoal, or way-point, based. An APF algorithm must express its output as way-points.

Lagoudakis & Maida [8, 9] used a neural-network-based APF [5, 6] in an incremental path planner. They also used the obstacle expansion technique introduced in [4].

Stentz & Hebert [13] 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 [1]. 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*. The vehicle turns to avoid an obstacle as soon as it is detected. The GDF has the following features.

1. Field strength changes linearly with (city-block) path-length from the goal.
2. Obstacles are avoided by virtue of not falling on the trajectory path created by the gradient. However because of maneuverability uncertainties in the robot, the planned route may not exactly match the realized route. Regions of repulsion from obstacles are used only when the bot deviates from the planned path.
3. The linear gradient is generated by a simple and fast algorithm (linear on number of grid cells).
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 maneuverability uncertainties, computing a fine-grained exact path is not useful.

Figure 1 summarizes our initial design choices. The top-left flow-field was generated by the algorithm in [9]. Trajectories are smooth. However, the flow-field robustness is quite sensitive to parameter choices and scales poorly to arenas whose grid size is larger than 50×50. The top-right figure shows a GDF. Trajectories have sharp turns but performance is stable with respect parameter adjustments and the algorithm scales well with arena size. Field flows were obtained by applying minus $\arctan(x, y)$ to the x and y components of the stablized activation gradient. Further

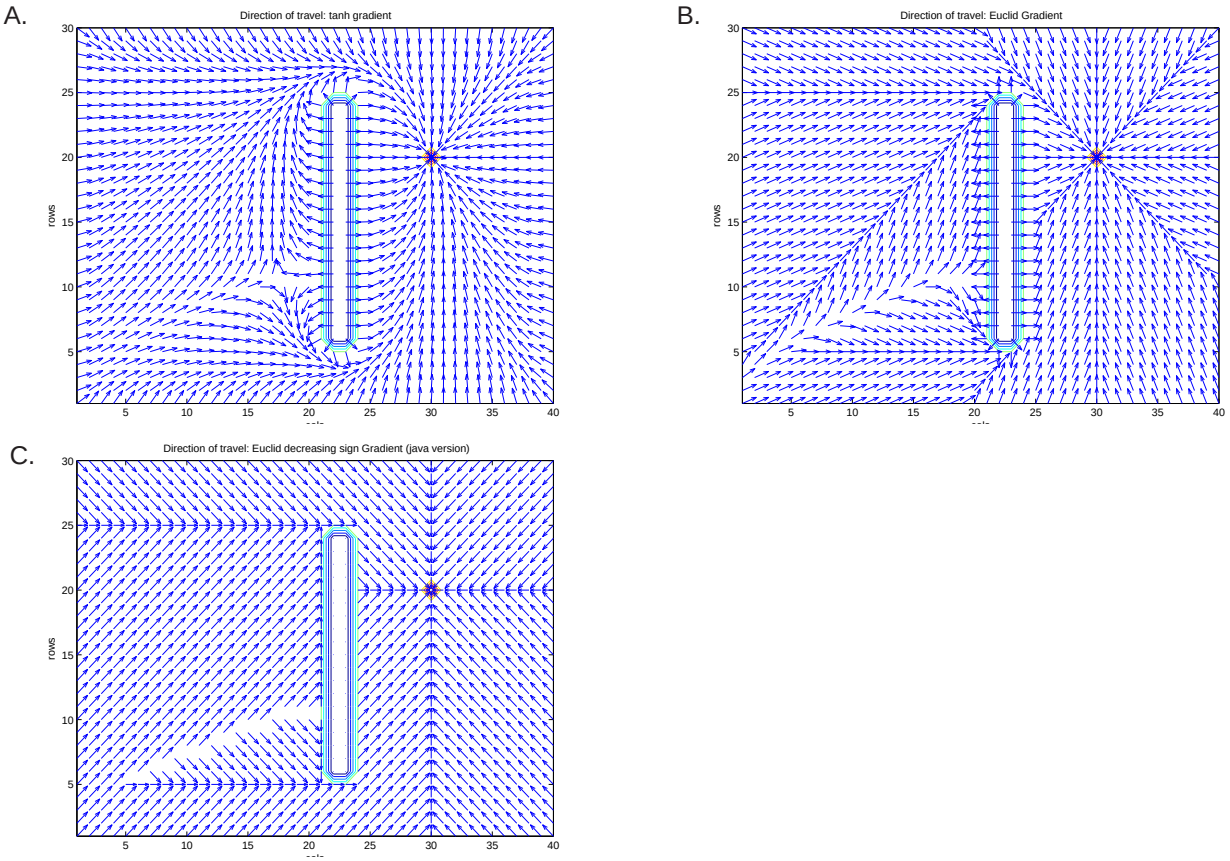

Figure 1: Initial design flow maps showing arena viewed from above. Subfigures show direction of travel (before subgoal extraction) toward goal location (30, 20) from every point in arena. Rectangular obstacle is positioned at $x = 23$ and is extended in the y-dimension. (A) Generated by a neural network. Trajectories are smooth but algorithm scales poorly to arena sizes larger than 50×50. (B) Flow-map was 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. When magnitude is not used, trajectories are straight lines joined by 45-degree turns (C). This is good for extracting way-points. Also, C shows aggressive turning to avoid obstacles. Compare locations (5, 15) in B and C.

