Specialization of Perceptual Processes

Ian D. Horswill

December 6, 1994

Abstract

In this report, I discuss the use of vision to support concrete, everyday activity. I will argue that a variety of interesting tasks can be solved using simple and inexpensive vision systems. I will provide a number of working examples in the form of a state-of-the-art mobile robot, Polly, which uses vision to give primitive tours of the seventh floor of the MIT AI Laboratory. By current standards, the robot has a broad behavioral repertoire and is both simple and inexpensive (the complete robot was built for less than $20,000 using commercial board-level components).

The approach I will use will be to treat the structure of the agent's activity—its task and environment—as positive resources for the vision system designer. By performing a careful analysis of task and environment, the designer can determine a broad space of mechanisms which can perform the desired activity. My principal thesis is that for a broad range of activities, the space of applicable mechanisms will be broad enough to include a number mechanisms which are simple and economical.

The simplest mechanisms that solve a given problem will typically be quite specialized to that problem. One thus worries that building simple vision systems will be require a great deal of *ad-hoc* engineering that cannot be transferred to other problems. My second thesis is that specialized systems can be analyzed and understood in a principled manner, one that allows general lessons to be extracted from specialized systems. I will present a general approach to analyzing specialization through the use of transformations that provably improve performance. By demonstrating a sequence of transformations that derive a specialized system from a more general one, we can summarize the specialization of the former in a compact form that makes explicit the additional assumptions that it makes about its environment. The summary can be used to predict the performance of the system in novel environments. Individual transformations can be recycled in the design of future systems.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisors, Rod Brooks and Lynn Stein, for their support and encouragement. My time as a graduate student would have suffered badly had they not been here for me at the right time. I would also like to thank the members of my committee—Tom Knight, Shimon Ullman, and Eric Grimson—for their time, patience, and support. I would particularly like to thank Eric for the time he put in talking to me and giving me suggestions when he was not (yet) even on the committee. This document also owes much to Patrick Winston for his comments on my talks.

Your fellow graduate students are the most important part of your graduate education. Phil Agre, David Chapman, and Orca Starbuck taught me much before they moved on. Phil was in many ways another advisor. The members of the Mobot lab, particularly Maja Mataric, Dave Miller (research scientist and honorary grad student), Anita Flynn, and Colin Angle, were very patient with me and did everything from reading paper drafts to showing me how to build a power supply. Dave Miller and Joanna Bryson read the TR and gave me useful comments. My various office mates, Eric Aboaf, Paul Resnick, David Michael, Jose Robles, and Tina Kapur gave me comments on papers, talks, ideas, and general nonsense, and put up with me when I was moody. The local vision people and honorary vision people, particularly Karen Sarachik, Mike Bolotski, Henry Minsky, Steve White, Sandy Wells, David Jacobs, and David Clemens, gave me useful advice, encouragement, and skepticism. Other members of the lab, Jeanne Speckman, Annika Pfluger, Laurel Simmons, Jonathan Meyer, and Ron Wiken made my life much easier. Patrick Sobalvarro, Carl de Marcken, and Dave Baggett made it much funnier. Sigrid Unseld was as good a friend as one can ask for.

The local hardware hackers, particularly Tom Knight, Mike Bolotski, Grinnell Moore, Mike Ciholas, Henry Minsky, Anita Flynn, and Colin Angle have earned my undying gratitude for putting up with my EE questions while I was building my robot.

A number of people have helped me through my thesis by giving me pep talks and encouragement at conferences. I would particularly like to thank Bruce Donald, Avi Kak, Stan Rosenschein, Takashi Gomi, Ron Arkin, Jim Hendler, and Drew McDermott.

My housemates over the years, Hilda Marshall, Craig Counterman, John Romkey, Rob and Cath Austein, Diana Walker, Rick and Rose Stout, and David Omar White, have remained good friends, even when I have done a poor job of keeping in touch. They have always been there for me.

Finally, I would like to thank my parents for all their love and care, and my wife, Louise, for marrying me.

Support for this research was provided in part by the University Research Initiative under Office of Naval Research contract N00014–86–K–0685, and in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014–85–K–0124.

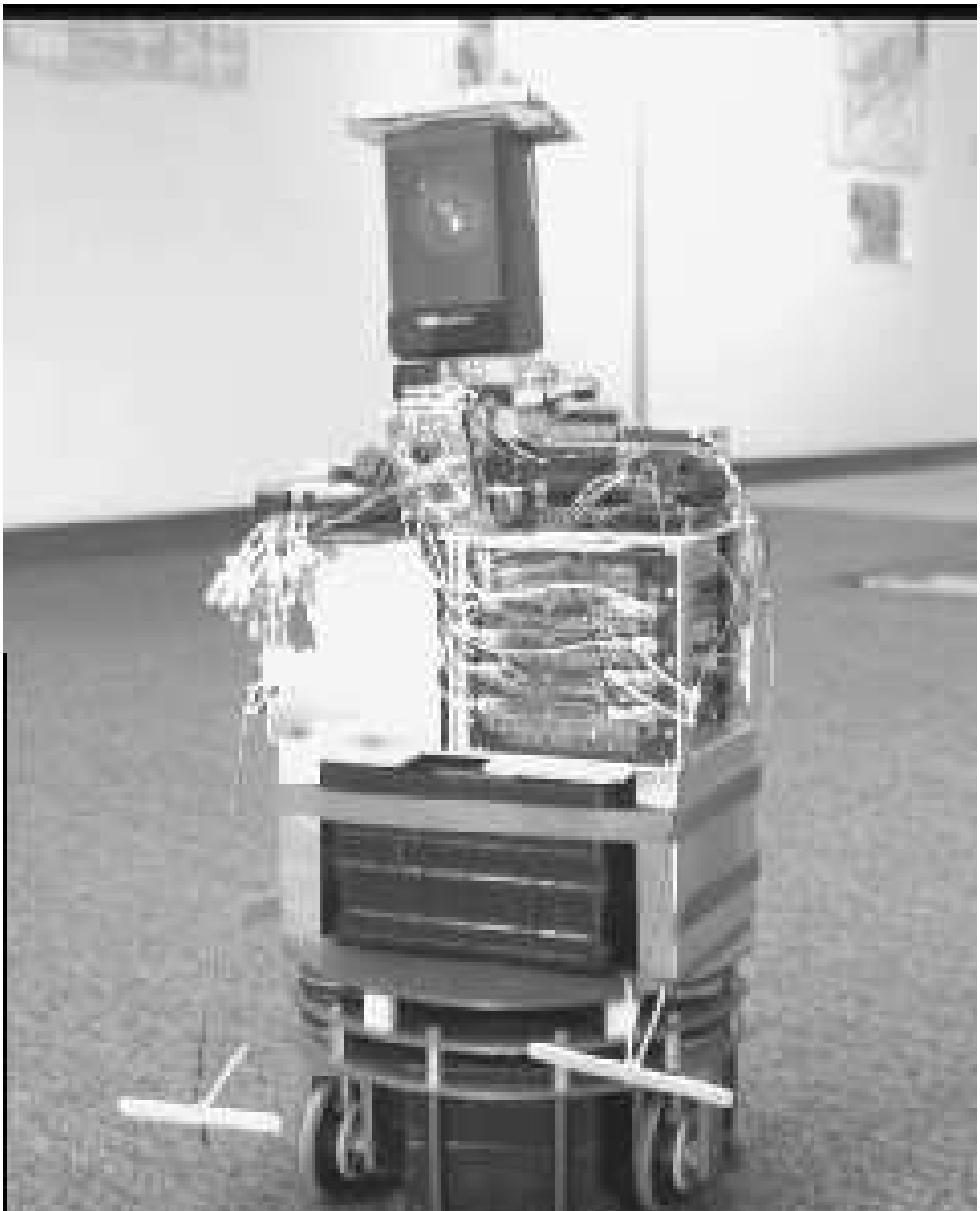
Preface: on Polly's youthful demise

This work documents, in part, the development and testing of the Polly robot. In it I claim that Polly was fast, robust, and had a very broad behavioral repertoire as measured by the current standards of the field. It should be stated for the record however that Polly is now all but dead; Its VMEbus boards are reluctant to boot properly and its base has many broken gear teeth. This speaks less of the inadequacies of the hardware than of the unusualness of its use. The electronics were never designed to be run off of batteries much less to be continuously shaken or periodically crashed into walls. The base, although ultra-reliable by the standards of robot research hardware, was not designed to exert $2g$ accelerations on 40 pounds of metal or to move that metal at a meter per second for hours a day for over a year.

Many of the algorithms developed for Polly live on. The low level navigation algorithms have been reimplemented in our lab for the Gopher, Frankie, and Wilt robots. Other labs are starting to use the algorithms and a commercial version of the collision avoidance system is even available. As of this writing, the basics of turn detection and control have been reimplemented on Frankie, although the full place recognition and navigation system will likely be considerably more sophisticated on Frankie than on Polly.

You might think it odd that one could feel wistful about the decomissioning of a large chunk of metal. But then you didn't spend as much time chasing it around the lab as I did.

Ian Horswill
Cambridge, September 1994



Polly the robot

Part I

Introduction and Approach

Chapter 1

Introduction

In this report, I will discuss the use of vision to support concrete, everyday activity. I will argue that a variety of interesting tasks can be solved using simple and inexpensive vision systems. I will provide a number of working examples in the form of a state-of-the-art mobile robot, Polly, which uses vision to give primitive tours of the seventh floor of the MIT AI Laboratory. By current standards, the robot has a broad behavioral repertoire and is both simple and inexpensive (the complete robot was built for less than $20,000 using commercial board-level components).

The approach I will use will be to treat the structure of the agent's activity—its task and environment—as positive resources for the vision system designer. By performing a careful analysis of task and environment, the designer can determine a broad space of mechanisms which can perform the desired activity. My principal thesis is that for a broad range of activities, the space of applicable mechanisms will be broad enough to include many that are simple and economical.

The simplest mechanisms that solve a given problem will typically be quite specialized to that problem. One thus worries that building simple vision systems will be require a great deal of *ad-hoc* engineering that cannot be transferred to other problems. My second thesis is that specialized systems can be analyzed and understood in a principled manner, one that allows general lessons to be extracted from specialized systems. I will present a general approach to analyzing specialization through the use of transformations that provably improve performance. By demonstrating a sequence of transformations that derive a specialized system from a more general one, we can summarize the specialization of the former in a compact form that makes explicit the additional assumptions that it makes about its environment. The summary can be used to predict the performance of the system in novel environments. Individual transformations can be recycled in the design of future systems.

1.1 Example

Suppose we view vision as being the problem of answering questions about the environment using images. Figure 1.1 shows an image of my office taken from my robot. We might want to determine whether the robot should turn left or right so as to avoid the objects in the scene. This amounts to the problem of finding which regions of the floor are free and which have objects on top of them. The correct answer is that the robot should turn left, since there are obstacles nearby on the right while there is clear floor on the left. The fundamental difficulty here, as with most vision problems, is that the projection process of the camera loses information, depth information in particular, and so we cannot uniquely determine the structure of the scene without additional information either in the form of extra images or of extra information assumed about the problem.

A common way of solving the problem would be to build a complete depth map of the scene using multiple images. We could then project the features in the depth map into the floor plane to determine which parts of the floor do and do not have obstacles on top of them.

The simplest version of this is two use two cameras in a stereo configuration. Distinctive features (usually edges) can be found in the two images and matched to one another. Given the matching of the features, we can compute each feature's shift due to parallax. From the camera positions and the parallax data, we can compute the positions of the surface patches in the world from which the individual features were imaged (see Barnard and Fischler [11]).

The stereo approach is a perfectly reasonable approach, but it does have two undesirable features. First, it is computationally expensive, particularly in the matching phase. It also requires that features be localized accurately and reliably, which usually means the use of high resolution data, which requires more computational power. A more important problem however is that the floor in this environment is textureless and therefore featureless. Figure 1.2 shows a map of the image in which pixels with significant texture (actually, significant intensity gradients) are marked in white. The floor is uniformly black. The stereo process cannot make any depth measurements in the region of the image which is most important to the robot because these are no features to be matched there.

This problem is easily remedied. Since the floor is always flat, it is reasonable for the stereo system to interpolate a flat surface in the absence of texture. However, it is important to remember that the stereo system is then working not because it is measuring the depth of the floor directly, but because it is making a smoothness assumption which happens to be true of the floor. This need not be true in the general case. The floor could slope gently or suddenly. There could even be a cliff. While a such sudden discontinuity in depth would typically generate image features along the discontinuity itself, there would still be no features on either side of the

Figure 1.1: Image of my office taken from the robot's camera. The dots in the lower middle of the image are artifacts due the quantization in the rendering process. The structure in the lower right hand portion of the image is a 5-legged office chair. The structures in the top-left are (left to right) a doorway viewed from an oblique angle, a small trash can, and a file cabinet. The homogeneous region in the lower and middle left is the carpet.

Figure 1.2: The pixels with significant texture. Pixels marked in white differ from their neighbors above and to the right by total of at least 15 grey levels out of a possible 510. Pixels marked in black differ by less than 15 grey levels. The image was first smoothed with a 3×3 filter to remove camera noise. Note that the floor is uniformly black.

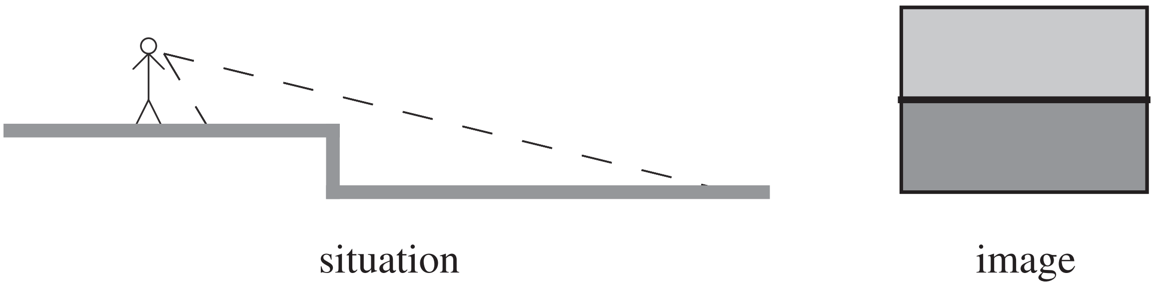


Figure 1.3: An observer views a cliff of a textureless surface (left). Although variations in lighting of the two sides of the cliff may produce a local variation in image brightness at the point of discontinuity (right), there is still no texture in the image above or below the discontinuity which would allow the observer to infer the depth, or even the presence, of the cliff.

discontinuity that could be used to detect the cliff (see figure 1.3).

This brings out two important points. First *truly general systems are extremely rare*, and so claims of generality should be considered carefully. Often the mechanisms we build have hidden assumptions which can fail to be true. These can be particularly difficult to diagnose because we typically choose test data that fit the assumptions. This it not to say that such assumptions are bad. Quite the contrary: they lead to great improvements in performance. Rather, we should make *informed decisions* about our use of specialization.

1.1.1 A coloring algorithm for navigation

When the stereo system works on the scene in figure 1.1, it works because the floor is flat and the obstacles have texture. We can make a different system to solve the problem, one that is much more efficient, by using these facts directly.

Let us treat the lack of texture on the floor as a positive feature of the environment. Notice that the floor forms a single, connected black blob at the bottom of figure 1.2. This blob is shown alone in figure 1.4). I will call this the carpet blob. The carpet blob is easily computed by starting at the bottom of the screen and tracing up each image column until a textured pixel is found. The set of pixels skipped over will be the blob.

Notice that the height of the blob varies with the amount of exposed floor in the corresponding direction so the number of pixels skipped in a given column gives us a rough and ready measure of the amount of free space in that direction. This suggests the following algorithm for solving the original problem: first, find the textured pixels in the image, then extract the carpet blob, and then turn in the direction in which the carpet blob is taller. This algorithm is the basis of much of Polly's navigation

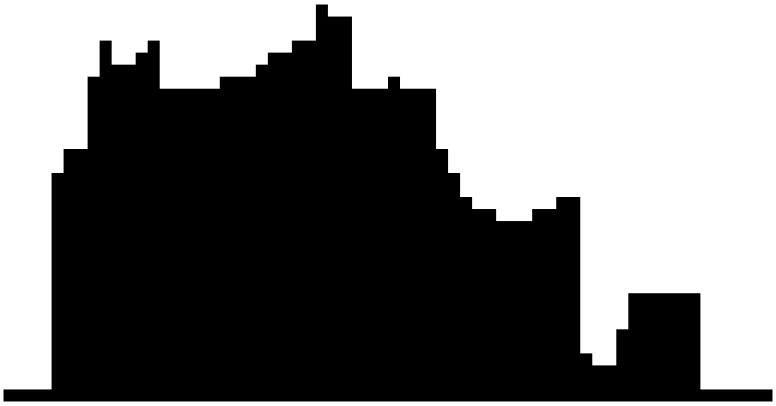


Figure 1.4: The set of textureless pixels corresponding to the carpet. Note that the blob is taller where there is more exposed carpet.

capabilities. It can easily be executed in real time on a low end personal computer.

1.1.2 Analysis of the coloring algorithm

We can understand the relationship of the stereo-based and blob-based algorithms as follows. Both systems determine whether there is more free space in front of the robot on the left side or the right side of the image. The stereo system measures this directly by computing a depth map and projecting it into the floor plane. Since we are only concerned with determining which side is larger, however, we do not need to know the exact distances in any particular units of measure. Any measure that increases monotonically with distance will work, provided that we use the same measure on both sides. We can thus substitute *any system* that computes a monotonic function of the distance for the stereo system. More importantly, we do not even need to know what the monotonic function is. It could vary from moment to moment so long as it was used uniformly within a given image. It has been known at least since Euclid that image plane height is a monotonic function of distance. This means, roughly, that if all the obstacles rest on the floor, then we can substitute the image plane height of the obstacle for the stereo system, provided that we have some way of labeling each pixel as being either obstacle or carpet. A general carpet detector that can recognize any carpet (or, equivalently, any obstacle) might be be more difficult to build than the stereo system. However, the carpet in this environment has a very predictable appearance: it has no texture. This means that we can substitute a texture detector for the general carpet detector or obstacle detector.

We can summarize this analysis with the following general principles:

- We can substitute any monotonic measure of a quantity for a calibrated measure, provided that the measure will only be used for comparisons.
- We can substitute height in the image plane for some other distance calculation, provided that all objects rest on the floor and there is some way of classifying pixels as being floor or object.
- We can substitute a texture detector for a floor detector, provided that the floor is textureless.

These principles concisely describe the specialization of the blob-based algorithm. Each describes a general transformation from a possibly inefficient algorithm to a more efficient one, along with the conditions on the task and environment which make it valid. These transformations can be applied to the design of other systems or used to predict and modify the performance of the original system. For example, if we wanted to use the blob-based algorithm in an environment with a textured carpet, we would have to abandon the last transformation, but we would still be able to use the other two. If there was some property other than texture which allowed carpet pixels to be easily classified, then we could use that property as the basis of a new transformation.

1.2 Preview of results

This report contributes to three areas: the design of efficient vision systems, the design of mobile robots, and the analysis of specialized systems.

1.2.1 Lightweight vision

Vision is traditionally thought of as being very expensive. A number of researchers have recently argued that vision can be significantly simplified by using the dynamics of the agent's interaction with its environment or by specializing the vision system to a particular task. (See Bajczy [8], Horswill [52], Ballard [9], Ikeuchi [53], or Aloimonos [4]. See also section 3.1.2.) We can simplify vision further, however, by taking into account specific properties of the agent's environment. We can view both the task and the environment as positive resources for the designer. The task imposes concrete constraints on the information which the system must extract from the image and the performance with which it must extract it. Far from being bad, these constraints are a good thing, for they tell the designer not only what is required of the agent, but also what is *not required* of it.

Knowing the constraints is important because it allows the designer to make trade-offs intelligently. Performance has many parameters. These parameters typically can not be optimized simultaneously, so the designer must decide what to optimize and what not to optimize. Knowing the task constraints allows the designer to make these choices effectively. A similar situation holds for choosing what information to extract from the image. Resolution is a useful case in point. For a given amount of computing power, the rate at which agent will be able to process frames will be bounded above by the inverse of the number of pixels in the image, assuming the program in the agent at least reads every pixel. The number of pixels is quadratic in the linear resolution of the image however. This means that of you want to double the accuracy with which you can localize a feature in the image, you must drop the rate at which you process images by at least a factor of four. Most vision research has focused on relatively high resolution images, whereas all the algorithms discussed in this work operate on very low resolution images (64×48 or 16×12).

Understanding the environment allows the designer to make optimizations such as the substitution of low level image properties for high level object properties. The algorithm discussed above is a case in point: because of the special properties of the environment, we can substitute a low level image property (texture) for a high level symbolic property (obstaclehood). Understanding the environment might also tell the designer that certain pathological situations will not occur or that certain performance requirements can be relaxed. All of these serve to broaden the space of mechanisms available to the designer.

For a given task and environment, the space of mechanisms which will solve that task in that environment is typically large. My principal claim is that for many real world task/environment pairs, the space is large enought to include a number of very efficient mechanisms. I call these very efficient mechanisms, "lightweight" vision systems.

1.2.2 Mobile robotics

The principal contribution of this report to the field of mobile robotics is to demonstrate that it is possible to build a robust and economical vision-based robot using only on-board computation and standard commerical hardware. The Polly robot uses a single 16MIP digital signal processor, a Texas Instruments TMS320C30, for nearly all its computation. A modified version of the chip is available for $35 in quantity. With each frame, the robot recomputes all its visual percepts, compares the scene to its complete database of landmarks, and updates its motor velocities.

Polly patrols the seventh floor of the MIT Artificial Intelligence Laboratory, searching for visitors who may want tours. When it finds a visitor, it offers them a tour. If they gesture properly, the robot leads the vistor around the lab, giving informative speeches as it recognizes landmarks. This task involves a very wide repertoire of behaviors by the current standards of the field. It can follow paths, recognize landmarks, detect the presence of people, interpret simple gestures, and choose paths to a specified landmark.

The robot is extremely well tested. The lower layers of its control system have been running for over a year and have seen several hundred hours of service. During that time, the environment has steadily changed its geometry and appearance: the student population rose and fell, furniture was rearranged, a small mountain range was built in part of the lab (really), carpets were shampooed, and office lighting was completely replaced. The robot tolerated all these changes, although it sometimes had to be modified slightly to cope with them (see chapter 11). The robot has also been tested in other, similar environments.

1.2.3 Analysis of specialization

There has been renewed interest within the AI community in the relationship between agents and their environments, particularly within the artificial life, biologically-based AI, and situated-action communities (see section 4.4 for a survey). We can analyze this relationship formally by deriving an agent that is specialized to its environment from a hypothetical general agent through a series of transformations that are justified by particular properties of the agent's environment. In performing the derivation, the designer divides the agent's specialization into a discrete set of reusable transformations, each of which is paired with an environment property that makes it valid. I call such properties "habitat constraints" because the set of such constraints define the agent's habitat. I will generally refer to the transformations as "optimizations." These are optimizations in the sense of compiler optimizations: they are transformations which improve performance, but do not necessarily yield optimal performance.

The advantage of the derivation is that it makes explicit the agent's implicit assumptions about its environment. These assumptions can then be used in the design or analysis of other agents. Thus it provides a way of recycling the experience gained in designing specialized systems. Although it is not possible in general to fully automate the design of specialized systems, this kind of post-hoc analysis can eventually allow us to develop cookbook methods for a broad range of problems.

The technique is not limited to the analysis of vision systems. I will present examples of its application to vision, motor control, and discrete action selection.

1.3 Structure of this report

The next three chapters present high level introductions to Polly, lightweight vision, and specialization, respectively. Chapter 2 describes Polly's task and environment, its high level software design, and discusses the language and hardware used to implement it. It also provides a detailed example of a run of the robot and a survey of other vision-based robot navigation systems. Chapter 3 discusses the history of vision research, particularly reconstructionism and the recent move toward task-based and active vision. It then discusses how task and environment can be used as resources for building lightweight vision systems. Chapter 5 draws out the general framework for analyzing specialization. Subsequent chapters are more detailed and technical.

Chapter 6 gives the formal basis for applying the transformational theory to a class of simple vision systems. Chapter 7 does the same with the problem of discrete action selection. The reader who is less interested in formalism may wish to skip these.

Chapters 8, 9, and 10 describe in detail the robot's vision, low level navigation, and high level navigation systems, respectively, and use the analytical framework to explain their performance.

Chapter 11 discusses a number of experiments performed on Polly, gives a sense of its reliability, and categorizes its failure modes.

Chapter 12 summarizes the key points of this work and gives conclusions.

Chapter 2

Introduction to the Polly system

Polly is a low-cost vision-based mobile robot built to explore the use of domain constraints in the design of lightweight vision systems. Polly lives on the 7th floor of the MIT AI Laboratory and gives simple tours of the 7th floor. Polly is interesting for a number of reasons. First, it has a relatively wide range of capabilities. It can navigate, recognize places, detect people, and understand their gestures. Second, it is simple and inexpensive, costing only $20K to build from off-the-shelf, board-level components (a faster version could now be built for less that $10K). Third, it is very fast, running between 0.75 and 1 meters per second, with its perception and control systems running at 15Hz. Finally, it performs its tasks almost exclusively using vision. Vision-based autonomous robots are relatively rare; Cheap, fast, autonomous systems with wide behavioral repertoires have not been built previously.

In this chapter, I will describe the task, environment, and basic structure of the robot. Section 2.1 describes the robot's task and habitat. Section 2.2 describes the high level architecture of the robot. Section 2.3 gives a detailed run of the robot and describes the operation of the robot during the run. Section 2.4 briefly describes the programming language used. Section 2.5 describes the basic hardware components of the robot, their capabilities, and their interconnections. Finally, section 2.6 describes previous visually guided mobile robots. Subsequent chapters will discuss the software components in more detail.

2.1 Task and environment

Polly's environment is the network of corridors on the seventh floor of the AI Laboratory at MIT (see figure 2.1). Its task is to patrol the lab, find visitors, and give them tours. Its patrol pattern is shown in figure 2.1. As it drives through the hallways, it searches for visitors who want tours. Since it can only look downward at the floor, it has to detect people by looking for their legs. The only way it can distinguish a visitor from a normal occupant is to rely on the fact that the normal occupants are probably sick of it and will therefore leave it alone, whereas visitors will actively investigate it.[1] For this reason, Polly only responds to people who stand directly in front of it. It ignores people who casually pass by it or who lean against the wall. When Polly does find a leg-like object directly in front of it, it introduces itself and offers a tour, saying that the person should wave their foot around if they want a tour. If the person indicates that s/he would like a tour by gesturing with their foot, Polly leads the person around the lab, making pithy comments and giving canned speeches when it recognized landmarks. For example, when it recognizes the T.V. lounge, it says "This is the T.V. lounge. We waste a lot of time here." When Polly returns to the place where it previously picked the visitor up, it thanks him/her and says goodbye. It then looks for another visitor.

Here is a typical scenario for Polly:

Event	Speech
Polly approaches visitor	Hello. I am Polly. Would you like a tour? If so, wave your foot around.
Visitor waves foot	Thank you. Please stand to one side.
Visitor moves	Thank you. Please follow me.
Polly drives	I can avoid obstacles, follow corridors, recognize places, and navigate from point to point.
Keeps driving	My vision system runs at 15 frames per second on a low cost computer
Robot passes vision lab	On the right here is the vision lab. By the way, I don't understand anything I'm saying.
Robot enters T.V. lounge	This is the T.V. lounge. We waste a lot of time here.
Passes office	This is Karen and Mike's office.
Passes office	This is the office of Anita Flynn.
Enters playroom	This is the playroom. This is the end of the tour. Thank you and have a nice day.
Robot drives off.	

Polly continually alternates between searching for visitors and giving tours until it a switch is thrown on its control panel. It then drives back to my office and parks by my desk.

[1] Or so the theory went when I designed the system. In practice, it is exactly the other way around: visitors, not wanting to cause any problems, get out of the robot's way; the denizens of the lab, however, are perfectly happy to harass it in an effort to test its limits.

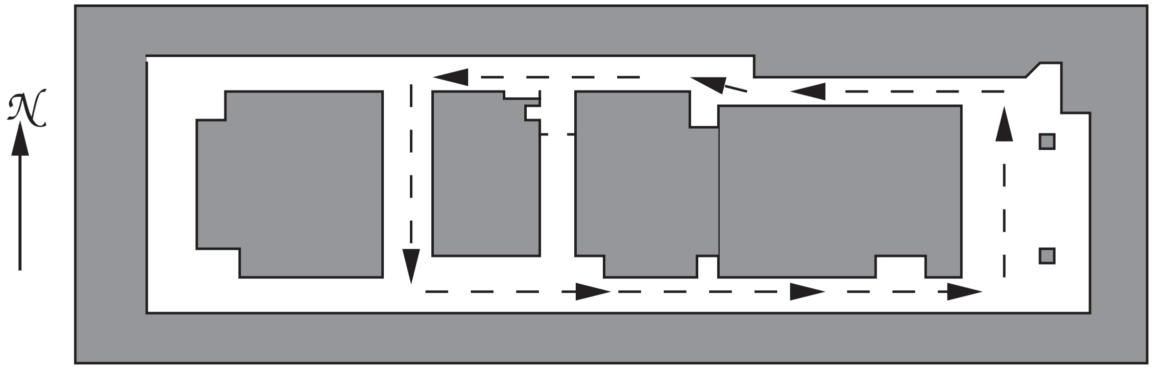


Figure 2.1: The layout of the 7th floor of the lab, and Polly's patrol pattern within it.

2.2 Software architecture

Polly is meant to be thought of as a group of communicating asynchronous processes connected by fixed links (see, for example, Brooks [20] or Rosenschein and Kaelbling [88]). These can be grouped into the high level structures shown in figure 2.2.

A group of related visual processes, the core vision system (CVS), transform images into high level percepts that encode specific relevant information such as whether the robot's path is blocked or whether there is a person in view. These percepts are effectively broadcast to all other parts of the robot.

Navigation is implemented by two groups of processes. The low-level navigation system (LLN), controls the motors and implements obstacle avoidance, path (wall or corridor) following, and switching from one path to another. It can also perform open-loop turns of specified angles. The high-level navigation system (HLN) matches landmarks to an internal map and performs goal directed navigation. When a higher-level process gives the HLN a goal landmark, the HLN directs the LLN through a series of corridors leading to the goal. The goals are usually specified by the `wander` system which implements the patrol pattern by alternately instructing the high level navigation system to go to opposite corners of the patrol circuit (the playroom and the vision lab, see figure 10.1).

A number of Polly's tasks, such as offering tours, require the robot to perform fixed sequences of actions. Each such sequence is implemented by a special kind of process called a "sequencer" (see section 10.4).

Another set of processes control the voice synthesizer. The `tour-announce-place` process gives a canned speech whenever the robot recognizes a landmark and is in the process of giving a tour. The `tour-chatter` process generates periodic small talk during tours such as "my vision system runs at 15 frames per second on a low cost computer," or "by the way, I don't understand anything I'm saying." The

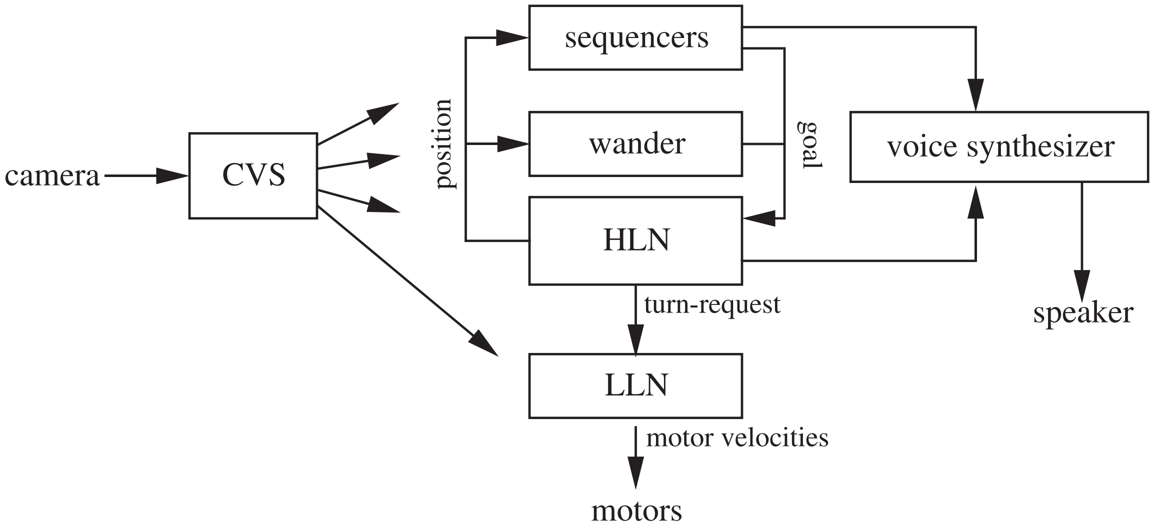


Figure 2.2: Polly's gross neuroanatomy.

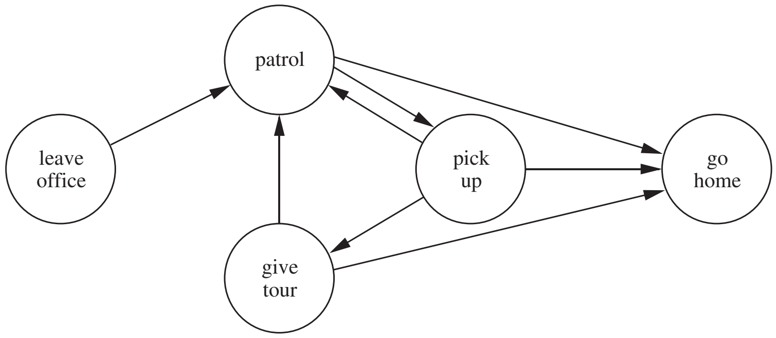


Figure 2.3: High level states and transitions

`messages` process generates random messages at random times when the robot is idle (not giving tours). This was done purely for the amusement of the author. The `crier` process can be programmed to give a specific message at specific interval. It effectively turns the robot into a "town crier." I have used it to announce talks. Finally the `chatter` process arbitrates requests from the other processes for access to the voice synthesizer. The design of these processes will not be discussed further.

The one piece of globally visible state in Polly is its high-level mode (see figure 2.3), which is stored in the global variable `global-mode`. The values of `global-mode` are 0, meaning the the robot is starting up or going home, 1, meaning that it is on patrol, 2 meaning that the robot is giving a tour, and 3, meaning that it is in the process of offering a tour. The global mode is used by the wander process to determine whether it should drive in the patrol pattern, by the speech system to determine whether it should give tour speeches, and by the pick-up routine to determine whether it should attempt a pick-up. It is written by the pickup routine, the tour routines, and the routines for starting up and going home. This could have been implemented using local state information, such as having the different processes check to see which other processes were enabled, or by using a hormone-like mechanism (see Brooks [21]). In this case, my needs were simple enough that a global variable sufficed.

2.3 Detailed example

The robot always begins near my desk, facing south (see figure 2.4). It leaves the office by performing a fixed set of steps specified by a sequencer. First it drives south until it is blocked by an obstacle. This leaves it near the file cabinet labeled (1) in figure 2.4. It then turns left and drives until it no longer sees a wall on the left, bringing it out the office door (point 2 in the figure). It then moves forward a few feet, turns left again, and drives until it is blocked and facing east, at which point it should be facing the right-hand wall (point 3 in the figure). It then turns right 120 degrees so as to face the pillar and couch, and moves forward. The robot's obstacle avoidance mechanisms are sufficient to move it into the channel between the couch and the wall and off into the corridor (see point 4 in the figure). At this point, Polly's normal navigation systems are activated. The robot's position is initialized to "Ian's office" and the wanderer sets the goal landmark to be the vision lab (see figure 10.1).

2.3.1 Patrolling

The robot now begins patrolling. It is in the corridor between "Ian's office" and "Elevator lobby" in figure 10.1, traveling west. The HLN determines that it is east of its goal, the vision lab, and so allows the LLN to continue to move west. When the robot reaches the elevator lobby, the wall on the right-hand side suddenly vanishes from view, indicating that the robot has reached a right turn. The robot checks its map for a right turn and determines that it is at the elevator lobby. The entry in the map for the elevator lobby says that the robot should veer to the right to remain in the corridor, so the HLN instructs the LLN to make a small right turn. The LLN does so and continues across the open space of the elevator lobby until it reaches the other side. There, the LLN realigns with the corridor and continues west. As the robot drives past the leftmost corridor in figure 10.1 (the corridor between the vision lab and the kitchen), the left-hand wall vanishes from view and the vision system signals the presence of a left turn. The HLN finds the turn in its map and updates its position.

When the robot reaches the vision lab, it detects another left turn and the HLN updates its position again. Now several things happen. First, the HLN notices that it has reached its goal. Then the wanderer notices that it has reached the vision lab

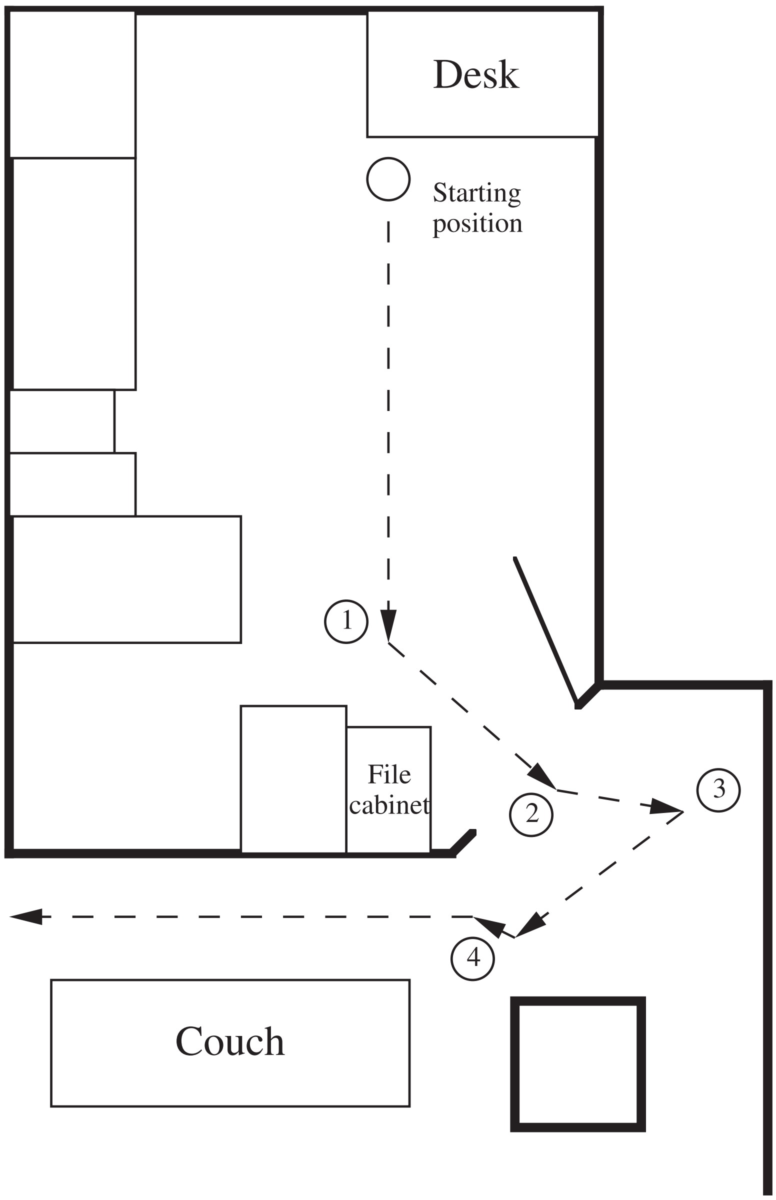


Figure 2.4: The robot's path as it leaves the office and enters the corridor. Both path and floor plan are only schematic—neither is drawn to scale.

