6 Planning and Navigation:

Where am I going? How do I get there?

6.1 Introduction

This text has focused on the elements of a mobile robot that are critical to robust mobility: the kinematics of locomotion; sensors for determining the robot's environmental context; and techniques for localizing with respect to its map. We now turn our attention to the robot's *cognitive level.* Cognition generally represents the purposeful decision making and execution that a system utilizes in order to achieve its highest-order goals.

In the case of a mobile robot, the specific aspect of cognition directly linked to robust mobility is *navigation competence*. Given partial knowledge about its environment and a goal position or series of positions, *navigation* encompasses the ability of the robot to act based on its knowledge and sensor values so as to reach its goal positions as efficiently and as reliably as possible. The focus of this chapter is how the tools of the previous chapters can be combined to solve this navigation problem.

Within the mobile robotics research community, a great many approaches have been proposed for solving the navigation problem. As we sample from this research background it will become clear that in fact there are strong similarities between all of these approaches even though they appear, on the surface, quite disparate. The key difference between various navigation architectures is the manner in which they decompose the problem into smaller sub-units. In Section (6.3) below, we describe the most popular navigation architectures, contrasting their relative strengths and weaknesses.

First, however, in the following Section we discuss two key additional competencies required for mobile robot navigation. Given a map and a goal location, *path planning* involves identifying a trajectory that will cause the robot to reach the goal location when executed. Path planning is a strategic problem-solving competency, as the robot must decide what to do over the long term in order to achieve its goals.

The second competency is equally important but occpies the opposite, tactical extreme. Given real-time sensor readings, *obstacle avoidance* means modulating the trajectory of the robot in order to avoid collisions. A great variety of approaches have demonstrated competent obstacle avoidance, and we survey a number of these approaches as well.

6.2 Competencies for Navigation: planning and reacting

In the Artificial Intelligence community planning and reacting are often viewed as contrary approaches or even opposites. In fact, when applied to physical systems such as mobile robots, planning and reacting have a strong complementarity, each being critical to the other's success. The navigation challenge for a robot involves executing a course of action (or plan) to reach its goal position. During execution, the robot must react to unforeseen events (e.g. obstacles) in such a way as to still reach the goal. Without reacting, the planning effort will not pay off because the robot will never physically reach its goal. Without planning, the reacting effort cannot guide the overall robot behavior to reach a distant goal- again, the robot will never reach its goal.

An information theorectic formulation of the navigation problem will make this complementarity clear. Suppose that a robot *R* at time *i* has a map M_i and an initial belief state b_i. The robot's goal is to reach a position *p* while satisfying certain some temporal constraints: $loc_g(R) = p; (g \leq n)$. Thus the robot must be at location *p* at or before timestep *n*.

Although the goal of the robot is distinctly physical, the robot can only really sense its belief state, not its physical location, and therefore we map the goal of reaching location *p* to reaching a belief state b_g, corresponding to the belief that $loc_g(R) = p$. With this formulation, a plan *q* is nothing more than one or more trajectories from b_i to b_g. In other words, plan *q* will cause the robot's belief state to transition from b_i to b_g, *if the plan is executed from a world state consistent with both* b_i *and* M_i.

Of course the problem is that the latter condition may not be met. It is entirely possible that the robot's position is not quite consistent with b_i, and it is even likelier that M_i is either incomplete or incorrect. Furthermore, the real-world environment is dynamic. Even if M_i is correct as a single snapshot in time, the planner's predictions regarding how *M* changes over time are probably imperfect.

In order to reach its goal nonetheless, the robot must incorporate new information gained during plan execution. As time marches forward, the environment changes and the robot's sensors gather new information. This is precisely where reacting becomes relevant. In the best of cases, reacting will modulate robot behavior locally in order to correct the planned-upon trajectory so that the robot still reaches the goal. Of course, at times, unanticipated new information will require changes to the robot's strategic plans, and so ideally the planner also incorporates new information as that new information is received.

Taken to the limit, the planner would incorporate every new piece of information in real time, instantly producing a new plan that in fact reacts to the new information appropriately. This theoretical extreme, at which point the concept of planning and the concept of reacting merge, is called *integrated planning and execution* and will be discussed in Section (6.3.2.3).

In the next two sub-sections, we begin by describing key aspects of planning and reacting as they apply to a mobile robot: path planning and obstacle avoidance. For greater detail, refer

to [5] and to chapter 25 of [16].

6.2.1 Path Planning

Even before the advent of affordable mobile robots, the field of path planning was heavily studied due to applications in the area of industrial manipulator robotics. Interestingly, the path planning problem for a manipulator with, for instance, 6 degrees of freedom (DOF) is far more complex than that of a differential drive robot operating in a flat environment. Therefore, although we can take inspiration from the techniques invented for manipulation, the path planning algorithms used by mobile robots tend to be simpler approximations due to the greatly reduced DOF. Furthermore, industrial robots often operate at the fastest possible speed because of the economic impact of high throughput on a factory line. So, the dynamics and not just the kinematics of their motions are significant, further complicating path planning and execution. In contrast, a majority of mobile robots today operate at such low speeds that dynamics are rarely considered during path planning, further simplifying the mobile robot instantiation of the problem.

Configuration Space

Path planning for manipulator robots and, indeed, even for most mobile robots, is formally done in a representation called ***configuration space***. Suppose that a robot arm (e.g. SCARA-robot) has k degrees of freedom (DOF). Every state or configuration of the robot can be described with k real values: $q_1, \ldots, q_k$. The k values can be regarded as a point p in a k-dimensional space called the configuration space C of the robot. This description is convenient because it allows us to describe the complex three-dimensional shape of the robot with a single k-dimensional point.

Now consider the robot arm moving in an environment where the workspace (i.e. its physical space) contains known obstacles. The goal of path planning is to find a path in the physical space from the initial position of the arm to the goal position avoiding all collisions with the obstacles. This is a difficult problem to visualize and solve in the physical space particularly as k grows large. But in configuration space the problem is straightforward. If we define the *configuration space obstacle* $\boldsymbol{O}$ as the subspace of C where the robot arm bumps into something, we can compute the *free space* $\boldsymbol{F=C-O}$ in which the robot can move safely.

Figure 6.1 shows a picture of the physical space and configuration space for a planar robot arm with two links. The robot's goal is to move its end effector from position *c1* to *c2*. The configuration space depicted is two dimensional because each of two joints can have any position from 0 to 2π. It is easy to see that the solution in C-space is a line from *c1* to *c2* that remains always within the free space of the robot arm.

For mobile robots operating on flat ground, we generally represent robot position with three variables: $(x,\ y,\ \theta)$ as in Chapter 3. But, as we have seen, most robots are non-holonomic, using differential drive systems or Ackerman steered systems. For such robots, the non-holonomic constraints limits the robot's velocity $(\dot{x},\ \dot{y},\ \dot{\theta})$ in each configuration (x, y, θ). For details regarding the construction of the appropriate Free Space to solve such path planning

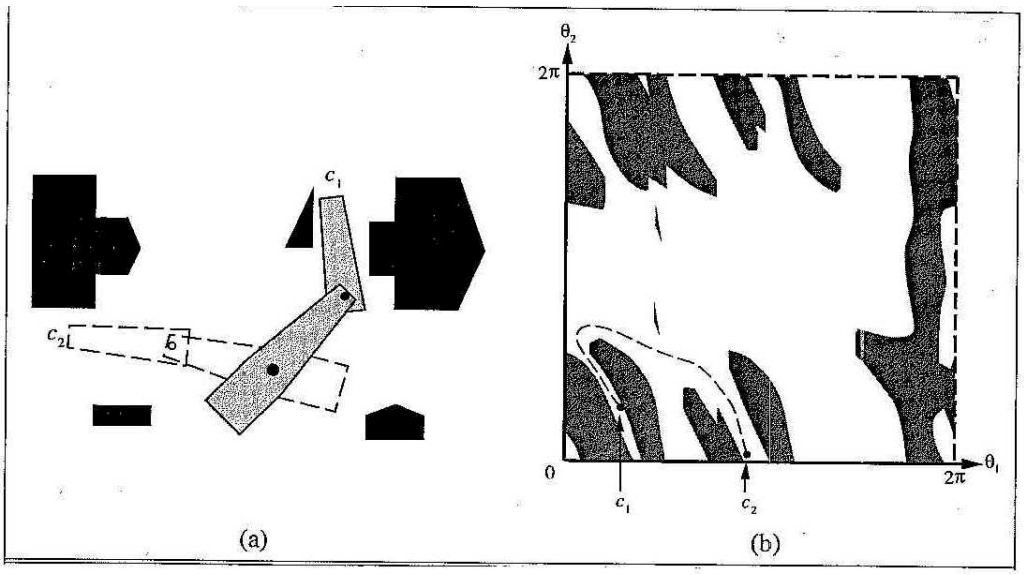

Fig 6.1 *Physical space (a) and configuration space (b) (adopted from [16] p. 792): a) A two-link planar robot arm has to move from the configuration point c1 to c2. b) The corresponding configuration space shows the free space and a path that achieves the goal*

problems, see [5], page 405.

In mobile robotics, the most common approach is to assume for path planning purposes that the robot is in fact holonomic, simplifying the process tremendously. This is especially commonplace for differential-drive robots because they can rotate in place and, so, a holonomic path can be easily mimicked if the rotational position of the robot is not critical.

Furthermore, mobile roboticists will often plan under the further assumption that the robot is simply a *point*. Thus we can further reduce the configuration space for mobile robot path planning to a two dimensional representation with just *x* and *y* axes. The result of all this simplification is that the configuration space looks essentially identical to a two-dimensional (i.e. flat) version of the physical space, with one important difference. Because we have reduced the robot to a point, we must inflate each obstacle by the size of the robot's radius to compensate. With this new, simplified configuration space in mind, we can now introduce common techniques for mobile robot path planning.

Path Planning Overview

The robot's environment representation can range from a continuous geometric description to a decomposition-based geometric map or even a topological map, as described in Section 5.5. The first step of any path planning system is to transform this possibly continuous environment model into a discrete map suitable for the chosen path planning algorithm. Path planners differ as to how they effect this discrete decomposition. We can identify three general strategies for decomposition:

- road map: identify a set of routes within the free space
- cell decomposition: discriminate between free and occupied cells

- potential field: impose a mathematical function over the space

The following Sub-sections present common instantiations of the road map and cell decomposition path planning techniques.

6.2.1.1 Road-Map Path Planning

Road-map approaches capture the connectivity of the robot's free space in a network of one-dimensional curves or lines, called *road-maps*. Once a road-map is constructed, it is used as a network of road (path) segments for robot motion planning. Path planning is thus reduced to connecting the initial and goal positions of the robot to the road network, then searching for a series of roads from the initial robot position to its goal position.

The road-map is a decomposition of the robot's configuration space based specifically on obstacle geometry. The challenge is to construct a set of roads that together enable the robot to go anywhere in its free space, while minimizing the number of total roads. We describe two road-map approaches below that achieve this result with dramatically different types of roads. In the case of the Visibility Graph, roads come as close as possible to obstacles and resulting paths are minimum-length solutions. In the case of the Voronoi Diagram, roads stay as far away as possible from obstacles.

Visibility Graph

The *visibility graph* for a polygonal configuration space *C* consists of edges joining all pairs of vertices that can see each other (including both the initial and goal positions as vertices as well). The unobstructed straight lines (roads) joining those vertices are obviously the shortest distances between them. The task of the path planner is thus to find the shortest path from the initial position to the goal position along the roads defined by the visibility graph (fig. 6.2).

Visibility graph path planning is moderately popular in mobile robotics, partly because implementation is quite simple. Particularly when the environment representation describes objects in the environment as polygons in either continuous or discrete space, the visibility graph search can employ the obstacle polygon descriptions readily.

There are, however, two important caveats when employing visibility graph search. First, the size of the representation and the number of edges and nodes increase with the number of obstacle polygons. Therefore the method is extremely fast and efficient in sparse environments, but can be equally slow and inefficient compared to other techniques when used in densely populated environments.

The second caveat is a much more serious potential flaw: the solution paths found by visibility graph planning tend to take the robot as close as possible to obstacles on the way to the goal. More formally, we can prove that Visibility graph planning is *optimal* in terms of the length of the solution path. This powerful result also means that all sense of safety, in terms of staying a reasonable distance from obstacles, is sacrificed for this optimality. The common solution is to grow obstacles by significantly more than the robot's radius, or alternatively modifying the solution path after path planning to distance the path from obstacles

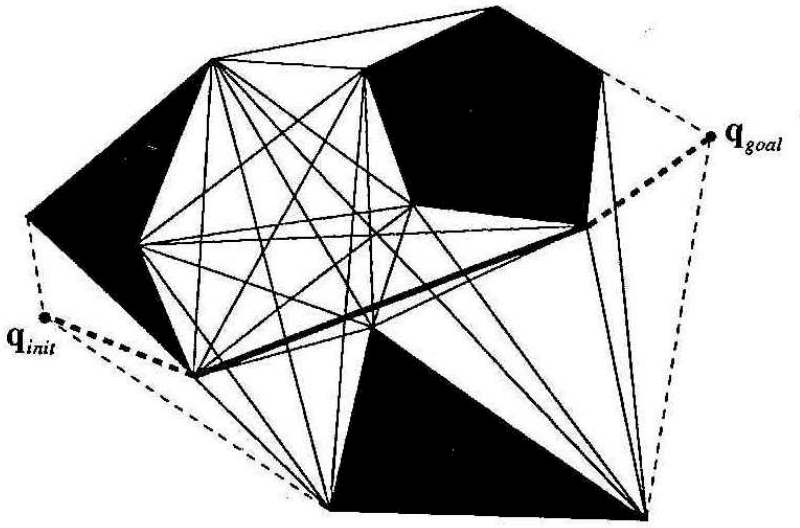

Fig 6.2 *Visibility graph (adopted from [5] p. 13):*
The nodes of the graph are the initial and goal points and the vertices of the configuration space obstacles (polygons). All nodes which are visible from each other are connected by straight line segments, defining the road map. This means there are also edges alonhg each polygon's sides.

when possible. Of course such actions sacrifice the optimal-length results of Visibility Graph path planning.