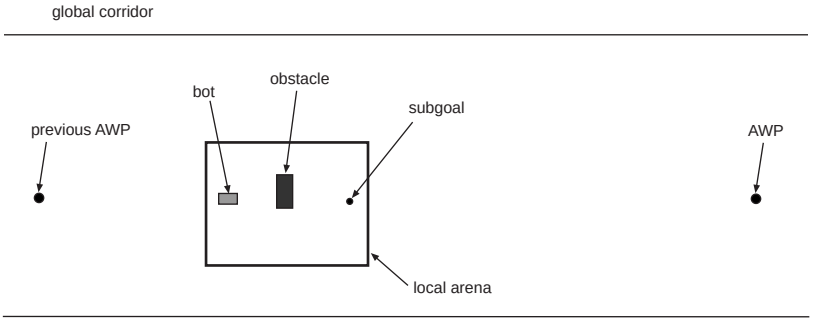

Figure 2: Local arena is constructed and used by the local navigation subsystem to circumvent dynamically detected obstacles. The GDF algorithm operates within a local arena or grid. As the bot moves, the arena moves with it. When the arena is constructed, a subgoal is chosen to be en route to distant AWP. The arena is updated on each local navigation cycle. An arena update can consist of building a new arena or adding dynamically acquired terrain descriptors to the current arena.

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 way-points from the flow fields (see Section 2.2).

The following describes the details and extensions of the CBLN module, its communication with the global navigation subsystem, and its performance in simulation and in CajunBot.

2 The local path-generation algorithm

As CajunBot travels, it visits a sequence of predetermined, global GPS waypoints. The specific 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 allows the bot to navigate around obstacles (and undesireable terrain) en route to the AWP (see Figure 2). It continuously 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 meters on a side, that moves along with the bot. CBLN communicates with the rest of the system via a realtime blackboard [3] implemented in shared memory (see Figure 3).

In the following, we refer to the global navigation 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 about 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. A later section describes exactly what L-nav reads and writes to the blackboard. Here, we say a few words about how the arena is initialized and then devote the remainder of the section to describing the path-generation computation. When the arena is initialized, 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 initialized. We now turn our discussion to computing the GDF, and then later to local waypoint extraction.

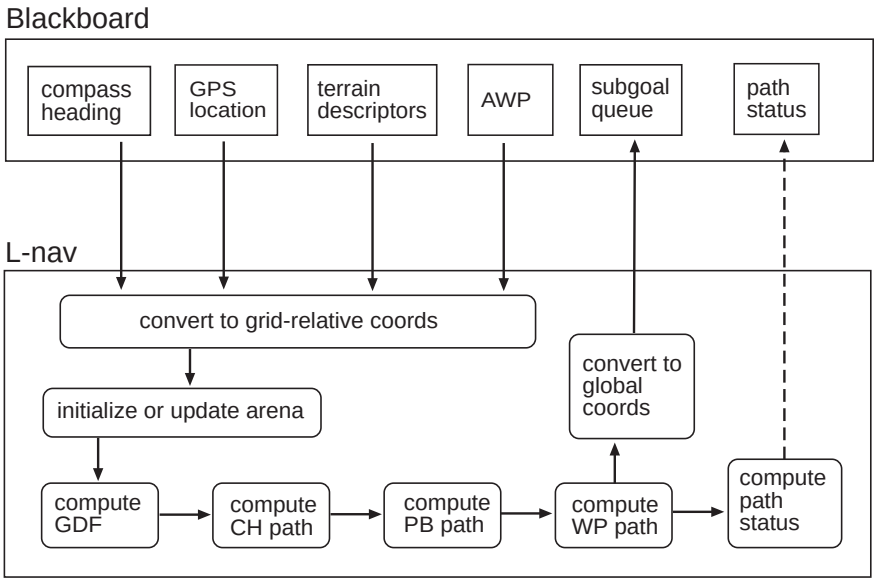

Figure 3: Information flow between the realtime blackboard and the CBLN implementation of L-nav. (The component showing the dashed arrow was not implemented.)

2.1 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 which 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 space containing known obstacles (as indicated by external sensors). Within the grid, the bot is modeled 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 center 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 greater than $r + \epsilon$ 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 2.2), 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 another 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 Section 4.2.1.

A path within the arena from the bot's current location to the subgoal region is constructed from a GDF. Before

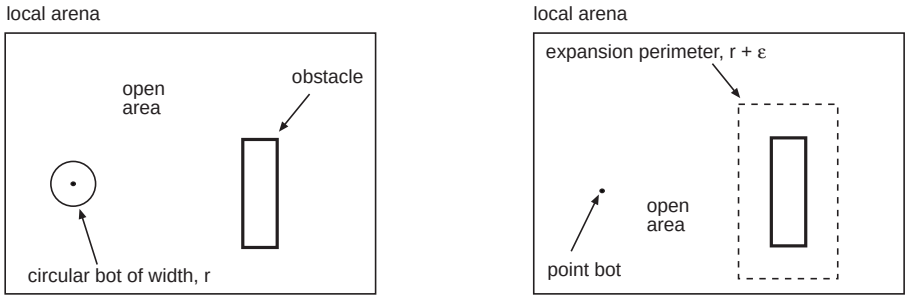

Figure 4: Use of an obstacle expansion region to simplify collision calculations.