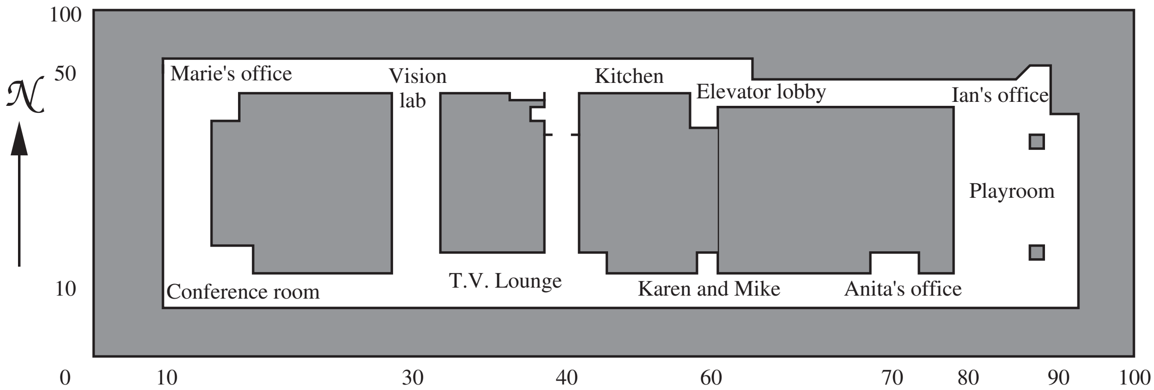


Figure 2.5: Polly's habitat, the 7th floor of the MIT AI laboratory. The diagram is not to scale, and the direction north has been defined for convenience, rather than geological accuracy.

and sets the goal to be the southern corner of the playroom. This, in turn, causes the HLN to wake up. The HLN determines that the goal is to the southeast. Since it is at a turn to the south, it instructs the LLN to turn 90 degrees to the left. The LLN turns left, and begins to align with the new (southern) corridor, as if nothing had happened, and proceeds along the corridor. When the robot reaches the southern wall (near the T.V. lounge in the figure), several things happen again. The LLN can no longer move any further and so it stops. The HLN notices that it has reached the southern wall, and so updates its position again. Since the goal is now east instead of southeast, the HLN tells the LLN to turn left again. The LLN turns, follows the wall through the T.V. lounge, and proceeds on its way. The robot continues, the HLN updating its position with the appearance of each left hand turn, until the robot reaches the playroom. Then the wanderer sets the goal back to the vision lab, the HLN tells the LLN to turn north, and the cycle repeats.

2.3.2 Giving tours

The robot continues to patrol until it encounters a person in the hallway. When this happens, a sequencer (`offer-tour`) is started. The sequencer inhibits the LLN and HLN and halts the robot. It then says "Hello, I am Polly. Would you like a tour? If so, wave your foot around." It then looks to see if there is any motion in the image. If not, it says "OK, have a nice day," and dis-inhibits the LLN and HLN. If there is motion, then it says "please stand to one side," and waits until it is no longer blocked. Then it says "OK, follow me," and starts a new sequencer, `give-tour`. `Give-tour` records the current position of the robot, and sets the robot's global mode to give-tour mode, which enables the `tour-chatter` and `tour-announce-place` processes. `Give-tour` waits for the robot to return to that position. In the mean time, the wanderer, HLN, and LLN continue as if nothing had happened. Each time the HLN recognizes a new place, `tour-announce-place` wakes up and gives the speech listed in the map for that place. This leaves long periods of boring silence while the robot drives from one place to another, so the `tour-chatter` process inserts chatter messages in the pauses, roughly every 15 seconds. Eventually, the robot returns to the place it started from. Then `give-tour` wakes up, says "That's the end of the tour. Thank you and have a nice day," sets the global mode to "patrol" (which disables `tour-announce-place` and `tour-chatter`), and turns itself off. The robot then continues on its way, looking for a new visitor.

2.4 Programming language issues

While Polly is meant to be thought of as a parallel system, in practice it is implemented on a serial processor. The processor is programmed in a subset of Scheme.[2] The main loop of the program repeatedly grabs a new image, processes it, computes a new set of motor commands, transmits the commands, and waits for a new image. Each execution of the loop is referred to as a "clock tick."

Parallel processes are simulated using Scheme procedures, one procedure per process. Each procedure is called once per clock tick by the main loop. Occasionally, several simple processes are folded into a single procedure for efficiency. Communication channels between the processes are simulated with global variables. Nearly all of these variables are of thought of as wires carrying signals: each wire is updated on each clock tick by a single process. For example, the variable `direction` is written every clock tick by the process which reads the robot's rotational odometer. It always holds the current direction of the robot. A few global variables, such as the variable which holds the goal position of the robot, are set only rarely but may be set by many different processes. These are best though of as latches.

Nearly all of the vision code is implemented using a set of macros for vector processing built on top of Scheme. Images are represented as 1D vectors in row-major (raster) format so that the pixel at address (x, y) in a 64×48 image is the $x + 64y$'th entry in the vector. Two images can then be compared using a vector equivalent of `mapcar`:

[2]The actual dialect is called "Senselisp" (see Horswill [47][48]). Senselisp is essentially Scheme with garbage collection and run-time type checking removed, and compile-time type inference, full macros, and pointer arithmetic added. This modifications allow a relatively simple compiler to produce very efficient code. Until recently, the Senselisp compiler produced better code than the C compiler sold by the manufacturer of the computer.

```
(define (compare-images in1 in2 out)
  (map-vector! out
               =
               in1
               in2))
```

This procedure takes two input vectors of the same length and applies the = function to successive pairs of elements of the two inputs, writing the results (true or false) to successive elements of the output. Vectors can be shifted around using pointer arithmetic, which allows adjacent pixels of an image to be compared using the vector macros. For example, the procedures

```
(define (bad-vertical-edge-detector in out)
  (map-vector! out
               =
               in
               (shift in 1)))
(define (bad-horizontal-edge-detector in out)
  (map-vector! out
               =
               in
               (shift in (raster-width in))))
```

detect vertical and horizontal edges by comparing a pixel with the pixel just to the right of it, and just below it, respectively. The function `raster-width` returns the number of pixels per line for a vector which is used to hold an image. `Shift` performs pointer-arithmetic. It takes a vector and an integer and returns the vector displaced by the specified number of elements. Note that this shifting will produce bizarre results at the boundary between the left and right sides of the image. In 2D, the left and right sides of the image are not connected, but in the 1D array, the first pixel of the 2nd line is right after the last pixel of the first, so the vertical edge detector above will find artifactual edges near the boundary. This could be avoided by using nested loops, but it is usually easier just to ignore the boundary pixels.

2.5 Hardware design

Because no commercial robot was available with an appropriate computer and frame grabber, I was forced to build the system myself from commercial board-level components. This was not as terrible as it sounds. It mostly involved making connectors to link one board to another or screwing, gluing, taping, or velcro'ing the various components to one another. Roughly a year was required to build the robot, but nearly all of the time was spent reading catalogs, ordering parts, or waiting for parts

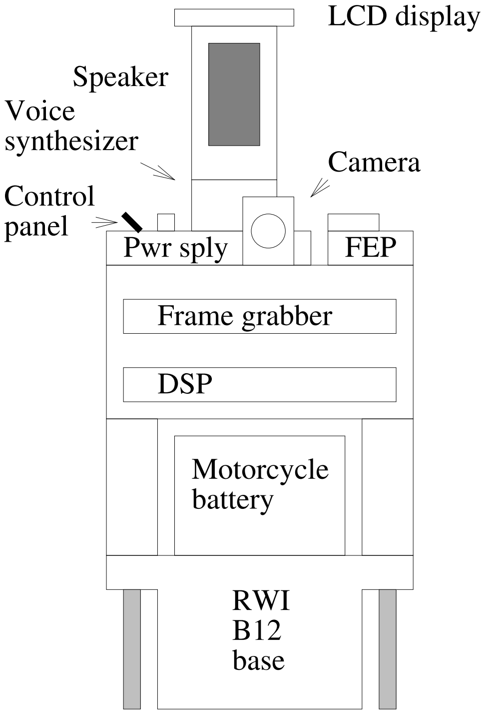


Figure 2.6: Basic components and layout of the robot.

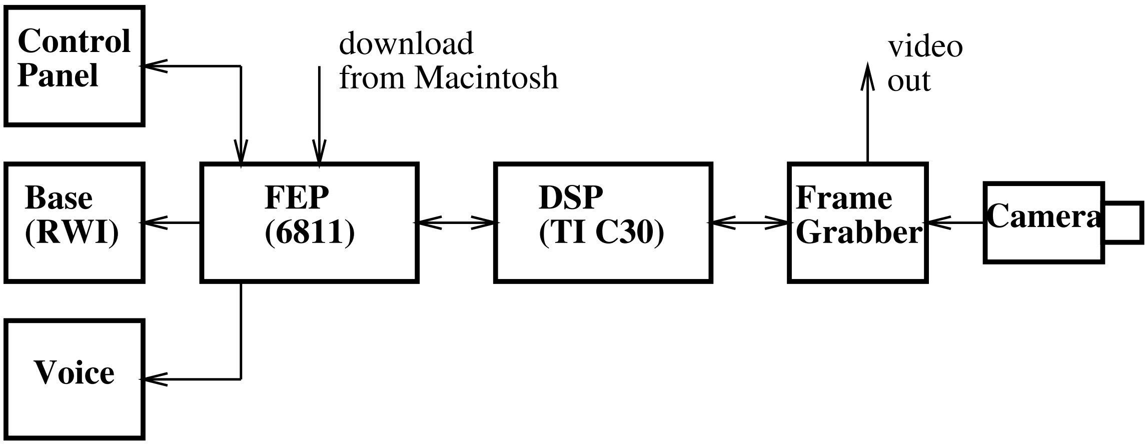


Figure 2.7: Computational components and data-paths.

to arrive. Less than a month was spent on actual assembly. Writing boot roms, down loaders, device drivers, and other systems software did take several months however.

The robot's hardware resources and physical layout are shown in figure 2.6. The robot's principal components are a commercial omni-directional robot base (Real World Interface model B12), a 32 bit digital signal processor (DSP) based on the Texas Instruments TMS320C30, with 65K words of SRAM (Pentek model 4283), a frame digitizing board (Data Translation model 1452), a black-and-white surveillance camera (Chinon CX-101), and a microcontroller front-end (Motorola MC68HC11) which connects the DSP to various low-speed peripherals (voice-synthesizer, input switches, LCD display).

The robot base contains motors, batteries, shaft encoders, a microcontroller to drive the motors, and a standard 9600 baud serial port over which to send commands to the microcontroller. The base has independent motors for driving forward and turning in place. Each motor can be controlled in force, velocity, or position space. The controller can also report the current position or velocity of either motor. The voice synthesizer accepts standard English text and converts it first to phonemes and then to audio.

Nearly all the interesting processing is done on board the DSP. The DSP is roughly a 16MIP machine. The DSP reads images from the frame grabber, performs all visual processing, and makes all control decisions. The DSP and the frame grabber are linked by a VMEbus. The camera generates the image at a resolution of roughly 350×250 pixels, but converts the image to a standard RS-170 analog video signal. The frame grabber then samples and digitizes the analog video signal at a resolution of 512×512 pixels. Since this is more data than the DSP needs or can possibly process, the DSP subsamples the image at a resolution of 64×48 pixels. Internal limitations of the frame grabber and VMEbus limit the frame rate to 15Hz. The camera is fixed in place so the robot cannot look around corners without pausing to turn its entire body.

With the exceptions described below, the 6811 microcontroller, referred to as the FEP (Front End Processor), is used only to interface the DSP to its peripherals. The DSP communicates with the 6811 over a built-in high-speed serial port. The 6811 does not have hardware to speak the DSP's serial protocol, so the 6811 implements the serial protocol in software. The DSP controls the base and other peripherals by sending a byte stream to the 6811 over the serial line. The 6811 forwards the bytes to the peripherals. Escape codes in the byte stream allow the DSP to switch from one peripheral to another. Other escape codes cause the 6811 to transmit the status of the front panel switches and the base's rotational odometer to the DSP.

The FEP only does two intelligent things. First, it parses the status output of the base's serial port to find reports of the rotate motor's current position. It converts these reports from hexadecimal to binary and sends them to the DSP upon request. This was done because the DSP only has one DMA controller and so cannot read and write its link to the FEP without taking an interrupt for every byte. To have the FEP forward every character written by the base to the DSP would have slowed the DSP down. It also would have required more complicated device drivers.

The other non-trivial task of the FEP is the polling of the bump switches. The FEP constantly polls the bump switches to check whether the robot has hit something. When the robot hits something, the FEP stops its normal activities and sends a sequence of commands to the base to reverse direction and turn away from the bumper that was hit. When the bumper deactivates, the FEP returns to its normal activity.

The program running on the DSP is approximately 1500 lines of Scheme code, plus drivers and data. The load image for the DSP is under 7K words (28K bytes). The program uses 5 image buffers, each of which is 3K words in size (64×48 pixels), so the total scratch space is less than 20K words (80K bytes).

2.6 Other visually guided mobile robots

A number of mobile robots have been built which use vision for navigation. Early systems tended to use vision for all navigation, but more recent systems have often used sonar for obstacle avoidance, and have relied on vision only for localizing themselves within a map, or for detecting a road or other path.

2.6.1 Systems with geometric maps

Many of these systems used vision to build up detailed geometric maps of the environments. The earliest of these was certainly Shakey, see Nilsson [76]. Early visions of Shakey used vision to construct grid-based maps of its environment, which was restricted to contain matte polyhedra.

The Stanford Cart (see Moravec [74]), used a novel nine-eyed stereo algorithm to build and maintain maps of the locations of feature points in its immediate surroundings. It then used the maps to plan safe paths around the features. The system worked surprisingly well, but unfortunately it required 15 minutes per frame to perform its computations because of the limited computing power available. It therefore had serious problems with environmental dynamics such as shadows moving with the sun.

Thorpe's FIDO system [100] used stereo vision to build grid-based local models of its environment. The robot used the model to plan and execute a safe path along a sidewalk.

Kriegman, Triendl, and Binford's Mobi system recognized hallways, walls, and doors in a two dimensional world-model [59][60]. The world model was built as the robot moved through its environment. The robot used 1D stereo to find the positions of strong vertical edges in the environment and tracked them over time using a Kalman filter. The system was able to achieve a cycle time on the order of ten seconds per frame.

Ayache and Faugeras [7], and later Kosaka and Kak [57] developed systems which used Kalman filtering with full 2D stereo to track the positions of vertical edges in a map and localize the robot's position. Kosaka and Kak were able to reasonable speeds using only a standard 16MIP workstation. Part of the efficiency of their system is due to the fact that it searches for edges only where its model says they should be. While this gives their system a large performance boost, it also means that it must rely on sonar for obstacle avoidance.

Braunegg's MARVEL system [18] used stereo data to build and maintain grid-based representations of its environment and determine its location by matching sensor data to the model. The most interesting thing about the system was its ability to tolerate slow changes in the environment by gradually mutating its model.

2.6.2 Non-geometric systems

Kortenkamp *et al.* describe a system for navigation using vision to match landmarks described by low resolution stereograms [56]. The system detects landmarks (corridor intersections and doorways) using a sonar-based algorithm, then distinguishes them using vision. The system navigates using the sort of topological map popularized by Kuipers and Byun [61] and Mataric [69].

Nelson [75] describes a novel system for navigation using a visual associative memory. The memory is trained to associate proper actions with overhead views from a planar world. The system can return to a home position by repeatedly looking up the proper action in the memory and executing it.

Engelson and McDermott describe a system for recognizing places using easily computed image signatures, rather than complicated models [37]. The signatures are convenient hash functions sufficient to distinguish views of different places.

A number of purely reactive systems have been developed. Bellutta *et al.* developed a corridor follower which estimates the position of the vanishing point in the image and servos on it to align with the corridor [16]. Storjohan *et al.* [95] developed a simple stereo-based system for obstacle avoidance based on back-projection of images. The system is given the correct correspondence between pixels in the two images for the case where both cameras are viewing a floor with no objects on it. The system warps the left image onto the right image, and compares them. If the space in front of the robot is unoccupied, them the right image and the warped left image should be identical. If they are not, then the offending pixels must contain obstacles. Coombs and Roberts have described a system which uses the presence of optic flow in the image to avoid obstacles [29]. Horswill built systems for following moving objects and following corridors using monocular vision and knowledge of the appearance of the background [52].

2.6.3 Outdoor road following

A large segment of the work in visual navigation is devoted to the problem of outdoor road following. A great deal of early road following work involved the construction of explicit 3D models of the road. A large amount of work has been done on recovering the three dimensional structure of visible road fragments from a single monocular view. These systems use the extracted contours of the road, together with some set of a priori constraints of road shape to recover the shape of the road. Waxman, LeMoingne, Davis, Liang, and Siddalingaiah describe a system for reconstructing road geometry by computing vanishing points of edge segments [111]. Turk, Morgenthaler, Gremban, and Marra used a simpler system based on the assumption that the vehicle and the visible portion of the road rested on the same flat plane [105]. This was called the *flat-earth* model. The flat-earth model allowed much faster processing and was sufficient for simple roads. Later, they substituted the constraint that the road have constant width for the constraint that the road lie in a plane. This allowed the system to handle roads which rose and fell and so it was termed the *hill and dale* model. Because the hill and dale model cannot account for curved roads, DeMenthon proposed an algorithm based on the *zero-bank* constraint which allows both hills and turns but does no allow the road to bank (turn about its own axis) [33].

Much of the work in outdoor road following has been done at Carnegie Mellon. Early work at CMU was done by Wallace, et. al. [110][109][108]. They use a sensor-based algorithm driven in image coordinates for motor control. They take the approach of extracting road edges rather than segmenting the road and using the slicing technique however. They also make extensive use of color information [108]. Using the CMU Warp parallel processor, a 10-100 MIPS floating-point processor optimized for low-level vision, they have reported speeds up to 1.08 km/hour using a servo-loop time of one frame every three seconds. More recently, Crisman [30] implemented a color-based road tracker which was able to properly distinguish road pixels from other pixels, even in the presence of complicated shadows. Pomerleau [81] has described a neural network that efficiently learns to follow roads using low resolution images.

Arkin has reported an architecture based on the schema concept [6]. A schema is a description of what action is appropriate to a given situation, similar to my notion of a tactical routine. Arkin defined a number of motor-schemas for moving along a path and avoiding obstacles which, when run concurrently, were able to navigate about the UMass campus.

Dickmanns *et al.* [34] has described a number of road following systems which can drive on the autobahn at speeds of up to 100km/hour. The systems use Kalman filtering to efficiently search for road edges within small windows of the image. By using multiple processors, their system is able to process images at 25 frames per second.

Chapter 3

Lightweight vision

In this chapter I will argue that vision can be very cheap and suggest some general techniques for simplifying visual processing. This is not an argument that all visual tasks can be solved cheaply. My goal is to convince the reader, particularly the reader who is not a vision researcher, that cheap real-time vision systems are feasible for a variety of tasks. This is, therefore, a theory of how to build task-specific vision systems as cheaply as possible, not a theory of how the human vision system works or of how to build a general programmable vision system. Nevertheless, the issues raised in building task specific vision systems overlap a great deal with recent discussions on the nature of general vision systems.

3.1 Background

Much of the previous work in vision has focused on the construction of modules of a hypothesized general vision system. Early work viewed vision as a domain-independent system for creating monolithic models of the outside world. Proposals vary as to the nature of the models, and the processing performed to create them, but the general approach is to use a series of transformations of the sensory input to move from what I will call the "surface structure" of the input to its "deep structure." I borrow the terms from Chomsky [27], who used them to refer to different levels of structure within sentences. I will use them as relative terms. For example, one might assume that a vision system takes images, transforms them into edge maps, then into depth maps via stereo matching, and finally into collections of geometric descriptions of individual objects. One often thinks of the objects as being "what's really out there," and of the image as being only a shadow; that the objects are the true reality and the images are mere appearance. Thus the object descriptions capture "what's really out there" better than the images, and so we can think of the object descriptions as the "deep structure" which is buried beneath the surface of the images.

In vision, the deep structure people have traditionally tried to extract is the detailed geometry of the environment (see Aloimonos and Rosenfeld [5] for a historical survey, or Marr [68] or Feldman [38] for examples of specific proposals). This is sometimes referred to as the "reconstruction approach." (see Aloimonos and Rosenfeld [5]). The approach has a number of appealing features. First, a complete description of the geometry of the environment, suitably annotated with other surface information such as color, seems to be a convenient form from which to compute any information about the environment which may be required. Another appealing characteristic is that is it fully domain independent, in the sense that any information needed about the environment can be derived from the model, provided the model is sufficiently detailed.

3.1.1 Problems with reconstruction

There are a number of problems with the reconstruction model of vision. The criticism that is the easiest to make and the most difficult to assess, is that reconstruction is hard to do from an engineering standpoint. Formally, it is an "ill-posed" inverse problem (Poggio and Torre [80]): the projection process loses information, and so inverting it requires additional assumptions such as smoothness of surfaces. These assumptions are enforced *e.g.* by requiring the spatial derivatives of the image or depth map to be small. This usually reduces the reconstruction problem to some form of constrained optimization in a high-dimensional space (see Poggio and Torre [80] or Horn [46] for discussions of general techniques for optimization in reconstruction). The resulting equations are frequently non-linear and so tend to be unstable (see Aloimonos and Rosenfeld [5]). Instability means that estimates of surface structure are highly sensitive to noise in image measurements. Estimates are also highly sensitive to deviations from the idealized smooth surface. Image discontinuities are a particular problem since discontinuities are the ultimate non-smoothness (see again Aloimonos and Rosenfeld, *ibid.*). To date, the reconstruction model of vision has never been fully instantiated, although many individual modules have been built with varying degrees of success. This does not prove that it *cannot* be done however.

A deeper criticism of reconstruction is that it is, at best, a partial theory. As an agent goes about its life, it will need to assess different aspects of its immediate situation. This is rather like answering a series of questions: *Is something about to attack me? What direction is the hallway? Are there any free seats here?* These questions will vary from moment to moment and from situation to situation. The agent will never need to answer the question *what is the monolithic model of the current state of the environment?* other than to use that model to answer some other question about its immediate situation. The important questions are always task-dependent and quite specific. Any agent that can answer the right questions at the right times will be successful, regardless of the organization of its perceptual system.

One of the major attractions of the reconstruction theory was that it gave us a domain-independent theory of visual processing. However, the information an agent needs from the environment is always domain-dependent. Thus reconstruction cannot be a complete theory of vision, but rather a claim that the processing of domain-dependent queries should begin by transforming sensor data into a canonical form: a single domain-independent representation which allows efficient processing of specific queries. The validity of this claim rests on three engineering issues:

1. whether the use of a canonical form really does simplify query processing,
2. whether the canonical form that makes processing easiest is a surface reconstruction, and
3. whether the savings accrued by using the canonical form are sufficient to justify the processing required to build the canonical form.

These issues are hard to evaluate. It is likely that they will have to be answered empirically. There are reasons to be skeptical that building a canonical form will be the simplest solution, however. General experience with representation indicates that expressiveness and explicitness trade off with one another in representation languages (see Levesque and Brachman [63] for a general discussion of this phenomenon in the context of knowledge representation languages). The first thing which is taught in many AI classes is the idea that a good representation should make the information important to a task explicit, and nothing else. If this is to be taken seriously, then a single canonical representation for all tasks is unlikely.

If it is hard to choose a single representation for all processing, it is equally hard to choose a single algorithm for constructing the representation. For any given problem, such as edge detection, there will tend to be many different algorithms, each of which makes different performance trade offs. One edge detector may be optimized to localize an edge as well as possible, at the cost of speed and accuracy in detecting the edge in the first place. Another edge detector may do the opposite. Again, it is unlikely that there will be a single approach which will give the right performance for all possible high level tasks.

All these problems can be answered by diluting reconstructionism. Indeed, few people ever believed the reconstruction theory in its most absolute form. It is almost certainly the case that some sort of shared intermediate processing is needed to build a system for answering a wide range of queries (see section 3.2.2, below). Such intermediate processing need not involve building a single monolithic representation, nor need it involve a fixed set of processes.

It should be noted that nothing I have said here applies if the task facing the designer is *to do reconstruction*; if one is trying to build digital terrain maps from satellite imagery, then one must, by definition, do reconstruction.

3.1.2 Active vision and task-based vision

Over the years, a number of researchers have focused on how agents can use the structure both of their task and of their own activity to simplify visual processing. Gibson's theory of direct perception [42][41] was based on the notion that the an agent's physiology was designed by nature to "resonate" with the particular information inherent in its environment. Cutting has further developed Gibson's theory of direct perception to take into account the agent's role of choosing which parts of the stimulus to attend to ("directed perception," see Cutting [31]).

Recently, a great number of machine vision researchers researchers have proposed various approaches to "active" and/or "task-based" vision. Bajczy [8] emphasized the use of active sensing strategies such as active focusing and zooming. In my master's thesis [52], I argued for basing the design of vision systems on concrete tasks. This chapter is an elaboration of that work. Ballard [9] and the vision group at Rochester University have developed an approach which they call "animate vision." They give many examples of how gaze control, agent motion, and so on can be used to simplify processing. Ikeuchi [53] has proposed an approach he calls "task oriented vision." Ikeuchi is in the process of developing a "vision compiler" which can generate custom vision applications given descriptions of visual tasks. Finally, Aloimonos [4] has developed an approach he calls "purposive and qualitative active vision," which stresses many of these same themes.

These proposals overlap a great deal. A common feature is the notion that visual machinery can be greatly simplified by computing only the information needed by the agent to perform its immediate task(s). Another common feature of these new proposals is the explicit use of agent dynamics to simplify visual processing. The earliest use of dynamics that I know of is Gibson's *The Senses Considered as Perceptual Systems* [41]. The earliest explicit use in the machine vision community of which I am aware is Bandopadhay *et al.* [10] who used active tracking of a feature point to simplify the equations for structure-from-motion. Perhaps the most systematic use of dynamics is in Dickmanns' Dynamic Vision methodology [34], in which vision is viewed as an optimal estimation problem in space-time. Dickmanns uses techniques from control theory and optimal estimation theory (*e.g.* Kalman filtering) to recover metric scene parameters.

The final common feature is a tendency to use qualitative information rather than metric information when possible. Qualitative information often has the twin advantages of being easier to extract from images and of being more numerically stable than metric information. The earliest proposal for the use of qualitative information of which I am aware is Thompson and Kearney [98].

3.2 Building lightweight vision systems

Often, we want to build the simplest possible system which can solve a given task in a given environment. In general, a given (task, environment) pair can be solved by a broad range of mechanisms with a broad range of cost and performance parameters. The better the designer's understanding of this range of mechanisms, the better she is able to choose the mechanism that best suits her needs. Anything which can broaden the range of mechanisms that could solve the problem is therefore a potential resource for the designer.

3.2.1 Resources for simplifying vision

If an agent need only compute the information it needs to perform its task, then we can treat that task as a resource which can be used to simplify visual processing. By "task" here, I mean the high level task of the agent. The task specifies, albeit loosely and indirectly, what information is required to perform it. It specifies it only loosely and indirectly because a task such as obstacle avoidance does not require any particular representation of space, obstacles, or the distance to obstacles, but rather places a set of operational constraints on which representations are sufficient for the task. One might use a depth map, or a Cartesian elevation map, or simply the distance and direction of the nearest obstacle. The units of measurement might be feet, meters, or some other unknown but repeatable system. This looseness should be viewed as a positive feature, not as an ambiguity in need of clarification through further task specification. The fact that the task underdetermines the choice of representation means that the designer has a wide range of choices available, some of which may be very efficient.

The task also provides the designer with concrete performance constraints. Again, these constraints are a good thing. Performance constraints not only tell the designer what is required, but also what is *not required*. Without concrete performance constraints, the designer can never know when to stop: since the system will never have perfect performance, there will be room for endless improvement, but this additional performance may be to no good end. More importantly, there is no single aspect of a system which corresponds to performance. Performance has many parameters. These parameters trade off with one another and so cannot be optimized simultaneously. The performance constraints of a task allow the designer to make intelligent choices about which aspects of performance to optimize and which to sacrifice.

One particularly important performance trade off is the relation between spatial sampling rate and temporal sampling rate. For a fixed amount of computational power, the designer cannot increase one without decreasing the other. Since the temporal sampling rate places a lower bound on response time, this can be a critical trade off. Many researchers have traded temporal resolution for spatial resolution.

Increasing spatial resolution can increase geometric accuracy, and so (one hopes) increase the general reliability of the agent. Unfortunately, this can lead to a vicious circle. Doubling the linear spatial resolution requires reducing the temporal sampling rate by at least a factor of four. If the agent makes control decisions at the same rate that it gets perceptual data, then the bandwidth of the control system will also have to drop by a factor of four. The agent will have to wait four times as long to find out if it made a mistake. If mistakes need to be corrected promptly, then the agent will also have less time in which to recover, and so mistakes become a much graver issue. Usually, the solution to this problem is to think very hard about a control decision before making it. But that means that the agent needs very accurate and detailed perceptual data about its environment, which means even more resolution, and so on.

High spatial resolution is needed for good geometric accuracy[1], but geometric accuracy is not always needed to perform the task. All the processing in Polly is performed on 64×48 images (or 16×12 for matching landmarks). Using these images, it would be difficult to estimate the robot's position relative to the walls with any precision. Fortunately, Polly's navigation task does not require it to know its exact position relative to the walls, only (1) whether it is moving roughly parallel to the walls and (2) whether it is too close to one of them (see chapter 9 for more discussion). Both of these can be checked efficiently using low resolution images.

The correlations between the deep structure of objects in the world and the surface structure of the image are another useful resource for simplifying vision. The orientation of a corridor is a property of its geometric structure, and so one way of finding it would be to determine the geometric structure of the corridor and then to extract the orientation from the geometric model. The geometric structure of the 3D world is not reflected in any simple manner in the structure of individual images. In the case of the corridor, however, its orientation happens to coincide with the location of the vanishing point of the corridor edges in the image. The vanishing point is a purely 2D notion which is only present in the (surface) image structure, but in this case, it happens to be correlated with the (deep) 3D structure of the corridor. Since the vanishing point is easily computed from the image, we can easily compute the orientation of the corridor.

The intended environment of the agent is another important resource for the designer. To survive in a wide range of environments, an agent has to be prepared for the worst cases of all environments. This means tighter performance constraints and larger information requirements. By restricting the range of environments in which the agent is to operate, we can often relax these constraints and use simpler systems. Surface structure correlations often hold only in restricted classes of environments. So, again, restricting the range of environments for an agent can allow optimizations

[1] It may also be needed in other cases such as when the only features in the scene are very very small.

which would not otherwise be possible (see chapter 5 for more discussion).

3.2.2 Distributed representation and mediation

One of the principle features of the recent work in active and task-based vision, and in many parts of AI in general, has been the move away from central mediating representations, such as monolithic models, toward more distributed representations in which different parts of the problem are solved by different computations which are largely independent (see Horswill and Brooks [51], Aloimonos [4], and Brooks[20]). These computations are often performed in parallel. One of the fears surrounding this practice is that a complicated agent which performs a wide range of tasks will need huge numbers of parallel processes. The issue of parallelism is outside the scope of this work, but the issue of distributed representation is very important here.

Distributed representation and central mediating representations have complementary strengths. One advantage of using separate computations and representations to perform separate tasks is that it relaxes design constraints. A subsystem that computes two pieces of information may have to satisfy the agent's performance constraints (speed, accuracy, and so on) for both pieces of information simultaneously. Thus, solving the problems separately may be much easier than solving them simultaneously. Sometimes, it is useful to use multiple computations to perform a single task. If the performance constraints of the task are too tight for any particular computation, then a set of independent systems that make different performance trade offs can accomplish the task reliably. For example, one system can give an approximate answer immediately, while another gives a more accurate answer later. Also, two systems with independent failure modes can be used together to perform a task reliably. Polly can align itself with the corridor both by finding the vanishing point of the corridor and by turning so as to balance the distance to each wall. If the vanishing point computation fails, the wall balancing algorithm compensates until the vanishing point computation recovers.

On the other hand, many tasks may have similar performance constraints and may perform the same intermediate steps. In that case, there is no reason not to share those intermediate steps. The use of mediation thus depends on the mix of tasks which the agent needs to perform, and the compatibility of their performance constraints. In my work, I have tended to build independent systems for extracting different pieces of information, and then fold together similar computations afterward.

3.3 Other related work

In recent years, a number of active and task-based vision systems have been built. A common task is the tracking of moving objects. Coombs [22] implemented a system for fixating and tracking objects using a stereo eye/head system. The system used disparity filtering to localize the object. Woodfill and Zabih [115] used optical flow to segment and track a moving object using monocular input. Horswill [52] used environmental constraints to segment and track a moving object. Aloimonos [4] describes a system, Medusa, which performs a number of tasks, including tracking.

Other researchers have built vision systems designed to extract useful high-level information without building detailed depth maps. Aloimonos [4] describes a number of motion algorithms which recover information useful for navigation without having to solve the general structure-from-motion problem. Nishihara [77] describes a minimalist approach to early vision, particularly stereo. Swain [96] describes a system for recognizing colored objects without using any geometric information whatsoever. Various researchers have developed systems for directly detecting occluding contours, without first measuring depth (see Mutch *et al.* [99], Spoerri [94], Toh and Forrest [103], and Wixson [114]). Horswill [49] used disparity filtering to implement proximity detection from stereo images. Aloimonos' Medusa system extracts a number of useful pieces of high-level information such as the presence of moving objects, the direction of translation, and looming [4].

Tsotsos has argued that task-based recognition is simpler than bottom-up recognition from a formal complexity standpoint [104]. Unfortunately, the proof is quite specific to the particular formalizations of the task-based and bottom-up problems. The former amounts to template matching when image position is not given and occlusion is not present, the latter to template matching when when the pose is given but occlusion is allowed. In the proof, it is the presence of occlusion which drives the complexity of the bottom-up approach. The results also depend on the requirement that the templates must match using both a correlation metric and a difference metric. If only one metric is used then, again, the complexity result falls.

One of the consequences of abandoning the model of vision as builder of monolithic models is that resource limitations are introduced into sensing. A robot can only point its camera in one direction at a time, the stereo system can only fixate one depth plane at a time, the color histogram unit can only process one task at a time, and so on. These computational and physical resources must be allocated and reallocated continually by the agent. The result is that sensing becomes a series of actions and so the traditional problems of action–planning, reaction, execution monitoring, management of multiple conflicting goals—are recapitulated for perception. This problem is generally referred to as the problem of the control of selective perception. It has recently spawned a vast literature (see the collections [32, section XI] and [91]). Approaches range from decision-theoretic models (see Rimey [101]) to full-blown symbolic planning (see Pryor and Collins [82]).

Visual routines are a popular framework for discussing these problems. The notion of visual routines is due to Ullman [106], who proposed the idea of a visual routine processor (VRP), which acts as a co-processor for higher levels of processing, and which has an instruction set of visual operations. To Ullman, visual routines were patterns of processing implemented by sequences of operations in the same manner as subroutines are implemented series of normal CPU instructions. Agre and Chapman modified and popularized the model within the greater AI community [3]. Agre and Chapman implemented a simulated VRP which they used for high-level vision. The control inputs of the VRP were tied directly to the central system and were treated in roughly the same way as any other effector. For Agre and Chapman, visual routines were not routines in the sense of "subroutines," but in Agre's sense of common patterns of interaction between agent and environment [2]. Later, Chapman implemented a second simulated VRP for his system, Sonja [26]. Chapman proposed this system, the Sonja Intermediate Vision System or SIVS, as a provisional computational model of biological intermediate vision [24].

Reece and Shafer implemented their own simulated VRP which they used to model active vision for driving [83]. Since then, the use of the term has broadened considerably, in effect, to include any sort of co-processor architecture. Swain [97] discusses how a number of existing active vision systems can be thought of as "active visual routines," but in this case, there is no visual routine processor!

3.4 Summary

Very general vision problems, such as building geometric models of arbitrary environments, can be very difficult and expensive to build. However, we can often build very simple and effective vision systems for specific tasks and environments. The structure of the task and the environment are important resources for designing these simple vision systems. As task and environment become more specific, the requirements on the vision system become looser, and so the space of possible mechanisms which can perform the task grows. Often, this space is very broad, and so some mechanism in the space is often very efficient. By making clear what design constraints do and do not hold, we can greatly simplify the design of our vision systems.

I have no general prescription for designing such systems, I have tried to outline the general resources available to the designer for simplifying vision problems. In the following chapters, I will give a number of examples of lightweight vision systems and discuss how they can be understood. With the proper analysis, the insights gained from the design of one system can be extracted and applied to the design of other systems.

Part II

Formal analysis of specialization

Chapter 4

Introduction to part II

4.1 Background

In the last decade AI has seen a surprising number of negative theoretical results. Formal planning has been shown to be computationally intractable, or even undecidable (see Chapman [23]). Many perception problems have been shown to be highly numerically unstable (see Aloimonos and Rosenfeld [5]). In general, most AI problems amount to search of large spaces, be they discrete spaces, as in the case of reasoning, or continuous spaces, as in the problems of inverse optics. Unfortunately, search is fundamentally hard. It is widely believed that no search problem which is at least as difficult as boolean satisfiability can be solved in polynomial time (Cook's NP-completeness result; see Hopcroft and Ullman [45] for an introduction).

This is a problem. If search is formally intractable, then either AI is impossible, as Penrose argues [79], or there must be a loophole somewhere. There are two candidate loopholes. There could be a loophole in search: there might be polynomial-time algorithms for solving search problems in the average case. This might be true if, for example, there were a good way of generating admissible heuristics for A^* in an automated manner. On the other hand, there might be a loophole "in life": it might be that most everyday problems facing agents are dramatically simpler, in some formal sense, than typical general search problems.