Voronoi Diagram

Contrasting with the Visibility Graph approach, a Voronoi Diagram is a complete road-map method that tends to maximize the distance between the robot and obstacles in the map. For each point in the free space, compute its distance to the nearest obstacle. Plot that distance in figure 6.3 as a height coming out of the page. The height increases as you move away from an obstacle. At points that are equidistant from two or more obstacles such a distance plot has sharp ridges. The Voronoi diagram consists of the edges formed by these sharp ridge points. When the configuration space obstacles are polygons, the Voronoi diagram consists of straight and parabolic segments. Algorithms that find paths on the Voronoi road map are complete just like Visibility Graph methods, because the existence of a path in the free space implies the existence of one on the Voronoi diagram as well (*i.e.* both methods guarantee completeness). However, the path in the Voronoi diagram is usually far from optimal in the sense of total path length.

The Voronoi Diagram has an important weakness in the case of limited range localization sensors. Since this path planning algorithm maximizes the distance between the robot and objects in the environment, any short-range sensor on the robot will be in danger of failing to sense its surroundings. If such short-range sensors are used for localization, then the chosen path will be quite poor from a localization point of view. On the other hand the visibility graph method can be designed to keep the robot as close as desired to objects in the map.

There is, however, an important subtle advantage that the Voronoi Diagram method has over

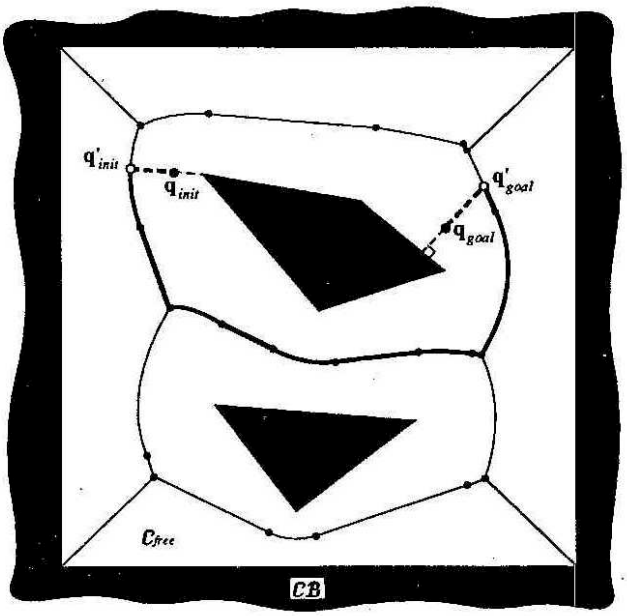

Fig 6.3 *Voronoi diagram (adopted from [5] p. 14):*
The Voronoi diagram consists of the lines constructed from all points that are equidistant from two or more obstacles. The initial q_{init} and goal q_{goal} configurations are mapped into the Voronoi diagram to q'_{init} and q'_{goal}, each by drawing the line along which its distance to the boundary of the obstacles increases the fastest. The direction of movement on the Voronoi diagram is also selected so that the distance to the boundaries increases fastest. The points on the Voronoi diagram represent transitions from line segments (minimum distance between two lines) to parabolic segments (minimum distance between a line and a point).

most other obstacle avoidance techniques: *executability*. Given a particular planned path via Voronoi Diagram planning, a robot with range sensors such as a laser rangefinde or ultrasonics can follow a Voronoi edge in the physical world using simple control rules that match those used to create the Voronoi Diagram: the robot maximizes the readings of local minima in its sensor values. This control system will naturally keep the robot on Voronoi edges, so that Voronoi-based motion can mitigate encoder inaccuracy. This interesting physical property of the Voronoi Diagram has been used to conduct automatic mapping of an environment by finding and moving on unknown Voronoi edges, then constructing a consistent Voronoi map of the environment [120].

6.2.1.2 Cell Decomposition Path Planning

The idea behind cell decomposition is to discriminate between geometric areas, or cells, that are free and areas that are occupied by objects. The basic cell decomposition path planning algorithm can be summarized as follows [16]:

- Divide *F* into simple, connected regions called "*cells*"
- Determine which opens cells are *adjacent* and construct a "*connectivity graph*"
- Find the cells in which the initial and goal configurations lie and search for a path in

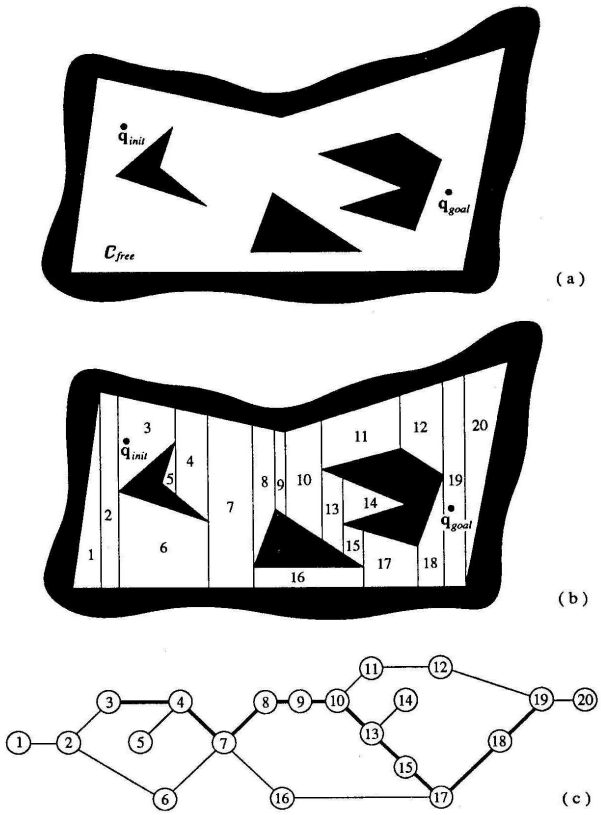

Fig 6.4 *Example of exact cell decomposition.*

the connectivity graph to join the initial and goal cell.

- From the sequence of cells found with an appropriate searching algorithm, compute a path within each cell, *e.g.* passing through the midpoints of the cell boundaries or by a sequence of wall following motions and movements along straight lines.

An important aspect of cell decomposition methods is the placement of the boundaries between cells. If the boundaries are placed as a function of the structure of the environment, such that the decomposition is *lossless*, then the method is termed *exact cell decomposition*. If the decomposition results in an approximation of the actual map, the system is termed *approximate cell decomposition*. In Section 5.5.2 we described these decomposition strategies as they apply to the design of map representation for localization. Here, we briefly summarize these two cell decomposition techniques once again, providing greater detail about their advantages and disadvantages relative to path planning.

Exact Cell Decomposition

Figure 6.4 depicts exact cell decomposition, whereby the boundary of cells is based on geometric criticality. The resulting cells are each either completely free or completely occupied, and therefore path planning in the network is complete, like the road-map based methods above. The basic abstraction behind such a decomposition is that the particular position of the robot within each cell of free space does not matter; what matters is rather the robot's ability to traverse from each free cell to adjacent free cells.

The key disadvantage of exact cell decomposition is that the number of cells and, therefore, overall path planning computational efficiency depends upon the density and complexity of

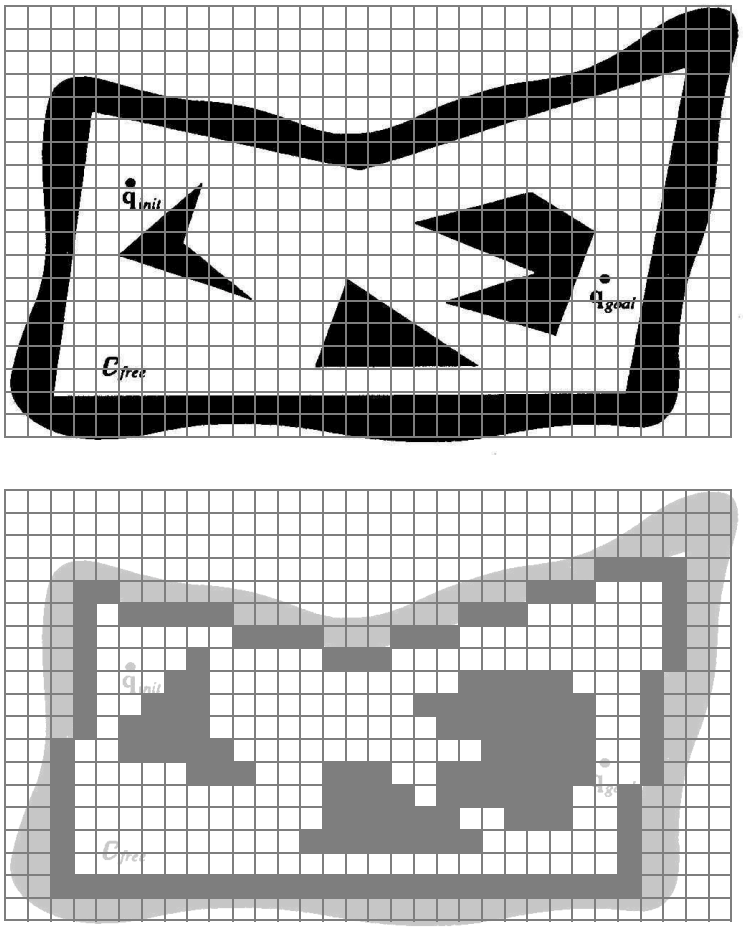

Fig 6.5 Fixed decomposition of the same space. (narrow passage disappears)

objects in the environment, just as with road-map based systems. The key advantage is a result of this same correllation. In environments that are extremely sparse, the number of cells will be small, even if the geometric size of the environment is very large. Thus the representation will be efficient in the case of large, sparse environments. Practically speaking, due to complexities in implementation, the exact cell decomposition technique is used relatively rarely in mobile robot applications.

Approximate Cell Decomposition

By contract, approximate cell decomposition is one of the most popular techniques for mobile robot path planning. This is partly due to the popularity of grid-based environmental representations. These grid-based representations are themselves fixed gridsize decompositions and so they are identical to an approximate cell decomposition of the environment.

The most popular form of this, shown in Figure 6.5, is the fixed-size cell decomposition. The cell size is not dependent on the particular objects in an environment at all, and so narrow passageways can be lost due to the inexact nature of the tessellation. Practically speaking, this is rarely a problem due to the very small cell size used (e.g. 5 cm on each side). The great benefit of fixed-size cell decomposition is the low computational complexity of path planning.

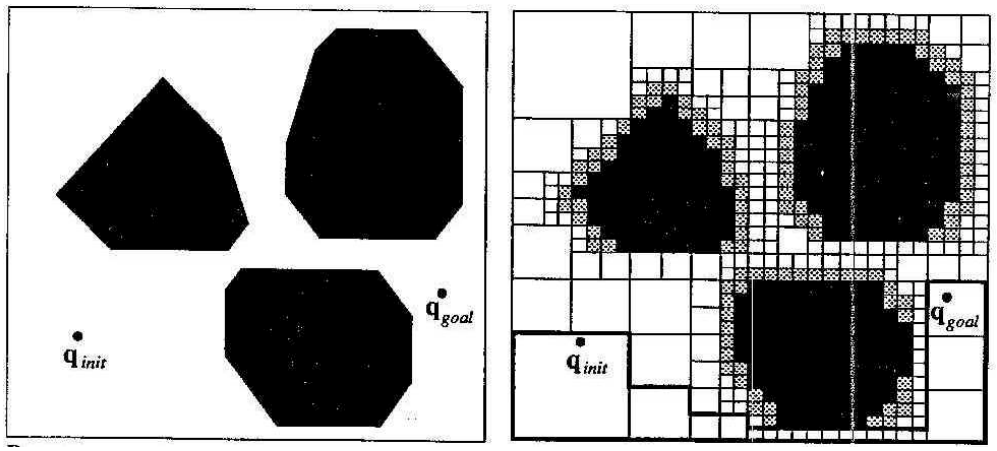

Fig 6.6 *Approximate variable-cell decomposition (adopted from [5] p. 18):*
a) The free space is externally bounded by a rectangle and internally bounded by three polygons.
b) The rectangle is decomposed into four identical rectangles. If the interior of a rectangle lies completely in free space or in the configuration space obstacle, it is not decomposed further. Otherwise, it is recursively decomposed into four rectangles until some predefined resolution is attained. The white cells lie outside the obstacles, the black inside and the grey are part of both regions.

For example, the *grassfire* transform is an efficient and simple-to-implement technique for finding routes in such fixed-size cell arrays. The algorithm simply employs wavefront expansion from the goal position outward, marking for each cell its distance to the goal cell [125]. This process continues until the cell corresponding to the initial robot position is reached. At this point, the path planner can estimate the robot's distance to the goal position as well as recovering a specific solution trajectory by simply linking together cells that are adjacent and always closer to the goal.

Given that the entire array can be in memory, each cell is only visited once when looking for the shortest discrete path from the initial position to the goal position. So, the search is linear in the number of cells only. Thus complexity does not depend on the sparseness and density of the environment, nor on the complexity of the objects' shapes in the environment. Formally, this grassfire transform is simply Breadth-First Search implemented in the constrainted space of an adjacency array. For more information on Breadth-First Search and other graph search techniques, refer to [16].

The fundamental cost of the fixed decomposition approach is memory. For a large environment, even when sparse, this grid must be represented in its entirety. Practically, due to the falling cost of computer memory, this disadvantage has been mitigated in recent years. The Cye robot is an example of a commercially available robot that performs all its path planning on a two-dimensional 2 cm fixed-cell decomposition of the environment using a sophisticated grassfire algorithm that avoids known obstacles and prefers known routes [121].