Category	Initial value	Clamped?
init_goal_act	0	yes
init_obst_act	$2 \times$ max_pl	yes
init_open_act	max_pl	no
init_expand_act	max_pl	no

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 $< C \cdot R$). Other activations are initialized, but not clamped, to values midway between goal and obstacle activation.

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 time steps. Activation of a given cell changes according to its own type and according to the activations of its four neighbors (north, south, east, and west). We first describe how grid activation changes over one time step within 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.

2.1.1 Grid-update sweep

Let i denote a cell at grid location (c, r). Let $\mathsf{act}(i, t)$ denote the 'activation' of i at time step, t (after t grid sweeps have occurred). The different cell types have different initial activations. $\mathsf{cat}(i)$ 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 $R \cdot C$ 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 set to a value between these bounds.

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

Let $\mathsf{nb}(i)$ be the set of city-block (CB) neighbors of i. These are the cells at locations: $(c+1, r)$, $(c-1, r)$, $(c, r+1)$,

and $(c, r-1)$. Let $\mathsf{nb\_act}(i,t)$ denote the set of the activations of the neighbors of i. That is,

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

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.

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

The interesting cases are the open cells and expansion cells. For an open cell, the cell reads the activations of its four neighbors 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 neighbors but remembers the maximum value instead of the minimum. It subtracts one from this maximum. Section 2.1.3 extends (3) to include cell cost.

2.1.2 Stable activation-field propagation

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 (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;

The update sweep is synchronous: new activations are computed for all of the cells before any values are changed. Each grid-update sweep updates the activations for each of the $C \cdot R$ cells once. Thus a stable GDF is obtained after $\mathsf{max\_pl} \cdot C \cdot R$ cell-activation updates. The following theorems describe the stability and convergence properties of the activations that emerge as the GDF propagates.

The preliminary definitions formalize the notion of grid city-block distance. Given a cell, i, the *city-block (CB) neighbors* 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 *city-block 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 neighbors. The *city-block distance* from cell i to cell j is the length of the shortest city-block path from i to j. Path length is defined as the number of elements in the sequence minus one. The next theorems describe how long it takes (number of grid update sweeps) for the GDF to stabilize. The path extraction algorithms in Section 2.2 assume the GDF is stabilized.

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 $2 \cdot \mathsf{max\_pl}$, then after* max_pl *update sweeps, the activation of each open cell codes its city-block distance to the nearest goal cell.* □

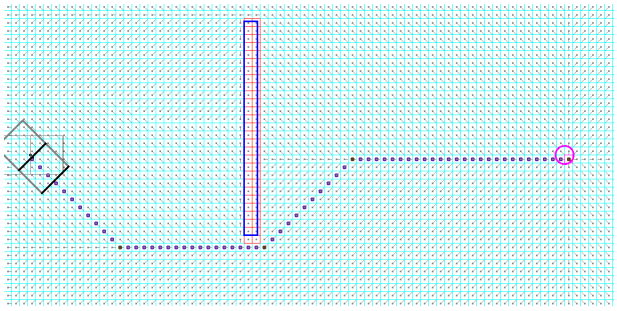

Figure 5: The CH path is composed of straight lines joined together at 45-degree angles. Background shows field flow map. This grid does not use expansion cells. Figure 6 shows a grid and associated CH path that uses expansion cells.

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,*

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

□

2.1.3 Incorporating a cost function

The update Formula (3) can be extended to include cell cost. Let $\mathsf{cost}(i) \geq 1$ 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.

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

Let $\max\_\mathrm{cost}$ be the largest cost value used in the grid. Maximum path length is now $\max\_\mathrm{cost} \times C \times R$. 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. 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.

2.2 Path extraction and local waypoint generation

2.2.1 Path extraction

After the GDF is stable, flow maps, paths, and waypoints are extracted. For a stable GDF, the activation value of an open cell, i, codes the CB distance to the nearest subgoal cell from i's location. From this information, it is possible

Sign of x–grad	Sign of y–grad	Local heading	Sign of x–grad	Sign of y–grad	Local heading	Sign of x–grad	Sign of y–grad	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

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.

---

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.

Trajectories are typically extracted from gradients by applying $\arctan(x, y)$ to the x and y gradient components of each arena location. For our GDF, minus $\arctan(x, y)$ would be used because the bot needs to travel down the gradient to approach the subgoal. 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 algorithm, only the signs of the x and y gradient components are computed for, 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 quantized into eight values known as the local, grid-relative compass headings. These quantized headings are used for path extraction (explained below). Drawing the local compass heading at each cell location generates the quantized flow map. The definitions below help characterize the paths.

An *annotated grid* is one in which each cell has been assigned a quantized 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. Grid square i is said to *target* j.

Because local headings are quantized, the GDF supports extraction of paths composed of line segments joined at 45- and 90-degree 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 CB distance, the rules of Table 2 allow the extraction of a CH path, defined below.

Definition 1 *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 $i + 1$.* □

Procedure to construct a CH path. Let G be an annotated grid and i be a particular cell. The objective is to construct a CH path from i to a subgoal cell. Assume that the GDF is stabilized. Initialize path P to the empty sequence. Put cell i at the end of P. While i is not a subgoal 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 cell in the path are neighboring cells within the grid. We do not want each adjacent cell in a CH path to be mapped into a separate waypoint. We need a path data structure which provides more support for waypoint extraction. For this purpose, the relevant feature of a path is where it changes direction. The next type of path — the PB path — saves only those sequence elements that represent changes of direction.