4.1.1 Loopholes in life

This loophole-in-life case deserves some elaboration. It does not mean that humans never solve hard problems. People prove theorems and play chess all the time. The question is what fraction of human activity is fundamentally hard and what fraction is fundamentally easy.

Complexity theory makes a distinction between problems and instances of problems. A problem is characterized by a set of instances together with their correct answers. Any single instance, considered as its own problem, is trivially computable. The instance has an answer whether we as programmers happen to know what it is or not. Therefore, there must exist a program that prints that answer, regardless of its input. That program, although boring, correctly solves the instance, and even does so in constant time and space. Single instances are uninteresting from a complexity-theoretic viewpoint.

One difficulty of AI research is that God doesn't tell us what the formal problem specification for life is. AI researchers have to consider specific instances ("scenarios"), infer the complete problem form the instances, and then design algorithms to fit the problems. However, the difficulty of the problems is extremely sensitive to the details of the phrasing of the problem. If two researchers can look at the same set of instances and infer radically different problems from them, then we have a serious methodological problem.

Consider the problem of getting to work in the morning. Most people agree that it has something to do with "navigation," whatever that is. But what is the ontology of navigation? Is navigation a process of getting from one set of Cartesian coordinates to another? Is it a process of getting from one fuzzily-defined area ("your bed") to another another ("around your office")? This choice has a profound impact on the computational difficulty of the problem. AI researchers may agree that agents need to get to work in the morning, but they are likely to violently disagree over whether agents need do Cartesian navigation.

If we believe that there is a loophole in life — that somehow, most real everyday tasks are relatively easy compared to the general case — then we must necessarily have made poor problem formalizations in the past. Our first task then, is to reopen the issue of formalization. We must explore not only different problem solutions, but also different problem *definitions*. Computer Scientists are often loath to do this. Computer scientists like to formally prove that they're right. Since the process of formalization is by definition itself informal, you can never prove that your formalization is correct.

The key question is: what is it about the instances we encounter regularly in our lives that makes them easy to solve? A number of overlapping answers have been proposed. Both Schank [90] and Agre [2] have argued at length that we tend to solve similar or even identical instances over and over, so we can keep recycling old solutions with minor modifications. Such a view means that we don't want to think about the navigation problem, but rather the meta-problem of solving long series of similar navigation problems in minimal amortized time.

A number of authors have stressed the importance of environmental dynamics in explaining intelligent activity. Some authors stress that the world is much simpler than the most general imaginable case (see, for example, Agre [2]). Other authors stress the complexity of the environment, arguing that the complexity of the environment allows agents to be simpler (see, for example, Simon [92]). These are not incompatible statements. The former is a statement about which of the imaginable possible worlds an agent must actually live in, whereas the latter is a statement about the presence of simplifying structures within the environment. Both groups agree that there are simplifying structures within the environment which allow agents to be simpler than would otherwise be necessary.

4.1.2 Computational minimalism

One way of understanding the comparative difficulty of two problem formalizations is to compare the simplest possible algorithms for each. Doing so allows us to build natural taxonomies of problems and their solutions. If we treat problems as task/environment pairs, then we can also taxonomize environments in terms of their relative difficulty for specific tasks.

In recent years, many researchers have tried to build the minimal mechanisms for specific task/environment pairs (see, for example, Agre [2], Brooks [20], Chapman [25], Connell [28], and Rosenschein and Kaelbling [88]). Some researchers, such as Connell, use minimalism as an engineering methodology. Agre, on the other hand, treats it largely as a psychological or anthropological methodology. I will focus principally on engineering issues.

From an engineering standpoint, the advantage of minimalism is that small changes in task or environment can greatly simplify the machinery needed to solve it. The disadvantage is that the same sensitivity makes the minimalist systems hard to understand. A system that works in one environment may fail to work in a slightly different environment. Worse, the effort expended in building a system for one environment may tell us nothing about how to build a system to perform the same task in a different environment.

Minimizing mechanism maximizes specialization. If computational minimalism is to succeed then we need a set of theoretical tools for analyzing the specialization of systems to their environments.

4.2 Transformational analysis

We can understand specialization by analyzing it, in the sense of dissecting it, into discrete chunks that can be understood in isolation. We can perform the analysis by treating the specialized system as a transformed, or optimized, version of a general system that performs the same task. Each transformation required to optimize the general system into the specialized system will be dependent on a specific property of the agent's environment. I will call these *computational* properties of the environment, not because the environment is doing computation, but because they have computational ramifications. The result of the analysis is then a series of independent optimizations, each of which simplifies a specific computational subproblem using a specific environment property.

To take a simplistic example, animals often need to detect the presence of other animals in the environment. If the (non-living) objects in the animal's environment are all static then motion in the environment is a cue to the presence of another animal. Consider two animals that are identical except that one detects other animals using shape, color, and smell information and the other simply by treating any motion as another animal. The animals behave identically within that environment and thus are "behaviorally equivalent" given that environment. There are differences however. The animal which uses motion may be more "economical" in some sense. If it were a robot, it might be cheaper to build, or take less power or physical space. The other animal has the advantage that it can potentially operate in environments violating the constraint that inanimate objects never move.

We can view the substitution of a motion-based animal detector for a shape-based one as a transformation of one mechanism into another, a transformation that preserves behavior in exactly the same way that rules of logical inference preserve truth value, substitution of identities in mathematics preserves denotation, or compiler optimizations preserve the input/output behavior of the program being compiled. An important difference however is that the transformation only preserves behavior in environments satisfying the constraint. We can view the constraint as representing a possible structure of environments, namely that inanimate objects do not move, and the transformation as the structure's computational significance[1]. We can characterize environments and compare them by giving the sets of useful constraints that they satisfy. We can characterize and compare agents by giving the sets of constraints that they require. The set of constraints assumed by an agent define the set of environments in which the agent can operate, that is, its "habitat," and for this reason I will refer to these constraints as "habitat constraints."

Application of a series of transformations produces a series of behaviorally equivalent systems making different engineering trade-offs and different assumptions about the world. We can compare behaviorally equivalent systems by giving a series of transformations by which one can be converted to the other. I will call such a series a "derivation" of the one from the other. A particularly useful way to analyze a specialized system S is to give a derivation of the system from a more general system G. Giving such a derivation

- makes explicit what additional assumptions are made by S,
- makes explicit what role those assumptions play in S's normal functioning,
- makes it easier to predict S's performance in novel environments,
- can make clearer how to modify S to operate in different environments.

[1]Or part of its computational significance, if it allows many different useful transformations.

Perhaps more importantly, the steps of the derivation can be reused in the design or analysis of other systems. In addition, if we define suitable notions of equivalence for agents components, then we can apply the analysis recursively to subsystems of S and G.

The general approach is as follows. We first define spaces of possible mechanisms and environments. We then decide what aspects of the behavior of a mechanism make it behaviorally equivalent to another mechanism. Different criteria will be required for different kinds of systems. Given these definitions, we can find transformations between mechanisms which preserve behavioral equivalence, either conditionally or unconditionally. If the transformation yields more efficient mechanisms, then we can view it as an optimization. Important classes of environments can be described in terms of the constraints they satisfy and the optimizations they facilitate. Optimizations can then be reused in the design of future systems.

4.3 Synthesis versus *post hoc* analysis

This work is frequently interpreted as a framework for automatic programming. However, it is intended only as a technique for *post hoc* analysis of existing mechanisms. My claim is that however you build the first mechanism, you should reduce it to a set of reusable lemmas for simplifying other mechanisms. This work does not itself give any rigorous methodology for building the first mechanism. It certainly is not a strong enough framework for doing automatic programming of the first mechanism. One might imagine automatic programming systems that use a stock of preexisting optimization lemmas to simplify new designs. However, the task of doing the original design and analysis is no easier or harder than many other tasks in engineering or mathematics.

4.4 Related work

Relatively little attention has been devoted to environmental specialization in computer science, mostly likely because it is only recently that we have begun to construct computational systems that are closely coupled to natural environments.

In biology, a great deal of attention has been given to the specialization of complete agents to their environments. Cybernetics, the progenitor of artificial intelligence, also focused on agent/environment interactions, although not necessarily on the properties of specific, complex environments [112]. Ideas from these areas are now being applied to artificial intelligence and robotics (see McFarland [70], Paton *et al.* [78]. Meyer and Guillot [71]).

In perceptual psychology, Gibson proposed an "ecological" theory of perception that stressed the role of the environment in forming an agent's perceptions. Gibson argued that the structure of the environment determines a set of invariants in the energy flowing through the environment and that these invariants can be directly picked up by the perceptual apparatus of the organism via a process akin to resonance.

Marr [68] argued that in order to properly understand the operation of a perceptual system (or more generally, of any intelligent system), we must understand the problem it solves at the level of a *computational theory*.[2] The computational theory defines the desired input-output behavior of the perceptual system, along with a set of *constraints* on the possible interpretations of a given input. The constraints were necessary because a single stimulus can usually be generated by an infinite number of possible situations. The virtue of a computational theory is that it abstracts away from the details of an individual mechanism. A single computational theory can be used to explain and unify many different mechanisms that instantiate it. To Marr, the role of the constraints within computational theories was to show how the structure of the environment made interpretation possible at all, not how to make it more efficient. Marr believed that the human visual system was a general mechanism for constructing three dimensional descriptions of the environment and so was relatively unconcerned with understanding how a system could be specialized to take advantage of useful, but unnecessary, properties of the environment. This work extends Marr's ideas by using constraints to explain optimizations at the implementation level.

Most formal models of environments use state-space descriptions of the environment, usually finite-state machines. Rosenschein and Kaelbling used finite state machines to represent both agent and environment (see Rosenschein [86][87], and Rosenschein and Kaelbling [88]). Their formalization allowed specialized mechanisms to be directly synthesized from descriptions of desired behavior and a formalization of the behavior of the environment. The formalization was powerful enough to form the basis of a programming language used to program a real robot. Later, Rosenschein developed a method for synthesizing automata whose internal states had provable correlations to the state of the environment given a set of temporal logic assertions about the dynamics of the environment. Donald and Jennings [36] use a geometric, but similar, approach for constructing virtual sensors.

Wilson [113] has specifically proposed the classification of simulated environments based on the types of mechanisms which can operate successfully within them. Wilson also used a finite state formalization of the environment. He divided environments into three classes based on properties such as determinacy. Todd and Wilson [102] used finite state machines to taxonomize grid worlds for a class of artificial agents created by a genetic algorithm. Littman [64] used FSM models to classify environments for reinforcement learning algorithms. Littman parameterized the complexity of RL agents in terms of the amount of local storage they use and how far into the future the RL algorithm looks. He then empirically classified environments by the

[2]Marr's actual story is more complicated than this, and used three levels of explanation, not two. See Marr [68].

the minimal parameters that still allowed an optimal control policy to be learned.

There is also an extensive literature on discrete-event dynamic systems (see Košecká [58] for a readable introduction), which also model the environment as a finite state machine, but which assume that transition information (rather than state information) is visible to the agents.

An alternative to the state-machine formalism can be found in the work of Dixon [35]. Dixon derives his semantics from first order logic, in which the world comes individuated into objects and relations, rather than on the state-space methods used here. Dixon's "open" approach also avoids the need to define the environment as a single mathematical structure. Like this work, Dixon's work attempts to formally model the assumptions a system makes about its environment. Dixon's interest however, is on what an individual program means rather than on comparing competing programs.

Several researchers have discussed how time-extended patterns of interaction with the environment (called "dynamics" by Agre [2]) can be used to reduce the computational burden on an agent. Lyons and Hendricks have discussed how to derive and exploit useful dynamics from a formal specification of the environment [67]. They use a uniform formalization of both agent and environment based on process algebra. Using temporal logic, they are able to identify useful dynamics and design reactive behaviors to exploit them. Hammond, Converse, and Grass discuss how new dynamics can be designed into an agent to improve the stability of the agent/environment system [44].

4.5 How to read part II

The top-level claims of this report are that (1) the use of task and environment in the design of special purpose vision systems can lead to dramatically simpler and more robust systems and (2) those systems can be analyzed and understood in a principled manner. Polly is meant to establish the plausibility of the first claim. Part II is meant to establish the plausibility of the second. Part III will apply the techniques of part II to the analysis of Polly.

The central claims of part II are that

1. Behavior-preserving transformations concisely describe the specialization of an agent to its environment.
2. Formal analysis in terms of transformations allows insights from the design of one special-purpose system to be applied to the design of another.

These ideas need not be applied formally; they can be used in everyday engineering problems in a relatively informal manner such as the analysis of the coloring algorithm given in section 1.1.2. Such analyses are useful, if somewhat hand-wavy. The goal of part II is to show that transformational analysis *can* be made formal and rigorous. Doing so requires the use of a fair amount of formalism. The formalisms themselves are not the focus of the work. Transformational analysis is the focus, and the formalisms are one set of possible tools for applying transformational analysis. They were chosen for simplicity of presentation more than theoretical power. The reader is free to adopt my formalisms, use other formalisms (see section 4.4), or to work in an informal manner.

The reader who feels bogged down by the math should feel free to skip it and go on to part III.

Chapter 5

Framework

5.1 Agents, environments, and equivalence

We will assume that we can reasonably separate the world into agent and environment. The world here need not mean the entire physical universe, only that portion of it which is relevant to our analysis. Let $\mathcal{A}$ denote some set of possible agents and $\mathcal{E}$ a set of environments. Each agent/environment pair will form a dynamic system with some behavior. We will also assume some task-specific notion of equivalence over possible behaviors. We will write $(a_1, e_1) \equiv (a_2, e_2)$ to mean that the behavior of a_1 operating in e_1 is equivalent to the behavior of a_2 in e_2. We can then say that two agents are equivalent if they are equivalent in all environments:

$$a_1 \equiv a_2 \quad \text{iff} \quad \forall e_1, e_2.(a_1, e_1) \equiv (a_2, e_2)$$

We will call them *conditionally equivalent* given some environmental constraint C if they are equivalent in all environments satisfying C. We will write this $a_1 \overset{\mathrm{C}}{\equiv} a_2$. Thus

$$a_1 \overset{\mathrm{C}}{\equiv} a_2 \quad \text{iff} \quad \forall e_1, e_2.C(e_1) \wedge C(e_2) \Rightarrow (a_1, e_1) \equiv (a_2, e_2)$$

Often, the designer has a particular behavior that they want the agent to achieve. Then the only useful behavioral distinction is whether the agent "works" or not, and so the $\equiv$ relation will divide the possible behaviors into only two classes, working and not working. Let the *habitat* H_A of agent A be set of environments in which it works. We will often refer to environment constraints as *habitat constraints*, since the habitat can be described as a constraint or conjunction of constraints.

5.2 Specialization as optimization

Suppose we want to understand an agent s that is somehow specialized to its environment. Although s might be more efficient than some more general system g, it may also have a smaller habitat, *i.e.* $H_s \subseteq H_g$. If we can find a sequence of mechanisms s_i and domain constraints C_i, such that

$$g \overset{C_1}{\equiv} s_1 \overset{C_2}{\equiv} s_2 ... \overset{C_n}{\equiv} s$$

then we have that $g \overset{C_1 \cap ... \cap C_n}{\equiv} s$. We can phrase this latter statement in English as: *within the environments that satisfy $C_1 ... C_n$, g and s are behaviorally equivalent–they will work in exactly the same cases.* This lets us express the habitat of s in terms of the habitat of g:

$$H_s \supseteq H_g \cap C_1 \cap ... \cap C_n$$

Note that the left- and right-hand sides are not necessarily equal because there may be situations where S works but g does not. One of the constraints on the right hand side might also be overly strong.

I will call such a sequence of equivalences, in which g is gradually transformed into s, a derivation of s from g, in analogy to the derivations of equations. We will restrict our attention to the case where each derivation step $s_{i-1} \overset{C_i}{\equiv} s_i$ can be seen as the result of applying some general optimizing transformation O_i that preserves equivalence given C_i, *i.e.* for which

$$\begin{aligned} s_i &= O_i(s_{i-1}), \text{ for each } i \text{ and} \\ a &\overset{C_i}{\equiv} O_i(a), \text{ whenever } O_i(a) \text{ is defined} \end{aligned}$$

Exhibiting such a derivation breaks s's specialization into smaller pieces that are easier to understand. It also places the constraints C_i in correspondence with their optimizations O_i, making the computational value of each constraint explicit. Teasing these constraints apart helps predict the performance of the agent in novel environments. If an environment satisfies all the constraints, the agent will work. If it does not, then we know which optimizations will fail, and consequently, which parts of the design to modify. In addition, if we can write a general lemma to the effect that $a \overset{C_i}{\equiv} O_i(a)$, then we can reuse O_i in the design of future systems. Such lemmas may be of greater interest than the actual agents that inspired them.

Note that we can equally well perform a derivation of one subsystem of an agent from another possible subsystem. For that reason, I will often use the term "mechanism" to mean either an agent or one of its subsystems.

5.3 Simple example

Let's apply this framework to the case of a specific kind of feedback control system: the first order control systems with one degree of freedom. Applying the framework requires specifying what it means for two systems to be equivalent.

Suppose we have a first order system with a single degree of freedom x. In this context, "first order" means we have direct control of $\frac{dx}{dt}$. Suppose we want to make x have some desired value. If the system converges to the desired value in finite time, then it is *stable* for that desired value. We will say two control systems are equivalent if they are stable for the same sets of values.[1]

While second order control problems (problems where we only have control of $\frac{d^2x}{dx^2}$) can be difficult, it seems that controlling the first order system should be more or less trivial. Intuitively, all we should have to get right is the sign of the control signal. While this isn't technically true,[2] the control signal can tolerate a wide range of variation and still converge. That means we don't need an accurate estimate of the the error. It turns out that any strictly increasing measure of the error will allow the system to converge, provide that it maps zero error to zero. We can formalize this as an optimization that substitutes uncalibrated estimates for calibrated ones. We will call this "decalibration:"

Theorem 1 *(Control decalibration) Let x be a physical system with one degree of freedom whose rate of change can be directly controlled. Then all control systems whose control laws are of the form*

$$\frac{dx}{dt}(t) = -f(x(t) - x_{set}) \tag{5.1}$$

where $f\colon\mathcal{R}\to\mathcal{R}$ is nondecreasing and $f(x) = 0$ iff $x = 0$, will cause $x(t)$ converge to x_{set} for arbitrary values of x_{set} and so all such control systems are equivalent under our definition.

Proof: Without loss of generality, we will assume that $x_{set} = 0$. We want to show that $\lim_{t\to\infty} x(t) = 0$. This equation must have a unique, continuous solution for a given initial value (see Brauer and Nohel [13], theorem 1.1). Note that x and its derivative must always have opposite sign, except when one is zero, in which case the other must also be zero. Thus x must be strictly decreasing over any interval in which x is strictly positive, and strictly increasing in any interval in which x is strictly negative. Suppose that $x(t_0) = 0$ and that $x(t)$ becomes positive before ever becoming non-positive in some interval $[t_0, t_1]$. Then $x(t_1) > 0$ and x is nonnegative in $[t_0, t_1]$. By the mean value theorem, there must be some $t \in [t_0, t_1]$ for which $x'(t) > 0$, a contradiction. Similarly, x cannot first become negative, and so $x(t)$ must be zero for all $t > t_0$. Thus x can never cross zero.

Since x will stay at zero one it reaches zero, we need only consider the case in which x stays either positive forever or negative forever. Suppose $x(t) > 0$ for all $t > t_0$, and so $x(t)$ is strictly decreasing for all $t > t_0$. Let $\epsilon > 0$. We want to

[1] This equivalence condition is not a necessary choice. One might want to include rate of convergence, maximum speed, or some other condition in the criteria.

[2] The system could converge to an incorrect value.

show that there exists some t_ϵ such that $x(t) < \epsilon$ for all $t > t_\epsilon$. Since x is strictly decreasing, we need only show that $x(t)$ eventually reaches ϵ. Suppose it does not. Then x must always be larger than ϵ, and so x' must always be more negative than $-f(\epsilon)$, meaning that x is bounded above by $x(t_0)-(t-t_0)f(\epsilon)$. But this drops below ϵ, and so x must too. Thus $\lim_{t\to\infty} x(t) = 0$. The limit holds by similar reasoning when x remains negative, and so must hold in general. □

The caveat to this theorem is that we live in a second order universe and so few physical systems can be accurately modeled as being first order systems with zero delay. However, it is often the case that the rate of change of the second order system is quickly and easily measured, while the absolute (or relative) position of the system is much more difficult to measure. An example might be a robot driving in a room. The robot can easily measure its speed by sensing currents or reading shaft encoders, but accurate information about the robot's position relative to obstacles might require the use of a vision system or other sensing modality whose latency is large compared to the accelerations which the motors can produce. In such situations, we can reduce the second order control problem to two first order control problems: one to control velocity using motor torques, and the other to control position by adjusting velocity. Velocity control can often be done fast enough to make the robot look like a first order system to vision. It is still necessary for the visual system to run fast enough to prevent oscillation however.

An important implication of this theorem is that such a system will be insensitive to errors in the calibration of their perceptual systems provided that (1) the system's estimate of the error is still monotonic in the actual error, and (2) the system still recognizes when there is zero error. Calibration can be a major problem for perceptual systems, particularly when the systems are driven around on robots which periodically crash into things, thus getting knocked out of calibration.

Chapter 6

Analysis of simple perceptual systems

In this chapter, we will perform a more detailed analysis of the coloring algorithm given in section 1.1.1. To do this, we need to flesh out the notions of environment, behavior, and behavioral equivalence. Throughout the paper, we will use a state space formalization of the environment. In this section, we will only be concerned with the environment states themselves, not with the possible transitions between them. We will also ignore the internal state of the agent. In section 7, we will add dynamics and internal state.

Let W be the set of possible world states. We will model environments as subsets of W (we will consider other state spaces in section 7.1). Thus $\mathcal{E} = 2^W$. Habitats, which we have defined as sets of environments, will then effectively just be (larger) regions of the state-space themselves. Habitat constraints, constraints over possible habitats, are then also effectively just subsets of W.

Since we are ignoring dynamics and internal state, we will consider only those perceptual systems that give information about the instantaneous world state. Thus a perceptual system is a mechanism that has an identifiable output with an identifiable set of possible states S such that the state of the output is causally determined by the state of the world. Effectively, the perceptual system computes a function from W to S. We will call that function the *information type* that the perceptual system computes. We will say that two perceptual systems are behaviorally equivalent if they compute the same information type. An information type is finite if its range is finite. Note that information types should not be confused with the concept of information as inverse probability used in classical information theory (see Hamming [43]). While the two are certainly compatible, classical information theory is concerned with measuring quantities of information, whereas our concern here is with distinguishing among different kinds of information.

6.1 Derivability and equivalence

Often what is interesting about an information type is what other information types can be computed from it. We will say that one information type $I':W \rightarrow S'$ is *derivable* from another, $I:W \rightarrow S$, if there exists a *derivation function* f for which $I' = f \circ I$. I_1 and I_2 are *equivalent* (written $I_1 \equiv I_2$) if they are interderivable.

The range of an information type is irrelevant to derivability; We can arbitrarily rename the elements of its range without changing what can be derived from it. Thus what really matters is the partition P_I it induces on the world states:

$$P_I = \{A \subseteq W \mid x, y \in A \Leftrightarrow I(x) = I(y)\}$$

The elements of the partition are the maximal sets of world states that are indistinguishable given only I.

Lemma 1 *The following statements are equivalent:*

1. *I_1 and I_2 are equivalent, that is, interderivable.*
2. *X is derivable from I_1 iff it is derivable from I_2, for all X.*
3. *The partitions P_{I_1} and P_{I_2} are identical.*
4. *I_1 and I_2 differ only by a bijection (a 1:1 onto mapping).*

Proof: First, recall that derivability is transitive. Now suppose that (1) holds. Then if X is derivable from I_2 and I_2 is, by (1), derivable from I_1, then X is derivable from I_1. Similarly, if X is derivable from I_1, then it must be derivable from I_2. Thus (1) implies (2). Now assume (2). Since derivability is reflexive, I_1 and I_2 must be interderivable, and so, equivalent. Now suppose that (3) is false. Then there must be a pair of world states w and w' such that either $I_1(w) = I_1(w')$ but $I_2(w) \neq I_2(w')$ or vice versa. Without loss of generality, we may assume that it is I_2 which differs. By interderivability, there is a derivation function f such that $I_2 = f \circ I_1$ and so $f(I_1(w)) \neq f(I_1(w'))$, which contradicts the assumption that $I_1(w) = I_1(w')$. Thus (3) must hold, and so (2) implies (3). Now note that for every information type $I:W \rightarrow S$, there is a trivial bijection between S and P_I given by the rule $s \mapsto I^{-1}(s)$ for all $s \in S$. Since the inverses and compositions of bijections are themselves bijections, (4) must follow from (3). Finally, note that (1) follows trivially from (4) since the 1:1 mapping would itself be the derivation function. □

We will say that I and I' are *conditionally identical given* C (written $I \stackrel{C}{=} I'$) if $I(w) = I'(w)$ for all $w \in C$. Note that $I \stackrel{W}{=} I$ and that $I_1 \stackrel{C_1}{=} I_2$ and $I_2 \stackrel{C_2}{=} I_3$ implies $I_1 \stackrel{C_1 \cap C_2}{=} I_3$. Finally, we will say that two perceptual systems are behaviorally equivalent if they compute the same information type and conditionally equivalent given C if their information types are conditionally identical given C.

6.2 Unconditional equivalence transformations

We will use a single box labeled with an information type I

$$\Rightarrow \boxed{I} \rightarrow$$

to represent a perceptual system that (somehow) computes I. The double arrow is meant to represent a connection to the environment. When we want to expose the internal structure in the system, we will use single arrows to represent connections wholly within the system. Thus

$$\Rightarrow \boxed{I} \rightarrow \boxed{f} \rightarrow \boxed{g} \rightarrow \ldots$$

represents a system which first computes I and then applies the transformations f, g, ... to it. Finally, we will denote predicates with a "?", thus

$$\Rightarrow \boxed{I} \rightarrow \boxed{> T?} \rightarrow$$

denotes a system which outputs true when $I(w) > T$, and false otherwise. These diagrams inherit the associativity of function composition:

$$\Rightarrow \boxed{f \circ I} \rightarrow \boxed{g} \rightarrow \;\equiv\; \Rightarrow \boxed{I} \rightarrow \boxed{f} \rightarrow \boxed{g} \rightarrow \;\equiv\; \Rightarrow \boxed{I} \rightarrow \boxed{g \circ f} \rightarrow$$

and so a simple optimization, which we might call "folding" (after constant-folding in compiler optimization), is the replacement of a series of computations with a single computation:

$$\Rightarrow \boxed{I} \rightarrow \boxed{f} \rightarrow \;\equiv\; \Rightarrow \boxed{f \circ I} \rightarrow$$

While folding is formally trivial, the technique is quite useful in practice. For example, rather than first computing pose information from an image and then running a grasp planner to choose a grasping strategy, one might use the object's 2D appearance to index directly into a table of grasping strategies. To abuse our notation somewhat, we might say that

$$\Rightarrow \boxed{\text{recognize model}} \rightarrow \boxed{\text{compute pose}} \rightarrow \boxed{\text{plan strategy}} \rightarrow$$

could be reduced to

$$\Rightarrow \boxed{\text{recognize view}} \rightarrow \boxed{\text{look up strategy}} \rightarrow$$

One example of an optimizing transformation is what might be called "decalibration." Estimating precise parameters such as depth can be difficult and can require precise sensor calibration. Often all that is done with this information is to compare it to some empirical threshold. For example, we might estimate the distance to an obstacle to decide whether we should swerve around it, or whether it is too late and we must brake to avoid collision. Generally, the designer arbitrarily chooses a threshold or determines it experimentally. In such situations, we can use *any* mechanism that computes distance in any units, provided that we correct the threshold.

Lemma 2 *(Decalibration) For any information type $I{:}W \rightarrow \mathcal{R}$ ($\mathcal{R}$ is the set of real numbers) and any strictly increasing function $f{:}\mathcal{R} \rightarrow \mathcal{R}$,*

$$\Rightarrow \boxed{I} \mapsto \boxed{> T?} \mapsto \quad \equiv \quad \Rightarrow \boxed{f \circ I} \mapsto \boxed{> f(T)?} \mapsto$$

Proof: By associativity, the right hand side is equivalent to

$$\Rightarrow \boxed{I} \mapsto \boxed{(> f(T)?) \circ f} \mapsto$$

and for all x, $f(x) > f(T)$ iff $x > T$, thus $(> f(T)?) \circ f = (> T?)$. $\Box$

Decalibration allows a calibrated mechanism to be replaced with an uncalibrated mechanism, in certain cases.

6.3 Transformations over simple vision systems

The coloring algorithm used image plane height to discriminate depth and a texture detector to find obstacles. In the remainder of this chapter, we will derive sufficient conditions for the validity of these techniques. We will show that image plane height is a strictly increasing function of object depth, provided the object rests on the floor and its projection into the floor is contained within its region of contact with the floor. We will also show that for floors whose surface markings have no spatial frequencies below ω and which are viewed from a distance of at least d, any low pass filter with a passband in the region $(0, d\omega)$ can be used to discriminate between objects and floor.

First, we need to define our coordinate systems, one camera centered, in which the forward direction direction ($\mathbf{z}$) is the axis of projection, and the other body-centered, in which the forward direction ($\mathbf{Z}$) is the direction of motion (see figure 6.1). We will assume that the camera faces forward, but somewhat down, and so the camera- and body-centered frames share their left/right axis, which we will call $\mathbf{X}$. We will call the up/down axes for the camera- and body-centered systems $\mathbf{y}$ and $\mathbf{Y}$, respectively. We will assume that the ground plane lies at $Y = 0$. We will denote the image with range set X by $\mathcal{I}(X)$ so the b/w images are $\mathcal{I}(\mathcal{R})$ and the color images are $\mathcal{I}(\mathcal{R}^3)$.

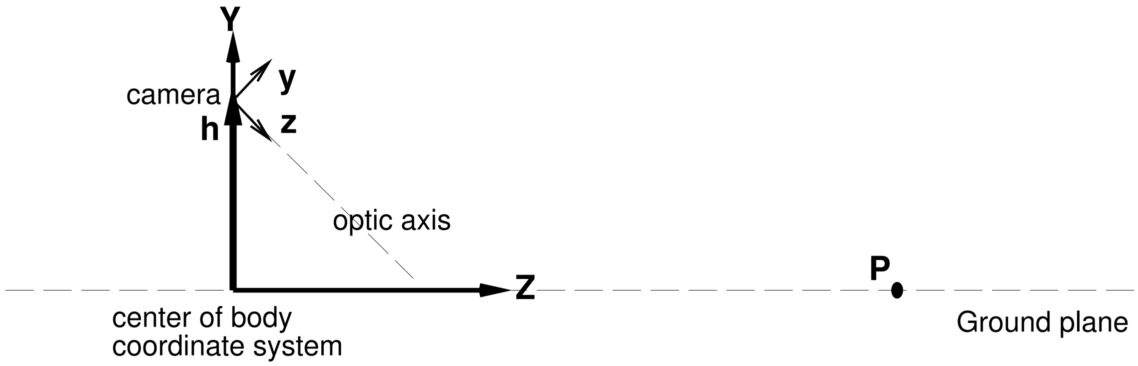


Figure 6.1: A camera viewing a ground plane. The **X** axis (not shown) comes out of the page and is shared by the the camera and body coordinate frames. The body coordinate frame is formed by **X**, **Y** and **Z**, the camera frame, by **X**, **y** and **z**. **z** is also the axis of projection, or optic axis, of the camera. **h** is the height of the camera and **P** is an arbitrary point on the ground plane.

The projection process can be specified in either of these coordinate frames. In camera-centered coordinates, the projection process maps a point (X, y, z) in the world to a point $(fX/z, fy/z)$ on the image plane, where f is the focal length of the lens. In the body-centered coordinate system, projection is best expressed with vector algebra. A point **P** in the world will be projected to the image plane point

$$\mathbf{p} = \frac{f(\mathbf{P} - \mathbf{h})}{\mathbf{z} \cdot (\mathbf{P} - \mathbf{h})}$$

(These are 3D coordinates; the 2D coordinates are obtained by projecting it onto the image plane axes **X** and **y**, yielding the coordinates $(\mathbf{X} \cdot \mathbf{p}, \mathbf{y} \cdot \mathbf{p})$).

Salience functions and figure/ground separation

Let O be a set of objects and $FG_O{:}W \rightarrow \mathcal{I}(\{T, F\})$ ("figure/ground") be the unique information type that, for all world states, returns an image in which pixels are marked "T" if they were imaged in that world state from one of the objects O, otherwise "F." A perceptual system that can compute FG_O within its habitat can distinguish O from the background. FG_O can be arbitrarily difficult (consider the case where O is the set of chameleons or snipers). Fortunately, there are often specific cues that allow objects to be recognized in specific contexts. We will call these cues *salience functions*. An information type is a salience function if it is conditionally equivalent to FG_O given some constraint (a "salience constraint"). The use of such simple, easily computed functions to find particular classes of objects is common both in AI (see Swain [96], Turk *et al.* [105], [30], Horswill and Brooks [51], Woodfill and Zabih [115]) and in the biological world (see Roitblat [85] for a good introduction).

The coloring algorithm uses the texture detector as a salience function. We want to determine what salience constraint is required for a given texture detector. For simplicity, we will restrict ourselves to Fourier-based measures of texture. Effectively, a texture detector examines a small patch of the image. We can approximate the projection of a small patch with

$$(X, y, z) \mapsto (fX/z_0, fy/z_0)$$

where z_0 is the distance to the center of the surface patch. A sufficiently small patch can be treated as a plane with a some local coordinate system (x', y'). Suppose the patch's reflectance varies as a sinusoid with frequency vector $\vec{\omega}$. Then its reflectance R at a point (x', y') on the patch is given by:

$$R(x', y') = \frac{1}{2}\left(\sin\frac{x'}{\omega_x} + \sin\frac{y'}{\omega_y}\right) + \frac{1}{2}$$

If we view the patch:

- from a unit distance,
- through a lens of unit focal length,
- from a direction normal to the patch,
- with the X axis aligned with the x' axis, and
- with even illumination of unit intensity

then the image intensity will simply be

$$I(x, y) = R(x, y)$$

Now consider the effect of changing the viewing conditions. Doubling the distance or halving the focal length halves the size of the image.

$$I(x, y) = R(\frac{x}{2}, \frac{y}{2}) = \frac{1}{2}\left(\sin\frac{x}{2\omega_x} + \sin\frac{y}{2\omega_y}\right) + \frac{1}{2}$$

The image is still a sine wave grating, but its projected frequency is doubled. Rotating the patch by and angle θ around the X axis shrinks the projection along the Y axis by a factor of $\cos\theta$, producing a sine wave of frequency $(\omega_x, \frac{\omega_y}{\cos\theta})$:

$$I(x, y) = R(x, y\cos\theta) = \frac{1}{2}\left(\sin\frac{x}{\omega_x} + \sin\frac{y\cos\theta}{\omega_y}\right) + \frac{1}{2}$$

Rotating the patch about the Y axis shrinks the X axis of the projection. Rotating about the optic axis simply rotates the frequency vector.

Thus a sine wave grating viewed from any position appears as a grating with identical amplitude but with a frequency vector modified by a scaling of its components

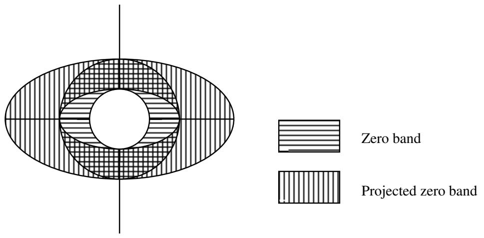


Figure 6.2: The effect of perspective projection on local frequency distributions.

and possibly a rotation. Since the projection process is linear, we can extend this to arbitrary power spectra: the power spectrum of the patch's projection will be the power spectrum of the patch, rotated and stretched along each axis (see figure 6.2). Frequency bands of the patch are transformed into elliptical regions of the frequency domain of its projection. Bounds on the possible viewing conditions yield bounds on how much the frequency bands can be deformed.

The *background texture constraint* (BTC) requires that all surface patches of the background have surface markings whose power spectra are bounded below by ω, that all objects have surface markings with energy below ω, and that no surface in view is closer than d focal lengths, and that the scene is uniformly lit. We have that

Lemma 3 *Any thresholded linear filtering of the image with a passband in the interval* $(0, d\omega)$ *is a salience function given the background texture constraint.*

Proof: By assumption, no patch of the background has energy in the band $(0, \omega)$, but all objects do. By the reasoning above, when any patch, either object or background, is viewed fronto-parallel from distance d, the band $(0, \omega)$ projects to the band $(0, d\omega)$. Thus a patch was imaged from an object iff its projection has energy in this band. But note that increasing the distance or changing the viewing orientation can only increase the size of the projected frequency ellipse. Thus for any distance greater than d and any viewing orientation, a patch will have energy in $(0, d\omega)$ iff it was imaged from an object. Thus a thresholded linear filter is a salience function given BTC. □

The corollary to this is that any thresholded linear filter with passband in $(0, d\omega)$ is conditionally equivalent to a figure/ground system given the background texture constraint.

Depth recovery

Depth can be measured in either a camera-centered or a body-centered coordinate system. We will call these "camera depth" and "body depth," respectively. The camera depth of a point $\mathbf{P}$ is its distance to the image plane, $\mathbf{z} \cdot (\mathbf{P} - \mathbf{h})$. Body depth,

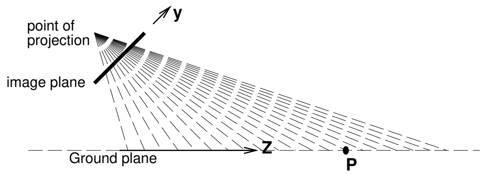


Figure 6.3: Monotonicity of image plane height in body depth. Rays projected from the point of projection to points on the ground plane pass through successively higher points on the image plane as they move to more distant points on the ground plane.

on the other hand, is how far forward the robot can drive before it collides with the point, $\mathbf{Z} \cdot \mathbf{P}$. We will concern ourselves with body depth.