Figure 6.6 illustrates a variable-size approximate cell decomposition method. The free space is externally bounded by a rectangle and internally bounded by three polygons. The rectangle is recursively decomposed into smaller rectangles. Each decomposition generates four

identical new rectangles. At each level of resolution only the cells whose interiors lie entirely in the free space are used to construct the connectivity graph. Path planning in such adaptive representations can proceed in a hierarchical fashion. Starting with a coarse resolution, the resolution is reduced until either the path planner identifies a solution or a limit resolution is attained (e.g. k • size of robot). In contrast to the exact cell decomposition method, the approximate approach does not guarantee completeness, but it is mathematically less involving and thus easier to implement. In contrast to the fixed-size cell decomposition, variable-size cell decomposition will adapt to the complexity of the environment, and therefore sparse environments will contain appropriately fewer cells, consuming dramatically less memory.

6.2.1.3 Potential Field Path Planning [see 5]

Potential field path planning creates a field, or gradient, across the robot's map that directs the robot to the goal position from multiple prior positions. This approach was originally invented for robot manipulator path planning and is used often and under many variants in the mobile robotics community. The potential field method treats the robot as a point under the influence of an artificial potential field $U(q)$. The robot moves by following the field, just as a ball would roll downhill. The goal (a minimum in this space) acts as an attractive force on the robot and the obstacles act as peaks, or repulsive forces. The superposition of all forces is applied to the robot, which, in most cases, is assumed to be a point in the configuration space (fig. 6.17). Such an artificial potential field smoothly guides the robot towards the goal while simultaneously avoiding known obstacles.

It is important to note, though, that this is more than just path planning. The resulting field is also a control law for the robot. Assuming the robot can localize its position with respect to the map and the potential field, it can always determine its next required action based on the field.

The basic idea behind all potential field approaches is that the robot is attracted toward the goal, while being repulsed by the obstacle that are known in advance. If new obstacles appear during robot motion, one could update the potential field in order to integrate this new information. In the simplest case, we assume that the robot is a point, thus the robot's orientation θ is neglected and the resulting potential field is only two dimensional (x, y). If we assume a differentiable potential field function $U(q)$, we can find the related artificial force $F(q)$ acting at the position $= (x, y)$.

$$F(q) = -\nabla U(q) \tag{6.1}$$

where $\nabla U(q)$ denotes the gradient vector of U at position q.

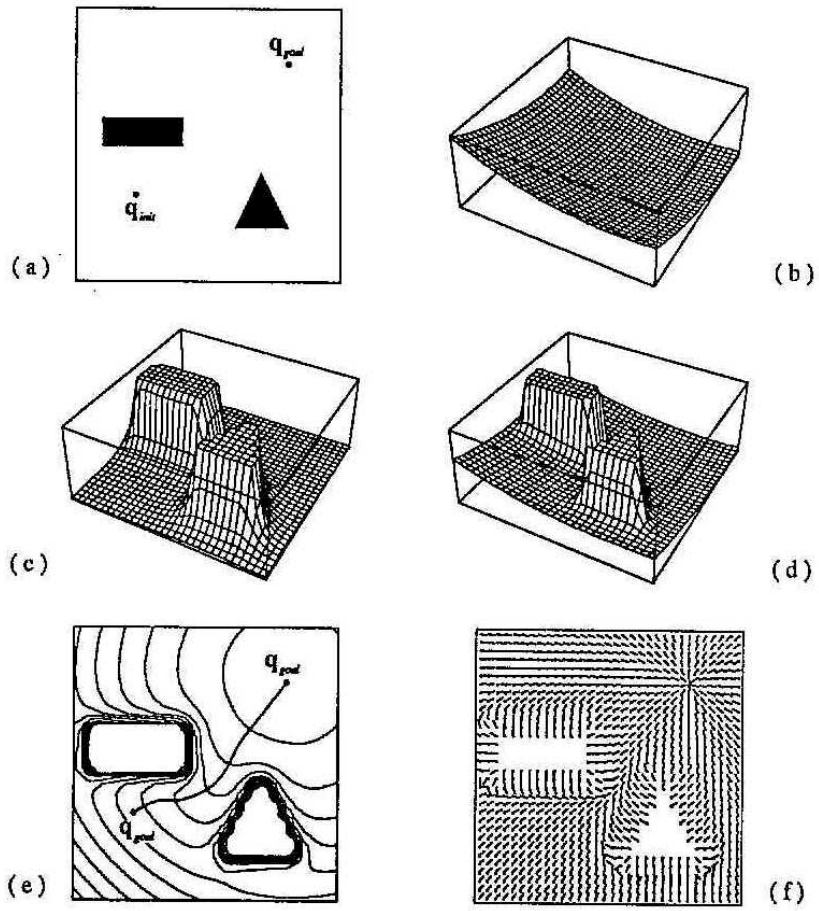

Fig 6.7 *Typical potential field generated by the attracting goal and two obstacles (source [5]).*
a configuration of the obstacles, start and goal
b attracting field due to the goal attractor
c repulsing field generated by the obstacles
d resulting potential field
e equipotential plot and path generated by the field
f gradient vector orientations

$$\nabla U = \begin{bmatrix} \frac{\partial U}{\partial x} \\ \frac{\partial U}{\partial y} \end{bmatrix} \tag{6.2}$$

The potential field acting on the robot is then computed as the sum of the attractive field of the goal and the repulsive fields of the obstacles:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{6.3}$$

Similarly, the forces can also be separated in a attracting and repulsing part:

$$\begin{aligned} F(q) &= F_{att}(q) - F_{rep}(q) \\ &= -\nabla U_{att}(q) - \nabla U_{rep}(q) \end{aligned} \tag{6.4}$$

Attractive Potential

An attractive potential can for example be defined as a parabolic function.

$$_{att}(q) = \frac{1}{2} k_{att} \cdot \rho_{goal}^2(q) \tag{6.5}$$

where k_{att} is a positive scaling factor and $\rho_{goal}(q)$ denotes the Euclidean distance $\|q - q_{goal}\|$. This attractive potential is differentiable, leading to the attractive force F_{att}

$$F_{att}(q) = -\nabla U_{att}(q) \tag{6.6}$$

$$= -k_{att} \cdot \rho_{goal}(q) \nabla \rho_{goal}(q) \tag{6.7}$$

$$= -k_{att} \cdot (q - q_{goal}) \tag{6.8}$$

that converges linearly towards 0 as the robot reaches the goal.

Repulsive Potential

The idea behind the repulsive potential is to generate a force away from all known obstacles. This repulsive potential should be very strong when the robot is close to the object, but should not influence its movement when the robot is far from the object. One example of such a repulsive field is:

$$_{rep}(q) = \begin{cases} \frac{1}{2} k_{rep} \left(\frac{1}{\rho(q)} - \frac{1}{\rho_0} \right)^2 & \text{if } \rho(q) \leq \rho_0 \\ 0 & \text{if } \rho(q) \geq \rho_0 \end{cases} \tag{6.9}$$

where k_{rep} is again a scaling factor, $\rho(q)$ is the *minimal distance* from q to the object and ρ_0 the *distance of influence* of the object. The repulsive potential function U_{rep} is positive or zero and tends to infinity as q gets closer to the object.

If the object boundary is convex and piecewise differentiable, $\rho(q)$ is differentiable everywhere in the free configuration space. This leads to the repulsive force F_{rep}:

$$_{rep}(q) = -\nabla U_{rep}(q) \tag{6.10}$$

$$= \begin{cases} k_{rep}\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)\frac{1}{\rho^2(q)}\frac{q - q_{obstacle}}{\rho(q)} & \text{if } \rho(q) \leq \rho_0 \\ 0 & \text{if } \rho(q) \geq \rho_0 \end{cases}$$

The resulting force $F(q) = F_{att}(q) + F_{rep}(q)$ acting on a point robot exposed to the attractive and repulsive forces moves the robot away from the obstacles and toward the goal (fig. 6.7). Under ideal conditions, by setting the robot's velocity vector proportional to the field force vector, the robot can be smoothly guided towards the goal, similar to a ball rolling around obstacles and down a hill.

However, there are some limitation with this approach. One are local minima that appear dependent of the obstacle shape and size. Another problem might appear if the objects are concave. This might lead to a situation for which several minimal distances $\rho(q)$ exist, result in oscillation between the two closest points to the object. For more detailed analyses of potential field characteristics refer to [5].

The Extended Potential Field Method

Khatib and Chatila proposed the *Extended Potential Field* approach [69]. Like all potential field methods this approach makes use of attractive and repulsive forces that originate from an artificial potential field. However, two additions to the basic potential field are made: the *rotation potential field* and the *task potential field*.

The rotation potential field assumes that the repulsive force is a function of the distance from the obstacle and the orientation of the robot relative to the obstacle. This is done using a gain factor which reduces the repulsive force when an obstacle is parallel to the robot's direction of travel, since such an object does not pose an immediate threat to the robot's trajectory. The result is enhanced wall following, which was problematic for earlier implementations of potential fields methods.

The task potential field considers the present robot velocity and from that it filters out those obstacles that should not affect the near-term potential based on robot velocity. Again a scaling is made, this time of all obstacle potentials when there are no obstacles in a sector named *Z* in front of the robot. The sector *Z* is defined as the space which the robot will sweep during its next movement. The result can be smoother trajectories through space. An example comparing a Classical Potential Field and an Extended Potential Field is depicted by Fig. 6.8.

A great deal of variations and improvements of the potential field methods have been proposed and implemented by mobile roboticists [67] [58]. In most cases, these variations aim to improve the behavior of potential fields in local minima while also lowering the chances

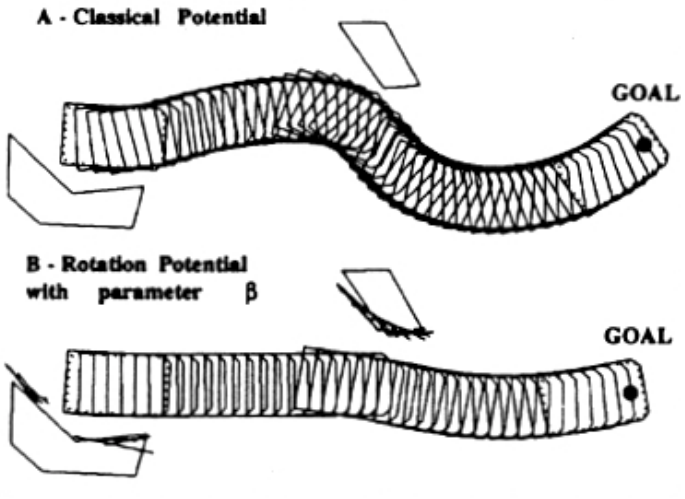

Fig 6.8 *Comparison between a Classical Potential Field and an Extended Potential Field, source [69].*

of oscillations and instability when a robot must move through a narrow space such as a doorway.

Potential fields are extremely easy to implement, much like the grassfire algorithm described in Section 6.2.1.2. Thus it has become a commonplace tool in mobile robot applications in spite of its theoretical limitations.

This completes our brief summary of path planning techniques that are most popular in mobile robotics. Of course, as the complexity of a robot increases (e.g. large DOF nonholonomics) and, particularly, as environment dynamics becomes more significant, then the path planning techniques described above become inadequate for grappling with the full problem scope. However, for robots moving in largely flat terrain, the mobility decision-making techniques roboticists use often fall under one of the above categories.

But a path planner can only take into consideration the environment obstacles that are known to the robot *in advance*. During path execution the robot's actual sensor values may disagree with expected values due to map inaccuracy or a dynamic environment. Therefore, it is critical that the robot modify its path in real time based on actual sensor values. This is the competence of *obstacle avoidance* which we discuss below.

6.2.2 Obstacle Avoidance

Local obstacle avoidance focuses on changing the robot's trajectory as informed by its sensors during robot motion. The resulting robot motion is both a function of the robot's current or recent sensor readings *and* its goal position and relative location to the goal position. The obstacle avoidance algorithms presented below depend to varying degrees on the existence of a global map and on the robot's precise knowledge of its location relative to the map. Despite their differences, all of the algorithms below can be termed obstacle avoidance algorithms because the robot's local sensor readings play an important role in the robot's future trajectory. We first present the simplest obstacle avoidance systems that are used successfully in mobile robotics. The Bug Algorithm represents such a technique in that only the most recent robot sensor values are used, and the robot needs in addition to current sensor

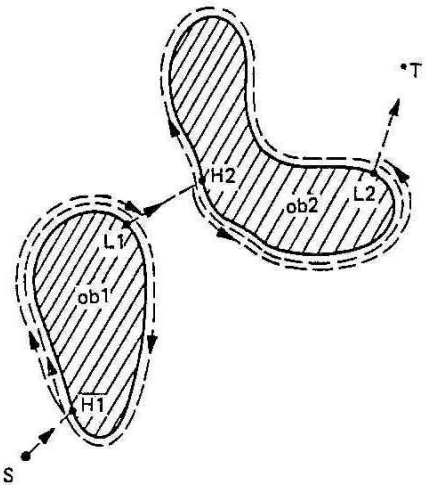

Fig 6.9 ***Bug1*** *algorithm with H1, H2, hit points and L1, L2, leave points [51].*

values only approximate information regarding the direction of the goal. More sophisticated algorithms are presented afterwards, taking into account recent sensor history, robot kinematics and even dynamics.

6.2.2.1 Bug Algorithm [51, 52]

The Bug algorithm is perhaps the simplest obstacle avoidance algorithm one could imagine. The basic idea is to follow the contour of each obstacle in the robot's way and thus circumnavigate it.

With Bug1, the robot fully circles the object first, then departs from the point with the shortest distance toward the goal (fig. 6.9). This approach is of course very inefficient but guarantees that the robot will reach any reachable goal.

With Bug2 the robot begins to follow the object's contour, but departs immediately when it is able to move directly toward the goal. In general this improved Bug algorithm will have significantly shorter total robot travel as shown in path (fig. 6.10a). However, one can still construct situations in which Bug2 is arbitrarily inefficient (*i.e.* nonoptimal).