Definition 2 *Let P be a CH path where 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. □

Path type	Abbrev	Function
City-block	CB	Cell activations code shortest CB distance to subgoal.
Compass heading	CH	Local headings extracted from CB activations using gradient sign.
Path bends	PB	Filtered CH path to retain cells where direction changes.
Waypoints	WP	Filtered PB path to retain cells matching spacing and reachability requirements.

Table 3: Summary of path types used to obtain waypoints.

Definition 3 *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.* □

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 subgoal cell. Add the subgoal to the sequence. □

2.2.2 Waypoint extraction

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 4 *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 above, 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 maneuverability, and bot heading.

To summarize, three path-sequence types are used (see Table 3). These are CH, PB, and WP, where $\mathrm{WP} \subseteq \mathrm{PB} \subseteq \mathrm{WP}$. 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.

2.2.3 Waypoint stability

A path to the subgoal is constructed when the bot is at a particular point in the arena. If the bot is on a cell within that path (not necessarily the first cell) and then constructs a new path starting from the cell it is currently at, then the new path is a subpath of the previous path. This yields a form of path stability. The bot continuously recomputes its path. As long as the bot lives in a grid world and follows the previously computed path, the new path matches the previous path.

However, the real bot lives in continuous physical space and its current position must be mapped to a grid cell as the bot travels. This can create complications. If the bot is moving horizontally or vertically in grid-relative coordinates,

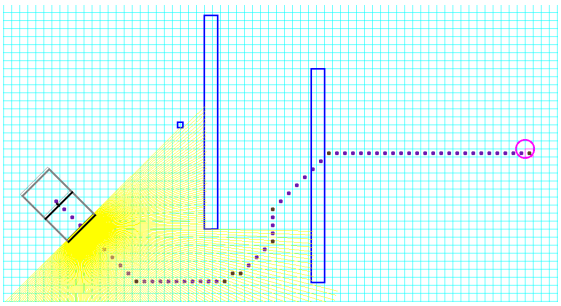

Figure 6: Incremental obstacle detection. Laser range finder does not detect the full extent of the distant obstacle. Path is constructed on the basis of the information available. The arena must be continuously updated and a new path generated as the robot travels. Path bends in the middle of the path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

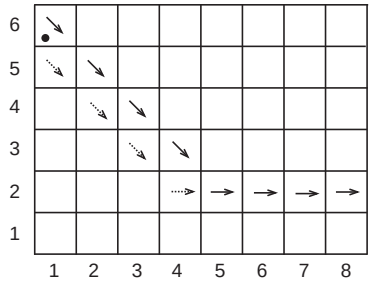

Figure 7: Initial bot location is at black dot within bottom-left of square (1, 6). Initial path is shown in black arrows with first waypoint at center of path bend (5, 2). If bot moves in continous space along path toward waypoint, bot could enter boundaries of grid square (1, 5). A path originating at this square is different from the previous path. The new path, indicated by dashed arrows, is parallel to the original path and merges with the original path. The new waypoint is at (4, 2), displaced horizontally from the original waypoint by an amount equal to the grid spacing. If the bot continues, it could reach the boundary of square (2, 5) and reconstruct the original path with the original waypoint. This leads to waypoint oscillation by an amount equal to the grid spacing.

there will be no complications. In this case, when the bot's physical position is mapped to a grid cell, that cell will be in the existing path sequence. If the bot constructs a new path from that point, then the new path will be a subpath of the original path. This property yields the stability described above.

However, if the bot travels diagonally in grid-relative coordinates, then even if the bot follows the path, mapping its position in continuous space to a grid cell does not necessarily place it on a cell in the existing path (see Figure 7). Its position could map to a cell adjacent to the path. If a new path is generated from this location, it will be slightly different. It will generate a segment that is parallel to the corresponding segment in the previous path, but displaced horizontally or vertically (in grid-relative coordinates) by an amount equal to the grid spacing. After the first bend, the two paths will merge.

The significance is that the first local waypoint extracted is located at that path bend. Depending on which path is constructed, the waypoint will be different and will shift location by an amount equal to the grid spacing. This problem can be eliminated by mapping the bot's global location to a diamond on a diamond-shaped tiling of the arena (rather than grid-shaped) during the path-extraction phase while the bot is traveling on a grid-relative diagonal heading. In other words, when the bot is traveling horizontally or vertically use a rectilinear grid. When the bot is traveling diagonally, use a diamond-shaped grid. This is only needed for path extraction and not for propagating the GDF.

3 Communication between L-nav and the blackboard

L-nav communicates with the rest of the CajunBot system by reading and writing to the realtime blackboard implemented in shared memory (Figure 3). L-nav is one of 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 for that data type. 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 from the digital terrain map, and a reference to the AWP. Data on the blackboard is expressed in global coordinates whose units are meters. The L-nav read-and-write processes must translate between the blackboard global coordinates and the grid-relative arena coordinates whose units are centimeters. The list below summarizes the complete L-nav cycle (from Section 2) with its read and write components included.

1. Read the path-relevant information from the blackboard and convert it to grid-relative coordinates.
2. Initialize or update the arena using the information from Step 1 (plus cached terrain descriptor information from the previous four L-nav cycles).
3. Propagate a stable GDF and extract subgoal waypoints as described in Section 2.
4. Write the waypoints (expressed in global coordinates) to the blackboard.