Consider a world of flat objects lying on a ground plane. Then both object points and ground plane points have zero Y coordinates. The points must be linear combinations of $\mathbf{X}$ and $\mathbf{Z}$. Since both $\mathbf{z}$ and $\mathbf{Z}$ are perpendicular to $\mathbf{X}$, the X component of the point will make no contribution either to camera depth or to body depth and we can restrict our attention to the one dimensional case, shown in figure 6.1, of a point $\mathbf{P} = n\mathbf{Z}$. Its body depth is simply n, while its camera depth $\mathbf{z}\cdot(n\mathbf{Z}-\mathbf{h})$ depends on camera placement. We can see by inspection, however, that the camera depth is linear in n and so camera depth and body depth are related by a linear mapping. More surprisingly, image plane height is a strictly increasing function of body depth. This can be seen from figure 6.3. It can also be shown analytically. The image plane height of $\mathbf{P}$ is

$$\mathbf{y} \cdot \left(\frac{f(n\mathbf{Z} - \mathbf{h})}{\mathbf{z} \cdot (n\mathbf{Z} - \mathbf{h})} \right) = \frac{\mathbf{y} \cdot (fn\mathbf{Z} - f\mathbf{h})}{n\mathbf{z} \cdot \mathbf{Z} - \mathbf{z} \cdot \mathbf{h}} = \frac{n\alpha - \delta}{n\beta - \gamma}$$

for $\alpha = f\mathbf{Z} \cdot \mathbf{y}$, $\beta = \mathbf{z} \cdot \mathbf{Z}$, $\gamma = \mathbf{z} \cdot \mathbf{h}$, and $\delta = f\mathbf{h} \cdot \mathbf{y}$. Differentiating with respect to n, we obtain

$$\frac{\alpha(n\beta - \gamma) - \beta(n\alpha - \delta)}{(n\beta - \gamma)^2} = \frac{\beta\delta - \alpha\gamma}{(n\beta - \gamma)^2}$$

When the camera looks forward and $\mathbf{P}$ is in front of the agent, we have that $n, \beta, \delta > 0$, and $\gamma\alpha < 0$, so the derivative is strictly positive.

The *ground plane constraint* (GPC) requires that the camera view a set of the objects O resting on a ground plane G, and that for each $o \in O$, o is completely in view and o's projection in G is its set of points of contact with G.[1] Thus pyramids resting on their bases would satisfy the restriction, but not pyramids resting on their points. Given GPC, we can use least y coordinate as a measure of the depth of the closest object. Let Body-Depth$_O$ be the information type that gives the correct body

[1]Formalizing the notion of contact can be difficult (see for example Fleck [39], chapter 8), but we will treat the notion as primitive, since the particular formalization is unimportant for our purposes.

depth for pixels generated by one of the objects O, or ∞ for pixels generated by the background.

Lemma 4 *Let R be a region of the image. Then* $\min_R \circ$Body-Depth$_O$ *is conditionally equivalent to* $\min\{y : FG_O(x,y)$ *for some* $(x,y) \in R\}$ *given GPC, modulo a strictly increasing function.*

Proof: Note that there can only be one minimal depth, but there can be many minimal-depth object points. However, it must be the case that some contact point (an object point touching the floor) has minimal depth, otherwise there would be an object point whose ground plane projection was not a contact point, a contradiction. Let p be a minimal-depth contact point. We want to show that no object point can have a smaller projected y coordinate than p. Since the y coordinate is invariant with respect to changes in the X coordinate, a point which projects to a lesser y coordinate than p must have either a smaller Z coordinate or a smaller Y coordinate. The first would contradict p's being a minimal-depth point while the latter would place the point below the ground plane. Thus p must have a minimal y projection. We have already shown that for contact points the y projection is strictly increasing in body depth. $\Box$

A trivial corollary to this lemma is that the height of the lowest figure pixel in an image column gives the distance to the nearest object in the direction corresponding to the column.

Chapter 7

Analysis of action selection

In this chapter we will apply transformational techniques to action-selection tasks with the goal of demonstrating a number of formal conditions under which we can reduce deliberative planning systems to reactive systems. We will continue to model the environment as a dynamic system with a known set of possible states. First, we will add actions (state transitions) to the environment, making it a full state-machine. We will then model both deliberative planning systems and reactive systems as variants of the control policies of classical control theory (see Luenberger [65] or Beer [15]). This gives us a uniform vocabulary for expressing both types of systems. We can then examine various formal conditions on the environment that allow simplifications of the control policy (*e.g.* substitution of a reactive policy for a deliberative one)

Again, the focus of chapter is the use of transformational analysis, not the specifics of the notation used below. The notation is needed to establish a framework within which to apply the transformations. The notation used here is largely equivalent to those used by Rosenschein and Kaelbling [88], and by Donald and Jennings [36]. It was chosen for largely for compactness of presentation. The formal trick of externalizing the agent's internal state also turns out to be useful.

7.1 Environments

We will now allow different environments to have different state spaces and will treat actions as mappings from states to states. An environment will then be a state machine $E = (S, A)$ formed of a state space S and a set of actions A, each of which is a mapping from S to S.

For example, consider a robot moving along a corridor with n equally spaced offices labeled 1, 2, and so on. We can formalize this as the environment $\mathcal{Z}_n = (\{0, 1, ..., n-1\}, \{inc_n, dec, i\})$, where i is the identity function, and where inc_n and dec map an integer i to $i+1$ and $i-1$, respectively, with the proviso that $dec(0) = 0$

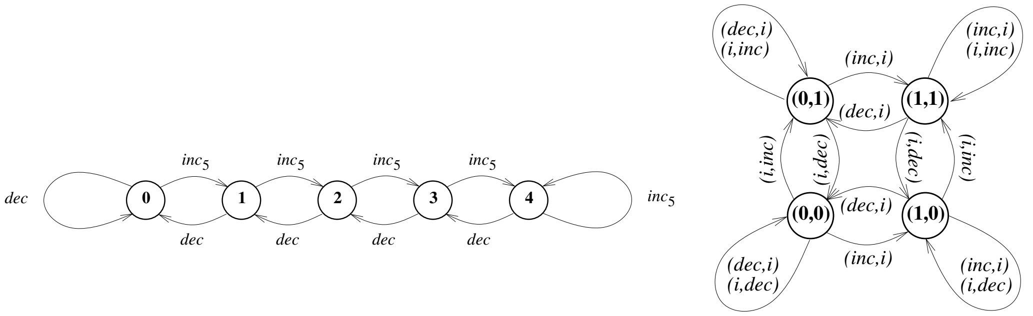


Figure 7.1: The environment Z_5 (left) and and the serial product of $\mathcal{Z}_2$ with itself, expressed as graphs. Function products have been written as pairs, *i.e.* $inc \times i$ is written as (inc, i). Identity actions (i and $i \times i$) have been omitted to reduce clutter.

and $inc_n(n-1) = n-1$ (see figure 7.1). Note that the effect of performing the identity action is to stay in the same state.

7.1.1 Discrete control problems

We will say that a *discrete control problem*, or DCP, is a pair $D = (E, G)$ where E is an environment and G, the goal, is a region of E's state space. The problem of getting to the beginning of the corridor for our robot would be the DCP $(\mathcal{Z}_n, \{0\})$. By abuse of notation, we will also write a DCP as a triple (S, A, G). A finite sequence of actions $a = (a_1, a_2, ..., a_n)$ *solves* D *from initial state* s if $a_n(a_{n-1}(...a_1(s))) \in G$. D em is solvable from s if such a sequence exists. D *is solvable* (in general) if it is solvable from all $s \in S$.

7.1.2 Cartesian products

Often, the state space of the environment is structured into distinct components that can be acted upon independently. The position of the king on a chess board has row and column components, for example. Thus we would like to think of the king-on-a-chess-board environment as being the "product" of the environment $\mathcal{Z}_8$ with itself (since there are eight rows and eight columns), just as $\mathcal{R}^2$ is the Cartesian product of the reals with themselves. However, consider an environment in which a car drives through an 8×8 grid of city blocks. We would also like to think of this environment as being the product of $\mathcal{Z}_8$ with itself. Both the car and the king have 8×8 grids as their state spaces, but the car can only change one of its state components at a time, whereas the king can change both by moving diagonally.

We will therefore distinguish two different Cartesian products of environments, the *parallel product*, which corresponds to the king case, and the *serial product*, which corresponds to the car case. Let the Cartesian product of two functions f and g be $f{\times}g\colon (a,\ b) \mapsto (f(a),\ g(b))$, and let i be the identity function. For two environments $E_1 = (S_1, A_1)$ and $E_2 = (S_2, A_2)$, we will define the parallel product to be

$$E_1 \,\|\, E_2 \;=\; (S_1{\times}S_2, \{a_1{\times}a_2 : a_1 \in A_1, a_2 \in A_2\})$$

and the serial product to be

$$E_1 \rightleftharpoons E_2 \;=\; (S_1{\times}S_2, \{a_1{\times}i : a_1 \in A_1\} \cup \{i{\times}a_2 : a_2 \in A_2\})$$

The products of DCPs are defined in the obvious way:

$$(E_1, G_1) \,\|\, (E_2, G_2) \;=\; (E_1 \,\|\, E_2, G_1{\times}G_2)$$

$$(E_1, G_1) \rightleftharpoons (E_2, G_2) \;=\; (E_1 \rightleftharpoons E_2, G_1{\times}G_2)$$

The state diagram for $\mathcal{Z}_2 \rightleftharpoons \mathcal{Z}_2$ is shown in figure 7.1.

We will say that an environment or DCP is parallel (or serial) *separable* if it is isomorphic to a product of environments or DCPs.

7.1.3 Solvability of separable DCPs

The important property of separable DCPs is that their solutions can be constructed from solutions to their components:

Claim 1 *Let D_1 and D_2 be DCPs. Then $D_1 \rightleftharpoons D_2$ is solvable from state (s_1, s_2) iff D_1 is solvable from s_1 and D_2 is solvable from s_2.*

Proof: Consider a sequence S that solves the product from (s_1, s_2). Let S_1 and S_2 be the sequences of actions from D_1 and D_2, respectively, that together form S, so that if S were the sequence

$$(a{\times}i,\ i{\times}x,\ i{\times}y,\ b{\times}i,\ i{\times}z,\ c{\times}i)$$

then S_1 would be (a, b, c) and S_2 would be (x, y, z). S must leave the product in some goal state (g_1, g_2). By definition, g_1 and g_2 must be goal states of D_1 and D_2 and so S_1 and S_2 must be solution sequences to D_1 and D_2, respectively. Conversely, we can construct a solution sequence to the product from solution sequences for the components. $\square$

The parallel product case is more complicated because the agent must always change both state components. This leaves the agent no way of preserving one solved subproblem while solving another. Consider a "flip-flop" environment $F = (\{0,1\}, \{flip\})$ where $flip(x) = 1{-}x$. F has the property that every state is accessible from every other state. $F \rightleftharpoons F$ also has this property. $F \,\|\, F$ does not however. $F \,\|\, F$

has only one action, which flips both state components at once. Thus only two states are accessible from any given state in $F \| F$, the state itself and its flip. As with the king, the problem is fixed if we add the identity action to F. Then it is possible to leave one component of the product intact, while changing the other. The identity action, while sufficient, is not necessary. A weaker, but still unnecessary, condition is that F have some action that always maps goal states to goal states.

Claim 2 *Let D_1 and D_2 be DCPs. If $D_1 \| D_2$ is solvable from state (s_1, s_2) then D_1 is solvable from s_1 and D_2 is solvable from s_2. The converse is also true if for every goal state of D_1 and D_2, there is an action that maps to another goal state.*

Again, let S be a solution sequence from (s_1, s_2). Now let S_1 and S_2 be the sequences of obtained by taking the first and second components, respectively, of each element of S. Thus, if S is

$$(a{\times}x,\ b{\times}y,\ c{\times}z)$$

then we again have that S_1 is (a, b, c) and S_2 is (x, y, z). Again, S_1 and S_2 are solution sequences for their respective component problems. Similarly, we can form a solution to the product from solutions to the components by combining them element-wise. To do this, the solutions to the components must be of the same length. Without loss of generality, let S_1 be the shorter solution. Since there is always an action to map a goal state to a goal state, we can pad S_1 with actions that will keep D_1 within its goal region. The combination of S_2 and the padded S_1 must then be a solution to the product. □

7.2 Agents

We will assume an agent uses some *policy* to choose actions. A policy p is a mapping from states to actions. We will say that p:

- *generates a state sequence* s_i when $s_{i+1} = (p(s_i))(s_i)$ for all i.
- *generates an action sequence* a_i when it generates s_i and $a_i = p(s_i)$ for all i.
- *solves D from state s* when p generates a solution sequence from s.
- *solves D* when it solves D from all states.
- *solves D and halts* when it solves D and for all $s \in G$, $(p(s))(s) \in G$.

For example, the constant function $p(s) = dec$ is a policy that solves the DCP $(\mathcal{Z}_n, \{0\})$ and halts.

7.2.1 Hidden state and sensors

A policy uses perfect information about the world to choose an action. In real life, agents only have access to sensory information. Let $T\colon S \to X$ be the information type (see section 6) provided by the agent's sensors. The crucial question about T is what information can be derived from it. We will say that an information type is *observable* if it is derivable from T.

To choose actions, we need a mapping not from world states S to A, but from sensor states X to A. We will call such a mapping a *T-policy.* A function p is a T-policy for a DCP D if $p \circ T$ is a policy for D. We will say that p *T-solves* D from a given state if $p \circ T$ solves it, and that p T-solves D (in general) if it T-solves it from any initial state.

7.2.2 Externalization of internal state

We have also assumed that the agent itself has no internal state–that its actions are determined completely by the state of its sensors. In real life, agents generally have internal state. We will model internal state as a form of external (environmental) state with perfect sensors and effectors. Let the *register environment* R_A over an alphabet A be the environment whose state space is A and whose actions are the constant functions over A. We will write the constant function whose value is always a as $\mathcal{C}_a$. The action $\mathcal{C}_a$ "writes" a into the register. We will call $E \parallel R_A$ the *augmentation* of E with the alphabet A. An agent operating in the augmentation can, at each point in time, read the states of E and the register, perform an action in E, and write a new value into the register.

Using external state for internal state is not simply mathematical artifice. Agents can and do use the world as external memory. An agent need only isolate some portion of the world's state (such as the appearance of a sheet of paper) which can be accurately sensed and controlled. Humans do this routinely. Appointment books allow people to keep their plans for the day in the world, rather than in their scarce memory. Bartenders use the position of a glass on the bar to encode what type of drink they intend to mix and how far they are into the mixing (see Beach [14]). For an example of a program that uses external state, see Agre and Horswill [1].

7.3 Progress functions

A *progress function* is a measure of distance to a goal. In particular, a progress function Φ for a DCP $D = (S, A, G)$ is a non-negative function from S to the reals for which

1. Φ is nonnegative, *i.e.* $\Phi(s) \geq 0$ for all s.

2. $\Phi(s) = 0$ iff $s \in G$.
3. For any initial state i from which D is solvable, there exists a solution sequence $S = (a_1, ...a_n)$ along which Φ is strictly decreasing (*i.e.* $\Phi(a_j(...(a_1(i)))) > \Phi(a_{j+1}(a_j(...a_1(i))))$ for all j).

The term "progress function" is taken from the program verification literature, where it refers to functions over the internal state of the program that are used to prove termination of loops. Progress functions are also similar to Liapunov functions (see Luenberger [65]), admissible heuristics (see Barr and Feigenbaum [12], volume 1, chapter II), and artificial potential fields (see Khatib [54] or Latombe [62]).

We will say that a policy p *honors* a non-negative function Φ, if Φ steadily decreases it until it reaches zero, *i.e.* for all states s and some $\epsilon > 0$, either $\Phi((p(s))(s)) < \Phi(s) - \epsilon$ or else $\Phi(s) = \Phi((p(s))(s)) = 0$. A policy that honors Φ can be thought of as doing hill-climbing on Φ and so will run until it reaches a local minimum of Φ. When Φ also happens to be a progress function for the DCP, that local minimum will be a global minimum corresponding to the goal:

Lemma 5 *Let $\Phi: S \rightarrow \mathcal{R}$ be non-negative and let p be a policy for a DCP D that honors Φ. Then p solves D and halts exactly when Φ is a progress function on D.*

Proof: Consider the execution of p from an arbitrary initial state i. On each step, the value of Φ decreases by at least ϵ until it reaches 0, after which it must remain zero. Thus Φ must converge to zero within $\frac{\Phi(i)}{\epsilon}$ steps after which the state of the system is confined to the set $\Phi^{-1}(0)$. We need only show that $\Phi^{-1}(0) \subseteq G$ iff Φ is a progress function for D. If Φ is a progress function $\Phi^{-1}(0) \subseteq G$ holds by definition. To see the converse, suppose $\Phi^{-1}(0) \subseteq G$. We want to show that from every state from which D is solvable, there is a solution sequence that monotonically decreases Φ. The sequence generated by p is a such a sequence. □

Progress functions can be generated directly from policies. The *standard progress function* $\Phi_{p,D}$ on a policy p that solves D is the number of steps in which p solves D from a given state. An important property of product DCPs is that we can construct progress functions for products from progress functions for their components:

Lemma 6 *If Φ_1 is a progress function for D_1 and Φ_2 is a progress function for D_2, then $\Phi: (x, y) \mapsto \Phi_1(x) + \Phi_2(y)$ is a progress function for the serial product of the DCPs.*

Proof: Since $\Phi_1 \geq 0$ and $\Phi_2 \geq 0$, we have that $\Phi \geq 0$. Similarly, Φ must be zero for exactly the goal states of the product. Now suppose the product is solvable from (s_1, s_2). Then there must exist solution sequences for the components that monotonically decrease Φ_1 and Φ_2, respectively. Any combination of these sequences to form a solution to the product must then monotonically decrease Φ, and so Φ must be a progress function for the product. □

Again, the parallel case is more complicated:

Lemma 7 *If Φ_1 is a progress function for D_1 and Φ_2 is a progress function for D_2, and for every goal state of D_1 and D_2 there is an action that maps that state to a goal state, then $\Phi\colon(x,y)\mapsto\Phi_1(x)+\Phi_2(y)$ is a progress function for the parallel product of the two DCPs.*

Proof: Again, we have that $\Phi\geq 0$ and that Φ is zero for exactly the the goal states of the product. Now consider a state (s_1,s_2) from which the product is solvable. There must be solution sequences S_1 and S_2 to the component problems along which Φ_1 and Φ_2, respectively, are strictly decreasing. Without loss of generality, assume that S_1 is the shorter. Of the two solutions. We can pad S_1 and combine the solutions to produce a solution to the product. The padding cannot change the value of Φ_1, and so the value of Φ must be strictly decreasing along the combined solution. $\square$

7.4 Construction of DCP solutions by decomposition

7.4.1 Product DCPs

We now have the tools to construct solutions to product DCPs from the solutions to their components:

Lemma 8 *Let p_1 be a policy which solves D_1 and halts from all states in some set of initial states I_1, and let p_2 be a policy which solves D_2 and halts from all states in I_2. Then the policy*

$$p(x,y)=p_1(x)\times p_2(y)$$

solves $D_1\parallel D_2$ and halts from all states in $I_1\times I_2$. (Note that here we are using the convention of treating p, a function over pairs, as a function over two scalars.)

Lemma 9 *Let p_1 be a policy which solves D_1 from all states in some set of initial states I_1, and let p_2 be a policy which solves D_2 from all states in I_2. Then any policy for which*

$$p(x,y)=p_1(x)\times i \text{ or } i\times p_2(y)$$

and

$$\begin{aligned} y\in G_2, x\notin G_1 &\Rightarrow p(x,y)=p_1(x)\times i\\ x\in G_1, y\notin G_2 &\Rightarrow p(x,y)=i\times p_2(y)\end{aligned}$$

will solve $D_1\rightleftharpoons D_2$ and halt from all states in $I_1\times I_2$.

Proof: We can prove both lemmas using progress functions. Let Φ_{p_1,D_1} and Φ_{p_2,D_2} be the standard progress for p_1 and p_2 on D_1 and D_2, respectively. Their sum must be a progress function for the product. This follows directly for the serial case, and from the fact that p_1 and p_2 halt for the parallel case. Since the policies for both products clearly honor the sum, they must solve their respective products. Note that the constraint given in the second lemma is sufficient, but not necessary. □

7.4.2 Reduction

We can often treat one environment as an abstraction of another; The abstract environment retains some of the fundamental structure of the concrete environment but removes unimportant distinctions between states. An abstract state corresponds to a set of concrete states and abstract actions correspond to complicated sequences of concrete actions.

Let a projection of an environment $E = (S, A)$ into an abstract environment $E' = (S', A')$ be a mapping $\pi \colon S \to S' \cup \{\perp\}$. π gives the abstract state for a given concrete state or else $\perp$ if it has no corresponding abstract state. π^{-1} gives the concrete states corresponding to a given abstract state. For sets of states, we will let $\pi^{-1}(S) = \cup_{s \in S}\ \pi^{-1}(s)$.

We define a π-implementation of an abstract action a' to be a policy that reliably moves from states corresponding to an abstract state s' to states corresponding the abstract state $a'(s')$ *without visiting states corresponding to other abstract states.* Thus for any s' for which $a'(s')$ is defined, the implementation solves the DCP

$$(\pi^{-1}(\{s', \perp, a'(s')\}),\ A,\ \pi^{-1}(a'(s')))$$

Note that we do not require p to stay in $\pi^{-1}(a'(s'))$ upon reaching it.

Given π-implementations $p_{a'}$ of each abstract action a', we can use an abstract policy p' to solve problems in the concrete environment by emulating the abstract actions. We need only look up the abstract state corresponding to our current concrete state, look up the abstract action for the abstract state, and run its implementation. This suggests the policy

$$p(s) = p_{p'(\pi(s))}(s)$$

This concrete policy works by taking the state s, looking up its abstract state $\pi(s)$, computing the proper abstract action $p'(\pi(s))$, and then computing and running the next concrete action in its implementation $p_{p'(\pi(s))}$. Note that since this policy has no internal state, it effectively recomputes the abstract action each time it chooses a concrete action. This is no problem when the concrete environment is in a state that corresponds to an abstract state, but the π-implementations are allowed to visit states that have no abstract state. To handle this problem, it is necessary to add a register to the environment to remember what abstract action is presently being performed. The policy for the augmented environment computes a new abstract action whenever the environment is in a concrete state with a corresponding abstract state. It stores the name of the new abstract action in the register for later use, while also executing it its implementation. When the environment is in a concrete state with no abstract state, it uses the abstract action stored in the register and preserves the value in the register:

Lemma 10 *Let $D = (S, A, G)$, $D' = (S', A', G')$ be DCPs, π be a projection of D into D', and for each action $a' \in A'$, let $p_{a'}$ be a π-implementation of a' in D. If p' is a policy which solves D', then the policy*

$$p(s, a) = \begin{cases} p_a(s) \times \mathcal{C}_a & \text{if } \pi(s) = \perp \\ p_{p'(\pi(s))}(s) \times \mathcal{C}_{p'(\pi(s))} & \text{otherwise} \end{cases}$$

solves the augmentation of D with the alphabet A', from any state in $\pi^{-1}(S')$.

Proof: Let $\Phi_{p',D'}$ be the standard progress function for p' on D' and let $s \in P^{-1}(S')$. Then $\Phi_{p',D'}(\pi(s))$ is the number of abstract actions need to solve the problem from the concrete state s. If $\Phi_{p',D'}(\pi(s)) = 0$, then the problem is already solved, so suppose that p solves the problem from states s for which $\Phi_{p',D'}(\pi(s)) = n$ and consider an s for which $\Phi_{p',D'}(\pi(s)) = n + 1$. The policy p will immediately compute $p'(\pi(s))$, store it into the register. Call this action a'. The policy p will also immediately begin executing $p_{a'}$. Since this policy is a p-implementation of a', the system must reach a state in $\pi^{-1}(a'(\pi(s)))$ in finite time, which is to say that it will reach the next state in D'. By assumption, p' can solve D' from this high level state in n steps, and so p must be able to solve D from s, and so, by induction p solves D for all $s \in P^{-1}(S')$. $\square$

We will say that D is *reducible* to D' if there exists a projection π of D into D' and π-implementations of all of actions in D'. If D is reducible to D' then we can easily convert any solution to D' into a solution to D.

Part III

The design of Polly

Chapter 8

The core visual system

Nearly all perceptual processing in Polly is done by a small group of processes which I will call the core visual system (CVS). The core visual system is responsible for computing depth information, the direction of the corridor, detecting people, detecting motion, and sanity checking of the visual system's inputs. Every 66ms, the CVS processes a 64×48 image and updates a large number of "percepts" (see table 8.1). Most percepts are related to navigation. The CVS is implemented in the Polly code by the procedures `low-level-vision` and `derived-aspects`, which call other procedures to compute the individual percepts.

Figure 8.1 shows the principal components of the CVS. The CVS subsamples the image, smoothes it, and passes it to several parallel computation streams. One stream (in grey) computes a depth map of the scene and compresses the map into a number of scalar values such as distance to the closest object in the robot's path (`center-distance`) and whether there is a left turn in view (`left-turn?`). The computation of the depth map assumes the presence of a single, textureless carpet. The system will underestimate depth when two different textureless carpets abut. The CVS compensates for this problem using a second, parallel, computation to check for carpet boundaries. When it detects a boundary, it instructs the depth map computation to ignore the boundary. The CVS also computes the vanishing point of the lines forming the corridor. The depth map and vanishing point are used by the low level navigation system. A symmetry detector searches the image for tall skinny symmetric regions. These regions typically correspond to people's legs. The symmetry detector reports the direction of the most symmetric region (`person-direction`) and a binary value indicating whether the size and symmetry of the region are strong enough to be a person (`person?`). Finally, a motion unit is used to detect foot gestures. A pair of nod-of-the-head detectors were implemented and tested, but could not be used on the robot because of hardware problems (see 8.5).

Table 8.1 shows the suite of high level percepts generated by the CVS. Most

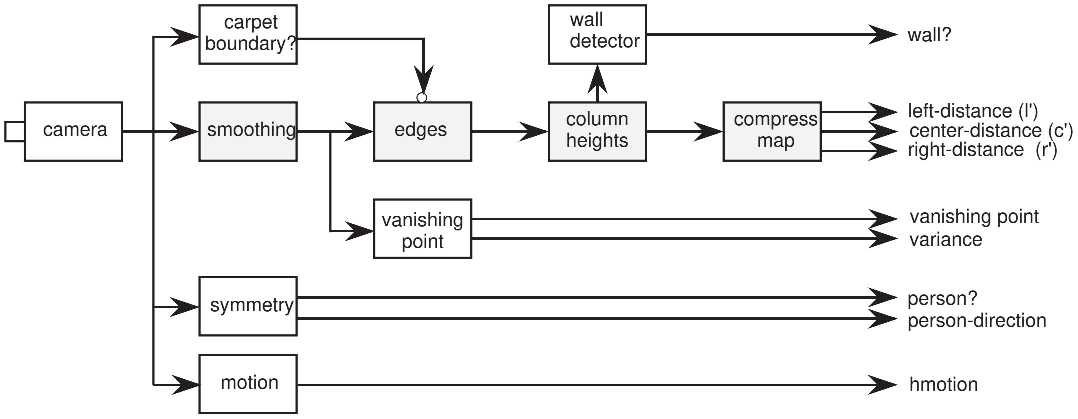


Figure 8.1: The major components of the core visual system (CVS).

`open-left?`	`open-region?`	`person-direction`
`open-right?`	`blind?`	`wall-ahead?`
`blocked?`	`light-floor?`	`wall-far-ahead?`
`left-turn?`	`dark-floor?`	`vanishing-point`
`right-turn?`	`person-ahead?`	`farthest-direction`

Table 8.1: Partial list of percepts generated by the visual system.

percepts are derived from the depth map. `Blocked?` is true when the closest object ahead of the robot is nearer than a threshold distance (about three feet). `Open-left?` and `open-right?` are true when the there are no objects in the respective direction closer than a threshold distance. `Left-turn?` and `right-turn?` are true when `open-left/right?` is true and the robot is aligned with the corridor. Alignment with the corridor is actually determined by the low level navigation system. The LLN judges the robot to be aligned if the robot has driven straight for a sufficient period of time. It would be preferable to use visual data, but the robot cannot always determine the axis of the corridor from a single image: if the corridor is blocked, or the robot has reached the end of the corridor, there will be insufficient information to accurately judge the corridor's orientation. `Wall-ahead?` and `wall-far-ahead?` are true when there is a flat region ahead in the depth map. `Dark-floor` and `light-floor` are true when the the bottom center image pixel intensity is above or below a given threshold. Both percepts are computed with hysteresis to compensate for transient variations in lighting or albedo.

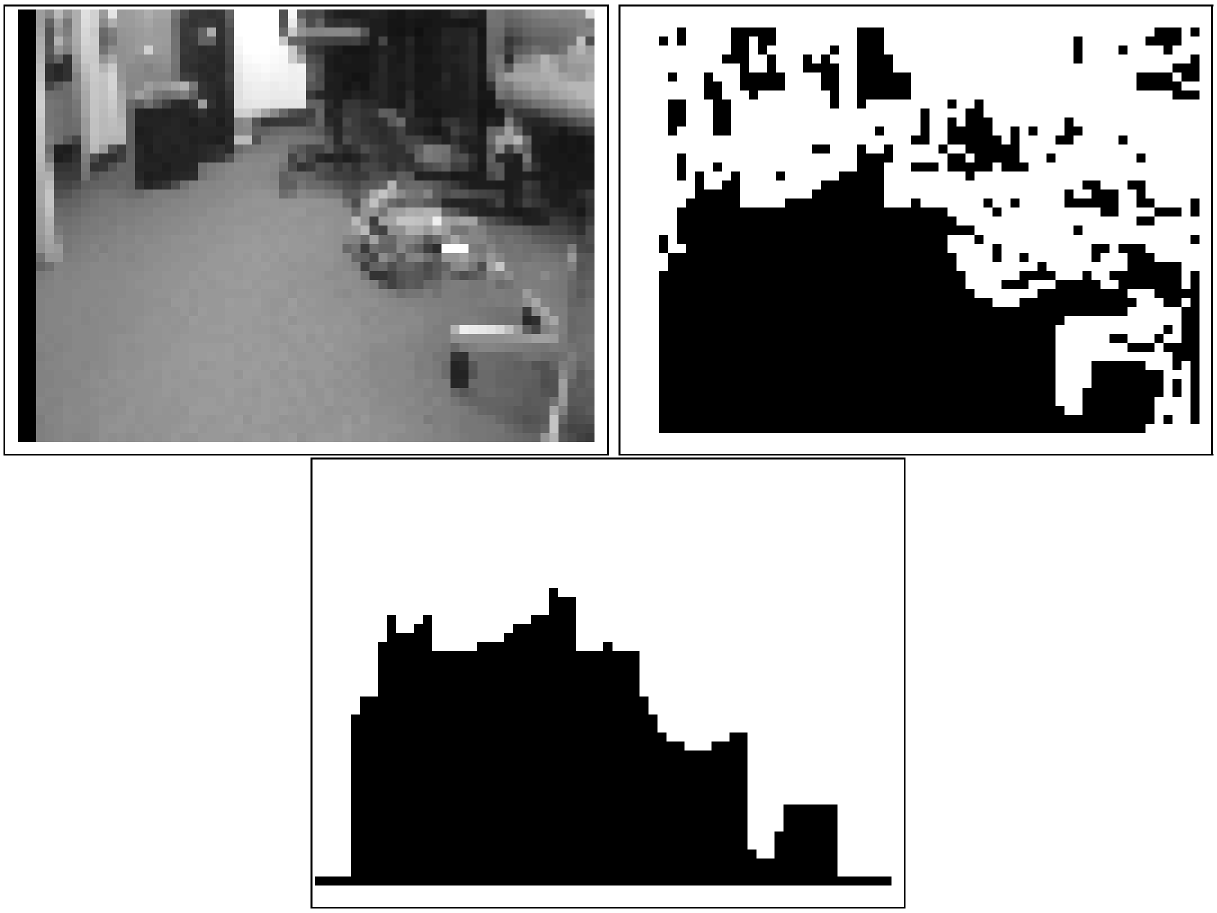


Figure 8.2: The robot computes a depth map by labeling floor pixels (above right) and finding the image plane height of the lowest non-floor pixel in each column. The result is a monotonic measure of depth for each column (above center).

8.1 Computation of depth

Since most of Polly's computation is devoted to navigation, a large fraction of its visual processing is devoted to finding the empty space around it. The central representation used for this purpose is the radial depth map (RDM). The RDM gives the distance to the nearest obstacle for each column of the image. Since image columns correspond to directions in the ground (X-Z) plane, the representation is equivalent to the sort of radial map one would get from a high resolution sonar ring (hence the name).

The central pipeline of the CVS (shown in grey in figure 8.1) computes depth information. The robot starts by finding the edges in the image. Then, for each column of the image, it computes the height of lowest edge pixel in that column.

Under the right set of assumptions, this height will be a measure of the distance to the closest obstacle in view within that column (see below, and also section 6.3). Figure 8.2 gives an example of the depth map calculation.

We can understand the assumptions made by this system and their ramifications by performing a derivation. Consider an arbitrary system for computing a radial depth map. For example, we might first use a stereo system to extract a 2D depth map, then collapse the 2D data into a radial map:

$$\Rightarrow \boxed{\text{stereo}} \mapsto \boxed{\text{project}} \mapsto$$

Such an approach might be quite effective, but it makes its own set of assumptions about the world (e.g. smoothness of surfaces or presence of dense texture). By lemma 4, p. 68, we can reduce any such system to

$$\Rightarrow \boxed{\text{FG}} \mapsto \boxed{\text{column heights}} \mapsto$$

where "FG" is some computation which performs figure/ground separation. Fortunately, Polly's environment also satisfies the background texture constraint because the carpet has no visible texture. By lemma 3, p. 66, we can use any linear filter whose pass band is restricted to the zero-band of the carpet to solve the figure/ground problem. An edge detector is such a filter, so we can reduce the system to

$$\Rightarrow \boxed{\text{edges}} \mapsto \boxed{\text{column heights}} \mapsto$$

The derivation is summarized in table 8.2. The derivation shows that the the background texture constraint is used to to simplify figure/ground separation. More importantly, it shows that the constraint is used for nothing else. If we wish to run the system in an environment that does not satisfy the background texture constraint but does satisfy the ground plane constraint, then we can use any salience constraint that holds of the domain. For example, if the background has a distinctive color or set of colors, we can use a color system such as Swain's color histogram method [96] to find the carpet:

$$\Rightarrow \boxed{\text{color}} \mapsto \boxed{\text{column heights}} \mapsto$$

If we wanted to build a system that worked in both domains, we could implement both the color system and the edge detector and switch between them opportunistically, provided there was sufficient information to determine which one to use. One could even implement the stereo system in parallel with these systems and add another switch.

The particular edge detector used by Polly is a thresholded gradient detector. The detector was chosen because it compiled to a very short loop on the DSP. Because

Constraint	Computational problem	Optimization
Ground plane	depth perception	use height
Background-texture	figure/ground separation	use texture

Table 8.2: Habitat constraints used for depth-recovery.

any edge detector should work (by lemma 3), we are free to make the choice based on computational efficiency. The exact test is for $|\frac{\partial I}{\partial x}(x,y)| + |\frac{\partial I}{\partial y}(x,y)| > 15$ after smoothing with a 3×3 low-pass filter to remove noise. The possible range of gradients is 0 to 510, so this is a very sensitive detector. To avoid driving into dark regions where edge detection is unreliable (because the estimates of the intensity derivatives become noisy), the edge detector automatically marks any pixel darker than 40 grey levels. This helps prevent the robot from driving into dark rooms. The edge detector is implemented by the `find-dangerous` procedure, whose source code is shown in figure 8.3. The procedure is so named because it labels pixels to be avoided.

The CVS compresses the depth map into three values—`left-distance`, `right-distance`, and `center-distance`—which give the closest distance on the left side of the image, right side, and the center third, respectively. Other values are then derived from these three. For example, `open-left?` and `open-right?` are true when the corresponding distance is over threshold. `Left-turn?` and `right-turn?` are true when the depth map is open on the correct side and the robot is aligned with the corridor.

8.2 Detection of carpet boundaries

The background texture constraint fails at the boundaries between two carpets. At such boundaries, the edge detector fires even though there is no object present, and so the robot thinks it's blocked. Polly explicitly detects this condition and treats it as a special case. The procedure `carpet-boundary?` (shown in figure 8.4) checks for the presence of a carpet boundary. If it returns true, then `find-dangerous` will ignore horizontal edges. Since the carpet boundary is horizontal in the image when the robot approaches it, this is sufficient to cause the robot to pass the carpet boundary. Once it passes the boundary, `carpet-boundary?` returns false, and `find-dangerous` once again becomes sensitive to horizontal lines.

The actual test used by Polly is overly simplistic but adequate for the job. The robot examines a 10×15 window at the bottom center of the image. It searches for a horizontal edge of medium contrast within the window. If there is such an edge, and no pixel is brighter than 120 grey levels, then it treats the image as a carpet boundary scene. The requirements that the edge have medium contrast and that no pixel be too bright prevent the system from classifying baseboards, which are dark

```
(define (find-dangerous image out)
  (let ((dark-threshold 40)
        (edge-threshold 15))
    (if suppress-horizontal
        (map-vector! out
                     (lambda (up left center)
                       (if (and (< (abs (- left center))
                                   edge-threshold)
                                (> center dark-threshold))
                           0 255))
                     (shift image (- *image-width*))
                     (shift image -1)
                      image)
      (map-vector! out
                   (lambda (up left center)
                     (if (and (< (+ (abs (- left center))
                                    (abs (- up center)))
                                 edge-threshold)
                              (> center dark-threshold))
                         0 255))
                   (shift image (- *image-width*))
                   (shift image -1)
                   image))))
```

Figure 8.3: Source code for finding obstacle pixels. The code has been simplified by removing compiler declarations. `suppress-horizontal` is a wire set by the carpet boundary detector when it detects a horizontal boundary between two carpets. This causes the robot to ignore horizontal lines (see section 8.2).

next to a bright wall, as carpet boundaries. A more intelligent test would certainly be a good idea.

8.3 Vanishing-point detection

When Polly is in a corridor, the visual system also generates the x-coordinate of the vanishing point of the corridor. The vanishing point coincides with the axis of the corridor and so can be used for steering (this will be discussed further in section 9.2). The vanishing point computation works by finding each edge pixel in the image, fitting a line to the edge, intersecting it with the top of the screen, and computing the mean of the intersections (see figure 8.5).

As with the other visual computations in Polly, the computation of the vanishing point is simplified by Polly's knowledge of the appearance of the environment. We can make this knowledge explicit by deriving Polly's vanishing point computation from a more general computation. I will start from the system of Bellutta *et al.* [16], which extracts vanishing points by running an edge finder, extracting straight line segments, and performing 2D clustering on the pairwise intersections of the edge segments. We can represent the system schematically as:

$$\Rightarrow \boxed{\text{Canny edges}} \mapsto \boxed{\text{lines}} \mapsto \boxed{\text{intersect}} \mapsto \boxed{\text{cluster}} \mapsto$$

We can simplify the system by making stronger assumptions about the environment. We can remove the step of grouping edge pixels into segments by treating each edge pixel as its own tiny segment:

$$\Rightarrow \boxed{\text{Canny edges}} \mapsto \boxed{\text{intersect}} \mapsto \boxed{\text{cluster}} \mapsto$$

This will weight longer lines more strongly, so the lines of the corridor must dominate the scene for this to work properly.