Practical Application: example of Bug2

Because of the popularity and simplicity of Bug2, we will present a specific example of obstacle avoidance using a variation of this technique. Consider the path taken by the robot in Figure 6.10a. One can characterize the robot's motion in terms of two states, one that involves moving toward the goal and a second that involves moving around the contour of an obstacle. We will call the former state goalSeek and the latter wallFollow. If we can describe the motion of the robot as a function of its sensor values and the relative direction to the goal for each of these two states, and if we can describe when the robot should switch between them, then we will have a practical implementation of Bug2. The following pseudocode provides the highest-level control code for such a decomposition:

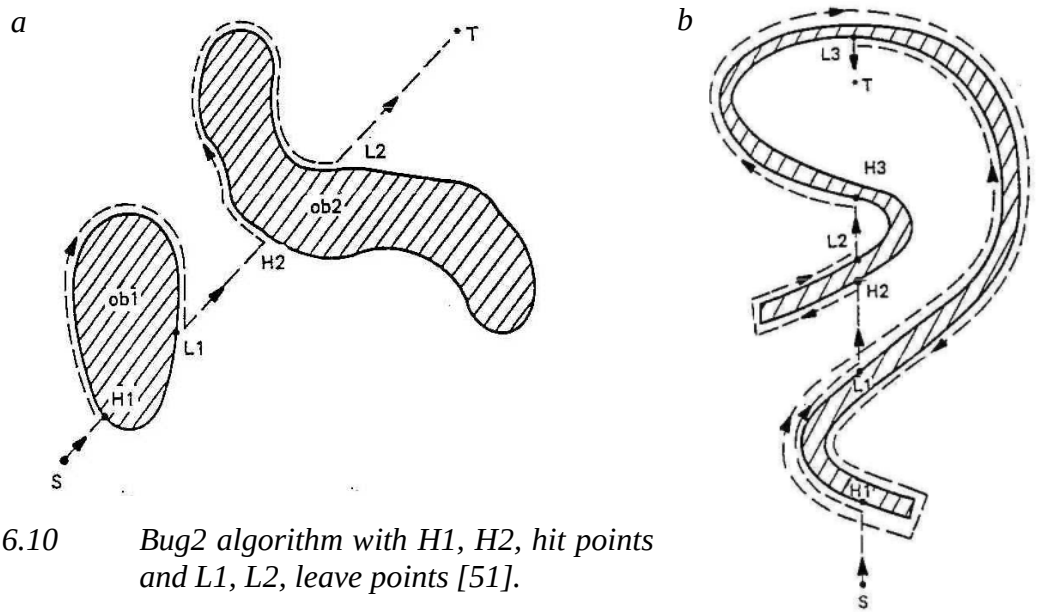

Fig 6.10 *Bug2 algorithm with H1, H2, hit points and L1, L2, leave points [51].*

```
public void bug2(position goalPos) {
        position robotPos = robot.GetPos(&sonars);
        int forwardVel, rotationVel;
        int goalDist = getDistance(robotPos,goalPos);
        angle goalAngle = Math.atan2(goalPos, robotPos);
        goalAngle = goalAngle - robot.GetAngle();

        if (goalDist < atGoalThreshold) {
                System.out.println("At goal!");
                robot.SetVelocity(0,0);
                robot.SetState(DONE);
                exit();
        }
        forwardVel = ComputeTranslation(sonars);
        if (robot.GetState() == GOALSEEK) {
                rotationVel = ComputeGoalSeekRot(goalAngle);
                if (ObstaclesInWay(goalAngle, sonars)) robot.SetState(WALLFOLLOW);
        }
        if (robot.GetState() == WALLFOLLOW) {
                rotationVel = ComputeRWFRot(sonars);
                if (!ObstaclesInWay(goalAngle, sonars)) robot.SetState(GOALSEEK);
        }
        robot.SetVelocity(forwardVel, rotationVel);
        bug2(goalPos);
} // end bug2() //
```

In the ideal case, when encountering an obstacle one would choose between left wall following and right wall following depending on which direction is more promising. In this simple example we have only right wall following, a simplification for didactic purposes that ought not find its way into a real mobile robot program.

Now we consider specifying each remaining function in detail. Consider for our purposes a robot with a ring of sonars placed radially around the robot. This imagined robot will be differential-drive, so that the sonar ring has a clear "front" (aligned with the forward direction of the robot). Furthermore, the robot accepts motion commands of the form shown above, with a rotational velocity parameter and a translational velocity parameter. Mapping these two parameters to individual wheel speeds for each of the two differential drive chassis' drive wheels is a simple matter.

There is one condition we must define in terms of the robot's sonar readings, `obstaclesInWay()`. We define this function to be true whenever any sonar range reading in the direction of the goal (within 45 degrees of the goal direction) is short:

```
private boolean ObstaclesInWay(angle goalAngle, sensorvals sonars) {
        int minSonarValue;
        minSonarValue = MinRange(sonars, goalAngle - (pi/4), goalAngle + (pi/4));
        return (minSonarValue < 200);
} // end ObstaclesInWay() //
```

Note that the function `ComputeTranslation()` computes translational speed whether the robot is wall following or heading toward the goal. In this simplified example, we define translation speed as being proportional to the largest range readings in the robot's approximate forward direction:

```
private int ComputeTranslation(sensorvals sonars) {
        int minSonarFront;
        minSonarFront = MinRange(sonars, -pi/4, pi/4);
        if (minSonarFront < 200) return 0;
        else return (Math.min(500, minSonarFront - 200));
} // end ComputeTranslation() //
```

Note that there is a marked similarity between this approach and the potential field approach described in Section 6.2.1.3. Indeed, some mobile robots implement obstacle avoidance by treating the current range readings of the robot as force vectors, simply carrying out vector addition to determing the direction of travel and speed. Alternatively, many will consider short range readings to be repulsive forces, again engaging in vector addition to determine an overall motion command for the robot.

When faced with range sensor data, a popular way of determining rotation direction and speed is to simply subtract left and right range readings of the robot. The larger the difference, the fast the robot will turn in the direction of the longer range readings. The following two rotation functions could be used for our Bug2 implementation:

```
private int ComputeGoalSeekRot(angle goalAngle) {
        if (Math.abs(goalAngle) < pi/10) return 0;
        else return (goalAngle * 100);
} // end ComputeGoalSeekRot() //

private int ComputeRWFRot(sensorvals sonars) {
        int minLeft, minRight, desiredTurn;
        minRight = MinRange(sonars, -pi/2, 0);
        minLeft = MinRange(sonars, 0, pi/2);
        if (Math.max(minRight,minLeft) < 200) return (400); // hard left turn
        else {
                desiredTurn = (400 - minRight) * 2;
                desiredTurn = Math.inttorange(-400, desiredTurn, 400);
                return desiredTurn;
        } // end else
} // end ComputeRWFRot() //
```

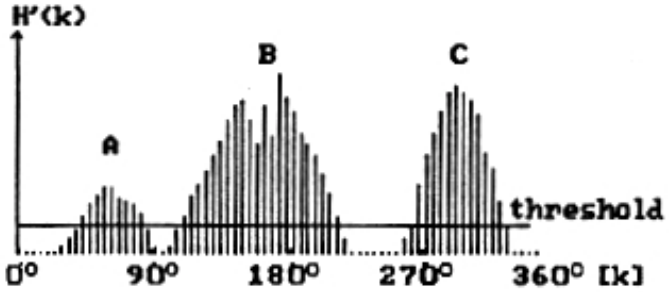

Fig 6.11 Polar histogram, source [62].

Note that the rotation function for the case of right wall following combines a general avoidance of obstacles with a bias to turn right when there is open space on the right, thereby staying close to the obstacle's contour. This solution is certainly not the best solution for implementation of Bug2. For example, the wall follower could do a far better job by mapping the contour locally and using a PID control loop to achieve and maintain a specific distance from the contour during the right wall following action.

Although such simple obstacle avoidance algorithms are often used in simple mobile robots, they have numerous shortcomings. For example, the Bug2 approach does not take into account robot kinematics, which can be especially important with non-holonomic robots. Furthermore, since only the most recent sensor values are used, sensor noise can have a serious impact on real-world performance. The following obstacle avoidance techniques are designed to overcome one or more of these limitations.

6.2.2.2 Vector Field Histogram

Borenstein together with Koren developed the Vector Field Histogram (VFH) [53]. Their previous work which was concentrated on potential fields [61] was abandoned due to the method's instability and inability to pass through narrow passages. Later Borenstein, together with Ulrich, extended the VFH algorithm to yield VFH+ [54] and VFH*[Roland, reference the VFH* reference I describe in my text to you here..].

One of the central criticisms of Bug-type algorithms is that the robot's behavior at each instant is generally a function of only its most recent sensor readings. This can lead to undesirable and yet preventable problems in cases where the robot's instantaneous sensors readings do not provide enough information for robust obstacle avoidance. The VFH techniques overcome this limitation by creating a local map of the environment around the robot. This local map is a small occupancy grid, as described in Section (5.7), populated only by relatively recent sensor range readings. For obstacle avoidance, VFH generates a polar histogram as shown in figure 6.11 . The x-axis represents the angle at which the obstacle was found and the y-axis represents the probability that there really is an obstacle in that direction based on the occupancy grid's cell values.

From this histogram a steering direction is calculated. First all openings large enough for the vehicle to pass through are identified. Then a cost function is applied to every such candidate opening. The passage with the lowest cost is chosen. The cost function G has three

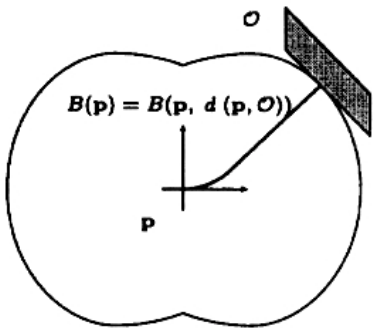

Fig 6.13 Shape of the bubbles around the vehicle, source [56].

terms:

$$= a \cdot \text{target\_direction} + b \cdot \text{wheel\_orientation} + c \cdot \text{previous\_directio} \qquad (6.11)$$

target_direction: Alignment of the robot path with the goal.

wheel_orientation: Difference between the new direction and the current wheel orientation.

previous_direction: Difference between the previously selected direction and the new direction.

The terms are calculated such that a large deviation from the goal direction leads to a big cost in the term “target direction”. The parameters *a, b, c* in the cost function *G* tune the behavior of the robot. For instance a strong goal bias would be expressed with a large value for *a*. For a complete definition of the cost function refer to [61].

In the VFH+ improvement one of the reduction stages takes into account a simplified model of the moving robot’s possible trajectories based on its kinematic limitations (*e.g.* turning radius for an Ackerman vehicle). The robot is modelled to move in arcs or straight lines. An obstacle thus blocks all of the robot’s allowable trajectories which pass through the obstacle (fig. 6.12a). This results in a masked polar histogram where obstacles are enlarged so that all kinematically blocked trajectories are properly taken into account (fig. 6.12c).

6.2.2.3 The Bubble Band Technique

This idea is an extension for nonholonomic vehicles for the Elastic Band Concept suggested by Quinlan and Khathib [56]. The original Elastic Band Concept applied only to holonomic vehicles and so we focus only on the Bubble Band extension made by Khathib, Jaouni, Chatila and Laumod [57].

A bubble is defined as the maximum free-space around a robot that can be guaranteed during execution of its trajectory. The bubble is generated using a simplified model of the robot in conjunction with range information available in the robot’s map. Even with a simplified model of the robot’s geometry, it is possible to take into account the actual shape of the robot when calculating the bubbles size (6.13). Given such bubbles, a band, or string of bubbles

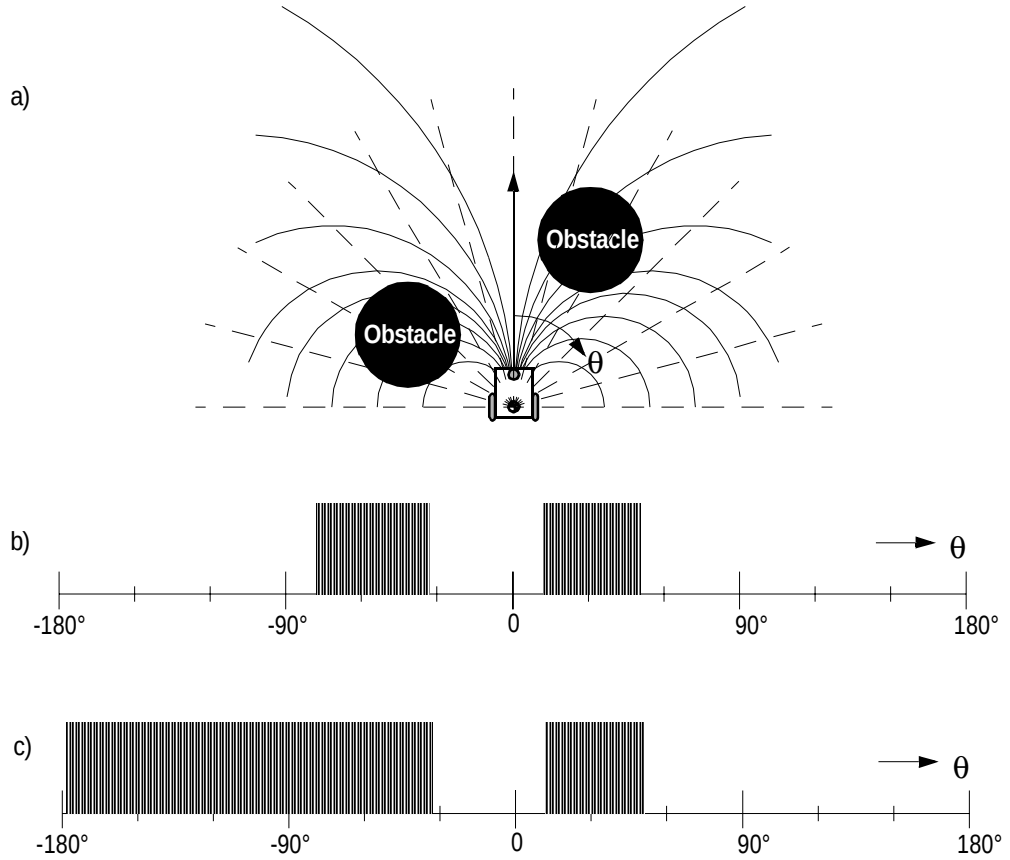

Fig 6.12 *Example of blocked directions and resulting polar histograms [54].*
a) robot and blocking obstacles
b) Polar histogram, source
b) Masked polar histogram, source.