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

3.1 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 global AWP. 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

Return code	Meaning
result_normal	A path was found.
result_trapped	There is no path.
result_arena_too_small	Obstacles span width of arena.
result_goal_in_obstacle	Subgoal unreachable because within obstacle.
result_bad_input	Bad input.

Table 4: L-nav return codes.

distance between the bot and subgoal is 1500 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 3.2.2). 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. It signals this information through a return code shown in Table 4. Unfortunately, in the current implementation, this information is not written to the blackboard.

3.1.1 Conversion between grid-relative and global coordinates

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.

1. 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.
2. 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.
3. The transformation does not have a scaling component.

3.2 Blackboard interaction

3.2.1 No-go and cost-descriptor input

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 nonempty, 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 twenty 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

[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 [11].

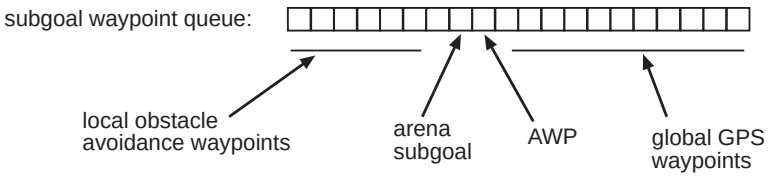

Figure 8: Subgoal waypoint queue. The bot achieves these waypoints in sequence. The queue is continuously updated.

arena. This design was chosen experimentally and offers a compromise between reactive performance and persistence of memory.

3.2.2 Waypoint queue

Whereas 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 twenty waypoints as shown in Figure 8. The queue has two parts delimited by the location of the AWP. In front of the AWP are the local obstacle avoidance waypoints that were generated in the arena. Immediately in front of the AWP is the arena subgoal. After the AWP, are the next GPS waypoints in the bot's preset itinerary. A queue size of twenty 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.

Subgoal waypoint beyond AWP As mentioned in Section 3.1, 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 can detect this situation and then discard that AWP in order to prevent backtracking.

There are two ways this can go wrong. First, if the subgoal is considerably beyond the AWP, then reaching it may take the bot considerably off course as it travels to the subsequent AWP (see Figure 9A). This problem is mitigated by having the arena persist for a maximum of twenty L-nav cycles. Thus, the bot is guarrantteed to head toward the subsequent AWP within twenty 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 9B.

4 Performance

CajunBot was tested in the DARPA Grand Challenge Qualification and Inspection Demonstration (QID) obstacle course. Because CajunBot was built in such a short time, the L-nav software was not run on the physical CajunBot

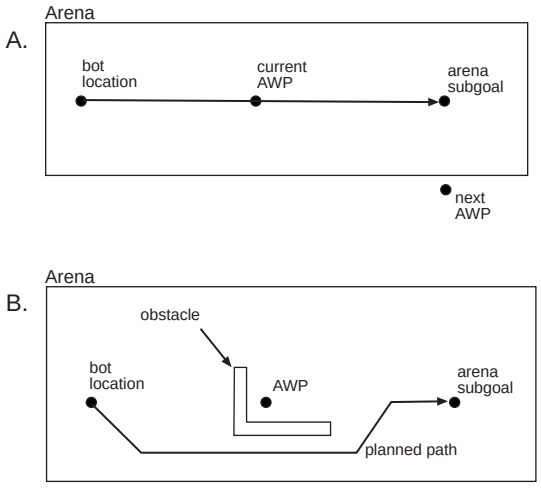

Figure 9: Arena subgoal issues. (A) Bot deviates from next AWP if it tries to achieve the arena subgoal. (B) Bot entirely misses AWP if it avoids obstacle en route to arena subgoal.

---

until after QID started. Consequently, all of the L-nav testing was done in simulation. This section describes how the simulations were developed and how CajunBot performed in the QID.

4.1 Simulation development

There were six phases of simulation development before the final L-nav module was deployed on CajunBot. These were:

4.1.1 Potential field visualization

Potential fields were initially generated in a JAVA prototype and visualized using MATLAB. Examples are shown in Figure 1. We started with a neural-network-generated potential field but found that a simple linear potential field that coded city block distance to the goal was more useable. We subsequently learned that this type of potential field was first used in [1]. We initially created flow maps using the gradient sign because we didn't immediately think of using the more standard minus $\arctan(x, y)$ function. One would expect the more standard $\arctan$ flow maps to have better properties than the gradient-sign flow maps, but this wasn't obvious from observing the flow maps. Certainly, the $\arctan$ flow maps had a seemingly desireable smoothness property but the gradient-sign maps had a nice aggressive avoidance property. Further, although the gradient-sign path trajectories had sharp turns, the trajectories themselves seemed very trustworthy and sensible. At this point, we abandoned the neural network potential fields in favor of the linear potential fields. However, in order to evaluate flow-map generation from the potential field, we needed a different testing tool.

4.1.2 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 10. As the simulator developed, we gradually added more realism to the bot and environment such as including maneuverability limitations, skid steering, and steering delays. Internals of the core algorithm were aggressively visualized in order to reveal anomalies and bugs. We also systematically added increasingly complex obstacle configurations, such as that shown in Figure 10. In these simulations, the bot performed better with the gradient-sign flow maps than with the $\arctan$ flow maps. A bit of a crisis emerged when steering delays were introduced because 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 2.2 were developed. When waypoint-based control was added to the local arena, the steering-oscillation issues did improve.