If the edges are strong, then any edge detector will suffice. Polly uses a gradient threshold detector simply because it compiles to very efficient code:

$$\Rightarrow \boxed{|\nabla I|} \mapsto \boxed{\text{intersect}} \mapsto \boxed{\text{cluster}} \mapsto$$

Here I is the image, ∇I is the spatial gradient of the image, and so $|\nabla I|$ is the magnitude of the gradient of the image. If the tilt-angle of the camera is held constant by the camera mount, then the vanishing point will always have the same y coordinate, so we can reduce the clustering to a 1D problem.

$$\Rightarrow \boxed{|\nabla I|} \mapsto \boxed{y\ \text{intersect}} \mapsto \boxed{\text{1D cluster}} \mapsto$$

```
(define (carpet-boundary? image)
  (with-vars-in-registers
    (let ((edges 0)
          (edge-thresh 9)
          (bad-thresh 25)
          (region-width 10)
          (region-height 15)
          (top-line -1)
          (bad-edges 0)
          (bottom-line 0))
      (let ((im (shift image 1947)))
        ;; im points to row 30, pixel 27 of image.
        (countdown (lines region-height)
            ;; Process a line.
            (countdown (pixel 10)
              ;; Process a pixel; check intensity and gradient.
              (let ((center (in-register data (vector-ref im 0))))
                (let ((delta (abs (- (vector-ref im 64) center))))
                  (when (> center 120)
                    (incf bad-edges))
                  (when (> delta bad-thresh)
                    (incf bad-edges))
                  (when (> delta edge-thresh)
                    (incf edges)
                    (when (< top-line 0)
                      (set! top-line lines))
                    (when (> lines bottom-line)
                      (set! bottom-line lines)))
                  (shiftf im 1 ))))
          ;; Move to next line.
          (shiftf im 54)))
      (and (> edges 7)
           (< edges 30)
           (= bad-edges 0)
           (< (- bottom-line top-line) 7)))))
```

Figure 8.4: Source code for the carpet boundary detector. Note that `countdown` is the same as `dotimes` in Common Lisp, except that it counts backward.

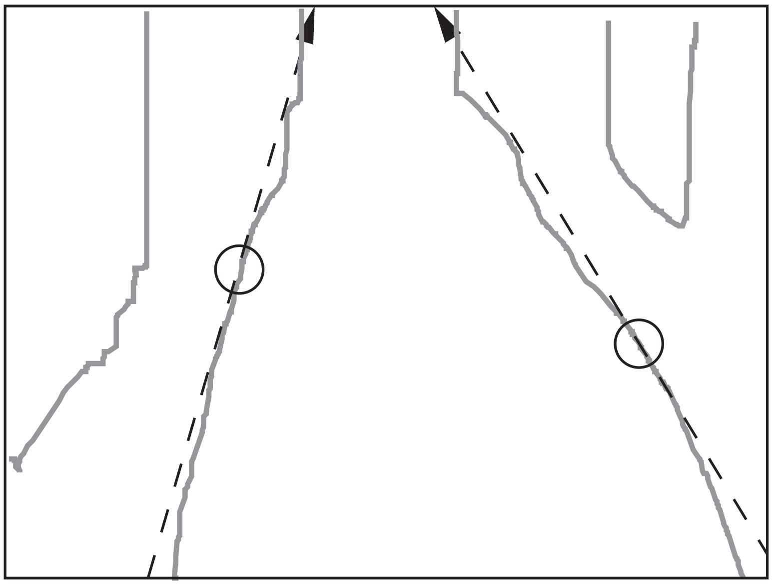


Figure 8.5: The vanishing point computation: edge fragments at individual pixels (shown in circles) are extended (dashed lines) and intersected with the top of the image to find the horizontal location of their intersections (arrowheads). The mean of the horizontal locations is used as the vanishing point.

Finally, if we assume that the positions and orientations of the non-corridor edges are uniformly distributed, then their y-intersections will have zero mean. If we replace the clustering operation, which looks for modes, with the mean of the y-intersections, the result will be a weighted sum of the means of the corridor and non-corridor edges. Since the mean of the latter is zero, the result will be the mean of the corridor edges multiplied by an unknown scale factor which will depend on the ratio of corridor to non-corridor edges. Thus, while the result will typically underestimate the magnitude of the vanishing point, it will get its sign right. As we will see in the next chapter, that will be sufficient for our purposes. The resulting system is thus:

$$\Rightarrow \boxed{|\nabla I|} \rightarrow \boxed{y \text{ intersect}} \rightarrow \boxed{\bar{x}} \rightarrow$$

The derivation is summarized in table 8.3.

The CVS also reports the variance of the y-intersections as a confidence measure. This entire computation is performed in the Polly code by the procedure `vanishing-point`. The procedure computes the mean and variance directly in a single pass over the input array. The procedure contains its own edge detector which is tuned to find diagonal edges (see figure 8.6).

```
(define (find-vanishing-point image)
   ;; Start at the end of the 45th line of the image.
  (let ((image (shift image (- (* 46 *image-width*)
                               2)))
            (sum 0)
            (sum-squares 0)
            (points 0))
    (countdown (y 45)
      ;;; Scan a line.
      (countdown (x (- *image-width* 1))
        (let* ((dx (- (vector-ref image 1)
                      (vector-ref image 0)))
               (dy (- (vector-ref image *image-width*)
                      (vector-ref image 0))))
          (when (and (> (abs dx) 10)
                     (> (abs dy) 10))
            ;; We have a reasonable edge point.
            (let ((x-intercept (+ (quotient (* y dy)
                                            dx)
                                  x)))
              (when (and (> x-intercept -1)
                         (< x-intercept 64))
                (set! sum (+ sum x-intercept))
                (set! sum-squares (+ sum-squares
                                     (* x-intercept x-intercept)))
                (set! points (+ points 1))))))
                ;; Next pixel.
                (set! image (shift image -1)))
            ;; Skip over the pixel at the beginning of the line.
            (set! image (shift image -1)))
    ;; Done with the image.  Now compute the mean and variance.
    (if (and (> points 20)
             (< points 256))
        (let ((mean (quotient sum points)))
          (let ((variance (- (quotient sum-squares points)
                             (* mean mean))))
            (make-pair mean variance)))
      (make-pair 31 1000))))
```

Figure 8.6: Source code for the vanishing-point computation. The code walks over the image, from bottom to top, end of line to beginning, finding all the reasonable edges and computing the mean and variance of their y intersections. The mean and variance are returned as a pair (two 16-bit quantities in a 32-bit word). If there are too many or too few edges, the code gives up and returns a large variance. Note that the code has to skip the first pixel of every line because there is no pixel to the left of it.

Constraint	Computational problem	Optimization
Long corridor edges	line finding	use pixels as lines
Strong corridor edges	edge detection	use cheap detector
Known camera tilt	clustering	1D clustering
Uniform non-corridor intersections	clustering	use mean

Table 8.3: Habitat constraints used by the vanishing point computation.

8.4 Person detection

In order to offer tours to people, Polly needs to detect their presence as it moves through the corridors of the lab. At present, the robot can only look downward, so it can only see people's legs. Thus person detection reduces to leg detection. Polly relies on the fact that people generally stand, and so it need only search for straight, vertical legs, which appear as tall, narrow symmetric objects. Symmetric objects are found in two passes. First, it searches for tall, skinny symmetric regions of the image. Then, it tests the most symmetric region to see if it is a distinct object.

8.4.1 Symmetry detection

This is performed by the Scheme procedure `find-symmetric`. It is a simplified version of the technique used by Reisfeld *et al.* [84]. The simplest measure of symmetry about the vertical axis at a point (x, y) is:

$$-\int_0^l \frac{\partial I}{\partial x}(x+r,y)\frac{\partial I}{\partial x}(x-r,y)dr$$

where l is the width of the region being searched for symmetry. This will give a large value if the pixels $(x+r,y)$ and $(x-r,y)$ tend to have significant horizontal intensity gradients with opposite signs. Thus both a bright region in front of a light region and a dark region in front of a light region will have positive scores, but a homogeneous region will have zero score and a region with a ramp-shaped intensity will have a negative score.

One disadvantage of this measure is that it is linear. If we superimpose a non-symmetric pattern on a symmetric one, their scores will cancel one another. This is a problem because the edges of a leg may be symmetric, while the surface markings of the pant leg may not be. We can alleviate this problem to some extent by only counting pixel pairs which are symmetric. We can do this by adding a min to the integrand:

$$-\int_0^l \min\left(0, \frac{\partial I}{\partial x}(x+r,y)\frac{\partial I}{\partial x}(x-r,y)\right) dr$$

Another problem with the linearity of the computation is that a single pair of strong edges can generate a huge symmetry value for an otherwise non-symmetric region. This can be alleviated by adding a compressive nonlinearity to the integrand. Reisfeld *et al.* [84] used a log function for the nonlinearity. Polly uses a max because it is somewhat faster to compute:

$$-\int_0^l \max\left(-\alpha, \min\left(0, \frac{\partial I}{\partial x}(x+r,y)\frac{\partial I}{\partial x}(x-r,y)\right)\right) dr$$

Here α is the maximal symmetry value which the system will give to a single pixel pair.

Having scored all pixels for the vertical symmetry of the local regions centered around them, we can then find vertical lines about which the image is strongly symmetric by integrating the pixel scores along columns of the image. The result is then a vector of symmetry values for each column defined by:

$$s(x) = \sum_y \sum_{r=1}^{l} \max\left(-\alpha, \min\left(0, \frac{\partial I}{\partial x}(x+r,y)\frac{\partial I}{\partial x}(x-r,y)\right)\right)$$

where the derivatives are estimated as differences of adjacent pixels.

8.4.2 The protrusion test

Having tested the different image columns for symmetry, the visual system still needs to distinguish between objects which are symmetric and subregions of objects which are symmetric. Since Polly specifically looks for people standing in hallways, and such people will show up as distinct bulges in the radial depth map, the visual system requires that a symmetric region align with a bulge in the depth map. This test is performed by the procedure `protrusion-near?` which tests the depth map for a negative depth edge followed by a positive depth edge within a specific width around the region. If such edges are found, then there is such a protrusion and the visual system asserts the `person?` signal and drives the `person-direction` wire with the x-coordinate of the column with maximal symmetry.

The protrusion test is not actually a sufficient test to guarantee that a given region is a person's leg. Any tall, skinny, vertical object, such as a chair leg, will pass the protrusion test. To avoid these false positives, the pick-up system only operates while it is in hallways. That is, trusts the `person?` signal when the `corridor?` signal is also asserted.

The protrusion test can also generate false negatives. When the person leans against the wall, the protrusion detector will generally missed them and the pick up system won't offer them a tour. Fortunately, this is useful for our purposes since we only want the robot to offer tours to people who actually approach the robot.

8.5 Gesture interpretation

When Polly offers a visitor a tour, it needs to be able to receive an answer. Since it cannot hear, it looks for a particular gesture. Originally, this was to be a nod of the head. I built two working nod detectors which ran in real time on a macintosh using a static camera. In the end, however, I was forced to use waves of the foot instead of nods. This was largely forced by hardware a considerations. The robot is so short that the upward-looking camera saw only the bottom of the visitor's chin and so thus nods ended up being Z-axis oscillations rather than Y-axis oscillations. Z-axis oscillations are much more difficult to measure. The upward looking camera also has had to cope with a great deal of glare from the overhead lights. This glare made it very difficult to properly image the face. The right solution would have been to raise the camera to human eye-level, but that was impossible to due to mechanical stability issues. Finally, the upward-looking camera broke late in the design of the system so it was decided to give up on the nod detector and use the wave detector.

8.5.1 Foot waving

Detecting foot waves turned out to be a trivial problem because anyone who was actually interacting with the robot was standing still. Further, there is typically no other motion in the scene. Thus the only motion in the scene is intentional shaking of the leg and so a simple motion detector sufficed as a wave-of-the-leg detector (in contrast to the various nod detectors). The detector used measures total image change from frame to frame:

$$m_{\text{total}}(t) = \sum_{x,y} |I(x,y,t) - I(x,y,t-1)|$$

and applies a low-pass filter to the resulting series $m_{\text{total}}(t)$.

The low-pass filtered motion is computed by the procedure `total-hmotion` and its instantaneous value is store in `hmotion`.

8.5.2 The first nod detector

Detecting nods is a more difficult problem. Suppose that there is a person facing the camera. For simplicity, we will assume orthographic projection so that a point $(x,y,z)^T$ in the world is projected to a point $(x,y)^T$ in the image, and points higher in the image have larger y coordinates. Finally, we will assume that the objects in view are rigid or at least piecewise rigid. We will treat the motion of a given object at a given point in time as a combination of a translation $\mathbf{T}$ and a rotation ω about a point **P**.[1] A point **R** on the object thus moves with an instantaneous velocity of

$$\mathbf{T} + (\mathbf{R} - \mathbf{P}) \times \omega$$

Since a nod is an alternating rotation of the head, a nod is then a motion of the head in which ω has constant direction but varying sign, **R** remains constant, and **T** remains zero. The conceptually simplest approach to detecting nods would be to find the head in the image, determine its pose (position and orientation in 3-space), and to track the pose through time to recover the motion parameters **T**, **R**, **P**, and ω, and then test these parameters for rotational oscillation:

$$\Rightarrow \boxed{\text{find head}} \rightarrow \boxed{\text{3D pose}} \rightarrow \boxed{\text{track}} \rightarrow \boxed{\text{oscillating?}} \rightarrow$$

This system would be expensive and difficult to implement, although it would certainly be conceptually possible. The main problems are with finding faces and determining pose. A particular problem with the latter is that it is easy to confuse translation with a rotation. In particular, it is very difficult to distinguish the cases of my bending (rotating) my head downward, and my head translating down 3cm, but still pointing forward. Both would involve the points of my facing moving downward along the Y axis, while staying at the same point along the X axis. Of course, the translation would require my neck to suddenly grow 3cm shorter or for me to do deep knee bends.

An simpler system is:

$$\Rightarrow \boxed{\text{normal flow}} \rightarrow \boxed{\textstyle\int_{\text{image}}} \rightarrow \boxed{\text{bandpass}} \rightarrow \boxed{\text{envelope det}} \rightarrow$$

It computes the vertical component of *normal flow field* (see below) of the image, then integrates the field over the entire image to obtain a single net vertical motion, and looks for oscillations in the net motion. The optic flow of an image is the apparent two-dimensional motion of the texture in the image. It is the projection into the image plane of the motion vectors of the actual objects in the world. Since the 3-space motion of a point **R** is simply

$$\mathbf{T} + (\mathbf{R} - \mathbf{P}) \times \omega$$

the optic flow of **R**'s projection is simply the x and y coordinates of this vector, or

$$\mathbf{f_R} = \begin{bmatrix} T_x + (R_y - P_y)\omega_z - (r_z - p_z)\omega_y \\ T_y + (R_z - P_z)\omega_x - (R_x - P_x)\omega_z \end{bmatrix}$$

[1]This is a gross oversimplification of the kinematics of jointed objects, but it will suffice for our purposes.

The normal flow field is the component of the optic flow field in the direction of the image's intensity gradient. The useful property of the normal flow field is that it is very easy to compute. If we let $I(x, y, t)$ represent the brightness of a point (x, y) in the image, at time t, then the vertical component of the normal flow field is given by

$$-\frac{\frac{\partial I}{\partial t}(x, y, t)}{\frac{\partial I}{\partial y}(x, y, t)}$$

Since the derivatives can be approximated by subtracting adjacent points in time or space, respectively, we can compute this value very quickly.

We can explain why (and when) this system works by performing a derivation. We start from the original system:

$$\Rightarrow \boxed{\text{find head}} \rightarrow \boxed{\text{3D pose}} \rightarrow \boxed{\text{track}} \rightarrow \boxed{\text{oscillating?}} \rightarrow$$

Recall that computing the 3D pose is difficult since translations and rotations both generate vertical motions in the image. However, since the head is not physically capable of translating, any up/down motion of the head in the image *must* be due to a rotation, unless the person is bending their knees, jumping up and down, bowing rapidly, or standing on an oscillating platform. If we assume that people don't do these things, then we may safely interpret any vertical oscillation of a head as a nod. Let's call this the *head kinematics constraint*: that the translation vector $\mathbf{T}$ of the head is nearly always zero, and that it never oscillates. The head kinematics constraint allows us to use the 2D position of the head (*i.e.* the position of the head in the image itself), instead of its 3D pose. An oscillation of the 2D position will indicate a nod. Thus we can replace the "3D pose" module with a module that finds the centroid of the image region occupied by the head:

$$\Rightarrow \boxed{\text{find head}} \rightarrow \boxed{\text{centroid}} \rightarrow \boxed{\text{track}} \rightarrow \boxed{\text{oscillating?}} \rightarrow$$

Thus we've reduced the problem to finding 2D oscillations of the head. Finding the head can be difficult in itself however. Fortunately, we're not looking for arbitrary heads, only oscillating ones. Thus we need not bother searching static parts of the image. Indeed, if we assume vertical oscillations are rare, *i.e.* people don't nod their hands or play with yo-yos, then we can use the oscillatory motion itself to find the head. We'll call this the *motion salience constraint*: no motion parameter of any object is allowed to oscillate, save for the ω parameter of a head. The motion salience constraint removes the need for a full recognition engine and allows us to use a system like:

$$\Rightarrow \boxed{\text{find oscillating points}} \rightarrow \boxed{\text{find head}} \rightarrow$$

Constraint	Problem	Optimization
Head kinematics	motion disambiguation	use 2D motion
Motion salience	head detection	use oscillation
Horizontal lines	flow computation	substitute normal flow
Nod dominance	head detection	use net flow

Table 8.4: Constraints used in the derivation of the first nod detector, and the problems they were used to simplify.

which finds the oscillating pixels, and then searches for head-shaped regions of oscillating pixels. We can find the oscillating points by first computing the 2D motion (optic flow) at each point in the image, and then testing the motion for vertical oscillation. The test can be done, in turn, by applying a bandpass filter and an envelope detector at each point. The resulting system is thus:

$$\Rightarrow \boxed{\text{optic flow}} \rightarrow \boxed{\text{bandpass}} \rightarrow \boxed{\text{envelope det}} \rightarrow \boxed{\text{find head}} \rightarrow$$

Computing optical flow can still be fairly expensive however. The normal flow field is much easier to compute and, fortunately, its vertical component will always have the same sign as the vertical component of the optical flow, provided that the normal flow is non-zero. The normal flow will be zero when the actual motion is nonzero only if there is no texture in the image at that point, or if the y (vertical) derivative of the image intensity is zero. Fortunately, faces have considerable vertical intensity variation and so this is not a problem. Thus we can reduce the system to:

$$\Rightarrow \boxed{\text{normal flow}} \rightarrow \boxed{\text{bandpass}} \rightarrow \boxed{\text{envelope det}} \rightarrow \boxed{\text{find head}} \rightarrow$$

Even looking for oscillations at each point might be too expensive, however. If we assume that the motion of the head will dominate any other motion in the image, then we can look for oscillations in the net vertical motion of the entire image:

$$\Rightarrow \boxed{\text{normal flow}} \rightarrow \boxed{\int_{\text{image}}} \rightarrow \boxed{\text{bandpass}} \rightarrow \boxed{\text{envelope det}} \rightarrow$$

which is exactly the system we sought to derive. The assumption that the head motion dominates the motion in the rest of the image is needed to rule out the case of motions in different parts of the image canceling with one another to generate the appearance of oscillatory motion when in fact, no single part of the image was oscillating.

The derivation is summarized in table 8.4. I have implemented this system on a Macintosh personal computer. The system uses 64×48 grey-scale images at a rate of approximately 5 frames per second. The system performs reliably provided that the subject nods their head several times to give the bandpass filter a chance to respond. Unfortunately, this can lead to dizziness if it is used frequently, so a different system was needed.

8.5.3 The second nod detector

The first nod detector was not adequate for general use because the bandpass filter required many cycles to decide that a head nod was really happening. This is a well known problem with linear filters: the narrower the pass band, the longer it takes the system to respond. A non-linear oscillation detector such as a zero-crossing detector would probably do better. The result would be something like this:

$$\Rightarrow \boxed{\text{normal flow}} \rightarrow \boxed{\textstyle\int_{\text{image}}} \rightarrow \boxed{\text{zero-crossing det}} \rightarrow \boxed{\text{envelope det}} \rightarrow$$

However we would expect the net flow to have a relatively large number of random zero crossings, even when the head is not moving. We can deal with this problem by backing off of the net flow optimization and returning to computing the oscillations on a per-pixel basis:

$$\Rightarrow \boxed{\text{normal flow}} \rightarrow \boxed{\text{zero-crossing det}} \rightarrow \boxed{\text{envelope det}} \rightarrow \boxed{\text{find head}} \rightarrow$$

The flow and zero-crossing detectors have been implemented on Polly. Unfortunately, the robot is so short, and the viewing angle of the camera so wide, that it can only see the bottom of a person's chin, so it has been difficult to get the flow detector to respond to any head movements, much less to detect nods.

8.6 Summary

Polly's vision system uses a number of parallel processes to efficiently compute just the information needed for its task. These processes are made more efficient by taking advantage of the special properties of the environment. We can understand the use of these properties by deriving the systems from more general ones. The complete list of constraints used in the derivations is given in table 8.5.

Constraint	Computational problem	Optimization
Ground plane	depth perception	use height
Background-texture	figure/ground separation	use texture
Long corridor edges	line finding	use edge pixels
Strong corridor edges	edge detection	use cheap detector
Known camera tilt	clustering	1D clustering
Uniform intersections	clustering	use mean
Head kinematics	motion disambiguation	use 2D motion
Motion salience	head detection	use oscillation
Horizontal lines	flow computation	use normal flow
Nod dominance	head detection	use net flow

Table 8.5: Habitat constraints assumed by the core visual system and the problems they helped to simplify. Note that "known camera tilt" is more a constraint on the agent, than on the habitat.

Chapter 9

Low level navigation

The LLN is the bottom layer of the robot's control system. It consists of a speed (forward-velocity) controller, an open-loop (ballistic) turn controller, a corridor follower, a wall follower, and a small amount of code to arbitrate between them (see figure 9.1). These systems are controlled by the signals `speed-request`, `turn-request`, and `inhibit-all-motion?`. `Speed-request` controls the maximum forward velocity and has no effect on steering, other than to disable the wall follower and corridor follower when it is zero. When `turn-request` is zero (no turn requested), the corridor follower (or wall follower if only one wall is visible) has complete control of steering. When `turn-request` is driven with a desired turn angle, the open-loop turn controller issues a turn command to the base and inhibits all steering for a time proportional to the size of the turn. `Inhibit-all-motion?` inhibits all LLN components, forcing the robot to stand still. It is asserted during visitor pickup (see section 10.5) to prevent the low-level navigation system from avoiding the visitor.

9.1 Speed controller

The robot's rate of forward motion is controlled by the procedure `speed-control`. The two constraints on forward velocity are that the robot should move forward when there's nothing in its way, and that it should stop when there is something close to it. We want the robot to move at full speed when the nearest obstacle is more than some safe distance, d_{safe}, and to stop when it's less than some distance d_{stop}. We use the rule

$$speed = \min\left(v_{max}, \frac{v_{max}}{d_{safe} - d_{stop}}(d_{center} - d_{stop})\right)$$

where d_{center} is the distance to the closest object in the center of the image (the `center-distance` output of the visual system). The robot smoothly decelerates as it approaches an obstacle and will back up if it gets too close to the obstacle. Backing

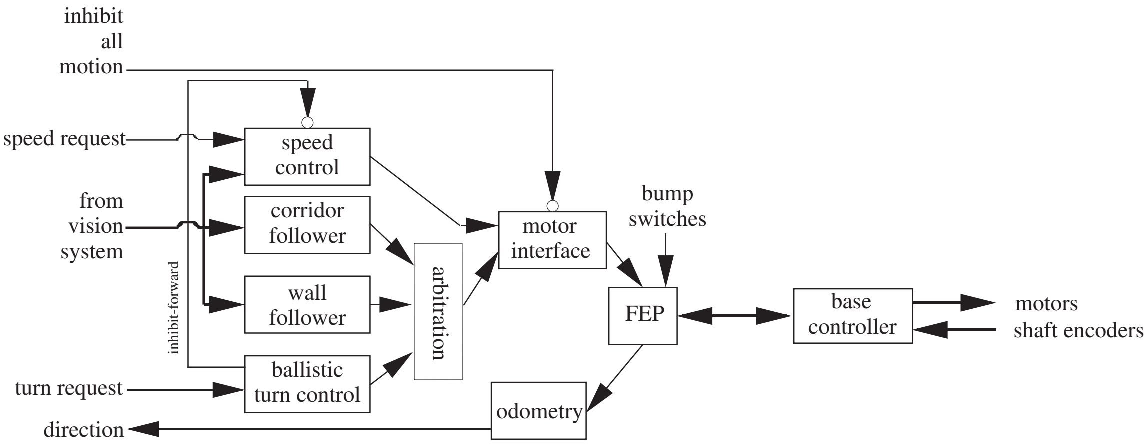


Figure 9.1: Components of the low level navigation system (LLN).

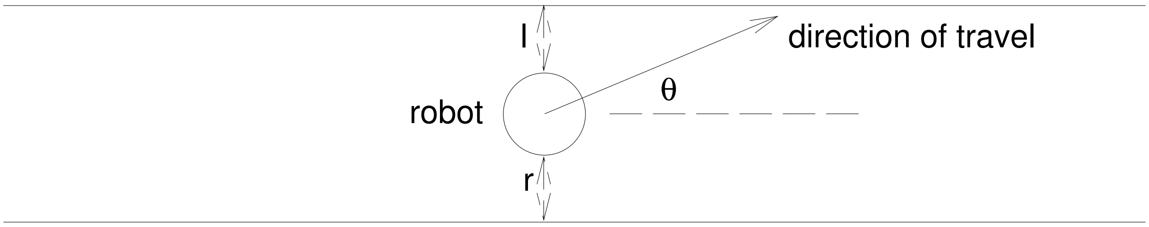


Figure 9.2: The corridor following problem. The robot needs to turn so as to simultaneously keep l and r large and θ small.

up is useful because it allows one to herd the robot from place to place. The maximum speed, v_{max}, is set by the input `speed-request`.

The speed controller is modulated by two other inputs. If `inhibit-forward?` is asserted, the robot only drives backward, if at all. `Inhibit-forward?` is driven by the ballistic turn controller (see section 9.5). When `inhibit-all-motion?` is set, all motors are disabled, and the robot will not even back away from approaching threats.

9.2 Corridor follower

Corridor following consists of two basic subproblems: aligning with the axis of the corridor and staying comfortably far from the walls on either side. In figure 9.2, these problems correspond to keeping θ small and l and r large, respectively. The robot base gives us control of the turn rate ($\frac{d\theta}{dt}$) and the speed (s) and so the variables l, r, and θ are coupled in the kinematics of the base:

$$\frac{dl}{dt} = -\frac{dr}{dt} = s \sin\theta$$

so we cannot change l or r without first making θ nonzero. Polly's corridor follower uses separate control systems for wall distance and θ and sums their outputs.

9.2.1 Aligning with the corridor

Polly uses a trivial algorithm for aligning with the corridor. It uses the control law

$$\frac{d\theta}{dt} = -\alpha\left(\mathrm{vp}_x - \frac{\mathrm{width}}{2}\right)$$

where vp_x is the core vision system's estimate of the vanishing point (carried in the variable `vanishing-point`), α is a gain parameter, and "width" is the width of the image in pixels. The vanishing point needs to be biased by width/2 because the coordinate system of the image is centered on the left-hand side of the image, not in the middle.

An obvious and direct way of performing this task would be to first construct a 3D model of the environment, then find the walls of the corridor in the model and compute θ, and finally multiply θ by some gain to drive the turning motor. We can represent this schematically as:

$$\Rightarrow \boxed{\text{3D model}} \rightarrow \boxed{\text{corridor}} \rightarrow \boxed{\theta} \rightarrow \boxed{\text{gain}} \rightarrow \frac{d\theta}{dt}$$

This is not a particularly efficient design however, since 3D models are both difficult and computationally expensive to build. Intuitively, building an entire model of the environment, only to compress it down to a single number, θ, seems a waste of energy. Any system that turns to minimize θ, that is, any system of the form,

$$\Rightarrow \boxed{\theta} \rightarrow \boxed{\text{gain}} \rightarrow \frac{d\theta}{dt}$$

will work. By decalibration (lemma 1, p. 58), we can substitute *any monotonic function of* θ for our estimate of θ, provided that we get zero right. Thus we can use any system of the form:

$$\Rightarrow \boxed{f(\theta)} \rightarrow \boxed{\text{gain}} \rightarrow \frac{d\theta}{dt}$$

where f is a monotonic function for which $f(0) = 0$. Since $\mathrm{vp}_x - \frac{\mathrm{width}}{2}$ is such a function, we can use it, and reduce the above system to

$$\Rightarrow \boxed{\text{vanishing point}} \rightarrow \boxed{\text{gain}} \rightarrow \frac{d\theta}{dt}$$

which is the system that Polly uses.

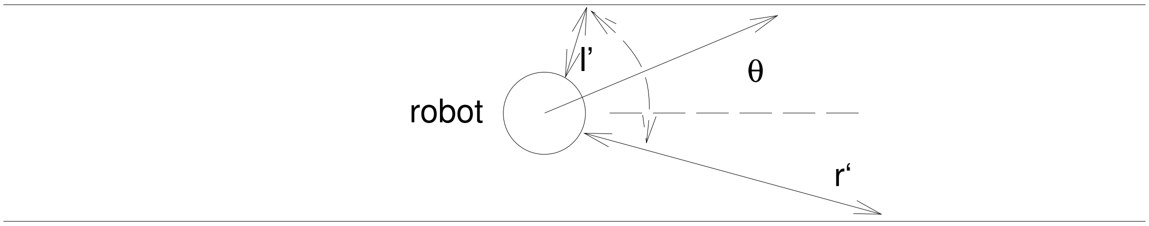


Figure 9.3: The nearest points in view of the robot. The dashed curved line indicates the robot's field of view. l' and r' are the distances to the nearest points in view on the left and right walls, respectively. Note that as the robot rotates clockwise in place, l' gets larger and r' gets smaller.

9.2.2 Avoiding the walls

One way to avoid the walls is to drive away from the wall that looks closest. If we let l' be the distance to the closest point in view on the left, and r' the distance to the closest point in view on the right (see figure 9.3), then we might steer so as to make l' and r' be equal.. Polly does this using the control law

$$\frac{d\theta}{dt} = -\beta(\texttt{left-distance} - \texttt{right-distance}) \tag{9.1}$$

where β is a gain parameter. Recall however, that `left-distance` and `right--distance` are not the same as l' and r', rather they are equal to $f(l')$ and $f(r')$ for some unknown monotonic f. We would like to show that this doesn't matter. Unfortunately, we can't use lemma 1 directly, because it only allows us to replace $l' - r'$ with $f(l' - r')$, *not* $f(l') - f(r')$. Fortunately, for any given position of the robot, l' will be a strictly increasing function of θ, while r' will be a strictly decreasing function of θ. Thus their difference is also strictly increasing. But then $f(l') - f(r')$ must also be a strictly increasing function of θ. Moreover, $f(l') - f(r')$ is zero exactly when $l' - r'$ is zero. Thus both $l' - r'$ and $f(l') - f(r')$ are effectively decalibrated versions of θ, and both will converge to the same value of θ for a given position in the corridor.[1]

9.2.3 Integrating the control signals

The corridor-follower sums the control signals to align with the corridor and avoid the walls, yielding the final control law:

$$\frac{d\theta}{dt} = -\alpha\left(\texttt{vanishing-point} - \frac{\text{width}}{2}\right) - \beta\left(\texttt{left-distance} - \texttt{right-distance}\right)$$

[1]This reasoning assumes the robot's field of view is wholly to the right of the robot in the figure, and that both walls are in the field of view. For very large or small fields of view, or for large values of θ, this may not be true.

The integration is modulated by the variance of the vanishing point and the presence of obstacles. If the variance of the estimate of the vanishing point is high then α is set to zero. This prevents the robot from steering in odd directions or oscillating when the robot enters non-corridor areas. When the robot is blocked by an obstacle, it makes all steering turns at full speed. This allows the robot to quickly turn around the obstacle without causing stability problems at other times.

9.3 Wall follower

When the robot enters an open room or a very wide corridor, it will be unable to see the opposite wall. In such situations, the corridor follower would treat the opposite wall as being at infinity but would still try to balance the distances of the two walls. It would drive until *neither* wall could be seen and then continue on in a straight path. It would have no idea where it was going. Polly has a separate control system to handle this case. The control systems turns so as to keep the wall that is in view at a fixed distance. The control law is then simply

$$\frac{d\theta}{dt} = -\gamma(f(l') - d_0)$$

for the case where the left wall is in view, or

$$\frac{d\theta}{dt} = \gamma(f(r') - d_0)$$

when the right wall is in view. Here γ is a gain parameter and d_0 is the desired (decalibrated) distance. Again, $f(l')$ and $f(r')$ are stored in the `left-distance` and `right-distance` variables.

9.4 General obstacle avoidance

There is no general obstacle avoidance routine in the system. However, both the corridor follower and the wall follower work to control the distance to the nearest thing on either side. In the normal case, the nearest objects on each side are the walls. However, if an obstacle is in the way, it will be the nearest thing on one or the other side and the robot will avoid it. This is a local navigation strategy equivalent to the method of artificial potential fields (see Khatib [54]). It has the advantage of being fast and easy to compute and the disadvantage that the left and right distances can sometimes balance exactly, even when then robot is blocked by an obstacle. Such cases are dealt with by the unwedger unit in the high level navigation system (see section 10.1.2).

9.5 Ballistic turn controller

When the robot reaches a junction and needs to switch from one corridor to another, it issues an open-loop turn command to the base. This command is issued by the ballistic turn controller whenever the input line `turn-request` is non-zero. `Turn-request` is normally held low by the ballistic turn controller, but can be driven with a specific value (a number of degrees to turn) by the high level navigation system to force a turn. After issuing an open loop turn, the controller maintains control of the turning motors for a time proportional to the size of the turn. This gives the microcontroller in the base time to servo to the correct direction and decelerate.

One potential problem with ballistic turns is that the robot can turn to face a wall. If the wall is textureless and completely fills the visual field of the robot, then it will appear to be an empty field and the robot will happily try to drive into it. In theory, the speed controller should back away from the wall as it turns toward it. In practice, it is possible for the robot to turn so fast that the speed controller literally never sees the wall coming. To prevent this, the ballistic turn controller asserts the signal `inhibit-forward?` during large turns. This prevents the speed controller from ever driving forward in the first place, while still allowing it to back up if something gets too close.

9.6 Steering arbitration

Arbitration is performed by the procedure `turn-controller`. During ballistic turns, it inhibits both the wall follower and the corridor follower, allowing the ballistic turn controller to finish its turn. If no ballistic turn is in progress, it uses the output of either the corridor follower (if both `open-left?` and `open-right?` are false), or the wall follower (if one is true). As with the speed controller, the turn controller stops completely when `inhibit-all-motion?` is asserted.

9.7 The FEP bump reflex

In addition to the visual collision avoidance implemented in the DSP, the FEP (the 6811 front-end processor) also monitors a pair of bump switches. If either switch closes, the FEP immediately performs a fixed sequence of actions: first it halts the base, then it reverses the base, and finally, it issues a 45 degree open-loop turn away from the activated bumper. During this time, the DSP is disconnected from the base. When the bump switch opens, the FEP resumes normal processing and allows the DSP to drive the base.

The bump reflex was implemented in the FEP because processing latency was a critical issue. Polly's normal top speed is 1m/s. Since the DSP only samples its sensors at 15Hz, the robot can move up to 7cm between samples. The bump switches extend roughly 12cm from the base and require 1cm of travel to trigger. Even at top speed, the DSP could not initiate a braking action until there were only 4cm between the robot and the obstacle. There is no way the robot can possibly brake in 4cm. Fortunately, the FEP can sample the bump switches at approximately 1kHz, and initiate braking actions immediately upon bumper contact. In practice, even this is insufficient however, and so the robot often collides with obstacles, albeit at lower velocity. While the base is physically capable of decelerating fast enough to stop, the required deceleration will topple the robot. This is not a pleasant experience for either robot or owner.

9.8 Odometric sensing

The only other sensing on the robot is odometric. The RWI base provides high resolution shaft encoders for dead reckoning turns and forward motions. The vast majority of this information is ignored by the current system. The inherent unreliability of odometry was one reason for its limited use. In the course of driving from one end of the building to the other, the rotational odometer can drift by as much as 45 degrees, even when the base is reasonably well aligned. Another reason was simply that I was more interested in vision than odometry and so chose to spend my time engineering the vision side.

All that said, Polly does use the rotational shaft encoder to determine direction. The robot assumes that it starts out pointed south and so the first shaft encoder reading, call it ω_0, it receives will be a south reading. It can then compute its rotation relative to that direction by taking the difference of its current reading and ω_0. Because of encoder drift, the robot only computes orientation to within 90 degrees. It then drives the wire `direction` with one of the values `north`, `south`, `east`, or `west`. Odometer drift can still cause the robot to misestimate its direction however, so the system recalibrates itself: whenever the robot determines that it is in a long corridor and the visual system reports that it is aligned with the axis of the corridor, it assumes it is exactly aligned with one of the compass points, and recomputes ω_0. This has proven very effective for compensating for drift. Using this technique, Polly has survived drift rates of up to 4 degrees/second.

Chapter 10

High level navigation

Navigation is the central problem of current mobile robot research. If you have a robot that can drive from place to place, then the first thing you probably want it to do is to drive to a particular place. The problem remains largely unsolved. Navigation suffers from many of the same problems of intractability and sensitivity to error that other AI problems suffer from. In this chapter, I will discuss how Polly manages to navigate reasonably reliably and efficiently. Polly navigates in a particularly benign, but unmodified, environment. As with most systems in Polly, what's interesting is not it's precise navigation algorithm, but the basic structures of the environment which allow it to function. I will begin by discussing a fairly conventional formalization of the navigation problem and some of the reasons for its difficulty. Then I will add progressively more structure to the problem and discuss how that structure allows the use of simpler mechanisms. In doing so, I will derive a very simple, idealized, navigation algorithm for a class of worlds to which the MIT AI Laboratory belongs. Finally, I will give the details of Polly's navigation algorithm and describe how it does and does not implement the idealized algorithm.