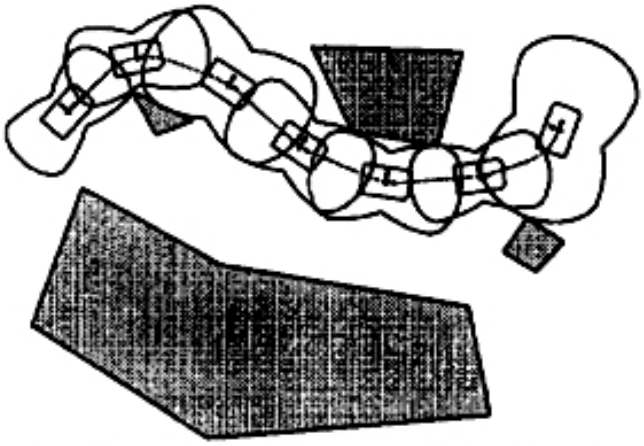

Fig 6.14 *A typical bubble band, source [56].*

can be used along the trajectory from the robot's initial position to its goal position to show the robot's expected free space throughout its path (see Fig. 6.14).

Clearly, computing the bubble band requires a global map and a global path planner. Once

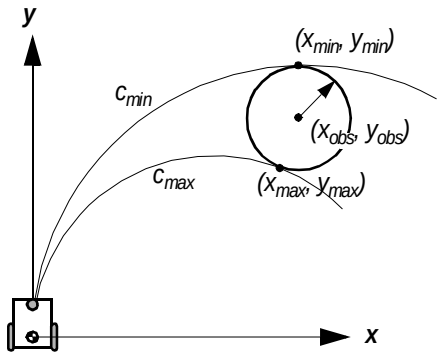

Fig 6.15 Tangent curvatures for an obstacle, source [66].

the path planner's initial trajectory has been computed and the bubble band is calculated, then modification of the planned trajectory ensues. The bubble band is takes into account forces from objects in the model and internal forces. These internal forces try to minimize the "*slack*" (energy) between adjacent bubbles. This process, plus a final smoothing operation, makes the trajectory smooth in the sense that the robot's freespace will change as smoothly as possible during path execution.

Of course, so far this is more akin to path planning optimization than obstacle avoidance. The obstacle avoidance aspect of the bubble band strategy comes into play during robot motion. As the robot encounters unforeseen sensor values, the bubble band model is used to deflect the robot from its originally intended path in a way that minimized bubble band *tension.*

An advantage of the bubble band technique is that one can account for the actual dimensions of the robot. However, the method is most applicable only when the environment configuration is well-known ahead of time, just as with off-line path planning techniques.

6.2.2.4 Curvature Velocity Techniques

The Basic Curvature Velocity Approach

The *Curvature Velocity Approach* (*CVM)* from Simmons [66] enables the actual kinematic constraints and even some dynamic constraints of the robot to be taken into account during obstacle avoidance, which is an improvement over more primitive techniques. CVM begins by adding physical constraints from the robot and the environment to a velocity space. The velocity space consists of rotational velocity ω and translational velocity v, thus assuming that the robot only travels along arcs of circles with curvature $= \omega / v$.

Two types of constraints are identified: those derived from the robot's limitations in acceleration and speed, typically: $-v_{max} < v < v_{max}$, $-\omega_{max} < \omega < \omega_{max}$; and, second, the constraints from obstacles blocking certain v and ω values due to their positions. The obstacles begin as objects in a Cartesian grid but are then transformed to the velocity space by calculating the distance from the robot position to the obstacle following some constant curvature robot trajectory, as shown in figure 6.15. Only the curvatures that lie within c_{min} and c_{max}

are considered since that curvature space will contain all legal trajectories.

To achieve real-time performance the obstacles are approximated by circular objects and the contours of the objects are divided into few intervals. The distance from an endpoint of an interval to the robot is calculated and in between the endpoints the distance function is assumed to be constant.

The final decision of a new velocity (v and ω) is made by an objective function. This function is only evaluated on that part of the velocity space that fulfills the kinematic and dynamic constraints as well as the constraints due to obstacles. The use of a Cartesian grid for initial obstacle representation enables straightforward sensor fusion if, for instance, a robot is equipped with multiple types of ranging sensors.

CVM takes into consideration the dynamics of the vehicle in useful manner. However a limitation of the method is the circular simplification of obstacle shape. In some environments this is acceptable while, in other environments, such a simplification can cause serious problems. The CVM method can also known to suffer from local minima since no *a priori* knowledge is used by the system.

The Lane Curvature Method

Simmons and Ko presented an improvement of the CVM which they named The Lane Curvature Method, LCM [55] based on their experiences with the shortcomings of CVM. CVM had difficulty guiding the robot through intersections of corridors. The problems stemmed from the approximation that the robot moves only along fixed arcs, whereas in practice the robot can change direction many times before reaching an obstacle.

LCM calculates a set of desired lanes, trading off lane length and lane width to the closest obstacle. The lane with the best properties is chosen using an objective function. The local heading is chosen in such way that the robot will transition to the best lane if it is not in that lane already.

Experimental results have demonstrated better performance as compared to CVM. One caveat is that the parameters in the objective function must be chosen carefully to optimize system behavior.

6.2.2.5 Dynamic Window Approaches

Another technique for taking into account robot kinematics is the Dynamic Window obstacle avoidance method. Two such approaches are represented in the literature. The Dynamic Window Approach [63] by Fox and Burgard and the Global Dynamic Window approach [64] by Brock and Khatib.

The Local Dynamic Window Approach

In the Local Dynamic Window Approach the kinematics of the robot is taken into account by searching a well-chosen velocity space. The velocity space is all possible sets of tuples (v, ω) where v is the velocity and ω is the angular velocity. The approach assumes that

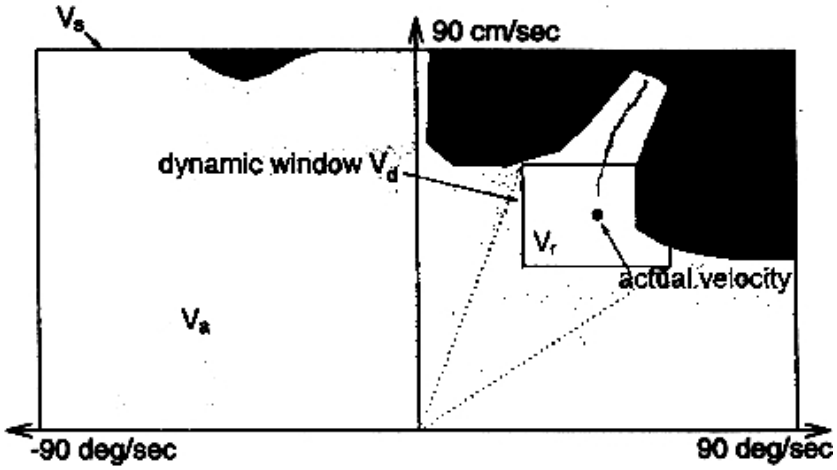

Fig 6.16 *The dynamic window approach, source [63]. The rectangular window shows the possible speeds (v, ω) and the overlap with obstacles in configuration space*

robots move only in circular arcs representing each such tuples.

Given the current robot speed the algorithm first selects only those tuples (v, ω) that will ensure that the vehicle can come to a stop before hitting an obstacle. These velocities are called *admissible* velocities. The next step is to further reduce the admissible velocities using a *dynamic window*. The dynamic window consists of all tuples (v, ω) that can be reached within the next sample period, given the current velocity and the acceleration capabilities of the robot. By doing this the robot dynamics is taken under consideration. In figure 6.16, a typical dynamic window is presented. Note that the shape of the dynamic window is rectangular. This follows from the approximation that the acceleration capabilities for translation and rotation are independent.

A new motion direction is chosen by applying an objective function to all the admissible velocity tuples in the dynamic window. The velocity tuple which maximizes the length of the trajectory until an obstacle is reached, and the alignment of the robot with the goal direction, is chosen. The objective function O has the form:

$$= a \cdot heading(v, \omega) + b \cdot velocity(v, \omega) + c \cdot dist(v, \quad (6.12)$$

heading: Measure of progress towards the goal location.

velocity: Forward velocity of the robot -> encouraging fast movements.

dist: Distance to the closest obstacle in the trajectory.

The Global Dynamic Window Approach

The Global Dynamic Window Approach adds, as the name suggests, global thinking to the algorithm presented above. This is done by adding a minima-free function named ***NF1*** to the objective function O presented above. NF1 is a function computed using a wave-propagation technique virtually identical to grassfire (see figure 6.17 and Section 6.2.1.2) devel-

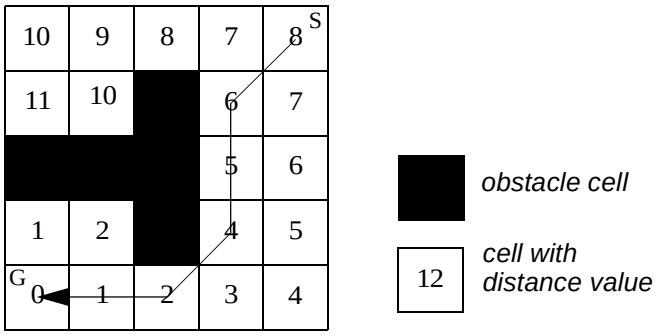

Fig 6.17 An example of the distance transform and the resulting path as it is generated by the NF1 function (S means start, G goal)

oped by Latombe [65]. It operates on a grid representing the environment, including known obstacles. NF1 labels the cells in the occupancy grid with the total distance L to the goal. To make this faster the Global Dynamic Window Approach calculates the NF1 only on a selected rectangular region which is directed from the robot towards the goal. The width of the region is enlarged and recalculated if the goal cannot be reached within the constraints of this chosen region..

This allows The Global Dynamic Window Approach to achieve some of the advantages of global path planning without complete *a priori* knowledge. The occupancy grid is updated from range measurements as the robot moves in the environment. The NF1 is calculated for every new updated version. If the NF1 cannot be calculated due to the fact that the robot is surrounded by obstacles the method degrades to the Dynamic Window Approach. This keeps the robot moving so that a possible way out may be found and NF1 can resume.

The Global Dynamic Window Approach promises real-time, dynamic constraints, global thinking and minima free obstacle avoidance at high speed. An implementation has been demonstrated with an omnidirectional robot using a 450 MHz onboard PC. This system produced a cycle frequency of about 15 Hz when the occupancy grid was 30×30 m with a 5 cm resolution. Average robot speed in the tests was greater than 1 m/s.

6.2.2.6 The Schlegel Approach to Obstacle Avoidance

Schlegel [68] presents an approach that considers the dynamics as well as the actual shape of the robot. The approach is adopted for raw laser data measurements and sensor fusion using a Cartesian grid to represent the obstacles in the environment. Real-time performance is achieved by the use of precalculated lookup tables.

As with previous methods we have described, the basic assumption is that a robot moves in trajectories built up by circular arcs, defined as curvatures i_c. Given a certain curvature i_c Schlegel calculates the distance l_i to collision between a single obstacle point $[x, y]$ in the Cartesian grid and the robot, depicted in Fig. 6.18. Since the robot is allowed to be any shape this calculation is time consuming and the result is therefore precalculated and stored in a lookup table.

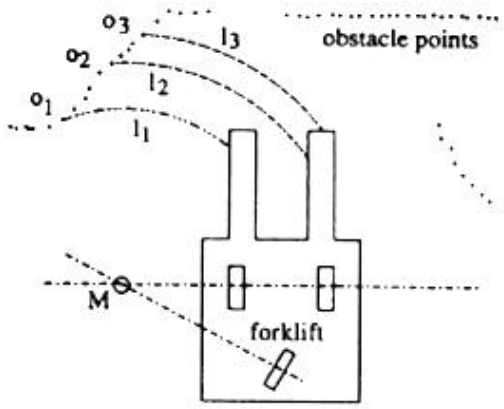

Fig 6.18 *Distances l_i resulting from the curvature i_c when the robot rotates around M (source [68]).*

For example, the search space window V_s is defined for a differential drive robot to be all the possible speeds of the left and right wheel, $_r$, v_l. The dynamic constraints of the robot are taken into account by refining V_s to only those values which are reachable within the next time-step, given the present robot motion. Finally an objective function chooses the best speed and direction by trading off goal direction, speed, and distance until collision.

During testing Schlegel used a wavefront path planner. Two robot chassis were used, one with synchro-drive kinematics and one with tricycle kinematics. The tricycle drive robot is of particular interest because it was a forklift with a complex shape that had a significant impact on obstacle avoidance. Thus the demonstration of reliable obstacle avoidance with the forklift is an impressive result. Of course a disadvantage of this approach is the potential memory requirements for the lookup table. In their experiments, the authors used lookup tables of up to 2.5 Mb using a 30×30 m Cartesian grid with a resolution of 10 cm.

6.2.2.7 Other Approaches

The approaches described above are some of the most popularly referenced obstacle avoidance systems. There are, however, a great many additional obstacle avoidance techniques in the mobile robotics community. For example Tzafestas and Tzafestas [70] provide an overview of fuzzy and neurofuzzy approaches to obstacle avoidance. Inspired by nature, Chen and Quinn [59] present a biological approach in which they replicate the neural network of a cockroach. The network is then applied to a model of a four wheeled vehicle.

The Liapunov functions form a well known theory that can be used to prove stability for nonlinear systems. In the paper of Vanualailai, Nakagiri and Ha [60] the Liapunov functions are used to implement a control strategy for two point masses moving in a known environment. All obstacles are defined as anti-targets with an exact position and a circular shape. The anti-targets are then used when building up the control laws for the system. However, this complex mathematical model has not be tested on a real-world robot to our knowledge.