4.1.3 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++. Now we were faced with translating the L-nav system into C++ without introducing bugs. The low-level potential-field software was translated and then tested by visualizing it in MATLAB using the methods in Section 4.1.1. However, to test the other components we needed either to link it with the existing JAVA testbed and visualization environment (perhaps via JNI), or translate the environment into C++. Since such a large percentage of the core algorithm was aggressively visualized, it was decided that translating the JAVA testbed into C++ and OPENGL/GLUT was the most prudent approach. It was also decided that the team would develop a comprehensive CajunBot simulation system [10] that mimicked the physical CajunBot. The design objective for this system was to solve the full system integration and testing problem. It was intended that any code that ran correctly in the *comprehensive simulator* should be trustworthy in the physical CajunBot. Also, in parallel, we continued the existing methodology of developing a series of *targeted simulations*, each of which was designed to study and answer questions about a specific problem.

4.1.4 G-nav control of L-nav

Although it was expected that G-nav and L-nav should easily work together if their means of interaction were waypoint-based, we had not yet developed an algorithm to realize this. The previous C++ testbed was targeted to model L-nav. We decided to enhance it to include a G-nav testbed. This testbed modeled the bot operating in a global environment supplied with obstacles and operating under GPS-waypoint control. Whenever an obstacle came within the bot's sensor range, G-nav would call L-nav to obtain a sequence of subgoal waypoints that would bypass the obstacle(s). The arena construction heuristics and coordinate transformations described in Section 3.1 were initially formulated and studied using this testbed. This testbed also enabled testing of some elements of the terrain descriptor interface. However, most importantly, this testbed allowed for simultaneous observation and visualization of the internals of G-nav and L-nav working together.

4.1.5 Blackboard interaction between L-nav and G-nav

In the previous simulator, G-nav invoked L-nav whenever an obstacle was in sensor range. In the physical CajunBot, L-nav and G-nav were to be concurrent processes interacting via the blackboard in shared memory. The objective

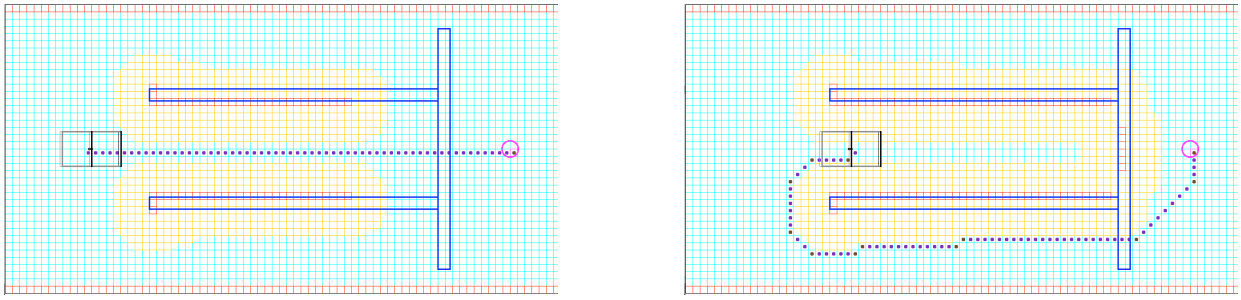

Figure 10: Dead end canyon. (Left) Bot enters canyon to approach subgoal. The canyon has a dead end which is beyond sensor range. (Right) When dead end is detected, L-nav generates a new path to subgoal. If bot follows this path, it will hit canyon wall while turning.

---

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 visualize G-nav and L-nav working together. This stage of development yielded the designs shown in Figures 3 and 8, 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 being extracted by another concurrent process. Also, this simulation did not use the G-nav software in the physical CajunBot because it was not deemed necessary to answer the concurrency issues of L-nav.

4.1.6 Comprehensive CajunBot simulator

At this point, the code for the L-nav concurrent process was transferred to the comprehensive Cajunbot simulator [10] to address as many system integration issues as possible. 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 standpoint 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 simply receiving bad data from other processes. Earlier simulations should have used a richer spectrum of testing data.

The comprehensive CajunBot simulation did have limitations, however. For instance, it could not simulate the richness of the sensor input provided by the physical CajunBot operating in a real environment. In addition, there were some concurrency surprises that were not revealed in the comprehensive simulation. Perhaps one reason for this was that the comprehensive simulation needed to allocate resources to simulate the testbed environment, whereas this was not necessary in the software that ran on the physical CajunBot. The differing processing loads on the two systems could have lead to the observed concurrency discrepancies.

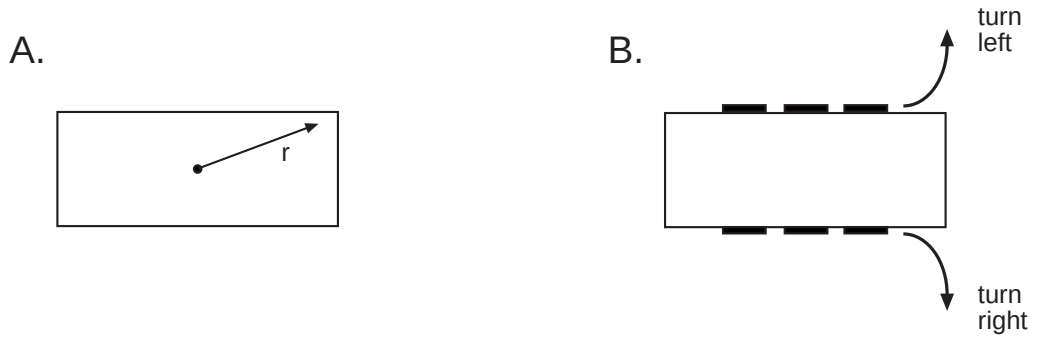

Figure 11: Elongated bot and limited maneuverability. (A) Radius of a circle circumscribing an elongated bot that can turn in place. (B) When turning left (right), skid steer bot rotates about left (right) side.