Consider the problem of piloting a robot about the office environment shown in figure 10.1. At any given moment, the robot must decide given its destination how fast to turn and how fast to move forward or backward. Polly uses the policy of

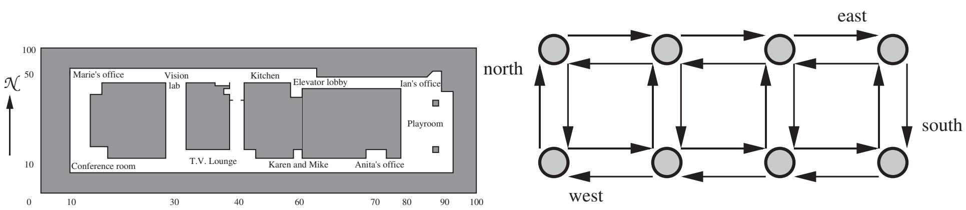


Figure 10.1: Approximate layout of the 7th floor of the AI lab at MIT (left) and its topological structure (right).

following corridors except when it reaches intersections. At intersections it compares the coordinates of the intersection to the coordinates of its goal (presumed to be another intersection) and turns north when the goal is to the north, south when the goal is to the south, and so on:

$$p_{polly}(sensors) = \begin{cases} stop & \text{if at goal} \\ turn\text{-}north & \text{if north of goal and at turn to north} \\ turn\text{-}south & \text{if south of goal and at turn to south} \\ \ldots & \\ turn\text{-}north & \text{if south of goal and pointed south} \\ turn\text{-}south & \text{if north of goal and pointed north} \\ \ldots & \\ follow\text{-}corridor & \text{otherwise} \end{cases}$$

The details of the perception and control systems are given in [50].

10.0.1 Derivation from a geometric path planner

Geometric path planning is a common technique for solving this type of problem. Given a detailed description of the environment, a start position, and a goal position, a path planner computes a safe path through the environment from start to goal (see Latombe [62]). Once the path has been planned, a separate system follows the path. Geometric planning is versatile and can produce very efficient paths, but is not computationally efficient. It also requires detailed knowledge of the environment which the perceptual system may be unable to deliver.

We can clarify the relationship between a path planning system and Polly's reactive algorithm by deriving Polly's algorithm from the planning system. Let $\mathcal{N}$ be the DCP whose states are (position, orientation) pairs and whose actions are small (translation, rotation) pairs such as the robot might move in one clock tick. Clearly, Polly can be modeled as an $\mathcal{N}$ policy. However, the planner can equally well be modeled as an $\mathcal{N}$ policy. A planner/executive is simply a policy that uses internal state to compute and execute a plan. The planning portion uses scratch memory to gradually compute a plan and store it in a plan register, while the executive reads the finished plan out of the register and executes each segment in turn. Thus a planner/executive architecture has the form:

$$\begin{aligned} p_0(s, plan, scratch) &= \begin{cases} i \times \text{plan}_{\mathcal{N}}(s, scratch) & \text{if } plan \text{ incomplete} \\ \text{execute}(plan) \times i & \text{otherwise} \end{cases} \\ \text{execute}(plan) &= \text{head}(plan) \times \mathcal{C}_{\text{tail}(plan)} \end{aligned}$$

An agent in $\mathcal{N}$ will spend nearly all its time in corridors. The only real choice points in this environment are the corridor intersections. Thus only the graph of corridors and intersections $\mathcal{N}'$, need be searched, rather than the full state space of $\mathcal{N}$ (see figure 10.1). By lemma 10, we can augment the environment with a register to hold the current north/south/east/west action and replace p_0 with the policy

$$p_1(s, action) = \begin{cases} p_{p'_1(I(s))}(s) \times \mathcal{C}_{p'_1(I(s))} & \text{if at intersection} \\ p_{action}(s) \times \mathcal{C}_{action} & \text{otherwise} \end{cases}$$

where:

- $I(s)$ is the intersection at state s
- the different p_{action} policies implement following north, south, east, and west corridors, respectively, and
- p'_1 is an arbitrary $\mathcal{N}'$ policy.

The lemma requires that the goal always be a corridor intersection and that the robot always be started from a corridor intersection. We could now solve $\mathcal{N}'$ by adding plan and scratch registers and using a plan/execute policy:

$$p'_1(intersec, plan, scratch) = \begin{cases} i \times \text{plan}_{\mathcal{N}'}(intersec, scratch) & \text{if } plan \text{ incomplete} \\ \text{execute}(plan) \times i & \text{otherwise} \end{cases}$$

We can simplify further by noting that $\mathcal{N}'$ is isomorphic to $\mathcal{Z}_4 \rightleftharpoons \mathcal{Z}_2$, that is, the corridor network is a 4×2 grid. By lemma 9, we can replace p'_1 with any policy that interleaves actions to reduce grid coordinate differences between the current location and the goal. We can then remove the plan and scratch registers from p_1 and reduce it to

$$p_2(s, action) = \begin{cases} p_{p'_2(I(s))}(s) \times \mathcal{C}_{p'_2(I(s))} & \text{if at intersection} \\ p_{action}(s) \times \mathcal{C}_{action} & \text{otherwise} \end{cases}$$

where p'_2 is any $\mathcal{N}'$ policy satisfying the constraints that (1) it only stops at the goal, and (2) it only moves north/south/east/west if the goal is north/south/east/west of $I(s)$.

There are still two important differences between p_2 and p_{polly}: Polly uses a different set of actions ("turn north" instead of "go north") and it has no internal state to keep track of its abstract action. While it appears to use a qualitatively different policy than we have derived, it does not. Within a short period of beginning a *north* action, an agent will always be pointed north. Similarly for *east*, *south*, and *west* actions. The orientation of the robot effectively is the *action* register and turn commands effectively write the register. There's no need for internal memory. Polly stores its state in its motor.

We can summarize the transformations used in the derivation as follows (see table 10.1). The constraint that the environment consist of a network of corridors and that the goal be a corridor intersection allows us to replace geometric planning

Constraint	Optimization
ground plane constraint	use height for depth estimation
background-texture constraint	use texture for obstacle detection
corridor network	replace planning in $\mathcal{N}$ with planning in $\mathcal{N}'$
grid structure	replace planning with difference reduction
orientation correlation	store state in orientation

Table 10.1: Summary of constraints and optimizations used in Polly's navigation system.

with planning in the corridor graph. The isomorphism of the corridor graph to a grid allows us to replace planning with difference reduction. Finally, the correlation of the robot's orientation with its internal state allows us to store the current action in the orientation.

It is important to note that either, both, or neither of the subproblems (the abstracted environment and corridor following) could be solved using deliberative planning; the two decisions are orthogonal. If both are implemented using planners, then the resulting system is effectively a hierarchical planner (see Sacerdoti [89] or Knoblock *et al.* [55]). Polly's environment happens to allow the use of simple reactive policies for both, so it is a layered reactive system (Brooks [20]). In an environment with a more complicated graph topology, one could reverse the second optimization and use a deliberative planner, leaving the first optimization intact. The result would then be a hybrid system with planning on top and reacting on the bottom (see Spector and Hendler [93], Lyons and Hendriks [67], Bresina and Drummond [19], or Gat [40] for other examples). On the other hand, one could imagine an environment where the individual corridors were cluttered but were connected in a grid. In such an environment, the abstract problem could be solved reactively, but corridor following might actually require deliberative planning.

10.1 Navigation in Polly

We have seen how Polly's navigation problem is easier than the general case of navigation because of specific properties of the environment. These properties allow an idealized policy, p_{polly}, to solve the navigation problem without any planning. In fact, using only two bits of state information (the direction register telling whether to go north, south, east, or west). Polly implements a version of this policy using the network of parallel processes shown in figure 10.2. The navigator chooses corridors to steer toward the goal. When the robot comes to an intersection, it signals the ballistic turn controller to align with the new corridor. In effect, the low-level navigation system implements p_{north}, p_{south}, p_{east} and p_{west}, the navigator implements

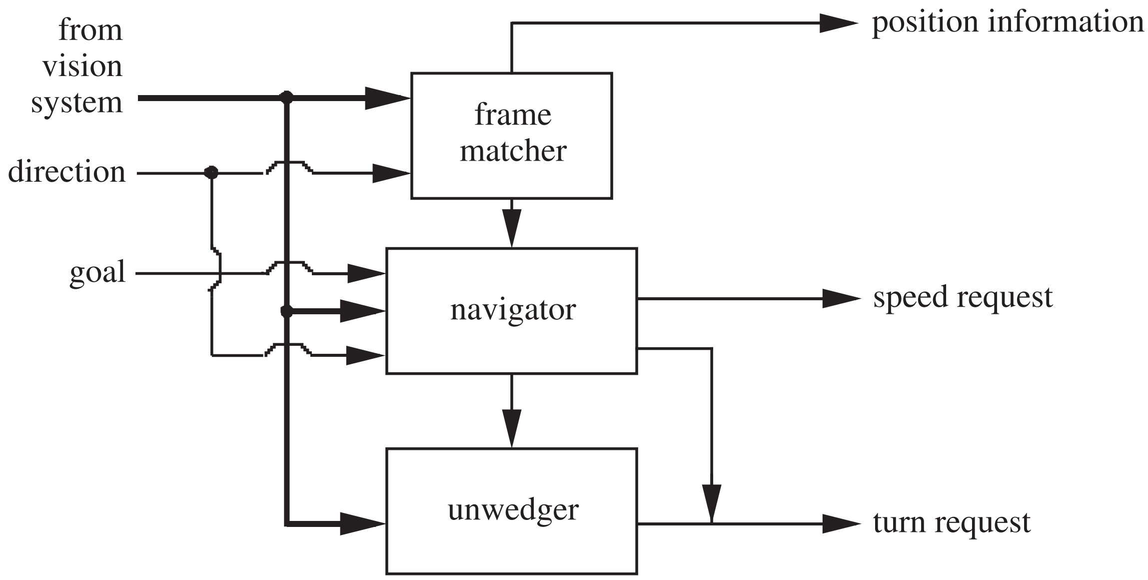


Figure 10.2: The high level navigation system.

p', the orientation of the robot implements the internal state, and the ballistic turn controller implements the write circuitry for the internal state.

10.1.1 The navigator

The navigator receives messages on inputs `goal-x` and `goal-y` and sends messages on the wires `speed-request` and `turn-request`, which are handled by the speed controller and ballistic turn controller, respectively. The navigator also monitors the outputs `last-x`, `last-y`, and `frame-strobe?` from the place recognition system. The first two of these hold the coordinates of the last recognized landmark. `Frame-strobe?` is asserted for one clock tick whenever a new landmark is recognized.

When the navigator's goal inputs are zero, the navigator inhibits motion by setting `speed-request` to zero. When the goal inputs are non-zero, the navigator sets `speed-request` to full speed, and continually checks `frame-strobe`. When a new landmark is reached, it checks whether the landmark is an intersection (the place recognition system allows other types of landmarks), and if so, whether a left or right turn would point in the direction of the goal. If so, it waits 2 seconds and issues a turn (by setting `turn-request`). The 2 second delay is needed because the vision system detects the turn before the robot reaches the intersection. The navigator also informs the unwedger of what direction it would turn, if it it should reach an intersection (see below).

While the robot is moving, the navigator continually checks whether it has reached its goal (`last-x`=`goal-x` and `last-y`=`goal-y`). When it reaches its goal, it clears `goal-x` and `goal-y`, stopping the robot.

Because the place recognition system can fail, the navigator also continually checks whether it has overshot its goal. If so it initiates a u-turn.

10.1.2 The unwedger

One problem with the local navigation strategy used by the corridor follower is that it can get stuck in local minima. This happens, for example, when the robot is perfectly perpendicular to a wall, in which case the difference between the space on the left and space on the right will be zero and the robot will not turn, even though it is blocked. The unwedger takes care of this problem. When the robot is blocked for more than two seconds, the unwedger initiates a 45 degree ballistic turn. The turn is in the same direction that the navigator would turn if the navigator were at an intersection. This is useful because the robot often *is* at intersections when it sees a wall in front of it, but it cannot tell because it is unable to turn its head. If the navigator has already aligned itself with the goal along one of the axes, then it will be traveling along the other axis and will not want to turn at an intersection. In this case, the unwedger is forced to turn in the direction of the last turn it saw. This is a useful strategy for getting out of a *cul de sac*.

In most cases the turn initiated by the unwedger will be sufficient to direct the corridor follower toward a new corridor. If the robot has driven into a dead end, then the unwedger will fire every two seconds until the robot has turned itself around, at which point the robot leaves the way it came. Occasionally, Polly aligns itself with a corner and the unwedger and corridor follower fight — the unwedger turning away from the corner, and the corridor follower turning toward it. This problem is rare enough to be ignored.

10.2 Place recognition

Polly keeps track of its position by recognizing landmarks and larger-scale "districts." These places are given to it in advance. The lab and some of its landmarks are shown in figure 10.3. The job of the place recognition system is to determine, on each clock tick, whether it has entered the region corresponding to a landmark or district, and if so, what its qualitative coordinates are. The coordinates are only qualitative because the navigator only requires that they order the landmarks properly along the north/south and east/west axes. Therefore the coordinates can be warped by any monotonic, or rather strictly increasing, deformation. The coordinates of the different landmarks are shown in figure 10.3.

Information about landmarks is stored in an associative memory that is exhaus-

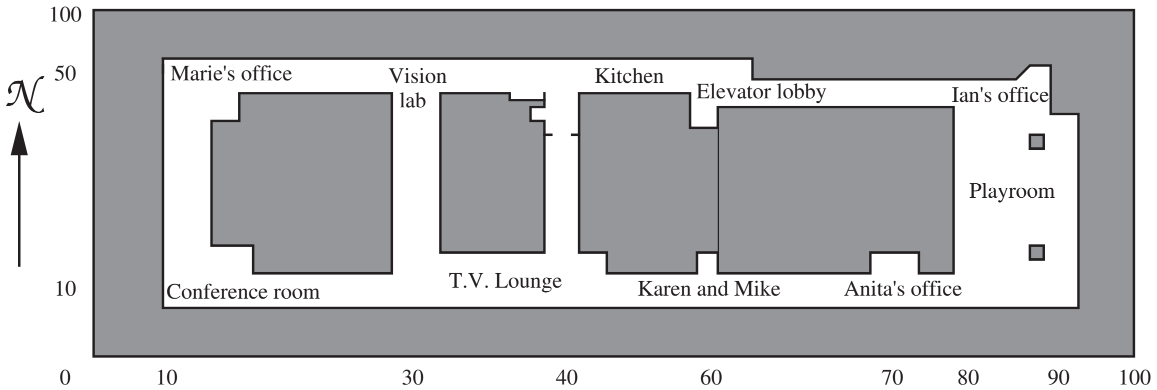


Figure 10.3: Landmarks in Polly's environment and their qualitative coordinates.

Kitchen	
Position	(50, 40)
Direction	west
Veer	0
Image	...

Corridor 1	
Position	(40, 40)
Direction	west
Veer	0
Features	left

Elevator lobby	
Position	(60, 40)
Direction	east
Veer	45
Features	right, wall

Figure 10.4: Example place frames.

tively searched on every clock tick (66ms). The memory consists of a set of frame-like structures, one per possible view of each landmark (see figure 10.4). In some cases, more than one frame per view is given to compensate for variations in lighting or geometry over time. Each frame gives the expected appearance of a place from a particular direction (north/south/east/west). All frames contain a place name, qualitative coordinates, a direction and some specification of the landmark's appearance: either a 16×12 grey-scale image or a set of qualitative features (left-turn, right-turn, wall, dark-floor, light-floor). No explicit connectivity information is represented. Frames can also be tagged with a speech to give during a tour or an open-loop turn to perform. The latter is used to deal with the jog in the corridor at the elevator lobby. The complete set of frames describing the 7th floor is given in appendix A.

While at first glance this may seem to be an inefficient mechanism, it is in fact quite compact. The complete set of 32 frames for the 7th floor requires approximately 1KW of storage. The system can scan all the frames and find the best match at 15Hz using only a fraction of the CPU.

The code for the frame matcher is shown in figure 10.5. The matcher process is implemented by the procedure `frame-matcher`. On each clock tick, the frame matcher computes a score for each frame. If the best frame's score is below threshold (low scores are good), the frame matcher takes it to be the robot's new position. The matcher then asserts the output `frame-strobe?` for one clock tick, latches the matched frame into the output `current-frame`, and latches its coordinates into `last-x` and `last-y`. To prevent false matches, the matcher disables itself when it is not moving or not in a corridor, has been recently blocked, or has recently matched another place. The latter two are implemented using the internal counter `frame-debounce`, that holds the number of clock ticks (15ths of a second) to wait before matching. The time to wait after matching a given frame is stored in one of the frame's slots. Its default value is 2 seconds (30 ticks).

The procedure `match-frame` computes scores (see figure 10.6). The score is the sum of three penalty terms:

1. position difference from the last recognized landmark
2. direction difference between the frame and the current odometry, and
3. appearance difference

The position penalty depends on direction: if the robot is moving east and the frame is west or north of the last position, then the penalty will be higher than if it were east of the last position.

Polly uses two ways of encoding landmark appearance. Image frames contain 16×12 grey-scale images of their respective landmarks, each taken from some specific position. An image frame has exactly one image, although a landmark could have multiple image frames. The appearance difference for an image frame is the sum of the squared differences of its pixels with the current image.

A feature frame contains a vector of binary features (left turn, right turn, dark floor, wall ahead) computed by the visual system. The appearance difference for a feature frame is small if the current feature vector and the frame's feature vector are exactly the same, otherwise large. Feature frames are only matched when a new feature appears or disappears in the world. The requirement that the feature bits change amounts to a requirement that the robot leave one intersection or open space before matching the next one. Because of the nature of the wall detector (it is only sensitive to walls at a specific distance), the *disappearance* of a wall does not enable frame matching. This prevents false positives when the wall bit is intermittent.

The frame matching relies on the fact that the robot only views landmarks from specific directions. The restricted viewing angle is due both to the geometry of the building (long narrow corridors) and to the fact that the corridor follower tends to center the robot in the corridor. Roughly speaking, the hard parts of recognition are variation in lighting, variation in viewpoint, variation in the landmark itself, and occlusion. Polly fails in the latter two cases (as do most systems), but handles the other two by relying on the fact that they don't vary much in this domain. Lighting in office buildings is usually internally generated and kept at a constant level. Most viewing parameters are held fixed by the camera mount. Most of the rest are fixed by the corridor follower. The only viewpoint parameter that normally varies is the robot's distance. But that parameter is searched *mechanically* as the robot drives.

```
(define (frame-matcher)
  (when (and corridor?
             (> speed #x1800))
    (when blocked?
      (set! frame-debounce 30))
    ;; FRAMES is the address of the list of frames to be matched.
    ;; FRAME is a pointer to the current frame.
    (let ((frame frames)
          (best frames)
          (best-value 99999999))
      (decf frame-debounce)
      (set! frame-strobe? false)
      (when (< frame-debounce 0)
        (find-districts)
        (dotimes (n frame-count)
          (let ((matchval (match-frame mini frame)))
            (when (< matchval best-value)
              (unless (and (= (frame-x frame) last-x)
                           (= (frame-y frame) last-y))
                (setf best frame)
                (setf best-value matchval))))
          ;; Advance FRAME to the start of the next frame.
          (setf frame (shift frame (if (image-frame? frame)
                                       image-frame-length
                                     feature-frame-length))))
        (when (< best-value 70000)
          (setf frame-debounce (frame-place-size best))
          (setf frame-strobe? true)
          (setf current-frame best)
          (setf last-x (frame-x best))
          (setf last-y (frame-y best))))
      (set! last-match-bits feature-bits))))
```

Figure 10.5: Source code for frame-matcher process. Code has been slightly simplified by removing compiler declarations.

```
(define (match-frame mini-image frame)
  (let ((fimage (frame-image frame))
        (sum 0)
        (bits-changed? (not (= (logand feature-bits #b11110)
                               (logand last-match-bits #b11110)))))
    (when wall?
      (set! bits-changed? true))
    ;; Compute penalties based on estimated position and direction of
    ;; motion.
    (let ((delta-x (- (frame-x frame) last-x))
          (delta-y (- (frame-y frame) last-y)))
      ;; Encoding of direction is north=0, east=1, etc.
      (incf sum (* (abs delta-x)
                   (vector-ref direction
                               (if (> delta-x 0)
                                   ;  N    E   S    W
                                   #(3000 100 3000 3000)
                                   ;  N    E   S    W
                                 #(3000 3000 3000 100)))))
      (incf sum (* (abs delta-y)
                   (vector-ref direction
                               (if (> delta-y 0)
                                   ;  N    E   S    W
                                   #(100 3000 3000 3000)
                                   ;  N    E   S    W
                                 #(3000 3000 100 3000))))))
    ;; Large penalty for getting the direction wrong.
    (unless (= (frame-direction frame)
               direction)
      (incf sum 100000))
    (if (image-frame? frame)
        ;; MINI-IMAGE is the 16x12 version of the current image.
        (incf sum (compute-difference fimage mini-image))
      (incf sum (if (and (= (frame-features frame)
                            feature-bits)
                         bits-changed?)
                    60000
                  10000000)))
    sum)))
```

Figure 10.6: Source code for matching process. Code has been slightly simplified by removing declarations and code for computing image differences.

No computational search is needed, and so viewpoint-dependent template matching is sufficient. I will call this the *constant viewpoint constraint*: whenever the robot approaches a landmark, its approach will includes some specific viewpoint (*i.e.* the one in our image frame).

Polly can also recognize large-scale "districts" and correct its position estimate even if it cannot determine exactly where it is. There is evidence that humans use such information (see Lynch [66]). The robot presently recognizes the two long east/west corridors as districts. For example, when the robot sees a left turn while driving west, it must be in the southern east/west corridor, so its y coordinate must be 10 regardless of its x coordinate. This helps Polly recover from recognition errors. At present, the recognition of districts is implemented as a separate computation. It ought to be folded into the frame system. District recognition is implemented by `find-districts` (see figure 10.7), which is called by `frame-matcher`.

10.3 Patrolling

Patrolling is implemented by the `wander` process which alternately sets the navigator's goal to the playroom (when the robot gets as far west as the vision lab) and the vision lab (when the robot gets as far east as the playroom). `Wander` implements both the patrol pattern and the path for giving tours. `Wander` always sets the navigator's goal, except when the robot starts up or goes home.

10.4 Sequencers and the plan language

Since parallelism is the default in Polly, a separate mechanism is necessary to introduce seriality. Polly uses a simple plan language to specify fixed action sequences. The macro `define-sequencer` creates a such a sequence. It takes a sequence of condition/action pairs and defines a new process (Scheme procedure) to run the actions, and a program counter to keep track of where the process is in the sequence. The process checks its program counter on each clock tick. If it is negative, it does nothing. If it is a positive number n, it checks the nth condition. If the condition is true, it executes the nth action and increments the program counter. The basic condition/action pair is given with the form:

```
(when condition
  action)
```

The condition and action may be arbitrary pieces of scheme code, but must terminate quickly, to allow the rest of the system to run before the end of the clock tick. Various bits of syntactic sugar are available to make it more readable: (`wait` *condition*) and (`then` *action*) are ways of specifying pairs with null actions or conditions,

```
(define (find-districts)
  ;; Use a stricter criterion for turns so that we don't get doorways.
  (let ((left-turn? (> left-distance 25))
        (right-turn? (> right-distance 25)))
    (when (and aligned?
               ns-corridor?
               wall?)
      (when (= direction north)
        (set! last-y 40))
      (when (= direction south)
        (set! last-y 10)))
    (when (and ew-corridor?
               (not dark-floor)
               aligned-long-time?
               (not blocked?))
      (when (and (= direction west)
                 open-right?
                 (not open-left?)
                 in-corridor?)
        (set! last-y 10))
      (when (and (= direction east)
                 (not open-right?)
                 open-left?)
        (set! last-y 10))
      (when (and (= direction west)
                 (not open-right?)
                 open-left?)
        (set! last-y 40))
      (when (and (= direction east)
                 (not open-left?)
                 (< last-x 70)
                 open-right?)
        (set! last-y 40)))))
```

Figure 10.7: Source code for find-districts. Some of the repeated tests could be collapsed into single disjunctions, but it would have required the author to have done a better job implementing boolean tests in his compiler.

```
(define-sequencer (leave-office :import (...))
  (first (pdisplay-goal "Leave office"))
  (when blocked?
    ;; robot should now be at far wall
    (set! global-mode 1)
    ;; turn toward door
    (turn! -40))
  (sleep 2)
  (when open-left?
    ;; robot should be out of office
    (pdisplay-goal "Align"))
  (sleep 2)
  ;; Now we want to make ourselves parallel with the corridor.
  ;; turn toward east wall of playroom
  (then (turn! -60))
  (sleep 1)
  (when (and (= direction east)
             blocked?)
    ;; Should be aligned with east wall; turn around
    (turn! 150))
  (wait ew-corridor?)
  (sleep 5)
  ;; Robot now thinks it's aligned; set position.
  (then
    (set! last-x 80)
    (set! last-y 40)))
```

Figure 10.8: Source code for the sequencer for leaving the author's office and entering the corridor. `pdisplay-goal` is a macro for displaying a string on the LCD display.

respectively. `(Sleep` *seconds*`)` causes the process to be inactive for the specified period.

A sequencer is started, or "fired," when another process executes the form `(do!` *sequencer-name*`)`. `Do!` simply clears the program counter. The sequencer executes until it finishes the last condition/action pair, at which point it sets its program counter to -1 and waits to be fired again. Multiple firings (executions of `do!`) do not produce multiple copies of the sequencer, they just reset its program counter.

Figure 10.8 shows an example sequencer used to leave the office and enter the hallway. It is fired at boot time. The `:import` parameter is a compiler declaration.

```
(define-sequencer (offer-tour :import (...))
  (first (set! global-mode  3)
         (set! inhibit-all-motion? true))
 (when done-talking?
   (new-say "Hello.  I am Polly.  Would you like a tour?
            If so, wave your foot around."))
 (sleep 9)
 (then (if blocked?
          ;; The person's still there.
          (if (> hmotion 30)
              (do! give-tour)
            (begin (new-say "OK.  Have a nice day.")
                   (set! global-mode 1)
                   (set! inhibit-all-motion? false)))
        ;; The person's gone.
        (begin (set! global-mode 1)
               (set! inhibit-all-motion? false)))))
```

Figure 10.9: Sequencer code for offering a tour.

10.5 Giving tours

The sequencers `offer-tour` and `give-tour` implement tour-giving. When the robot is in patrol mode, in an east/west corridor, is blocked by a person, and has not offered a tour in the last 5 seconds, the `interact` process fires `offer-tour`. `Offer-tour` then fires `give-tour` if the visitor accepts (see figures 10.9 and 10.10).

```
(define-sequencer (give-tour :import (...))
  (first (new-say "OK.  Please stand to one side.")
         (if (= last-y 40)
           (begin (set! tour-end-x 80)
                  (set! tour-end-y 10))
           (begin (set! tour-end-x last-x)
                  (set! tour-end-y last-y))))
  (when (not blocked?)
    (new-say "Thank you.  Please follow me.")
    (set! inhibit-all-motion? false)
    (set! global-mode 2))
  (wait frame-strobe?)
  (wait (at-place? tour-end-x tour-end-y))
  (sleep 1.0)
  (then
    (new-say "That's the end of the tour.
              Thank you and have a nice day.")
    (set! global-mode 1)))
```

Figure 10.10: Sequencer code for giving a tour.

Part IV

Results

Chapter 11

Experiments with Polly

> The great thing about optimality criteria is there are so many to choose from!
> — Anonymous robotics conference attendee

Polly is one of the best tested vision-based robots to date. The low-level navigation system has seen hundreds of hours of testing in several different environments. The coloring algorithm has been ported to half a dozen other robots and run at conferences, in classrooms, and in auditoriums. The high-level navigation system and the tour-giving components are newer and so less well tested. As of the Spring of 1993, the robot had given over a hundred tours.

The performance of complex robot systems is extremely difficult to quantify in any meaningful manner. It has been suggested to me that I use the the robot's RMS deviation from the mid-line of the corridor to evaluate the low level navigation system. Unfortunately, staying in the center of the corridor is not part of the task. It is neither necessary, nor particularly desirable for the robot to stay always exactly in the center. The task requires only that it safely get from one side to another. Alternatively, one could try to prove the time- or power-optimality of the paths it generates, but again, it is unclear that the 10% variations one might find between corridor followers would ever matter to a real user. Worse, those real-world users who do care about power (or time), care about *total* power consumption, including consumption by the computers. Motor power consumption is actually dwarfed by computer power consumption on Polly (the electronics use about 40W, compared to the 5W used by the motors). Since optimizing the last 10% may require a thousandfold increase in computer power (or worse), it is unclear that time or power consumption of the path alone, is at all meaningful.

The most serious problem with quantification is that all these measures ignore the cases where the robot fails completely, e.g. because its environmental assumptions fail. Given a choice between a "90% optimal" robot that "fails 2% of the time" and a "100% optimal" robot that "fails 10% of the time", users will generally choose the 90% optimal one. No one has the slightest idea what "90% optimal" or "fails 2% of the time" really mean in any realistic global sense, much less how to measure them. There are far too many variables involved for one to be able to do controlled experiments.

For these reasons, much of the data presented here are, of necessity, anecdotal. Information tends to be more qualitative than quantitative, and much of it is, in the end, dependent on the particular environment in which I tested it. This is true of any complicated artifact interacting with a complicated environment. I see no real alternative to this.

Since most of the time the robot works fine, I have focused on classifying the errors which do occur. I have tried to catalog the different types of failure modes that I have observed in the system as it has moved outside of its design envelope. I have also tried to document the ways in which it recovers from problems, and the cases in which it cannot. While this information is less definitive than numerical tests of speed and accuracy, I feel that it is more edifying. In the end, it gives a more accurate picture of the performance of the system. None of this is to say that the robot is unreliable. Quite the contrary: the robot works well enough in the normal cases for which it is designed that we can begin to focus on its performance in pathological situations. Many of the failure modes discussed here, such as darkened rooms, the presence of specular reflection or the presence of isoluminant edges, would cause problems for most vision systems.

11.1 Speed

Polly is the fastest indoor vision-based robot to date. It is fastest both in terms of processing speed and rate of motion. Table 11.1 shows the processing and driving speeds for a number of robots in the literature. Since the different systems perform very different tasks, it is difficult to draw firm conclusions from the numbers. They are sufficient to show that Polly is a fast system by any standard, however.

11.1.1 Processing speed

The low spatial resolution used on Polly allows the use high temporal resolution (frame rate). A summary of the processing time for different levels of capability is given in table 11.2. The reader should note that the elapsed time does not decrease as the amount of processing decreases, thus the system is I/O bound. The I/O time was spent waiting for the serial link to the base, waiting for VMEbus transfers to and from the frame grabber, and waiting for the frame grabber to finish grabbing a frame. Although it is hard to test this directly, I believe that the principle limitation was that the frame grabber could not grab consecutive frames without double-buffering, which

System	Frames/sec	MIPS	Speed	Obstacle detection?
Polly	15	16	1 m/s	Yes
Stanford Cart (Moravec)	0.001	.25 (?)	0.003	Yes
FINALE (Kosaka & Kak)	0.03	16	0.2	No
Mobi	0.1	1	0.1	Yes
Thorpe *et al*	0.1	1	0.4	No
VTIS	< 1 ?	?	1–5.5	No
VaMoRs-87 (Dickmanns)	25	1.32 (?)	26.4	No
VaMoRs-91 (Dickmanns)	25	?	14	Yes
SCARF (Crisman)	0.1	16	?	No
ALVINN (Pomerleau)	?	?	26.4	No
AuRa path follower	0.18	0.5	0.18	No

Table 11.1: Processing and driving speeds of various visual navigation systems, and the problems they solve. The first group are indoor navigation systems intended for office buildings, although the Stanford Cart was also run outdoors. The second group are outdoor path followers, mostly road followers.

could not be implemented without sacrificing video output for debugging. Even if this restriction were removed, it would only have shifted the bottleneck to the base. At 9600 baud, there is only time to transmit 32 bytes per video frame time to the base. Since most base commands require at least 11 bytes of I/O, between input characters and output prompts, there would not be sufficient time to perform all the necessary transactions with the base at video rate. At 15fps, which is one half video rate, there is just barely enough time to reset the velocities and acceleration caps, and to poll the base for the odometry.

There is room for a great deal of improvement. A better compiler might be able to halve the processing time. Since the system was I/O bound in any case, it was not worth implementing the compiler improvements.

11.1.2 Driving speed

Driving speed was limited by the dynamics of the robot base, not by processing speeds. Although the robot's center of gravity is well below its midpoint, it can still fall over when decelerating, since all torques are applied at the very bottom of the robot. The result is that the wheels stop moving but the robot's card cage keeps moving and the robot falls over. This can be controlled by imposing acceleration caps at the cost of greatly increasing the stopping distance of the robot. The other problem is that internal control system of the base is calibrated for unloaded conditions. When the base tries to go from a stop to its maximum speed, the motors cannot provide sufficient force to accelerate the base at the rate the control system wants. When the

Test	Time (sec)	Frames/sec
Full system	67	15
Corridor follower	67	15
I/O only	67	15
No I/O	15	67
No VP	10	100

Table 11.2: Execution times for 1000 frames. "Full system" is all code presently implemented, including the person detector. "No I/O" is the corridor follower without any frame grabbing or output to the base (a single frame is grabbed at the beginning and processed repeatedly). "No VP" is the collision avoidance system run without I/O or the vanishing-point box. All execution times are for a Texas Instruments TMS-320C30-based DSP board (a Pentek 4283) running with no wait states. The processor has a 60ns instruction time. The first three lines are the same because the system cannot digitize frames faster than 15 FPS.

control system notices that it isn't moving fast enough, it signals an error and shuts down. Therefore speeds over 1m/s require pushing the robot to help it accelerate. The control system can be disabled, allowing the DSP to directly specify motor currents, but then the robot tends to pop "wheelies," and so again, human intervention is required to keep it from falling over. I believe the system has run as fast as 2.5m/s in this mode, but I have not calibrated velocity measurements. Apart from these stability and low-level control considerations, the navigation system appears to be able to pilot the robot as fast it is physically capable of moving.

Polly has been run on two different bases. The robot was generally run at 1m/s on the old base. Unfortunately, the base was damaged during the testing of the bump sensors and so a new base was substituted. The new base, while otherwise functionally equivalent, had a different gear ratio and internal control system. This new control system was grossly under-damped. The new base regularly accelerated to 2m/s regardless of the velocity it was given, at which point it would brake and fall over. It was necessary to add extra damping within the DSP's control system to compensate for this. Even so, the new base can only safely run at 0.75m/s.

11.2 Complete test runs

The night the pick-up system was first implemented, I videotaped four test runs. Traces of the test runs are shown in figures 11.2 through 11.7. The robot successfully gave the tour in three of the four tests.

In the first run, the visitor misunderstood the instructions given by the robot and thought the robot said to move out of the way. The visitor moved part-way out of

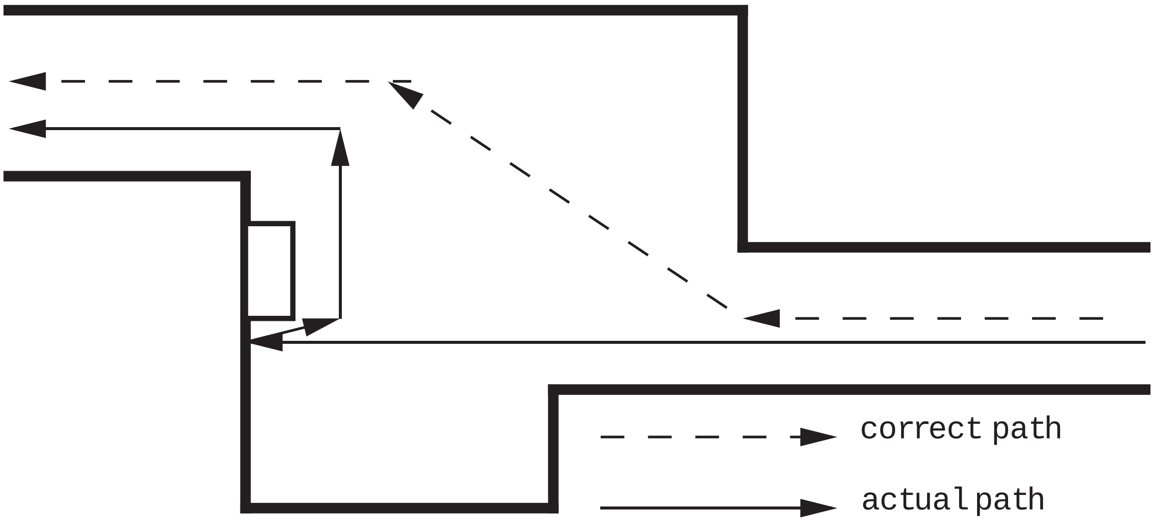


Figure 11.1: Detail of the area near the elevator lobby with the robot's intended path and actual path during the first test run..

the way, and the robot misinterpreted the motion as a wave of the foot. It correctly believed that the visitor wanted a tour, but for the wrong reasons. When the robot started the tour, it was too close to the elevator lobby to recognize it as a landmark. It had not recognized it previously because it was occluded by the visitor. The robot should have recognized the area and veered to the right, but instead continued forward until it was blocked by the wall. Fortunately, the unwedger and the navigator were able to successfully pilot the robot into the next corridor, in spite of the failure (see figure 11.1). Once the robot reached the kitchen, the place recognition system resynced itself, and the robot continued without incident.

In the second test, a place recognition error caused it to make an inappropriate turn from which it took a long time to recover. The run was terminated, and the robot was allowed to run, with the navigator turned off, until the place recognition system resynced itself. The robot made no errors after that point. The other test runs were performed properly.

11.3 Other environments

Much of Polly's efficiency is due to its environmental specialization. The robot is not specialized to a particular environment, but to a class of environments, its "habitat." The aim of the analytic part of this work is to understand the environmental features necessary to the robot's operation. Put another way, we want to understand what the robot's habitat really is. According to the analysis, the low-level navigation code should work in any environment with corridors with textureless carpets and where

Place	Action	Comments
corridor	drive west	
	stop	detected person
	“Hello...”	introduction speech
	pause	confused visitor moves out of the way
	“Please stand to one side”	Polly makes mistake visitor moves
	drives forward	
	“Thank you, please follow me”	visitor follows
elevator lobby	drives west	robot misses landmark
west wall	stop	
	turn right	unwedger takes over
	drive north	
north wall	stop	
	turn left	unwedger takes over robot now on course
	“I can avoid obstacles...”	chatter speech
kitchen	“...follow corridors...”	robot still making previous speech
intersection	“On the left here...”	kitchen place speech
	“My vision system runs...”	chatter speech
vision lab	turn left (south)	
	“On the right here is...”	Vision lab speech
	drives south	
T.V. lounge wall	stop	
	“By the way...”	chatter speech
	turn left (east)	unwedger takes over
	drives east	
couch	drives east	
end of couch	“This the T.V. lounge...”	place speech
end of lounge	“God, this place...”	chatter speech
	drives east	
Karen’s office	“This is...”	place speech
	drives east	
Anita’s office	“This is...”	place speech
	drives east	
playroom	turns left	
	“This is...”	place speech
	drives north	
near chair	stops	
	turns left	obstacle avoidance
	“This is the end...”	Goodbye speech
	drives	cruise mode

Figure 11.2: Transcript of first test run.