Table 6.1: Overview of the most popular obstacle avoidance algorithms

Method	Implementation with LRS[a]	Other requisites	Dynamics and actual shape considered	Global/ Local view	Known problems	Robot tests	*A Priori* knowledge required	Real-Time perform.
VFH+ [61]	Could easily be done	Histogram grid.	Very Simple dynamic approximation	Local	Dead-Ends	Non holonomic GuideCane	No	6 ms cycle time, 66 MHz 486 PC
The Bubble Band Approach [56]	?	Global Map, Path planner	Shape yes	Global	?	Simulations only	Yes, Model of the environment.	?
The Global Dynamic Window Approach [64]	Yes	NF1 algorithm, Histogram grim updated with latest range information	Dynamics yes Shape No	Global	Strange behavior in narrow passages. Can be fixed	Holonomic	No	66 msec cycle time, 450 MHz PC
The Curvature Velocity Method [66]	Yes	Histogram grid updated with latest range information	Dynamics yes Shape No	Local	Local minima, circular trajectories resulting in strange behavior	Non holonomic, Synchro-drive	No	12 msec cycle time, 66 MHz 486 PC
The Lane Curvature Method [67]	Yes	?	Dynamics yes Shape No	Local	Local minima	Non holonomic, Synchro-drive	No	8 Hz, 200 MHz Pentium Pro
The Extended Potential Field [69]	?	Environmental model with obstacles	Some dynamic	Global	Local Minima, wall following and door passing has been improved	?	Yes, Knowledge about the obstacles required	?
The Schlegel Approach [68]	Yes, and map	Histogram grid updated with latest range information	Dynamics yes Shape Yes	Local thinking of algorithm. Global of entire system	Possibly susceptible to local minima	Synchro-Drive, Tricycle	No, unless map is to be used.	125 ms,? MHz 2,5 Mb lookup table
Harmonic Potentials [58]	?	Environmental model with obstacles	No	Global	?	Simulations only	Yes, complete knowledge of the shape of the obstacles required	?

a.

6.3 Navigation Architectures

Given techniques for path planning, obstacle avoidance, localization and perceptual interpretation, how do we combine all of these into one complete robot system for a real-world application? One way to proceed would be to custom-design an application-specific, monolithic software system that implements everything for a specific purpose. This may be efficient in the case of a trivial mobile robot application with few features and even fewer planned demonstrations. But for any sophisticated and long-term mobile robot system, the issue of mobility architecture should be addressed in a principled manner. The study of *navigation architectures* is the study of principled designs for the software modules that comprise a mobile robot navigation system. Using a well designed navigation architecture has a number of concrete advantages:

Modularity for code reuse and sharing. Basic software engineering principles embrace software modularity, and the same general motivations apply equally to mobile robot applications. But modularity is of even greater importance in mobile robotics because in the course of a single project the mobile robot hardware or its physical environmental characteristics can change dramatically, a challenge most traditional computers do not face. For example one may introduce a Sick laser rangefinder to a robot that previously used only ultrasonic rangefinders. Or one may test an existing navigator robot in a new environment, where there are obstacles that its sensors cannot detect, thereby demanding a new path planning representation.

We would like to change part of the robot's competency without causing a string of side effects that force us to revisit the functioning of other robot competencies. For instance we would like to retain the obstacle avoidance module intact, even as the particular ranging sensor suite changes. In a more extreme example, it would be ideal if the non holonomic obstacle avoidance module could remain untouched even when the robot's kinematic structure changes from a tricycle chassis to a differential drive chassis.

Control localization. Localization of robot control is an even more critical issue in mobile robot navigation. The basic reason is that a robot architecture includes multiple types of control functionality (e.g. obstacle avoidance, path planning, path execution, etc). By localizing each functionality to a specific unit in the architecture, we enable individual testing as well as a principled strategy for control composition. For example consider collision avoidance. For stability in the face of changing robot software as well as for focused verification that the obstacle avoidance system is correctly implemented, it is valuable to localize all software related to the robot's obstacle avoidance process. At the other extreme, high level planning and task-based decision-making is required for robots to perform useful roles in their environment. It is also valuable to localize such high-level decision-making software, enabling it to be tested exhaustively in simulation and thus verified even without a direct connection to the physical robot. A final advantage of localization is associated with learning. Localization of control can enable a specific learning algorithm to be applied to just one aspect of a mobile robot's overall control system. Such targeted learning is likely to be the first strategy that yields successful integration of learning and traditional mobile robotics.

The advantages of localization and modularity prove a compelling case for the use of prin-

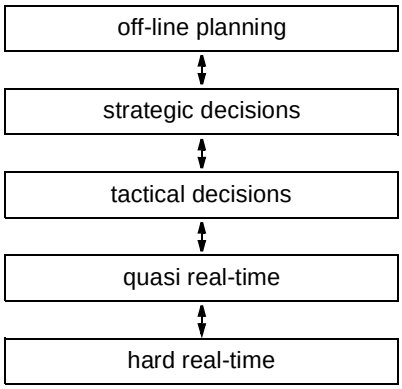

Fig 6.19 Generic temporal decomposition of a navigation architecture.

cipled navigation architectures. One way to characterize a particular architecture is by its decomposition of the robot's software. We can classify decompositions along two axes: temporal decomposition and control decomposition. In the next section, we define these two types of decomposition, then present an introduction to *behaviors*, which are a general tool for implementing control decomposition. Then in Section (6.3.2) we present three types of navigation architectures, describing for each architecture an implemented mobile robot case study.

6.3.1 Techniques for Decomposition

Decompositions identify axes along which we can justify discrimination of robot software into distinct modules. Decompositions also serve as a way to classify various mobile robots into a more quantitative taxonomy. *Temporal decomposition* distinguishes between real-time and non-real-time demands on mobile robot operation. *Control decomposition* identifies the way in which various control outputs within the mobile robot architecture combine to yield the mobile robot's physical actions. Below we describe each type of decomposition in greater detail.

6.3.1.1 Temporal decomposition

A temporal decomposition of robot software distinguishes between processes that have varying real-time and non-real-time demands. Figure 6.19 depicts a generic temporal decomposition for navigation. In this figure, the most real-time processes are shown at the bottom of the *stack*, with the highest category being occupied by processes with no real-time demands.

The lowest level in this example captures functionality that must proceed with a guaranteed fast cycle time, such as a 40 Hz bandwidth. In contrast, a quasi-real-time layer may capture processes that require, on average, 0.1 seconds response time, with large allowable worst-case inidividual cycle times. A tactical layer can represent decision-making that affects the robot's immediate actions and is therefore subject to some temporal constraints, while a strategic or off-line layers represent decisions that affects the robot's behavior over the long term, with little or no real temporal constraints on the module's response time.

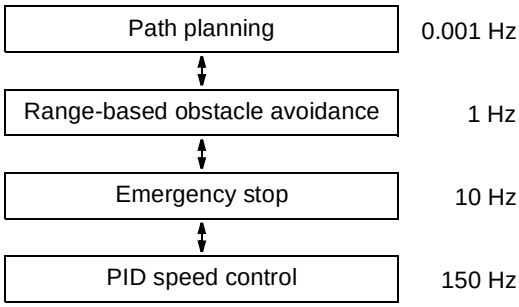

Fig 6.20 *Sample 4-level temporal decomposition of a simple navigating mobile robot. The second column next to the rectangles indicates realistic bandwidth values for each module.*

Four important, interrelated trends correlate with temporal decomposition. These are not set in stone; there are exceptions. Nevertheless, these general properties of temporal decompositions are useful:

Sensor response-time. A particular module's sensor response time can be defined as the amount of time between acquisition of a sensor-based event and a corresponding change in the output of the module. As one moves up the stack in Figure (6.19) the sensor response time tends to increase. For the lowest-level modules, the sensor response time is often limited only by the raw processor and sensor speeds. At the highest-level modules, sensor response can be limited by slow and deliberate decision-making processes.

Temporal depth. Temporal depth is a useful concept applying to the temporal window that affects the module's output, both backward and forward in time. *Temporal horizon* describes the amount of lookahead used by the module during the process of choosing an output. *Temporal memory* describes the historical time span of sensor input that is used by the module to determine the next output. Lowest-level modules tend to have very little temporal depth in both directions, whereas the deliberative processes of highest-level modules make use of a large temporal memory and consider actions based on their long-term consequences, making use of large temporal horizons.

Spatial locality. Hand in hand with temporal span, the spatial impact of layers increases dramatically as one moves from low-level modules to high-level modules. Real-time modules tend to control wheel speed and orientation, controlling spatially localized behavior. High-level strategic decision-making has little or not bearing on local position, but informs global position further into the future.

Context specificity. A module makes decisions as a function not only of its immediate inputs but also as a function of the robot's context as captured by other variables, such as the robot's representation of the environment. Lowest-level modules tends to produce outputs directly as a result of immediate sensor inputs, using little context and therefore being relatively context insensitive. Highest-level modules tend to exhibit very high context specificity. For strategic decision-making, given the same sensor values, dramatically different outputs are nevertheless conceivable depending on other contextual parameters.

An example demonstrating these trends is depicted in Figure 6.20, which shows a temporal

decomposition of a simplistic navigation architecture into four modules. At the lowest level PID control loop provides feedback to control motor speeds. An Emergency Stop module uses short-range optical sensors and bumpers to cut current to the motors when it predicts an imminent collision. Knowledge of robot dynamics means that this module by nature has a greater temporal horizon than the PID module. The next module uses longer-range laser rangefinding sensor returns to identify obstacles well ahead of the robot and make minor course deviations. Finally, the Path Planner module takes the robot's initial and goal positions and produces an initial trajectory for execution, subject to change based on actual obstacles that the robot encounters along the way.

Note that the cycle time, or bandwidth, of the modules changes by orders of magnitude between adjacent modules. Such dramatic differences are commonplace in real navigation architectures, and so temporal decomposition tends to capture a significant axis of variation in a mobile robot's navigation architecture.

6.3.1.2 Control decomposition

Whereas temporal decomposition discriminates based on the time behavior of software modules, control decomposition identifies the way in which each module's output contributes to the overall robot control outputs. Presentation of control decomposition requires the use of a simple version of *discrete systems* formalisms:

We will consider the overall system S to be comprised of a set M of modules m connected to one-another via their inputs and outputs. The system is *closed*, meaning that the input of every module m is the output of one or more modules in M. Each module has precisely one output and one or more inputs. The one output can be connected to any number of other module's inputs. More specifically, we define functions describing the connectivity of each module as follows:

$outs(m)$: the ordered list of modules with inputs equal to the outputs of m

$ins(m)$: the ordered list of modules with outputs connected to inputs of m

$outset(m)$: the unordered set representing $outs(m)$

$inset(m)$: the unordered set representing $ins(m)$

We further name a special module r in M to represent the robot. The module r contains exactly one input and one output line: $|outset(r)| = 1$; $|inset(r)| = 1$. Thus, the input and output of the robot represent the complete action specification for the robot and the complete perceptual output of the robot respectively. For simplicity we will refer to the input of r, $ins(r)$, as O and to the robot's sensor outputs $outs(r)$ as I. From the point of view of the rest of the control system, the robot's sensor values are the system's inputs, and the robot's actions are the system's outputs, explaining our choice of O and I.

Control decomposition discriminates between different types of control pathways through the discrete system comprising the robot's control software. At one extreme, depicted in Figure 6.21 we can consider a single perfectly linear, or sequential control pathway.

Such a serial system uses the internal state of all associated modules and the value of the robot's percept I in a sequential manner to compute the next robot action O. Formally we

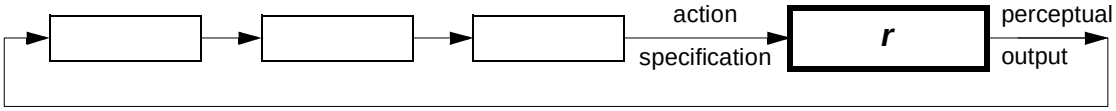

Fig 6.21 *Example of a pure serial decomposition.*

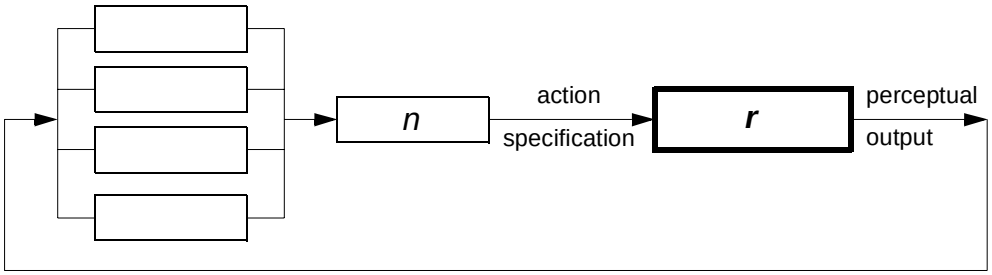

Fig 6.22 *Example of a pure parallel decomposition.*

can characterize a system as *pure serial* as follows:

The system comprised of modules *M* is *pure serial* iff:

$$\exists\,(m \in M)(ins(m) = [r]) \tag{6.13}$$

$$\exists\,(m \in M)(outs(m) = [r]) \tag{6.14}$$

$$\forall(m \in M)(|inset(m)| = 1) \tag{6.15}$$

$$\forall(m \in M)(|outset(m)| = 1) \tag{6.16}$$

A pure serial architecture has advantages relating to predictability and verifiability. Since the state and outputs of each module depend entirely on the inputs it receives from the module upstream, the entire system, including the robot, is a single well-formed loop. Therefore the overall behavior of the system can be evaluted using discrete forward simulation.

Figure 6.22 depicts the extreme opposite of pure serial control, a fully parallel control architecture. Because we choose to define *r* as a module with precisely one input, this parallel system includes a special module *n* that provides a single output for the consumption of *r*. We characterize a system as *pure parallel* using the following definition:

The system comprised of modules *M* is *pure parallel* iff:

$$outs(n) = [r] \tag{6.17}$$

$$\forall(m \in M)(m \neq r \wedge m \neq n \Rightarrow outs(m) = n) \tag{6.18}$$

$$\forall(m \in M)(m \neq r \wedge m \neq n \Rightarrow ins(m) = r) \tag{6.19}$$

Intuitively, the pure parallel system distributes responsibility for the system's control output *O* across multiple modules, possibly simultaneously. In a pure sequential system, the control flow was a linear sequence through a string of modules. Here, the control flow contains a *combination* step at which point the result of multiple modules may impact *O* in arbitrary ways.