4.2 Simulated Performance

4.2.1 Bot not aligned with path

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 10 (left) shows the bot traveling down a canyon toward goal. The canyon has a dead end which is beyond the bot's sensor range. On basis of current information, the most direct route to the goal is through the (unknown) dead end. When the sensors detect the dead end, a new local path is generated (right). However, the bot heading is not aligned with the new local path. If the bot tries to follow the path, it will collide with the canyon wall while turning. However, if the bot were aligned with path, there would be no problem. Hence a test must be made whenever the bot turns more than 125 degrees.

Why didn't the expansion region work? An expansion region for obstacle avoidance, as shown in Figure 4, works perfectly with a circular bot that can turn in place. Figure 11A 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 violates the above assumptions. CajunBot cannot turn in place and does not rotate about its center. CajunBot has skid steering (Figure 11B). That is, 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 degree turns.

4.3 Performance in field testing

Obstacle on waypoint There was a disabled van placed as an obstacle on top of a global waypoint. This of course tests whether the bot will discard the current AWP and move on to the next. Our vehicle did, in fact, circumvent the van in its first run, and it is instructive to examine why. As the van was approached, L-nav constructed a subgoal en route to the AWP, but 15 meters ahead of its current position. This actually places the subgoal *beyond* the global AWP and beyond the van. The subgoal waypoint was then put in the shared-memory queue in front of the AWP. So the

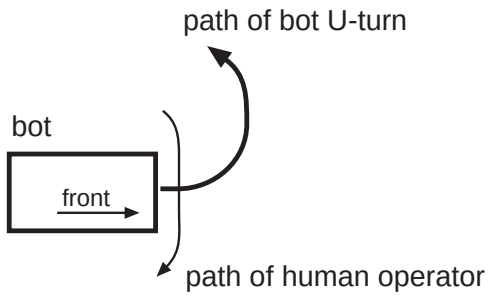

Figure 12: The path of the human operator is persistent. The bot cannot travel backwards. The bot wants to go around the perceived obstacle but, on the basis of the information it has, cannot do so without a collision.

G-nav system approached the subgoal before approaching the then active AWP. Once reaching the subgoal, it selected the next waypoint from the queue and this was the AWP centered within the van. The G-nav system first checked whether it already reached or passed this waypoint on its extended journey. G-nav realized that it was beyond this waypoint and abandoned it to press forward on its journey. **Author's note:** This is a guess. Perhaps examining the system logs would tell us whether this really happened.

Robot is trapped In this episode, the robot was in the starting shute. The system was running and waiting for a go signal. A DARPA operator was on the left side of the bot and, inexplicably, walked across the front of the bot to get to the other side while CajunBot was in start-up mode (We all thought this was remarkably dangerous and out of character for DARPA's stringent safety standards). When the bot finally received the go signal, it did nothing and did not move. Afterwards, the system log said the bot wanted to make a U-turn, but was trapped. My teammates initially suspected a bug in the local path-planning system. Upon reflection, it became apparent that the operator was a moving obstacle that walked across the entire sensory field of the laser range finder, and at very close range. The system had not been calibrated for moving obstacles, and so the operator was interpreted as a persistent, very-close-range static obstacle blocking the entire field (see Figure 12). **Author's note:** I wrote this description in April, before writing Section 3.2.1. Since the arena refreshed after twenty L-nav cycles, the ghost obstacle created by the operator should have been forgotten. So, I'm not sure what really happened. **Question:** Does the DTM have any persistence properties?

5 Discussion, limitations, future work

Conclusions

1. A local arena that supports eight quantized compass headings is adequate for practical waypoint extraction.
2. Aggressive avoidance is also a nice property of the L-nav path generation algorithm.

Limitations and future work

1. The logging facilities for L-nav need to be improved.

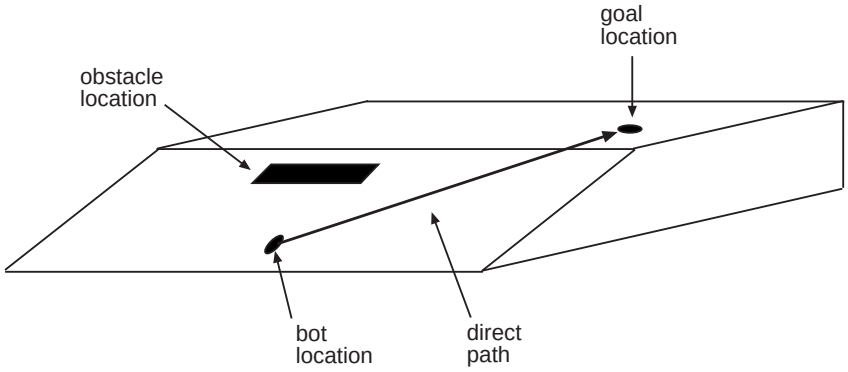

Figure 13: Local path planner tells bot to go directly to goal. Reactive system for balance reflex has higher priority and tells bot to align itself with the gradient. Another reflex tells bot to avoid obstacle. How does the reactive system resolve conflicting reflexes?