Place	Action	Comments
west wall	stops	
	turns right	obstacle avoidance
	drives north	
north wall	stops	
	turns left	unwedger takes over
	drives west	

Figure 11.3: Transcript of first test run (cont'd.)

Place	Action	Comments
corridor	drive east	
	stop	detected person
	"Hello..."	introduction speech
	pause	visitor waves foot
	"Please stand to..."	visitor moves
	drives forward	
	"Thank you, please follow me"	visitor follows
Anita's office	"This is the playroom..."	place recognition error
	turns left	
	drives north	drives into printer alcove

Figure 11.4: Transcript of second test run.

Place	Action	Comments
corridor	drive east	
	stop	detected person
	"Hello..."	introduction speech
	pause	visitor waves foot
	"Please stand to..."	visitor moves
	drives forward	
	"Thank you, please follow me"	visitor follows
Anita's office	"This is..."	place speech
	drives east	
	"I can avoid obstacles..."	chatter speech
playroom	turns left	
	drives north	
	"This is..."	place speech
	drives north	
near chair	stops	
	"My vision system..."	chatter speech
	turns left	obstacle avoidance
	drives	
west wall	stops	
	turns right	obstacle avoidance
	drives north	
north wall	stops	
	turns left	unwedger takes over
	drives west	
	"By the way..."	chatter speech
	drives west	
elevator lobby	veers right	
north wall	stops	
	turns left	obstacle avoidance
	drives west	
kitchen	"On the left..."	place speech
	drives west	
	"God,..."	chatter speech
vision lab	turns left	
	"On the right..."	place speech
	drives south	
T.V. lounge wall	stop	
	turn left (east)	unwedger takes over
	drives east	

Figure 11.5: Transcript of third test run.

Place	Action	Comments
couch	drives east	
end of couch	"This the T.V. lounge..."	place speech
	drives east	
Karen's office	"This is..."	place speech
	drives east	
	"This is the end..."	goodbye speech

Figure 11.6: Transcript of third test run (cont'd.).

all obstacles rest on the ground. I have conjectured that these properties are true of many office buildings, and I have tried to test the conjecture empirically by testing the robot in alternate environments.

Note that *no parameters were changed for any of these tests.* The only difference between the code run for these tests and the code run on the 7th floor is that the navigator was disabled (because the robot had no map for the other floors).

11.3.1 Tech Square

Tech Square is the common term for the building which houses both the AI lab and the Laboratory for Computer Science. The first test of Polly was to take it to different floors of Tech Square. The building has nine floors plus a basement. The corridor follower and obstacle avoidance work on all floors except the ninth floor and the basement. These floors have shiny linoleum tile rather than carpet. The tile is shiny enough that it acts like a dirty mirror when viewed from two feet above the ground. The result is that images of the overhead lights appear in the floor. Worse yet, they move with the robot so that if one happens to be close enough to the robot to make it back up, the robot will back into a wall. The roughness of the floor serves to diffuse the light somewhat, so it is possible that the edge detector could be tuned to a high enough frequency band to ignore the lights.

Many floors have carpet boundaries. The robot had no problems with the boundaries which appeared horizontal in its field of view because the carpet boundary detector was specifically tuned to horizontal boundaries, but some boundaries paralleled the walls. Such boundaries restricted the robot to the channel between the wall and the boundary. If the robot had turned 90 degrees at the right point, it probably could have crossed the boundary, but it did not do so. A more intelligent algorithm for detecting carpet boundaries would be a great asset.

Place	Action	Comments
corridor	drive west	
	stop	detected person
	"Hello..."	introduction speech
	pause	visitor waves foot
	"Please stand to one side"	visitor moves
	drives forward	
	"Thank you, please follow me"	visitor follows
elevator lobby	veers right	
north wall	stop	
	turn left	avoid obstacles
	"I can avoid obstacles..."	chatter speech
kitchen	"...follow corridors..."	robot still making previous speech
intersection	"On the left here..."	kitchen place speech
	"My vision system runs..."	chatter speech
vision lab	turn left (south)	
	"On the right here is..."	Vision lab speech
	drives south	
T.V. lounge wall	stop	
	"By the way..."	chatter speech
	turn left (east)	unwedger takes over
	drives east	
couch	drives east	
end of couch	"This the T.V. lounge..."	place speech
end of lounge	"God, this place..."	chatter speech
	drives east	
Karen's office	"This is..."	place speech
	drives east	
Anita's office	"This is..."	place speech
	drives east	
playroom	turns left	
	"This is..."	place speech
	drives north	
near chair	stops	
	turns left	obstacle avoidance
	"This is the end..."	Goodbye speech
	drives	cruise mode

Figure 11.7: Transcript of the last test run.

Place	Action	Comments
west wall	stops	
	turns right	obstacle avoidance
	drives north	
north wall	stops	
	turns left	unwedger takes over
	drives west	

Figure 11.8: Transcript of the last test run (cont'd.).

11.3.2 The Brown CS department

Polly was also tested at the Computer Science Department at Brown University. Again, the system worked well: it successfully followed corridors, avoided obstacles, including people, and moved through open spaces. There were problems, however. The robot's major problem was lack of light. The lights were dark enough to seriously inhibit the response of the video camera. Since many of the edges between the walls and the carpet were very low contrast to begin with, the edge detector would often miss them when running at low light levels. If robot saw one wall, but not the other, it would sometimes drive into the invisible walls. Again, this situation would be a problem for nearly any vision system.

Two aspects of the environment's geometry were problematic. One was the presence of a downward staircase. It is possible that the robot would have seen the staircase boundary and stopped, but I was not brave enough to test it. Also, some corridors were too narrow for the robot to turn around without backing into a wall in the process. The problem was that the corridors were narrower than Polly's minimum safe distance. The robot could perform in-place 180 degree turns, but when it reached dead ends and relied on the unwedger to turn it 45 degrees at a time, it had problems. This could be solved with a rear bumper.

11.4 Burn-in tests

I conducted a number of "burn-in tests" on the robot, by letting the robot run laps for 45 minutes at a time. I performed five controlled burn-in tests. The longest of which was 90 minutes. All runs worked well, running without incident for extended periods. Three of the five runs ran flawlessly until terminated by human intervention. The other two are described below.

In the first test, the robot ran for 45 minutes without incident. The test was terminates by a series of mishaps. The base, being under-damped, started up too fast and then broke, almost tipping. This generated an internal error in the firmware control system of the base, causing the base to halt and limp itself. I reset the base, but forgot that resetting the error also resets the odometer, so that the robot would think it was pointing south, which it was not. The robot then performed most of a lap in spite of having wildly incorrect odometry information. I reset the odometry when I realized the problem. However, I accidentally triggered the go-home function in the process. At the time, go-home could not be turned off and, worse, was not interlocked with the wanderer. The go-home sequencer and the wanderer fought for control of the navigator for a few more laps. When the wanderer eventually brought the robot near my office, the go-home routine forcibly parked the base, ending the run. The whole series of mishaps took about three laps to complete.

The final test was terminated by the failure of the motor battery. The battery appears to have been weakened by the racing in the control system of the new base. When the robot maneuvers through tight passages or oddly shaped areas, it makes frequent start and stops, causing the internal base control system to race. When racing, the base typically accelerates to 2m/s, then brakes to 0.5m/s. Racing pours huge amounts of energy into the motors to accelerate the robot, only to pour more energy into the motors to slow it down again. The racing seriously weakened Polly's motor batteries. Alas, the weak batteries lead to sluggish motor performance, which seems to cause the internal base control system to race even more. To make matters worse, the playroom had just been rearranged by workmen and so the freespace channel which the robot had to follow through the playroom was more complicated than usual (see figure 11.9). The complexity meant more starting and stopping. Toward the end of the run, the base repeatedly shut itself down because it could not obtain sufficient performance from the motors. The shutdowns required rather elaborate human intervention to restart the base, while maintaining the its odometric data.

Apart from these problems, the robot functioned well, running roughly 30 laps. The robot made three errors. The first was due to the overhead lights having been replaced with dimmer ones. The space between two lights had a strong enough shadow on the right side to drive the robot away (the robot avoids places which are dark enough to make the edge detector fail). The robot veered to the left in a sufficiently gradual manner that it still thought it was in a straight corridor, see figure 11.10. When it finally past the shadow, it was turned so far to the left that the right wall was out of view. Since it believed it was aligned the corridor, it concluded it had come to a landmark and turned accordingly. Once the problem was diagnosed, it could be compensated for by placing an obstacle across from the shadow to prevent the robot from veering. This solved the problem. Unfortunately, it is hard to compensate for the new lights. Dropping the darkness threshold did in fact prevent it from making the error, but also prevented the robot from halting when it approached dark obstacles in dark areas. It appears that either brighter lights or a more sensitive camera are the only answers.

The other two problems were due to shafts of bright morning sunlight coming

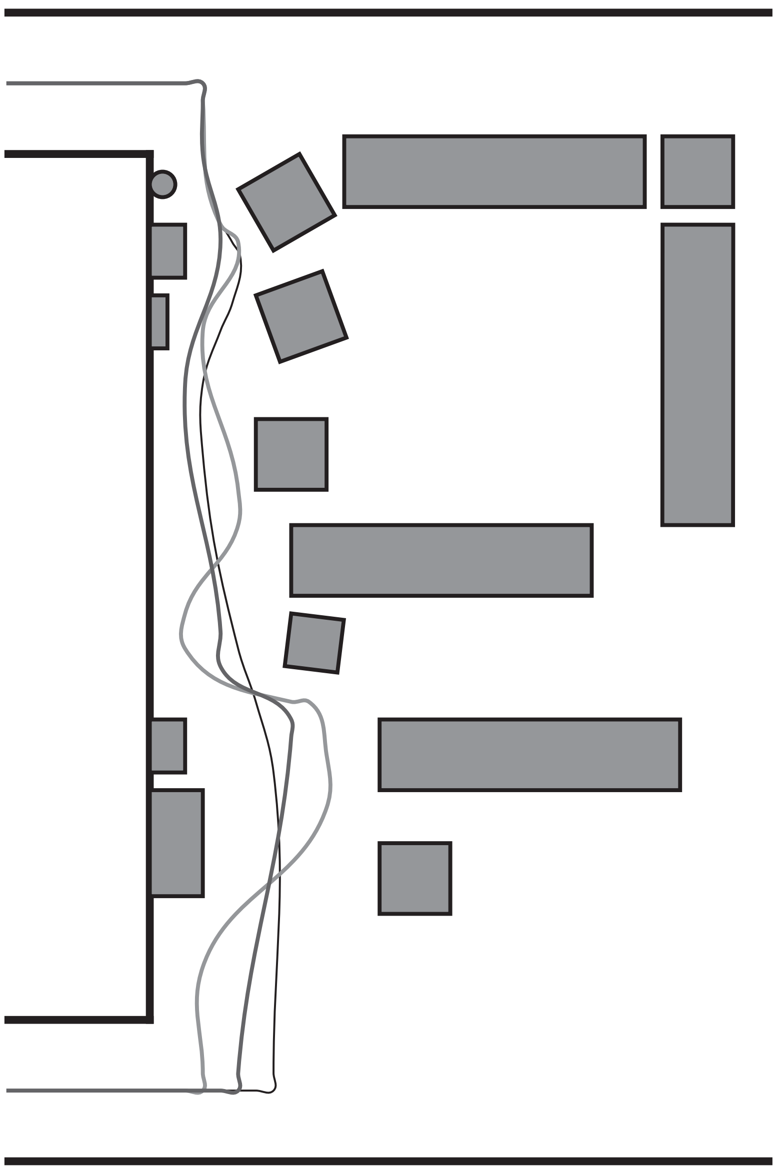


Figure 11.9: The layout of the playroom after begin rearranged by workmen, and the typical paths made by the robot. Note that NEITHER LAYOUT NOR PATHS ARE METRICALLY ACCURATE (neither could be measured accurately). Both are qualitatively accurate however, and the paths accurately represent which pieces of furniture redirected the robot's path.

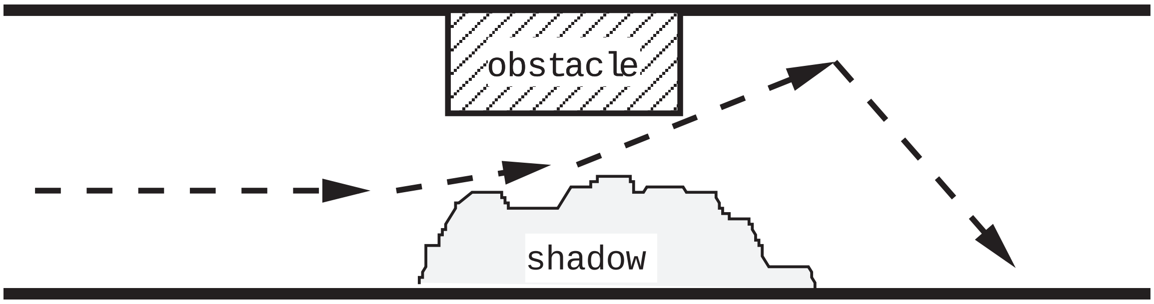


Figure 11.10: The shadow problem in the last burn-in run. When the overhead lights were replaced, one area of the corridor was dark enough that the robot could not see within it. The robot avoided the area, but lost sight of the right wall when it passed it and thought it had reached the elevator lobby. The robot then veered to its right as it should in the elevator lobby.

out of office doors. The shafts of sunlight saturate the camera pixels and appear to be obstacles. The robot halted and tried to find a way around the sunlight in vain. In one case, it tried to enter an office and grazed a doorway on its way in. Both problems were solved by closing office doors. A complete log of the run is given in appendix B.

11.5 Limitations, failure modes, and useful extensions

11.5.1 Low-level navigation

In general, all low-level navigation problems are obstacle detection problems. Fortunately, most of these are false positives rather than false negatives so the system is very conservative. The system's major failure mode is braking for shafts of sunlight. If sunlight coming through office doors in into the hallway is sufficiently strong it causes the robot to brake when there is in fact no obstacle. Shadows are less of a problem because they are generally too diffuse to trigger the edge detector.

False negatives can be caused by a number of less common conditions. The present system also has no memory and so cannot brake for an object unless it is actually within the camera's field of view. Some of the objects in the lab have the same surface reflectance as the carpet on which they rest, so they can only be distinguished in color. Since the robot only has a black and white camera, it cannot distinguish these isoluminant edges. The edge detector can also fail in low light levels. Of course, most vision systems are likely to miss an object if they cannot even find its edges so this failure mode should not be surprising.

The inability to see backward is the other major collision avoidance problem. Adding a rear bumper would solve most of the backing-into-wall problems.

11.5.2 The navigator

High-level navigation performance is determined by the accuracy of place recognition. In general, the system works flawlessly unless the robot gets lost. When the robot gets lost, the navigator will generally overshoot and turn around. If the robot gets severely lost, the navigator will flail around until the place recognition system gets reoriented. The worst case is when the place recognition system thinks it is east of its goal when it is actually at the western edge of the building (or west of the goal when it is east). In this case, the navigator unit and the unwedger continually make opposite course corrections. The navigator should probably be modified to give up in these situations. In general, the system would benefit from being able to explicitly notice that it's lost.

11.5.3 The unwedger

In general, the unwedger works very well. Its one failure mode appears when the robot gets into a corner. The corridor follower then tries to point the robot into the corner. When the unwedger turns the robot 45 degrees, it is insufficient to move it out of the attractor basin of the corner. Thus the robot loops, alternately turning toward and away from the corner. This could be fixed by increasing the turn angle, but that can cause problems in other situations. Perhaps the best solution would be to explicitly note the looping behavior and force a u-turn.

11.5.4 Place recognition

While recognition by matching images is quite general, it is fragile. It is particularly sensitive to changes in the world. If a chair is in view when a landmark template is made, then it must continue to be in view, and in the same place and orientation, forever. If the chair moves, then the landmark becomes unrecognizable until a new template is made. Another problem is that the robot's camera is pointed at the floor and there isn't very much interesting to be seen there. For these reasons, feature frames are preferred over image frames. The only image-based landmark is the kitchen. In ten trials, the robot recognized the kitchen eight times going west and ten times going east. Westward kitchen recognition fails completely when the kitchen is rearranged.

Both methods consistently miss landmarks when there is a person standing in the way. They also fail if the robot is in the process of readjusting its course after driving around an obstacle or if the corridor is very wide and has a large amount of junk in it. Both these conditions cause the constant-viewpoint constraint to fail. The former can sometimes cause the robot to hallucinate a turn because one of the walls is invisible.

Recognition of districts is very reliable, although it can sometimes become confused if the robot is driven in a cluttered open space rather than a corridor.

11.5.5 Camera limitations

Dynamic range is a major problem for vision systems. Polly suffers greatly from the fact that its camera and digitizer only have a dynamic range of 10:1 or 20:1, which means that any scene illuminated by both office lights and sunlight will necessarily lose pixels on one or both ends of the scale. These regions are effectively invisible.

Polly also suffers greatly from its limited field of view. Although its field of view is huge by most standards, it can still easily miss nearby obstacles because they are our of view. The field of view also causes intersection detection problems. Without being able to turn its head, it cannot look to see if it is really passing another corridor.[1] A wider field of view and/or the ability to steer the camera would be a great asset.

Finally, I would have liked very much to have been able to use color or stereo information. I was not able to do so because of hardware limitations in the frame grabber. I believe that both of these cues could greatly improve the performance of the robot.

11.5.6 Multi-modal sensing

Given the current state of vision technology, it is a bad idea to rely exclusively on vision for obstacle avoidance, particularly vision based on a camera with the limitations discussed above. I avoided other sensing modalities because of limited time, but the use of other modalities would be essential to the development of an industrial strength version of Polly. I have found that bump switches are extremely useful. Papering one's robot with bump switches seems like a very good idea. Unfortunately, bump switches are of little or no use at high speeds. A bump switch extending 10cm forward only gives a 10ms collision warning at 1m/s. Stopping in 10ms would require a $10g$ deceleration.

Finally, it would also be extremely useful to use translational odometry to disambiguate adjacent landmarks that are visually similar. Polly's hardware is physically capable of it, but it would have required much complicated device drivers.

[1]The robot could stop dead, turn, look, and turn again, but this would both waste time, and greatly increase the risk of its getting rear-ended by a human.

Chapter 12

Summary and conclusions

12.1 Why Polly works

Polly's efficiency and reliability are due to a number of factors. Specialization to a task allows the robot to compute only the information it needs. Specialization to a habitat allows the robot to substitute simple computations for more general ones.

Polly is an existence proof that a robust system with a large behavioral repertoire can be built using simple components specialized to their task and environment. It also demonstrates how we can analyze specialization so that we may better understand a system and transfer the insights gained in its design to the design of other systems.

There are also a number of architectural features in Polly's favor. Polly uses multiple strategies in parallel to reduces the likelihood of catastrophic failure. When the strategies have independent failure modes their combination can be quite robust. When the vanishing point computation generates bad data, the depth-balancing strategy compensates for it and the distance control system prevents collisions until the vanishing point is corrected.

Polly's control and perception loops run very fast (all visual percepts and motor commands are recomputed every 66ms) so it can rapidly recover from errors. We can think of Polly's control problem as being the problem of generating the next 66ms of safe path. Even at 1m/s, 66ms is only 6.6cm or about 3 inches. Other navigation systems that only process an image every few seconds or minutes, must compute path segments on the order of meters. Verifying the safety of a 6cm path is simply a lot easier than verifying a 1m path, particularly if the size of the freespace around the robot is less than a meter. Before committing to the 1m path, a robot needs to make very precise and reliable measurements to insure that that path will in fact be safe. Those measurements take a great deal of computing time. But if it takes the control system ten seconds to make a control decision, then the control system would have to commit to a path at least 10m long to maintain a speed of 1m/s. But that, in turn, requires even more precise measurements, and so on, resulting in a vicious circle.

The robot's small size significantly simplifies its navigation problems. Office buildings are designed human-sized creatures: the width of a doorway is determined by human shoulder width. My office door is 36 inches and my shoulders 20 inches. That leaves an 8 inch clearance on either side. Even a robot which is only 2 feet in diameter has only 6 inches of clearance, less than the minimum operating distance for standard Polaroid sonar sensors. The robot must rely on very short range sensors, such as infrareds, or on tactile feedback. Tactile feedback is very bad idea for a robot weighing 100 pounds or more. Polly, being only 1 foot wide, can survive a great deal slop in its navigation. Its small size and (relatively) light weight also simplify vehicle dynamics at high speeds.

12.2 Lightweight vision

There are few things one can prove about vision as a whole. I do not claim that lightweight systems are a replacement for reconstruction systems. Nor do I claim to have proven the long-term efficacy of task-based vision, active vision, or qualitative vision as research strategies. This report is best taken as a reminder that there are a number of resources available to the designer of vision systems that may have been forgotten.

The most important such resource in this work has been the structure of the environment. The presence of simplifying structures in the environment often allows simple computations to be substituted for expensive ones. This simplification can be done in a principled manner by making the structures explicit in the form of habitat constraints and describing the simplification in the form of a general lemma. The advantage of the lemma is that it allows other people to perform the same optimization in the future. It also shows what is important about the simplification and what is not. The particular edge detector used by my robot for figure ground separation is unimportant. What is important is the fact that any (thresholded) linear filter restricted to the right band will work.

The structure of the task is a complementary resource. It tells the designer what information is needed and what performance is required. More importantly, it tells the design what information is *not* needed and what performance is *not* required. Computing more information means computing more representations, which is obviously more expensive, or squeezing more information into the existing representations, which is often even worse. The first principle taught in most undergraduate AI classes is that a good representation makes the important information explicit, *and nothing else*. As one squeezes more and more information into a representation, the information becomes less explicit, so non-trivial processing is required just to look at the representation and extract the information that was needed in the first place.

Any representation with all the expressive power of the original image, is likely to be nearly as difficult to interpret.

Improving performance parameters, on the other hand, generally requires making trade-offs, usually either trading cost for performance, or trading one performance characteristic for another. Improving unimportant performance parameters is not only wasted effort, it is also a waste of other performance parameters.

Resolution is a useful case in point. Many researchers I have talked to have taken it for granted that images below 128×128 are mostly useless, whereas Polly uses resolutions as low at 16×12. Polly demonstrates that surprisingly good results can be obtained with surprisingly low resolutions (see Horswill and Brooks [51] and Pomerleau [81] for other examples). Obviously, some tasks and environments require higher resolution, but many do not. Nor does a system need to sample everything at the same resolution. If a smooth, finely textured object moves past the camera, then the intensity field will vary rapidly in both space and time and so will have to be finely sampled to estimate the motion field. The motion field itself however, will vary much more slowly and need not be sampled nearly as finely.

Fear of Vision

While it can be difficult to get people to admit to it in writing, there is a common attitude in both AI and robotics that vision is not even worth considering as a sensor because of its expense and unreliability. Of course sometimes it really is expensive. When tasked with constructing digital terrain maps from high resolution satellite images, the only option is to build a full stereo or shape from shading system and use whatever computational resources are necessary to do the job. However, simpler options are available for simpler tasks, such as corridor following. A reactive planning researcher wanting to give her planner something to execute, will be content with anything that does the job. Many researchers have put up with inadequate sensory suites, simply because they felt that vision was impractical. I hope this work succeeds in convincing such users that vision is safe to try.

12.3 Studying the world

I have argued that we need to study not only the structure of agents, but the structure of the environment and the relationships between agent structures and environment structures. In short, that we need to study the world. The approach I have used has been to define a formal semantics on a set of possible agents and a set of transformations over agents which preserve those semantics when some constraint holds. This approach allows the assumptions made by an agent to be separated and their computational significance drawn out. The assumptions (constraints) can then be cataloged for use in future designs and checked against different domains to make computational profiles of those domains. There are doubtless approaches to be explored.

One may object that this project is applicable only to specialized systems, not to truly general systems. However, truly general systems are extremely rare. Most "general" vision systems tacitly assume the world is non-specular or that surfaces are almost completely smooth. Others assume it is piecewise planar. Most planners assume that the world is completely stable. That requires, among other things, that the planner be the only agent in the world. Most also assume that the world is deterministic or only boundedly unpredictable, so the effects of an action can be easily determined in advance. These are not *necessarily* faults. They are habitat constraints, no more, no less. Habitat constraints can only be judged by their usefulness and their match to an agent's environment. The choice is not whether to use habitat constraints, but which ones to use. However one chooses constraints, they need to be stated explicitly along with their computational significance. This is true both for simple reactive robots and for complex reasoning systems. In short, we need to learn to be intelligent consumers of specialization.

We also need to understand the real underlying difficulties of the actual test domains of our systems. This can only be done by examining world and agent together. Failure to analyze our test domains seriously undermines our conclusions about our agents.

Studying the world can also help us design "general-purpose" systems. It helps make clear which tasks are hard or easy. Problems are often very hard in the general case, but if the majority of actual problem *instances* encountered by an agent are relatively simple, the agent may thrive by using simple methods whenever possible, saving its cognitive resources for the truly hard instances.

AI is and must be a natural science. It must continually make and test hypotheses about the nature of the external world. Our algorithms, representations, and formalizations of the world must eventually be compared with external reality. Doing so early reduces the risk of wasted effort. To understand intelligence, we must study not only ourselves but the world in which we live.

Appendix A

The frame database

The following is the database of frames presently in use by the robot. Each frame is defined by a `defframe` form which takes the name of the landmark, its coordinates, a set of features, and the direction the robot would be pointing if it were at that landmark and saw those features. The features can be an image, or they can be bits. The possible bits are left, right, wall, and dark (meaning there is a dark floor here. If the appearance of a place is variable, multiple frames can be used to specify the possible appearances. The `:veer` parameter is the amount to veer left or right (in degrees, right is positive), when reaching that landmark. The default veer is zero. The `speed` parameter is used to specify a string to send to the voice synthesizer when giving tours. Note that the spelling required to get the voice synthesizer to say the right thing can be rather odd. The `:place-size` parameter is the amount of time it should take the robot to drive through the landmark. The default is 2 seconds. The `:passage?` parameter means that the landmark is a grid point, and so the navigator should consider making turns at the landmark.

```
(defframe "Ian's office" ; starting point;  Ian's office.
  :x 90 :y 40
  :direction south
  :left? t)

(defframe "Ian's office"
  :x 90 :y 40
  :direction east
  :speech "This is eean's office."
  :place-size 20
  :left? t :wall? t :dark? t)

(defframe "T.K.'s office"
  :x 80 :y 40
```

```
  :direction west
  :place-size 10
  :passage? t
  :left? t :dark? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction west
  :veer 35
  :place-size 6
  :right? t :wall? t :dark? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction west
  :veer 45
  :place-size 6
  :right? t :wall? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction west
  :veer 45
  :place-size 6
  :right? t :dark? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction west
  :veer 45
  :place-size 6
  :right? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction east
  :veer 45
  :place-size 6
  :right? t :wall? t)

(defframe "Kitchen"
```

```
  :x 50 :y 40
  :direction west
  :speech "On the left here, we have the copier room and the kitchen."
  :image ...)

(defframe "Kitchen"
  :x 50 :y 40
  :direction east
  :image ...)

;; The control algorithms loose at this junction so the
;; place size is large to give them a chance to
;; stabilize.
(defframe "Corridor 1"
  :x 40 :y 40
  :direction west
  :passage? t
  :place-size 5
  :left? t)

(defframe "Corridor 2"
  :x 30 :y 40
  :direction west
  :passage? t
  :speech "On the right here is the vizon lab."
  :left? t)

;;; Kluge: this is to deal with the effects of having
;;; Marc Raibert's door open at the end of the hall on
;;; sunny days.  It overwhelms the AGC on the camera
;;; and makes the whole corridor appear to be dark like
;;; the playroom.
(defframe "Corridor 2"
  :x 30 :y 40
  :direction west
  :passage? t
  :speech "On the right here is the vizon lab."
  :left? t :dark? t)

(defframe "Corridor 1"
  :x 40 :y 40
```

```
  :direction east
  :passage? t
  :right? t)

(defframe "Corridor 2"
  :x 30 :y 40
  :direction east
  :passage? t
  :speech "On the right here is the vizon lab."
  :right? t)

(defframe "Corridor 2"
  :x 30 :y 40
  :direction east
  :passage? t
  :speech "On the right here is the vizon lab."
  :right? t :dark? t)

(defframe "Marie's office"
  :x 10 :y 40
  :direction west
  :veer -30
  :left? t)

(defframe "Marie's office"
  :x 10 :y 40
  :direction west
  :veer -40
  :left? t :right? t)

(defframe "Marie's office"
  :x 10 :y 40
  :direction north
  :right? t)

(defframe "Marie's office"
  :x 10 :y 40
  :direction north
  :right? t :left? t)

(defframe "Conference room"
```

```
  :x 10 :y 10
  :direction west
  :right? t)

(defframe "T.V. Lounge"
  :x 30 :y 10
  :speech "This is the T.V. Lounj."
  :direction east
  :passage? t
  :left? t)

(defframe "T.V. Lounge"
  :x 40 :y 10
  :speech "This is the T.V. lounj.  We waste a lot of time here."
  :direction east
  :lefT? t :place-size 10)

(defframe "T. V. Lounge"
  :x 40 :y 10
  :direction west
  :speech "This is the T.V. lounj.  We waste a lot of time here."
  :place-size 10
  :right? t)

(defframe "T.V. Lounge"
  :x 30 :y 10
  :passage? t
  :direction west :right? t)

(defframe "Anita's office."
  :x 70 :y 10
  :direction east
  :speech "This is the office of Anneeta Flin,
          superwoman of the Nineteez."
  :left? t :place-size 3)

(defframe "Anita's office"
  :x 70 :y 10
  :direction west
  :place-size 3
  :right? t)
```

```
(defframe "Robert's office"
  :x 71 :y 10
  :direction east
  :place-size 0
  :right? t :dark? t)

(defframe "Play room"
  :x 80 :y 10
  :speech "This is the Playroom."
  :direction east
  :passage? t
  :left? t :dark? t)

(defframe "Play room"
  :x 80 :y 10
  :direction east
  :passage? t
  :speech "This is the Playroom."
  :left? t :right? t :dark? t)

(defframe "Hi Karen"
  :x 60 :y 10 :direction west
  :right? t)

(defframe "Hi Karen"
  :x 60 :y 10 :direction east
  :speech "This is Karen and Mike's office."
  :left? t)
```

Appendix B

Log of the last burn-in run

7:50 AM Make freespace channel in playroom.

Started robot.

Robot consistently veers to left by room 739, loses the wall, and mistakes the room for the elevator lobby.

Experimented with changing lighting conditions in room 739. No effect.

8:10 Have determined that the problem is due to a shadow cast by the new lights. The shadow is dark enough to force the robot to avoid it.

Problem is solved by placing a new obstacle across from the shadow to balance it. Will try dropping the shadow threshold later.

8:30 Door to room 793 is opened. Morning sun casts a band of light strong enough to be a barrier into the hallway. Robot enters office, skims doorway, and halts. I reorient and reset the base and the robot continues. The robot believes it is at $(40, 40)$, when it is at $(30, 10)$. The place recognition system resets at the playroom and continues normally.

8:40 Uneven geometry of the playroom is making the robot do a lot of obstacle avoidance. The control system of the base is racing whenever it starts up after being blocked. No collisions so far. Damping in the DSP is compensating. Robot's path through playroom is quite variable.

8:45 Base control system seems worse. It "popped a wheelie" while accelerating from a stop in the playroom.

9:00 Another wheelie.

9:05 Base does another wheelie and shuts down. I manually reset the base and the robot continues normally.

9:07 Door to room 711 is opened. Sunlight swamps the video camera, and the robot stops and thrashes (trying to get past the sunlight). I disable the base.

9:09 Occupant of 711 leaves and closes the door. Robot restarted.

9:15 Base halts twice in a row. Possible battery problems. I reconfigure the playroom to be more corridor-like so the robot will not have to stop as much.

9:17 The robot runs the playroom without stopping.

9:20 Base halts. Restarted.

9:23 Base halts. Run terminated.

Appendix C

Polly's source code

This chapter contains the source code to the parts of Polly that are likely to matter to AI researchers. For brevity, it does not contain "system" code such as device drivers, utility functions, macro definitions, the FEP code, the assembler, or the Senselisp compiler. A few changes have been made: portions that were commented out have been deleted entirely. Some compiler directives have been removed. Some comments have been added.

The code is written in Senselisp, a statically-typed subset of Scheme that supports pointer arithmetic. Senselisp does not support garbage collection, so it must enforce stack discipline on activation records. This means that closures are effectively unsupported. Senselisp also includes a number of useful forms from Common lisp, such as `when` and `unless`. A summary of the linguistic peculiarities of Senselisp is given in figure C.1. Run-system features that are peculiar to Polly are given in figure C.2.

Caveat: Complicated `or` expressions exercise an obscure bug in Senselisp's register allocator. Since I had to graduate, I didn't fix the register allocator; I kluged around the problem. The reader will find occasional places where constructs like `(when (or A B) C)` are unpacked into `(when A C) (when B C)`. I apologize if this makes the code somewhat less readable.

C.1 The main loop

The top-level of the system is a series of initializations followed by the main loop. The main loop grabs a new image, processes it, computes new motor actions, and outputs them on the serial port to the FEP, which forwards them to the base.

Implementation notes

Serial I/O is a little odd on Polly. The C30's on-chip peripherals were sufficiently finicky that I decided to use a two-phase I/O system. First the CPU fills the output buffer, then the DMA controller transfers it to the serial port. No serial I/O calls

Compiler declarations	
`with-vars-in-registers`	informs the compiler to place everything possible in registers.
`with-hardware-looping`	enables compiler generation of special C30 zero-overhead looping instructions.
`in-register`	forces register allocation of argument.
Iteration constructs	
`countdown`	Macro; just like `dotimes` but counts backwards. It's faster.
Pointer arithmetic	
`shift`	performs pointer arithmetic on vectors
`%read-and-shift!`	equivalent of the `*p++` construct in C.
`%write-and-shift!`	same
`@++`	abbreviation for `%read-and-shift!`
`shiftf`	equivalent to `(set! p (shift p offset))`
Mapping	
`map-vector!`	Macro; like `mapcar`, but takes vectors as input and destructively modifies its output.
`do-vectors`	same, but no output argument. Procedure is called for effect only.
`map-region!`	like map-vector but only processes a sub-vector specified by start position and length parameters.
`do-regions`	same, but no output.
Other	
`ash`	arithmetic shift
`forge`	The equivalent of C type-casting; makes the compiler think its argument has a specified type.
`external`	forces a reference to a given assembly-language label. Useful for interfacing with the run-time system.
`when`	Macro; like common lisp - an if with a body consequent and null alternative.
`unless`	Macro; like common lisp - opposite of `when`.
`make-pair`	compresses two 16 bit integers into a single 32 bit integer.
`pair-a`, `pair-b`	extract components from pairs.

Figure C.1: Senselisp peculiarities

Character I/O	
`write-line`	writes a string to the serial port along with CRLF.
`write-line-formatted`	writes a string, then a hexadecimal number, then another string and a CRLF.
`display-line`	writes a string to a specified location on the robot's LCD display.
`display-formatted-line`	like write-formatted-line, but to display.
`display-packed-line`	Like display-line, but uses special packed strings.
`pdisplay`	Macro; packs its argument string and generates call to `display-packed-line`.
Other	
`switch-on?`	returns true if specified front-panel switch is on.
`true-time`	Macro; returns the number of consecutive clock ticks for which the argument predicate has returned true.
`define-box`	Macro; defines a named procedure and allocates space for the wires named as its outputs.

Figure C.2: Peculiarities of the Polly runtime system.

can be performed while the DMA controller is running. This greatly simplifies the I/O library, but it makes the code a little weird. The `wait-output` call is used to synchronize the CPU with the DMA controller to insure that it's safe for the CPU to write the output buffer again. `Start-output` re-initiates DMA.

Serial input is done by polling. I have since found that C30 interrupts are relatively easy to write and debug, but after my experiences with the serial port, I was reluctant to spend time debugging interrupt-driven serial I/O. The `fep-interface` call is used to poll the serial port. Note that there are two calls to it in the main loop to make sure that no bytes get lost.

The `state-loop` macro expands to an infinite loop plus some extra housekeeping code used by the `true-time` macro (see figure C.1).

C.1.1 tour-demo.lisp

```
(define (main)
  ;; Setup.
  (initialize-hardware)
  (initialize-library)
  (set! last-x 90)
  (set! last-y 40)
  (set! current-frame frames)
  (wait-output)
  (do! leave-office)
```

```
(start-output)

;; The real loop.
(state-loop
  (low-level-vision)
  (fep-interface)
  (derived-aspects)
  ;; This test should have gotten moved into the frame-matcher code.
  (when (and corridor?
             (looking-down?)
             (> speed #x1800))
    (frame-matcher))
  (unwedger)
  (wait-output)
  (fep-interface)
  (kluges)
  (odometry)
  (run-sequencers)
  (navigator)
  (interact)
  (update-display)
  (chatter)
  (messages)

  ;;; This is needed to prevent some kind of weird race condition
  ;;; I never determined.  Without it the FEP will sometimes crash
  ;;; when you bring it out of halt mode.
  (dotimes (n 10000))

  (motor-control)
  (start-output)))
```

C.2 The core vision system

The CVS is implemented in the files vision.lisp, which contains the vision routines themselves, and library.lisp, which contains the calls to the routines and the code to set the various global variables. The name "library" is due to historical reasons.

The routine `low-level-vision` computes most of the basic percepts such as the depth map. `Derived-aspects` then computes additional percepts from the percepts computed by `low-level-vision`.