One simple combination technique is temporal switching. In this case, called *switched parallel*, the system has a parallel decomposition but at any particular instant in time the output *O* can be computed as the specific output of one module:

$$\forall t \exists\, m(val_t(O) = f(val_t(outs(m)))) \tag{6.20}$$

The value of *O* can of course depend on a different module at each successive instant, but the complete value of *O* can always be determined based on the functions of a single module. For instance, suppose that a robot has an obstacle avoidance module and a path-following module. One overall control implementation may involve execution of path-following whenever the robot is more than 50 cm from all sensed obstacles and execution of the obstacle-avoidance module only when a sensor reports a range closer than 50 cm. One could describe such a robot as having *switched parallel* control. The disadvantage of complete switching is, of course, that the robot has no path-following bias when it is obstacle avoiding (and vice versa).

In contrast, the much more complex *mixed parallel* model allows control at any given time to be shared between multiple modules:

$$\mid t \exists\, (m_1,\ m_2)(val_t(O) = f(val_t((outs(m_1)),\ (outs(m_2)),\ ..))) \tag{6.21}$$

For example, the same robot could take the obstacle avoidance module's output at all times, converted to a velocity vector, and combine it with the path following module's output using vector addition. Thus the output of the robot would never be due to a single module, but rather the mathematical combination of both module's outputs.

Both the switched and mixed parallel architectures are popular in the behavior-based robotics community. Arkin [24] proposes the *motor-schema* architecture in which behaviors map sensor value vectors to motor value vectors. The output of all behaviors is combined, as in mixed parallel systems, using a linear combination of the behavior outputs. In contrast, Maes [ROLAND, see my Maes reference for our references in my notes to you] produces a switched parallel architecture by creating a *behavior network* construct in which a module is chosen discretely by comparing and updating activation levels for each behavior. The Subsumption architecture of Brooks [33] is another example of a switched parallel architecture, although the active model is chosen via a suppression mechanism rather than activation level modeling. For a further discussion see [24].

One overall disadvantage of parallel control is that verification of robot performance can be extremely difficult. Because such systems often include truly parallel, multi-threaded implementations, the intricacies of robot-environment interaction and sensor timing required to properly simulate all conceivable module-module interactions can be difficult or impossible to simulate. So, much testing in the parallel control community is performed empirically using physical robots.

An important advantage of parallel control is its biomimetic aspect. Complex organic organisms benefit from large degrees of true parallelism (e.g. the human eye), and one goal of the parallel control community is to understand this biologically common strategy and leverage it to advantage in robotics.

At the core of our understanding of biological systems as well as our ability to create practical robots lies the very important concept of robot behaviors. The next section introduces

the concept of behaviors, which is now a building block in most robot navigation architectures.

6.3.1.3 Behaviors as Functional Decompositions

From the earliest implemented robot architectures [104] to the most recent and complex [122], robot programmers have recognized that a mobile robot's activities can generally be divided into functionally separate tasks. Furthermore to accomplish each task a robot needs a particular set of skills. For example, a robot navigating indoor spaces may need to plug itself in for a recharge; navigate hallways; and carefully servo through doorways to deliver packages. An automated highway vehicle needs to be able to drive safely in a single lane on the highway; change lanes; merge onto the highway; and take exits off the highway.

The informal term, *behavior*, has widespread use in the robotics community and often refers to such robot skills. An important caveat is that *behavior* is used in reference to widely varying concepts. At one extreme, in the behavior-based community, a behavior can be thought of as a single perception-action pairing (e.g. a stimulus-response tuple). At the other extreme, for the deliberate planning community, a behavior is a high-level *action* that the robot planner can use in order to achieve its goals. For example finding aluminum cans may be a behavior. Happily, there is a middle ground that is somewhat acceptable to all robot software designers.

Furthermore, just as parallel control architectures can be either switched or mixed, the mobile robotics community has in some cases designed robots in which only one behavior is active at a time and, in other cases, architectures in which multiple behaviors are active simultaneously, either in a hierarchy or in a flat non-hierarchical sense.

Consider a box-pushing robot with four behaviors: *avoid obstacles, move-near-box, contact-box, push-box.* In order to go to a box and push it to the goal position, such a robot may activate these behaviors in the following three configurations:

1. *avoid obstacles* + *move-near-box*: Go to the box while avoiding obstacles enroute.
2. *contact-box*: Establish contact with the box face (no obstacle avoidance).
3. *push-box*: Push box and move to goal using a known clear path.

It is through a sequence of behavior activations that the robot is achieving its overall goal in the above example. In order to discuss the interaction between behaviors, we begin by first defining behaviors more formally, then defining the composability of multiple behaviors as well as the role of termination criteria.

Defining Behaviors

In this chapter we are particularly concerned about the performance of behaviors as elements in robot architectures. Therefore we present terminology for describing the performance of a behavior when it is active as the controller of a particular robot system. This will require constraint-based and state-based expressions. This is not a language used by behavior-based robotics designers; rather, this simple language will serve us as a useful tool in the analysis of behavior-based systems. For an in-depth study of behavior-based robotics see [(24)].

A set of constraints *C* is used to select a particular subset from the set of all possible states *S*. Formally we denote this subset of *S* consistent with *C* as: *states*(*C*). Furthermore, we need to manipulate possible paths through state space over time. Using *possible history* semantics, a *history* is a trajectory through state space *S*. A history from time *t=a* to *t=b* is a list of states from *S*, denoting the progression of world state from time *a* to time *b*. We use h_0 to denote the first state in the history.

With these tools in hand we can define *soundness*:

A behavior *beh* is *sound* with respect to a robot system executing *beh*, an initial constraints set *I*, domain constraints set *D*, and range constraints set *R* iff:

$$\forall (s \in states(I)) \tag{6.22}$$

$$\forall (h | (h_0 = s))(h \subseteq states(D) \Rightarrow h \subseteq states(R)) \tag{6.23}$$

I represents constraints over the initial conditions in which *beh* is to be activated. *D* represents constraints over every state in which *beh* is expected to function correctly. Together we call *I* and *D* the *applicability conditions* for the behavior. So a behavior is sound if and only if, when its applicability conditions are met then it maintains the range constraints.

Consider for example a hall-following behavior *hf* operating on a specific differential drive robot with a radial ultrasonic sensor ring. A robust implementation of *hf* is possible under the following generous applicability conditions:

I: The robot is within 5 cm of hallway centerline and both walls are present.

D:There are no obstacles in the hallway and at least one wall is contiguous.

A reasonable implementation of *hf* could achieve the following performance:

R:The robot will remain within 10 cm of centerline and will move at 10 cm/sec

Note from the definition above that behaviors can have very strict applicability conditions, with absolutely no guarantee of performance if they are activated in *inapplicable* states. For example, one could activate *hf* far off hallway centerline at a hallway intersection. The resulting robot motion may be disastrous, even if *hf* is sound.

Furthermore we make a careful distinction between *I* and *D* because they are generally not identical. It is common as in our above example that initial conditions be stricter than overall domain conditions. The behavior promises to keep the robot within 10 cm of centerline but requires 5 cm initially, perhaps to take an accurate initial measurement of the hallway geometry and in order to have lateral room to smoothly servo to the centerline while moving.

Behavior Composition

In some architectures, behaviors are hierarchical, with low-level behaviors such as obstacle avoidance and high-level behaviors such as sidewalk-following. Particularly in such hierarchical cases, the behaviors can be designed to operate as in a mixed parallel control system, with multiple behaviors active simultaneously.

Designing behaviors so that they can be *composed* successfully can be very challenging. With our behavior definition in hand, we can define the composability of a set β of behav-

iors as follows:

A set β of *n* behaviors is *composable* with respect to a robot system executing β iff:

The simultaneous activation of all behaviors in β is *sound* with respect to

$$\left(I' = \bigcup_{i=1}^{n} I_i\right);\left(D' = \bigcup_{i=1}^{n} D_i\right);\left(R' = \bigcup_{i=1}^{n} R_i\right) \tag{6.24}$$

and $states(R') \neq \varnothing$.

If $states(I') = \varnothing$ or $states(D') = \varnothing$ then β is composable in a degenerate manner since the composed behavior set is not applicable to any state. We are not interested in this case because it means the basic applicability criteria of the individual behaviors are inconsistent relative to one-another. Otherwise, if $states(R') = \varnothing$ then β is not composable. Consider for example the traditional example of box-pushing and obstacle avoidance. The range constraints *R* of box-pushing may include remaining within several mm of the box at all times, whereas the range constraints of obstacle avoidance may guarantee remaining at least 0.3 m from all objects in the environment at all times. In this example, the two behaviors are clearly incompatible, and thus not composable.

In practice the applicability conditions represented by R' and D' can be considerably stricter than the original applicability conditions of the individual behaviors. The basic challenge when composing behaviors is to verify that for the desired mobile robot application, these narrower applicability conditions can still be met practically.

In architectures that combine deliberate planning modules with behaviors that can be activated, the planning and reasoning systems must have a sufficiently rich representation of the behaviors to compute overall robot behavior in all applicable conditions, as well as the composability of various behaviors in mixed control architectures. But in addition, if behaviors are to run sequentially, appropriate termination mechanisms are required for transitions from one robot behavior to another. The next sub-section describes such termination criteria.

Termination Criteria

Suppose that a robot is currently executing under the control of behavior *b1* and must switch to *b2*. A key goal is to terminate *b1* such that the robot system state is consistent with the applicability criteria of *b2*. The design of termination criteria is therefore as important in achieving the desired robot performance as the design of individual behaviors that the roboticist plans to use. Termination criteria are often based on sensor events generated when features are extracted, as described in Section 4.3. For example, a hallway-navigating robot may extract hallway intersection features from its ranging data, generating perceptual events that can be used to modulate robot behaviors.

As an example, let us reconsider the example of Section 5.3. Figure 6.23 depicts a geometric map of a robot's office environment. Suppose that this robot has a switched control archi-

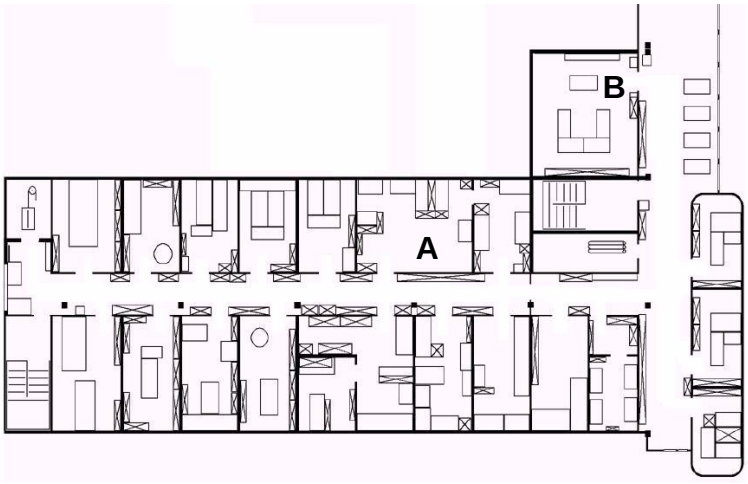

Fig 6.23 A Sample Environment

tecture with the following basic behaviors, or actions: *left-wall-follow*, *right-wall-follow*, *hall-follow, turn-left, turn-right, through-doorway*. The first three behaviors have no implicit termination; they continue until termination is triggered externally. The final three behaviors have explicit termination: *turn-left* turns 90° in place and immediately terminates while *through-doorway* servoes the robot through a doorway that it is facing and terminates when sensors determine that the robot has passed completely through the opening.

To navigate from *A* to *B* as shown on the map, the robot will need additional termination criteria. For example, suppose that the termination criterion *hallway-intersection* is based on extraction of range data showing that the robot is approximately centered at the intersection of two hallways or a hallway and a foyer. We will also use *in-hallway* and *in-room*, events that fire when the robot determines based on its sensor values that it has entered a hallway or room respectively.

Given all of the above tools, a behavior-based strategy for navigating from *A* to *B* could include the following sequence of behaviors and termination criteria:

left-wall-follow until *in-hallway*

hall-follow until *hallway-intersection*

turn-left

hall-follow until *hallway-intersection*

left-wall-follow until *in-room*

Recall that it is critical that each behavior terminate in a state that is consistent with the applicability criteria of the following behavior. Therefore *left-wall-follow*, when operating in the hallway when *in-hallway* is triggered, must leave the robot in a state appropriate for the initial constraints and domain constraints of *hall-follow*.

The power of behaviors lies in the fact that they provide an abstraction, stripping away the irrelevant detail of specific lowest-level robot outputs and instead providing functional detail that has impact on the robot's higher-level actions and missions. It is due to the convenience of this abstraction that you will see virtually every mobile robot architecture using some instantiation of behaviors as its "atomic" actions, although you will also note that the

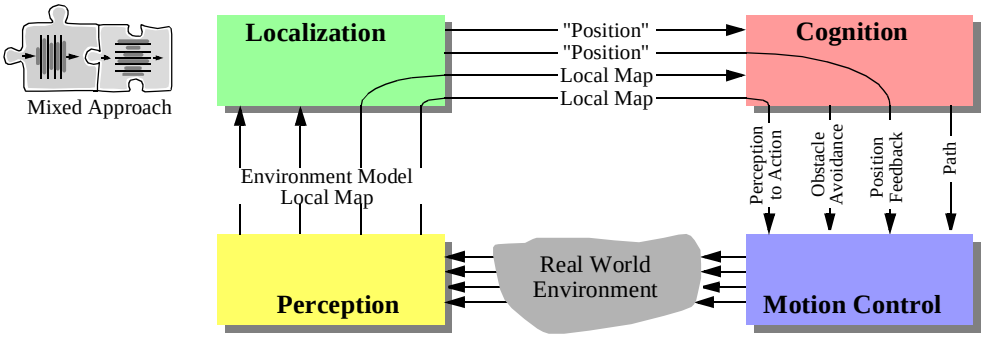

Fig 6.24 *The basic architectural example used throughout this text.*

particular implementation details vary tremendously.