---

2. L-nav does path-planning in the two-dimensional plane. However, the bot travels on the surface of 3D terrain.
3. Cost is based on cell cost, not arc cost. Need to integrate graph-based regional planning with geometry-based local planning [12].
4. Need to consider general architecture with reactive (reflexive), local, regional, and global navigation. Individual reflexes can conflict (e.g., gradient alignment verses obstacle avoidance as shown in Figure 13).
5. Reactive navigation should not translate to global coordinates.
6. Need to incorporate obstacle classification to distinguish harmless obstacles like small weeds and serious obstacles that look similar (e.g., barbed-wire).
7. Need to distinguish between moving obstacles and persistent obstacles.

Issues raised by Pablo in his April 24th email In this section, I discuss the role of L-nav in some of the issues that Pablo has raised.

Stopping "*The path planner needs to determine when the vehicle is unable to make progress and signal this condition.*" That's true. Table 4 shows that the CBLN module has a return code for when the robot is trapped. However, as shown in Figure 3, we forgot to establish a protocol to write the result back to shared memory so that it could be used by `Brain.C`.

Course boundary "*We are not honoring the course boundaries.*" Also, true. Given the existing L-nav, I think the best way to solve this is to have the CBLN wrapper program (the part of the program that adds the terrain descriptors to the arena) add course boundary information to the arena as it reads the RDDF file. The course boundary could be treated as an obstacle.

Stalemate "*The CB feels free to backtrack looking for a way out of a dead end even though there are no forks to backtrack to.*" First, I think this is in part a regional navigation issue as opposed to a local navigation issue, but let me explain what happens in the current implementation of CBLN. Recall from Section 3.2 that an existing arena is deemed stale after a maximum of twenty L-nav cycles and is rebuilt. Assume for the moment that this did not happen. In that situation, CajunBot would not feel free to back track to dead ends that are inside the arena (CBLN does not generate paths that go outside of the arena boundaries). It backtracks because of the limited persistence of the arena and its associated information.

Forgetting obstacles *"Once the vehicle has stopped, how does it detect that an obstacle has disappeared."* As described above, if CajunBot waits for a period of twenty L-nav cycles, then the CBLN module will forget the existence of the obstacle.

Tension between forgetting and persistence Pablo makes reference to an inherent conflict between remembering not to backtrack to dead ends and realizing that a non-persistent obstacle (e.g., a stopped vehicle) may have moved. This is a good question. To address this problem, we need some means to classify obstacles as persistent verses non-persistent.

Need for a library of scenarios Lastly, Pablo pointed out that we need a library of test scenarios in order to approach these questions systematically. I agree completely. The simulations described in Section 4.1.2 used a rudimentary library of local navigation scenarios. The simulations of G-nav control of L-nav described in Section 4.1.4 included an environment containing several G-nav/L-nav interaction scenarios, although it was not a systematic library. Also, in my opinion, the methodology of using a series of targeted simulations (augmented with aggressive visualization to answer targeted questions) helps make it clear which questions of been answered and which questions still remain open.

References

[1] J. Barraquand, B. Langlois, and J. C. Latombe. Numerical potential field techniques for robot path plannning. *IEEE Transactions on Systems, Man, and Cybernetics*, 22:224–241, 1992.

[2] Charles D. Cavanaugh. Design and integration of the sensing and control subsystems of CajunBot. In *7th International IEEE Conference on Intelligent Transportation Systems*, 2004.

[3] Daniel D. Corkill. Blackboard systems. *AI Expert*, 6(9):40–47, 1991.

[4] D. Feng, S. Singh, and B. H. Krogh. Implementation of dynamic obstacle avoidance on the cmu navlab. In *Proceedings of the 1990 IEEE Conference on Systems Engineering*, pages 208–211, 1990.

[5] R. Glasius, A. Kamoda, and C. Gielen. Neural network dynamics for path planning and obstacle avoidance. *Neural Networks*, 8(1):125–133, 1995.

[6] Roy Glasius. *Trajectory Formation and Population Coding with Topographical Neural Networks*. PhD thesis, Katholieke Universiteit Nijmegen, 1997.

[7] B. H. Krogh and D. Feng. Dynamic generation of subgoals for autonomous mobile robots using local feedback information. *IEEE Transactions on Automatic Control*, 34(5):483–493, 1989.

[8] Michail G. Lagoudakis. Mobile robot local navigation with a polar neural map. Master's thesis, University of Louisiana at Lafayette, 1998.

[9] Michail G. Lagoudakis and Anthony S. Maida. Neural maps for modile robot navigation. In *Proceedings of the 1999 IEEE International Joint Conference on Neural Networks*, 1999.

[10] Arun Lakhotia et al. The CajunBot simulator. Documentation for the CajunBot simulator, 2004.

[11] Anthony Maida. The post-QID CajunBot local-navigation module. Documentation for CBLN module, May 2004.

[12] Ananth Ranganathan and Sven Koenig. PDRRTs: Integrating graph-based and cell-based planning. www.cc.gatech.edu/ai/robot-lab/publications.html, 2004.

[13] A. Stentz and M. Hebert. A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots*, 2(2):127–145, 1995.