6.3.2 Case Studies: tiered architectures with behaviors

We have described temporal and control decompositions of robot architecture, and have introduced behaviors as the basic primitives most common in mobile robotics. Let us turn again toward mobile robot navigation with these tools in hand. Behaviors can clearly play an important role at the real-time levels of robot control, for example path-following and obstacle-avoidance. At higher temporal levels, more tactical tasks need to modulate the activation of behaviors in order to achieve robot motion along the intended path. Higher still, a global planner could generate paths to provide tactical tasks with global foresight.

In Chapter 1, we introduced a functional decomposition showing such modules of a mobile robot navigator from the perspective of information flow. The relevant figure is shown here again as Figure 6.24.

In such a representation, the arcs represent aspects of real-time and non-real-time competency. For instance, obstacle avoidance requires little input from the Localization module and consists of fast decisions at the Cognition level followed by execution in Motion Control. In contrast, PID position feedback loops bypass all high-level processing, tying the perception of encoder values directly to lowest-level PID control loops in Motion Control. The trajectory of arcs through the four software modules is providing temporal information in such a representation.

Using the tools of this chapter, we can now present this same architecture from the perspective of a temporal decomposition of functionality. This is particularly useful because we wish to discuss the interaction of strategic, tactical and real-time processes in a navigation system.

Figure 6.25 depicts a generic tiered architecture based on the approach of Pell et al. [122] used in architecting an autonomous spacecraft, *Deep Space 1*. This figure is similar to Figure (6.19) in presenting a temporal decomposition of robot competency. However the boundaries separating each module from adjacent modules are specific to robot navigation.

Path Planning embodies strategic-level decision making for the mobile robot. Path planning uses all available global information in non-real-time to identify the right sequence of local actions for the robot. At the other extreme, *Real-Time Control* represents competencies requiring high bandwidth and tight sensor-effector control loops. At its lowest level,

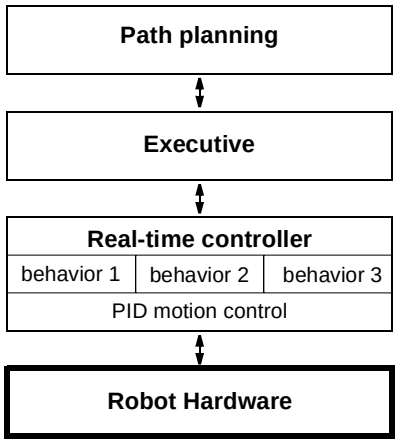

Fig 6.25 *A general tiered mobile robot navigation architecture based on a temporal decomposition.*

this level includes motor velocity PID loops. Above those, Real-Time Control also includes low-level behaviors that may form a switch or mixed parallel architecture.

In between the Path Planner and Real-Time Control tiers sits the *Executive*, which is responsible for mediating the interface between planning and execution. The Executive is responsible for managing the activation of behaviors based on information it receives from the Planner. The Executive is also responsible for recognizing failure, safing (to place the robot in a stable state) and even re-initiating the Planner as necessary. It is the Executive in this architecture that contains all tactical decision-making as well as frequent updates of the robot's short-term memory as is the case for localization and mapping.

It is interesting to note the similarity between this general architecture, used in many specialized forms in mobile robotics today, and the architecture implemented by Shakey, one of the very first mobile robots, in 1969 [104]. Shakey had *LLA* (Low-Level Actions) that formed the lowest architectural tier. The implementation of each LLA included the use of sensor values in a tight loop just as in today's behaviors. Above that, the middle architectural tier included the *ILA* (Intermedia-Level Actions), which would activate and deactivate LLA's as required based on perceptual feedback during execution. Finally, the top-most tier for Shakey was *STRIPS* (STanford Research Institute Planning System), which provided global lookahead and planning, delivering a series of tasks to the intermediate Executive layer for execution.

Although the general architecture shown in Figure 6.25 is useful as a model for robot navigation, variant implementations in the robotics community can be quite extreme. Below, we present three particular versions of the general tiered architecture, describing for each version at least one real-world mobile robot example.

6.3.2.1 Off-line planning

Certainly the simplest possible integration of planning and execution is no integration at all. Consider Figure 6.26, in which there are only two software tiers. In such navigation architectures, the Executive does not have a planner at its disposal, but must contain *a priori* all

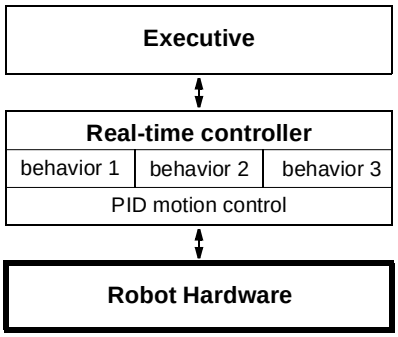

Fig 6.26 A two-tiered architecture for off-line planning.

relevant schemes for traveling to desired destinations.

The strategy of leaving out a planner altogether is of course extremely limiting. Moving such a robot to a new environment demands a new instantiation of the navigation system, and so this method is not useful as a general solution to the navigation problem. However such robotic systems do exist, and this can be useful in two cases:

Static route-based applications. In mobile robot applications where the robot operates in a completely static environment using a route-based navigation system, it is conceivable that the number of discrete goal positions is so small that the environment representation can directly contain paths to all desired goal points. For example in factory or warehouse settings, a robot may travel a single looping route by following a buried guide wire. In such industrial applications, path planning systems are sometimes altogether unecessary when a precompiled set of route-based solutions can be easily generated by the robot programmers. The Chips mobile robot is an example of a museum robot that also uses this architecture (88). Chips operates in a unidirectional looping track defined by its colored landmarks. Furthermore, it has only 12 discrete locations at which it is allowed to stop. Due to the simplicity of this environmental model, Chips contains an Executive layer that directly caches the path required to reach each goal location rather than a generic map with which a path planner could search for solution paths.

Extreme reliability demands. Not surprisingly, another reason to avoid on-line planning is to maximize system reliability. Since planning software can be the most sophisticated portion of a mobile robot's software system, and since in theory at least planning can take time exponential to the complexity of the problem, imposing hard temporal constraints on successful planning is difficult if not impossible. By computing all possible solutions off-line, the industrial mobile robot can trade versatility for effectively constant-time planning (while sacrificing significant memory of course). A different real-world example of off-line planning for this reason can be seen in the contingency plans designed for Space Shuttle flights. Instead of requiring astronauts to problem-solve on-line, thousands of conceivable issues are postulated on earth, and complete conditional plans are designed and published in advance of the Shuttle flights. The fundamental goal is to provide an absolute upper limit on the amount of time that passes before the astronauts begin resolving the problem, sacrificing of course a great deal of ground time and paperwork to achieve this worst-case performance guarantee.

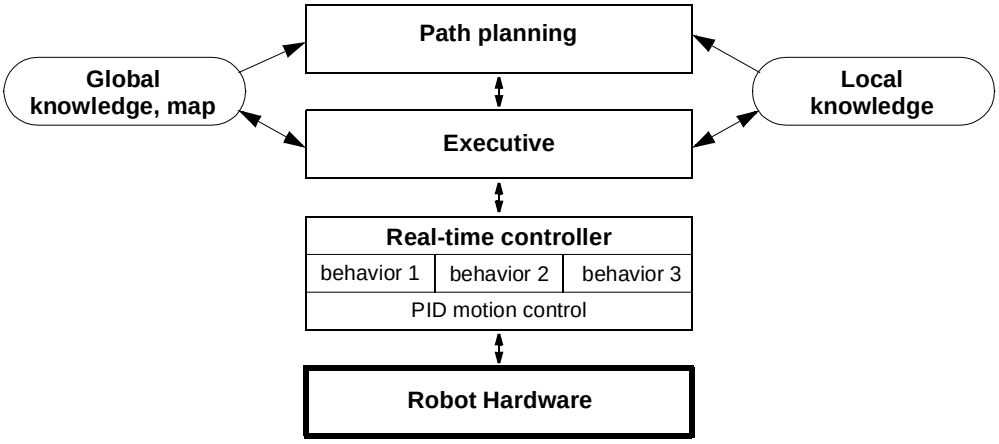

Fig 6.27 A three-tiered episodic planning architecture.

6.3.2.2 Episodic planning

The fundamental information-theoretic problem of planning off-line is that, during run-time, the robot is sure to encounter perceptual inputs that provide information, and it would be advantageous to take this additional information into account during subsequent execution. Episodic planning is the most popular method in mobile robot navigation today because it solves this problem in a computationally tractable manner.

As shown in Figure 6.27, the structure is three-tiered as in the general architecture of Figure 6.25. The intuition behind the role of the planner is as follows. Planning is computationally intensive, and therefore planning too frequently would have serious disadvantages. But the executive is in an excellent position to identify when it has encountered enough information (e.g. through feature extraction) to warrant a significant change in strategic direction. At such points, the executive will invoke the planner to generate, for example, a new path to the goal.

Perhaps the most obvious condition that triggers replanning is detection of a blockage on the intended travel path. For example, in [115] the path-following behavior returns failure if it fails to make progress for a number of seconds. The executive receives this failure notification, modifies the short-term occupancy grid representation of the robot's surroundings, and launches the path planner in view of this change to the local environment map.

A common technique to delay planning until more information has been acquired is called *deferred planning*. This technique is particularly useful in mobile robots with dynamic maps that become more accurate as the robot moves. For example, the commercially available Cye robot can be given a set of goal locations. Using its grassfire breadth first planning algorithm, this robot will plot a detailed path to the closest goal location only and will execute this plan. Upon reaching this goal location, its map will have changed based on the perceptual information extracted during motion. Only then will Cye's executive trigger the path planner to generate a path from its new location to the next goal location.

The robot Pygmalion implements an episodic planning architecture along with a more so-

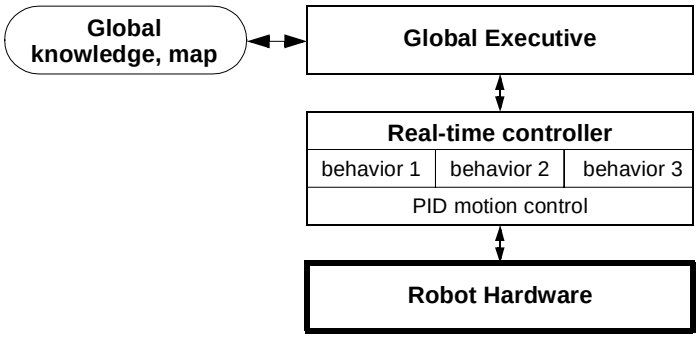

Fig 6.28 *An integrated planning and execution architecture in which planning is nothing more than a real-time execution step (behavior).*

phisticated strategy when encountering unforeseen obstacles in its way [Roland, a good reference for Pygmalion here]. When the lowest-level behavior fails to make progress, the Executive attempts to find a way past the obstacle by turning the robot 180° and trying again. This is valuable because the robot is not kinematically symmetric, and so servoing through a particular obstacle course may be easier in one direction than the other.

Pygmalion's environment representation consists of a continuous geometrical model as well as an abstract topological network for route planning. Thus, if repeated attempts to clear the obstacle fail, then the robot's executive will temporarily cut the topological connection between the two appropriate nodes and will launch the planner again, generating a new set of way points to the goal. Next, using recent laser rangefinding data as a type of local map (see Figure 6.27), a geometric path planner will generate a path from the robot's current position to the next way point.

In summary, episodic planning architectures are extremely popular in the mobile robot research community. They combine the versatility of responding to environmental changes and new goals with the fast response of a tactical executive tier and behaviors that control real-time robot motion. As shown in Figure 6.27, it is common in such systems to have both a short-term, local map and a more strategic global map. Part of the executive's job in such dual representations is to decide when and if new information integrated into the local map is sufficiently non-transient to be copied into the global knowledge base.

6.3.2.3 Integrated planning and execution

Of course the architecture of a commercial mobile robot must include more functionality than just navigation. But limiting this discussion to the question of *navigation* architectures leads to what may at first seem a degenerate solution.

The architecture shown in Figure 6.28 may look similar to the off-line planning architecture of Figure 6.26, but in fact it is significantly more advanced. In this case, the planner tier has disappeared because there is no longer a temporal decomposition between the original executive and the planner. Planning is simply one small part of the executive's nominal cycle of activities.

The idea of speeding up planning to the point that its execution time is no longer significant

may seem wishful. However, using specific algorithms in particular types of environments, such reductions in the cost of planning have been demonstrated. Consider the work of Stentz et al. [123]. These researchers designed a mobile robot control architecture for a large off-road vehicle traveling over partially known terrain at high speeds. Using advanced caching techniques from Computer Science, they optimized a grassfire path planning algorithm called *D** so that global path planning would be possible *within* the basic control loop of the Executive.

The result, depicted in Figure 6.28, is an architecture in which the local and global representations are the same, and in which the Executive has all global planning functionality required for the problem built-in. The advantage of this approach is that the robot's actions at every cycle are guided by a global path planner, and are therefore optimal in view of all of the information the robot has gathered thus far. Of course, the method is computationally challenging and will not be practical in more complex environments until processor speeds increase even further. It also has basic limits of applicability as environment size increases, but this has not been a barrier when applying this method to real-world scenario sizes.

The somewhat recent success of an integrated planning and execution method, *D**, underlines the fact that the designer of a robot navigation architecture must consider not only all aspects of the robot and its environmental task, but must also consider the state of processor and memory technology. This is indicative of the fact that mobile robot architecture design will remain an active area of innovation for years to come. All technological progress, from robot sensor inventions to microprocessor speed increases, is likely to catalyze new revolutions in mobile robot architecture as previously unimaginable tactics become practical.