C.2.1 vision.lisp

```
;;; This does a separable 3x3 low pass filter.
(define (smooth in out scratch)
  (with-hardware-looping
    (map-region! scratch
                 0 (- (vector-length scratch) 2)
                 (lambda (left middle right)
                   (+ left right (ash middle 1)))
                 (shift in -1)
                 in
                 (shift in -1))
    (map-region! out
                 64 (- (vector-length out) 128)
                 (lambda (up middle down)
                   (ash (+ up down (ash middle 1)) -4))
                 (shift scratch -64)
                 scratch
                 (shift scratch 64))))

;;; Given an edge image, compute the column heights and write them in VECTOR.
;;; The IMAGE argument is no longer used.
(define (find-distances edges image vector)
  ;;; Mark the top of the image so we can simplify the loop termination test.
  (with-hardware-looping
    ;; Mark the top of the edge image to insure that we have an edge in
    ;; every column.
    (map-region! edges 0 *image-width* (lambda () 255))

    (let ((sedges (shift edges (* *image-width* 45))))
      (countdown (column 63)
        ;; Find the height for column COLUMN.
        (let ((pointer (in-register address (shift sedges column)))
              (distance (in-register data 0)))
          ;; Scan up from the bottom until we find a non-zero pixel.
          (while (= (%read-and-shift! pointer -64) 0)
            (set! distance (+ distance 1)))
          ;; Write out the distance.
          (vector-set! vector column distance))))))

(define (find-vanishing-point image)
  (with-vars-in-registers
    (let ((image (shift image (- (* 46 *image-width*)
                                 2)))
          (sum 0)
```

```
    (sum-squares 0)
    (horizon 24)
    (points 0)
    (reciprocals (forge (vector integer) (external reciprocal-table))))
(let ((quotient (lambda (x y)
                  (ash (* x (vector-ref reciprocals y)) -16))))
  (countdown (y 45)
    ;;; Scan a line.
    (with-hardware-looping
      (countdown (x (- *image-width* 1))
        ;; Try a pixel; compute its gradients.
        (let* ((dx (in-register index (- (vector-ref image 1)
                                         (vector-ref image 0))))
               (dy (- (vector-ref image *image-width*)
                      (vector-ref image 0))))
          ;; Test gradients and reject if one is too small.
          (when (and (> (abs dx) 10)
                     (> (abs dy) 10))
            ;; We have a reasonable edge point.
            ;; Compute its x-intercept with the top of the screen
            ;; (which is assumed to be where the vanishing point is).
            (let ((x-intercept (+ (quotient (* y dy)
                                            dx)
                                  x)))
              ;; Make sure it's in view.
              (when (and (> x-intercept -1)
                         (< x-intercept 64))
                ;; It's in view; average it in.
                (set! sum (+ sum x-intercept))
                (set! sum-squares (+ sum-squares
                                     (* x-intercept x-intercept)))
                (set! points (+ points 1))))))
        ;; Next pixel.
        (set! image (shift image -1))))
    ;; End of the line.
    ;; Skip over the ignored pixel at the end of the line.
    ;; We skip it because we can't compute it's gradient.
    (set! image (shift image -1)))

  ;; We've done all the pixels.
  ;; Check that we got a sane number of edge pixels.
  (if (and (> points 20)
           (< points 256))
    ;; We did; return mean and variance.
```

```
          (let ((mean (quotient sum points)))
            (let ((variance (- (quotient sum-squares points)
                               (* mean mean))))
              (make-pair mean variance)))
          ;; We didn't; return center-of-screen and infinite variance.
          (make-pair 31 1000))))))

(define suppress-horizontal false)

;;; Find the naughty pixels we want to avoid.
;;; Also return the number of edge pixels.
(define (find-dangerous image out)
  (let ((dark-threshold 40)
        (bright-threshold 270)                ; no brightness threshold.
        (edge-threshold 15)
        (artifact-edges 50))
    (with-vars-in-registers
      (with-hardware-looping
        (let ((total-edges 0))
          (if suppress-horizontal
            (map-vector! out
                         (lambda (up left center)
                           (if (and (< (abs (- left center))
                                       edge-threshold)
                                    (> center dark-threshold)
                                    (< center bright-threshold))
                             0
                             (begin (incf total-edges)
                                    255)))
                         (shift image (- *image-width*))
                         (shift image -1)
                         image)
            (map-vector! out
                         (lambda (up left center)
                           (if (and (< (+ (abs (- left center))
                                          (abs (- up center)))
                                       edge-threshold)
                                    (> center dark-threshold)
                                    (< center bright-threshold))
                             0
                             (begin (incf total-edges)
                                    255)))
                         (shift image (- *image-width*))
                         (shift image -1)
```

```
                        image))
            (- total-edges artifact-edges))))))

;;; Compute a map of vertical symmetricalness about each pixel.
;;; IN is a grey-scale image.
;;; OUT is an image to write the per-pixel symmetry values into.
;;; RESULTS is a 64 element vector.  On exit, the ith element of results holds
;;;   the sum of the symmetry values for all pixels in column i.
(define (find-symmetry in out results)
  (let ((scan-width 8)                          ;number of pixels on either side to
                                                ;compare when computing symmetry
        (image-width 64)
        (lines-to-skip 15))                     ;don't bother with the top 15 lines.
    (with-vars-in-registers
      (let ((in (shift in      (* image-width lines-to-skip)))
            (out (shift out (+ (* image-width lines-to-skip)
                               scan-width)))
            (res results))
        (countdown (lines-to-go (- 48 lines-to-skip))
          ;; Do a line.
          (countdown (pixels-to-go (- image-width (* 2 scan-width)))
            ;; Compute the symmetry (SCORE) of a particular pixel.
            (let ((score 0))
              (let ((left1 (in-register address in))
                    (left2 (in-register address (shift in 1)))
                    (right1 (in-register address
                                         (shift in (* 2 scan-width))))
                    (right2 (in-register address
                                         (shift in (+ 1 (* 2 scan-width))))))
                ;; Compare derivatives of each pair of opposing pixels.
                (with-hardware-looping
                  (countdown (n scan-width)
                    (let ((left-deriv (- (@++ left1 1)
                                         (@++ left2 1)))
                          (right-deriv (- (@++ right1 -1)
                                          (@++ right2 -1))))
                      ;; Compare derivatives of a specific pair.
                      (incf score
                            (min 0 (* n left-deriv right-deriv)))))))
              ;; We've got a symmetry value.
              ;; Now impose min and max limits.
              (let ((true-score (min 255
                                     (ash (max (- score) 0)
                                          -2))))
```

```
                ;; And write it out.
                (vset! out 0 true-score)
                (vset! res 0 (+ (vref res 0)
                                (ash true-score -4)))))
            ;; Done with the pixel; shuffle the pointers.
            (shiftf in 1)
            (shiftf out 1)
            (shiftf res 1))

          ;; Done with the line.
          (set! res (shift results scan-width))
          (shiftf in (* 2 scan-width))
          (shiftf out (* 2 scan-width)))))))

;;; Return true if there is a bump in the distance vector near the specified
;;; offset.
(define (protrusion-near? distance-vec offset)
  (with-vars-in-registers
    (let ((edge-thresh 4)
          (first-neg-edge -1)
          (last-plus-edge -1)
          (d (shift distance-vec (- offset 6))))
      ;; Start 5 columns to the left of offset, and scan until 5 to the right.
      ;; Look for a negative-going distance edge followed by a positive-going
      ;; distance edge.
      (dotimes (n 11)
        (let* ((left (vref d n))
               (right (vref d (+ n 1)))
               (diff (- left right))
               (edge? (> (abs diff) edge-thresh))
               (neg-edge? (< diff 0)))
          (when edge?
            (if neg-edge?
              (when (< first-neg-edge 0)
                (set! first-neg-edge n))
              (set! last-plus-edge n)))))
      (and (> first-neg-edge -1)
           (> last-plus-edge -1)
           (> last-plus-edge first-neg-edge)))))

;;; Kluge to find the carpet boundary.
(define (carpet-boundary? image)
  (with-vars-in-registers
    (let ((edges 0)
```

```
        (edge-thresh 9)
        (bad-thresh 25)
        (region-width 10)
        (region-height 15)
        (top-line -1)
        (bad-edges 0)
        (bottom-line 0))
    (let ((im (shift image 1947)))
      (countdown (lines region-height)
          (countdown (pixel 10)
            (let ((center (in-register data (vector-ref im 0))))
              (let ((delta (abs (- (vector-ref im 64) center))))
                (when (> center 120)
                  (incf bad-edges))
                (when (> delta bad-thresh)
                  (incf bad-edges))
                (when (> delta edge-thresh)
                  (incf edges)
                  (when (< top-line 0)
                    (set! top-line lines))
                  (when (> lines bottom-line)
                    (set! bottom-line lines)))
                (shiftf im 1 ))))
        ;; Move to next line.
        (shiftf im 54)))
    (and (> edges 7)
         (< edges 30)
         (= bad-edges 0)
         (< (- bottom-line top-line) 7)))))

;;; Compute the sum of absolute differences between OLD and NEW.  This
;;; routine is misnamed.  It once tred to compute horizontal flow
;;; (hence the name), but difference images turned out to be more
;;; reliable on the whole.
(define fmotion 0)
(define (total-hmotion old new)
    (let ((sum 0))
      (do-vectors (lambda (old new right)
                    (let ((hmotion (abs (- new old))))
                      (when (> hmotion 20)
                        (incf sum hmotion))))
                  old
                  new
                  (shift new 1))
```

```
      (set! fmotion
            (+ (ash sum -6)
               (ash fmotion -1)))
      fmotion))

;;; Return the smallest element of a region of a distance vector.
(define (region-min vector start length)
  (let ((m (in-register data 9999999)))
    (do-regions start length
                (lambda (x)
                  (when (< x m)
                    (set! m x)))
                vector)
    m))

;;; Return the index of the largest element of vector in specified region.
(define (region-max-point vector start length)
  (with-vars-in-registers
      (let ((maximum 0)
            (max-point 0)
            (l length)
            (v (shift vector start)))
        (countdown (n l)
          (let ((x (@++ v)))
            (when (> x maximum)
              (set! maximum x)
              (set! max-point n))))
        (- l max-point))))

;;; Return the number of elements in vector whose values are near VALUE.
(define (region-value-count vector start length value)
  (with-vars-in-registers
    (let ((value value)                 ; get it in a register
          (count 0))
      (do-regions start length
                  (lambda (x)
                    (when (< (abs (- x value))
                             3)
                      (incf count)))
                  vector)
      count)))

;;; Smooth a distance map.
(define (smooth-1d in out)
  (map-vector! out
```

```
            (lambda ( l c r)
              (ash (+ l r (ash c 1)) -2))
            (shift in -1)
            in
            (shift in 1)))
```

C.2.2 library.lisp

```
;; The subsampled image from the camera (64x48).
(define image null-vector)
;; IMAGE averaged down to 16x12.
(define mini null-vector)
;; Low-pass filtered version of IMAGE.
(define smoothed null-vector)
(define old-smoothed null-vector)
(define motion null-vector)
(define reversals null-vector)
(define old-motion null-vector)
;; The edge map.
(define edges null-vector)
;; Radial depth map.
(define distance-vector null-vector)
;; Values of symmetry for each pixel.
(define symmetry-image null-vector)
;; Values of symmetry calculation for each column of the image.
(define symmetry-vector null-vector)

(define (initialize-library)
  (set! distance-vector (make-vector *image-width*))
  (set! image (make-image))
  (set! old-smoothed (make-image))
  (set! mini (make-mini-image))
  (set! smoothed (make-image))
  (vector-fill smoothed 0)
  (vector-fill old-smoothed 0)
  (set! edges (make-image))
  (set! symmetry-image (make-image))
  (set! symmetry-vector (make-vector *image-width*)))

;;;;; Visual System.

;;;; Tuning parameters.

(define-constant dark-floor-level 100)
(define-constant light-floor-level 100)
```

```
;; The number of noise edges introduced by frame grabber and dumb edge
;; algorithm.
(define-constant artifact-edges 150)

;;; We consider ourself "blind" if we see fewer than this number of edge
;;; pixels.
(define-constant blindness-threshold 50)

(define-box (low-level-vision
             :outputs (left-distance right-distance ;left-space
                                     ;right-space
                                     center-distance
                                     ;center-space
                                     wall-ahead?
                                     wall-far-ahead? boundary?
                                     ;farthest-direction
                                     dark-floor light-floor blind?
                                     edge-count vanishing-point variance
                                     reversals
                                     hmotion
                                     person? person-direction))
  ;; Get the image an preprocess it.
  (grab-and-start image)
  (shrink-image image mini)

  ;; Swap the the smoothed and old-smoothed buffers
  (let ((temp (in-register data old-smoothed)))
    (set! old-smoothed smoothed)
    (set! smoothed temp))
  (smooth image smoothed edges)

  ;;; Find the edges for the distance finder.
  (set! boundary? (and (= (logand direction 1)
                          1)
                       (carpet-boundary? smoothed)))
  (set! suppress-horizontal boundary?)
  (set! edge-count (find-dangerous smoothed edges))
  (set! blind? (< (true-time (> edge-count blindness-threshold)) 3))

  ;; Compute distances.
  (find-distances edges image distance-vector)
  (set! left-distance (region-min distance-vector 6 25))
  (set! right-distance (region-min distance-vector 31 25))
  (set! center-distance (region-min distance-vector 24 14))
```

```
  (set! wall-far-ahead? (> (region-value-count distance-vector 27 10 28)
                           8))
  (set! wall-ahead? (> (region-value-count distance-vector 24 16 25)
                       14))

  ;; Look for symmetry (i.e. people).
  (vector-fill-internal symmetry-vector 0 64)
  (find-symmetry smoothed symmetry-image symmetry-vector)
  (set! person-direction (+ 21 (region-max-point symmetry-vector 21 22)))
  (set! person? (and (> (vector-ref symmetry-vector person-direction)
                        100)
                     (protrusion-near? distance-vector person-direction)))

  ;; Find the vanishing point.  VP computation returns a pair.
  (let ((pair (in-register data
                           (find-vanishing-point smoothed))))
    (set! vanishing-point (pair-a pair))
    (set! variance (pair-b pair)))

  (let ((floor (vector-ref mini (- (* 16 12) 8))))
    (if dark-floor
      (when (> floor 120)
        (set! dark-floor false))
      (when (< floor 80)
        (set! dark-floor true)))
    (set! light-floor (> (true-time (> floor light-floor-level))
                         5)))
  (let ((temp (in-register data old-motion)))
    (set! old-motion motion)
    (set! motion temp))
  (set! hmotion (total-hmotion old-smoothed smoothed)))
(define-macro (2bit value pos)
  `(logand (forge integer ,value) ,(ash 1 pos)))

(define-box (derived-aspects
             :outputs (blocked? open-left? open-right? turning? aligned?
                       left-turn? right-turn?
                       aligned-long-time? in-corridor?
                       feature-bits previous-bits wall? open-region?
                       person-ahead?))
  (set! previous-bits feature-bits)
  (set! turning? (> (abs turn-rate) 2000))
  (set! aligned? (not turning?))
  (set! aligned-long-time? (> (true-time aligned?)
```

```
                              8))
(set! blocked? (< center-distance 15))
(set! in-corridor? (> (true-time (> center-distance 35))
                      5))
(set! open-left? (> left-distance 18))
(set! open-right? (> right-distance 18))

;;; Sorry.  This OR expression screwed up the register allocation on my
;;; compiler, so I kluged it by turning it into a logior.
(set! wall? (forge boolean (logior (forge integer wall-ahead?)
                                   (forge integer
                                          (> (true-time blocked?)
                                             10))
                                   (forge integer wall-far-ahead?))))

;;; Take a bunch of readings (wall?, open-left/right?, and dark-floor) and
;;; package them up as a 4-bit value we can match against the 4-bit feature
;;; values in the place frames.
(set! feature-bits (logior (2bit wall? 0)
                           (2bit open-left? 1)
                           (2bit open-right? 2)
                           (2bit dark-floor 3)))

(set! open-region? (= (true-time (and open-left? open-right?))
                      10))
(set! person-ahead? (and (not open-left?)
                         (not open-right?)
                         person?
                         corridor?
                         aligned?
                         (< (abs (- person-direction 32)) 10))) )
```

C.3 Low level navigation

The top-level entry point for the motor control system is `motor-control`. It calls `speed-control` and `turn-controller`, which call other routines in turn. `Turn-controller`, `ballistic-turn-controller`, and `speed-control` are the only routines that actually generate motor outputs. The other routines just pass numbers around representing speeds.

The `odometry` procedure talks to the FEP and processes any odometry information that has come in. It updates the `direction` wire, updates the direction on the LCD display, and corrects drift in the odometer's offset. The idea is that if the robot has been going in a straight line for a long time, it must be perfectly aligned with a corridor. That means we know the true value that the odometer ought to have and so we can compute the offset from the difference between the ideal and actual readings.

Implementation notes

The last part of this file is intended only for those who are intimately familiar with the RWI base control language and who want to know the dirty details of running a large payload at 1m/s.

C.3.1 motor-control.lisp

```
;;;;; MOTOR CONTROL

;;;; Tunable parameters
(define-constant maximum-speed #x2800)
(define-constant maximum-turn #x8000)
(define-constant (degrees->encoder degrees)
  (* degrees 328))
(define-constant (degrees->delay degrees)
  (ash (abs degrees) -2))


;;;; Latching inputs: (ballistic) turn and speed.

;;; The speed that the higher-levels want to move at.
;;; Thus continuously adjusts speed.
(define speed-request maximum-speed)

;;; For inhiniting forward motion when turning.
(define inhibit-forward? false)

;;; To stop the robot when e.g. looking for motion.
(define inhibit-all-motion? false)

;;; The turn which has been requested by the higher-levels.
;;; The moment this is set, the motor control unit will initiate a
;;; ballistic turn of the specified number of degrees. It will then
;;; ignore the input until the turn is complete. It will reset the input
;;; at the completion of the turn.
;;; (if no turn is specified, the corridor follower runs)
(define turn-request 0)

;;;; State variables.

;;; Number of ticks to wait until turn is finished.
(define turn-delay 0)
```

```
(define-box (motor-control :outputs (speed turn-rate))
  (set! speed (speed-control))
  (set! turn-rate (turn-controller)))

(define (turn-controller)
  (let ((sturn (steer))
        (bturn (ballistic-turn-controller)))
    (set! turn-request 0)
    (if (and (= bturn 0)
             (= turn-delay 0))
      (begin (set-turn-rate! sturn)
             sturn)
      bturn)))

(define (speed-control)
  ;;; Set the speed.
  (let ((speed (if blind?
                   -1500
                   (min speed-request
                        (max -3000
                             (* 1300
                                (- center-distance 7)))))))
    (when (and (> speed 0)
               inhibit-forward?)
      (set! speed 0))
    (when inhibit-all-motion?
      (set! speed 0))
    (set-speed! speed)
    speed))

(define (steer)
  (let ((turn 0))
    (when (and (not (= speed-request 0))
               (not inhibit-all-motion?)
               (= next-camera 0)
               (= current-camera 0))
      (set! turn
            (if (eq? open-left? open-right?)
              (follow-corridor)
              (follow-wall)))
      (when blocked?
        (set! turn
              (if (> (abs turn) 800)
                (if (> turn 0)
```

```
                 #x4000
                 #x-4000)
               0))))
    turn))

(define (ballistic-turn-controller)
  (when (> turn-delay 0)
    (decf turn-delay))
  (when (= turn-delay 0)
    (set! inhibit-forward? false))
  (if (and (not inhibit-all-motion?)
           (= current-camera 0)
           (= next-camera 0)
           (= turn-delay 0)
           (not (= turn-request 0)))
    (do-turn! turn-request)
    0))

;;; Start a ballistic turn.
(define (do-turn! turn)
  (start-open-loop-turn! turn)
  (set! turn-request 0)
  (set! turn-delay (degrees->delay turn))
  (set! inhibit-forward? true)
  (if (> turn 0) #x4000 #x-4000))

;;; Align with corridor or free space.
(define (follow-corridor)
  (let ((left-badness (max -1
                           (- 8 left-distance)))
        (right-badness (max -1
                            (- 8 right-distance)))
        (see-corridor? (< variance 64)))
    (+ (* (- left-badness right-badness)
          600)
       (if (and see-corridor?
                (not blocked?))
         (* (- vanishing-point 31)
            200)
         0))))

(define (follow-wall)
  (* 800
     (if open-left?
       (- right-distance 8)
```

```
      (- 8 left-distance))))

(define wedge-counter 30)
(define-constant wedge-time 30)
(define-box (unwedger :outputs (last-turn))
  (if (and blocked?
           (not inhibit-all-motion?)
           (not inhibit-forward?)
           (looking-down?)
           (not (= speed-request 0)))
    (decf wedge-counter)
    (set! wedge-counter wedge-time))
  (when (and open-left?
             (not open-right?))
    (set! last-turn -45))
  (when (and open-right?
             (not open-left?))
    (set! last-turn 45))
  (when (= wedge-counter 0)
    (set! wedge-counter wedge-time)
    (set! turn-request last-turn)))



;;;; Odometry.

(import odometer aligned?)
(define old-ode 0)
(define corridor-counter 0)
(define prev-direction north)
(define corridor-time (secs->ticks 2.0))

;; Direction of perfect south.
(define odometer-offset 0)
(define-box (odometry :outputs (direction last-direction corridor?
                                ew-corridor? ns-corridor?))
  ;; Ask FEP for a switch word, and the base for an odometry reading.
  (write-line "^RW")
  ;; Update direction.
  (set! last-direction direction)
  (set! direction (logand #b11
                          (+ south
                             (quotient (+ 57
                                          (ash (- odometer odometer-offset)
```

```
                                                    -8))
                                            115))))
  (when (> (true-time (and (> speed #x2000)
                           (< (abs (- old-ode odometer)) 300)))
           60)
    (when (= direction east)
      (set! odometer-offset (+ odometer 29550)))
    (when (= direction west)
      (set! odometer-offset (- odometer 29550))))
  (set! old-ode odometer)
  (when (= (true-time (= direction south)) 1)
    (pdisplay-line 4 0 "South"))
  (when (= (true-time (= direction west)) 1)
    (pdisplay-line 4 0 "West"))
  (when (= (true-time (= direction north)) 1)
    (pdisplay-line 4 0 "North"))
  (when (= (true-time (= direction east)) 1)
    (pdisplay-line 4 0 "East"))

  ;; Now figure out if we're in a corridor district.
  (when (not (= direction prev-direction))
    (set! corridor-counter corridor-time))
  (when (> turn-delay 0)
    (set! corridor-counter corridor-time))
  (when (and (not blocked?)
             (> speed 0))
    (decf corridor-counter))
  (set! prev-direction direction)
  (set! corridor? (< corridor-counter 0))

  (set! ew-corridor? false)
  (set! ns-corridor? false)
  (when corridor?
    (set! ew-corridor? (= (logand direction 1) 1))
    (set! ns-corridor? (= (logand direction 1) 0)))

  )

;;;;; This stuff is specific to the RWI base.

;;; This is used to manually damp the base's control system.
(define old-speed 0)

(define (set-speed! speed)
  ;; Don't accelerate too fast.
```

```
  (when (> speed old-speed)
    (set! speed (min speed (+ old-speed 500))))
  (set! old-speed speed)

  (when (= (logand *clock* 31) 0)
    (write-line "TA 3000"))
  (write-line-formatted "TV " 16 (min maximum-speed (abs speed)) "")
  (write-line (if (< speed 0)
                  "T-"
                  (if (= speed 0) "TH" "T+"))))

;;; Fast-rotate? causes set-turn-rate to reset the rotate acceleration.
(define fast-rotate? true)

(define (set-turn-rate! rate)
  (when fast-rotate?
    (set! fast-rotate? false)
    (write-line "RA 6000"))
  (set! turn-rate (ash turn-rate -1))
  (write-line-formatted "RV " 16 (min maximum-turn (abs rate)) "")
  (write-line (if (< rate 0)
                  "R-"
                  (if (= rate 0) "RH" "R+"))))

(define (start-open-loop-turn! turn)
  (write-line-formatted (if (> turn 0)
                             "R> " "R< ")
                        16
                        (degrees->encoder (abs turn-request))
                        "")
  (write-line "RA 8000")
  (write-line "RV 8000")
  (set! fast-rotate? true))


(define-box (fep-interface :outputs (odometer switches))
  (while (listen-port?)
    (let ((info 0))
      (set! info (read-port))
      (if (= (logand info 1) 1)
        ;; low-order bit set: it's a switch word.
        (set! switches info)
        ;; Odometer word.
        (set! odometer (- info #x80000000)))))
  (let ((old-value (vector-ref serial-port global-control)))
```

```
    (vset! serial-port global-control
           (logand old-value #b1111111111111111111111111))
    (assemble (nop)
              (nop)
              (nop)
              (nop))
    (vset! serial-port global-control old-value))
  )
```

C.4 High level navigation

C.4.1 place-recognition.lisp

```
;;;;; PLACE MEMORY (frame system).

(define-constant (square x) (let ((y (in-register data x))) (* y y)))

;;; Frame debouncing reduces the effects of random bits of furniture placed
;;; in large open spaces.  The furniture would otherwise make the world look
;;; like it had extra turns.
(define frame-debounce 0)

(define last-match-bits 0)

;;; Compare the current sensory data to all frames ...
(define-box (frame-matcher :outputs (last-x last-y current-frame
                                            frame-strobe?))
  ;; ... unless we're blocked by an obstacle ...
  (when blocked?
    (set! frame-debounce 30))
  (let ((frame frames)                      ;frames points to a vector of frames.
        (best frames)                       ;pointer to best frame so far
        (best-value 99999999))              ;best score so far
    (decf frame-debounce)
    (set! frame-strobe? false)
    ;; ... or FRAME-DEBOUNCE says we recently matched a frame.
    (when (< frame-debounce 0)
      ;; First, try to fix the X and Y axes independently by looking for
      ;; districts.
      (find-districts)

      ;; Now compare each frame.
      (countdown (n frame-count)
        (let ((matchval (match-frame mini frame)))
```

```
        (when (< matchval best-value)
          (unless (and (= (frame-x frame) last-x)
                       (= (frame-y frame) last-y))
            (setf best frame)
            (setf best-value matchval))))
      (setf frame (shift frame (if (image-frame? frame)
                                   image-frame-length
                                   feature-frame-length))))

    ;; If we found a match, update outputs.
    (when (< best-value 70000)
      (setf frame-debounce (frame-place-size best))
      (setf frame-strobe? true)
      (setf current-frame best)
      (setf last-x (frame-x best))
      (setf last-y (frame-y best))))
  (set! last-match-bits feature-bits)))

;; Match a specific frame to the current sensory data.
(define (match-frame mini-image frame)
  (with-vars-in-registers
    (let ((fimage (frame-image frame))
          (image (in-register address mini-image))
          (sum 0)
          (bits-changed? (not (= (logand feature-bits #b11110)
                                 (logand last-match-bits #b11110)))))
      (when wall?
        (set! bits-changed? true))
      ;; Compute penalties based on estimated position and direction of
      ;; motion.
      (let ((delta-x (- (frame-x frame) last-x))
            (delta-y (- (frame-y frame) last-y)))
        (incf sum (* (abs delta-x)
                     (vector-ref direction
                                 (if (> delta-x 0)
                                   #(3000 100 3000 3000)
                                   #(3000 3000 3000 100)))))
        (incf sum (* (abs delta-y)
                     (vector-ref direction
                                 (if (> delta-y 0)
                                   #(100 3000 3000 3000)
                                   #(3000 3000 100 3000))))))
      ;; Large penalty for getting the direction wrong.
      (unless (= (frame-direction frame)
```

```
            direction)
      (incf sum 100000))

    ;; IMAGE FRAME COMPARISON.
    (if (image-frame? frame)
      ;; This loop compares the image in an image frame to MINI-IMAGE.
      ;; The C30 doesn't support byte addressing, so the image templates
      ;; in the frames are stored in a packed format (4 8-bit pixels per
      ;; 32-bit word).  That's why there's so much shifting and masking
      ;; here.
      (with-hardware-looping
        ;; There are 16x12=192 pixels per template, so 48 words per template.
        (countdown (n 48)
          (let ((bytes (in-register data (@++ fimage))))
            ;; Compare a word's worth of pixels.
            (incf sum
                  (square (abs (- (logand bytes #xff) (@++ image)))))
            (incf sum
                  (square (abs (- (logand (ash bytes -8)
                                          #xff)
                                  (@++ image)))))
            (incf sum
                  (square (abs (- (logand (ash bytes -16)
                                          #xff)
                                  (@++ image)))))
            (incf sum
                  (square (abs (- (logand (ash bytes -24)
                                          #xff)
                                  (@++ image)))))))))

      ;; FEATURE FRAME COMPARISON.
      (incf sum (if (and (= (frame-features frame)
                            feature-bits)
                         bits-changed?)
                    60000
                    10000000)))
    sum)))

;;; Check if we can prove we're in a particular part of the building even
;;; without knowing our precise location.
(define (find-districts)
  ;; Use a stricter criterion for turns so that we don't get doorways.
  (let ((left-turn? (> left-distance 25))
        (right-turn? (> right-distance 25)))
```

```
;; Decide if we've reached the end of a N/S corridor.
(when (and aligned?
           ns-corridor?
           wall?)
  (when (= direction north)
    (set! last-y 40))
  (when (= direction south)
    (set! last-y 10)))

;; If we're in an E/W corridor and see a turn, we know which side of the
;; building we're on.
(when (and ew-corridor?
           (not dark-floor)
           aligned-long-time?
           (not blocked?))
  (when (and (= direction west)
             open-right?
             (not open-left?)
             in-corridor?)
    (set! last-y 10))
  (when (and (= direction east)
             (not open-right?)
             open-left?)
    (set! last-y 10))
  (when (and (= direction west)
             (not open-right?)
             open-left?)
    (set! last-y 40))
  (when (and (= direction east)
             (not open-left?)
             (< last-x 70)
             open-right?)
    (set! last-y 40)))))
```

C.4.2 kluges.lisp

```
(define (kluges)
  ;; This is because there isn't anything even remotely like a corridor in
  ;; the playroom.  This should have gotten worked into the district
  ;; recognition code, but never did.
  (when (and (> last-x 7)
             (= direction north)
             wall?)
    (set! last-y 40)))
```

C.4.3 navigator.lisp

```
;;; Simple algorithm for navigating to a specified place.
(define goal-x 0)
(define goal-y 0)

(define (display-packed-goal string)
  (pdisplay-line 5 0 "Goal:")
  (display-packed 5 6 string))

(define-box (navigator)
  (do-veering)
  (steer-to-goal))

;;; Place frames are tagged with small turns ("veers") that are executed
;;; VEER-DELAY ticks after recognizing the frame.  This takes care of the
;;; embedded turn in the northern E/W corridor (the one near the elevator
;;; lobby).
(define-constant veer-delay (secs->ticks 0.75))
(define (do-veering)
  ;; Check if FRAME-STROBE? rose, then fell, VEER-DELAY ticks ago.
  (when (= (true-time (not frame-strobe?)) veer-delay)
    ;; If so, turn.
    (set! turn-request (frame-veer current-frame))))

;;; This is the real meat of the navigator.  It computes the difference of
;;; current coordinates and goal coordinates for each axis and
;;; opportunistically makes turns to reduce the difference.
(define-box (steer-to-goal)

  ;; UPDATE THE DISPLAY.

  ;; Hardware issue: need to use true-time to debounce the switch.
  (when (= (true-time (switch-on? sw3)) 3)
    (pdisplay 7 0 "Navigation enabled."))
  (when (= (true-time (not (switch-on? sw3))) 3)
    (clear-line 7))

  ;; CHOOSE TURNS.
  (when (and (> (true-time (switch-on? sw3)) 5)
             (> goal-x 0))
    ;; We're not there yet, so start driving.
    (speed! #x5000)

    ;; Compute the difference between our current and desired positions.
    ;; Decide what turns would be useful to make.
```

```
(let* ((delta-x (- goal-x last-x))
       (time-since-strobe (true-time (not frame-strobe?)))
       (desired-x-direction (if (> delta-x 0) east west))
       (delta-y (- goal-y last-y))
       (desired-y-direction (if (> delta-y 0) north south))
       (left-direction (logand (- direction 1) #b11))
       (right-direction (logand (+ direction 1) #b11)))

  ;; CLEAR THE GOAL if we're there already.
  (when (and (= delta-x 0)
             (= delta-y 0))
    (set! goal-x 0)
    (set! goal-y 0)
    (speed! 0))

  ;; TELL THE UNWEDGER WHERE TO TURN.
  ;; When the unwedger activates, it executes the turn in LAST-TURN.
  (when (and (not (= delta-x 0))
             blocked?
             (= left-direction desired-x-direction))
    (set! last-turn -35))
  (when (and (not (= delta-x 0))
             blocked?
             (= right-direction desired-x-direction))
    (set! last-turn 35))
  (when (and (not (= delta-y 0))
             blocked?
             (= left-direction desired-y-direction))
    (set! last-turn -35))
  (when (and (not (= delta-y 0))
             blocked?
             (= right-direction desired-y-direction))
    (set! last-turn 35))

  ;; U-TURNS.  Do a u-turn if we've overshot.
  (when (and (not (= delta-x 0))
             ew-corridor?
             (> time-since-strobe 25)
             aligned-long-time?
             (not (= direction desired-x-direction))
             (not open-left?)
             (not open-right?))
    (set! turn-request 180))
  (when (and (not (= delta-y 0))
```

```
                ns-corridor?
                (> time-since-strobe 25)
                aligned-long-time?
                (not (= direction desired-y-direction))
                (not open-left?)
                (not open-right?))
       (set! turn-request 180))

     ;; Do a turn if at an intersection.
     ;; The turns are 80 degress, rather than 90, so that if we turn a
     ;; little bit early, we'll be pointed toward the far wall, rather
     ;; the near one.
     (when (= time-since-strobe 25)
       (when (and (frame-left-turn? current-frame)
                  (not (= delta-x 0))
                  (= left-direction desired-x-direction))
         (set! turn-request -80))
       (when (and (frame-right-turn? current-frame)
                  (not (= delta-x 0))
                  (= right-direction desired-x-direction))
         (set! turn-request 80))
       (when (and (frame-left-turn? current-frame)
                  (not (= delta-y 0))
                  (= left-direction desired-y-direction))
         (set! turn-request -80))
       (when (and (frame-right-turn? current-frame)
                  (not (= delta-y 0))
                  (= right-direction desired-y-direction))
         (set! turn-request 80))))))
```

C.4.4 wander.lisp

```
(define global-mode 0)
(define-constant wander-mode 1)
(define-constant tour-mode 2)
(define-constant offer-mode 3)

;; Drive in a loop around the lab.  The endpoints are determinate, but
;; the direction of motion (clockwise or counterclockwise) depends on
;; the initial configuration of the robot.
(define (wander)
  (when (> global-mode 0)
    ;; If we've overshot, turn around.
    (when (and (at-place? 10 40)
```

```
              frame-strobe?
              (= direction west))
      (set! turn-request 180))

    ;; If we've gotten to the vision lab, go to the playroom.
    (when (< last-x 40)
      (set-goal 80 10 "Playroom"))

    ;; If we've gotten to the playroom, go to the vision lab.
    (when (> last-x 70)
      (set-goal 30 40 "Vision lab"))))
```

C.5 Giving tours

C.5.1 sequencers.lisp

```
(define (run-sequencers)
  (go-home)
  (wander)
  (offer-tour)
  (give-tour)
  (leave-office)
  (tour-chatter))

;; When enabled, this drives from my desk out into the hallway.
(define-sequencer (leave-office :import (...))
  (first (pdisplay-goal "Leave office"))
  (when blocked?
    (set! global-mode 1)
    (turn! -40))
  (sleep 2)
  (when open-left?
    (pdisplay-goal "Align"))
  (sleep 2)
  (then (turn! -60))
  (sleep 1)
  (when (and (= direction east)
           blocked?)
    (turn! 150))
  (wait ew-corridor?)
  (sleep 5)
  (then
    (set! last-x 80)
```

```
    (set! last-y 40)
    (speed! 0)))

;; This tries to drive from an arbitrary place back to my desk.
(define-sequencer (go-home :import (...))
  (then (stop! give-tour)
        (set! global-mode 0))
  (go-to 90 40 "Home")
  (then (speed! #x5000))
  (when blocked?
    (turn! -120)
    (speed! #x5000))
  (sleep 10)
  (when (and (= direction north)
             blocked?)
    (pdisplay-goal "Done.")
    (set! inhibit-all-motion? true)
    (say "I'm home.")
    (speed! 0)))

;; Offer a tour to a person. If they wave their foot, fire GIVE-TOUR.
(define-sequencer (offer-tour :import (...))
  (first (set! global-mode 3)
         (set! inhibit-all-motion? true))
  (when done-talking?
    (new-say "Hello.  I am Polly.  Would you like a tour?
              If so, wave your foot around."))
  (sleep 9)
  (then (if blocked?
          ;; The person's still there.
          (if (> hmotion 30)
            (do! give-tour)
            (begin
             (new-say "OK.  Have a nice day.")
             (set! global-mode 1)
             (set! inhibit-all-motion? false)))
          ;; The person's gone.
          (begin (set! global-mode 1)
                 (set! inhibit-all-motion? false)))))

;; Coordinates of the place where we started the tour.
;; When we get back there, it's safe to stop.
(define tour-end-x 0)
(define tour-end-y 0)
```

```
;; Actually give the tour.
(define-sequencer (give-tour :import (...))
  (first (new-say "OK.  Please stand to one side.")
         (if (= last-y 40)
           (begin (set! tour-end-x 80)
                  (set! tour-end-y 10))
           (begin (set! tour-end-x last-x)
                  (set! tour-end-y last-y))))
  (when (not blocked?)
    (new-say "Thank you.  Please follow me.")
    (set! inhibit-all-motion? false)
    (set! global-mode 2))
  (wait frame-strobe?)
  (wait (at-place? tour-end-x tour-end-y))
  (sleep 1.0)
  (then
    (new-say "That's the end of the tour.
              Thank you and have a nice day.")
    (set! global-mode 1)))
```

C.5.2 interact.lisp

```
(define-constant hello-interval (secs->ticks 5))

(define hello-counter 30)

(define (interact)
  ;; Announce the current place, if we've just reached a new landmark.
  (when frame-strobe?
    (announce-place))

  ;; Go home if switch 4 is thrown.
  ;; True-time is used to debounce the switch.
  (when (= (true-time (switch-on? sw4)) 5)
    (do! go-home))
  (when (= (true-time (switch-off? sw4)) 5)
    (stop! go-home))

  ;; If the right list of magic conditions holds, offer a tour.
  (when (and person?
             (= turn-delay 0)
             ew-corridor?
             blocked?
             (switch-off? sw1)
             (< (abs (- person-direction 31)) 10)
```

```
              (= global-mode 1)
              (= hello-counter 0))
     ;; Fire the offer-tour sequencer.
     (do! offer-tour))

   ;; This prevents repeated offering of tours.  It was important back when
   ;; Polly was programmed to say "hello" any time it saw a person at a
   ;; distance.  It's probably not needed anymore.
   (when (> hello-counter 0)
     (decf hello-counter))
   (when (active? offer-tour)
     (set! hello-counter hello-interval)))

;; Put the current place on the screen.
(define (announce-place)
  (tour-announce-place)
  (display-packed-line 3 0 (frame-name current-frame)))

;; Send the speech for the current place to the voice synthesizer, provided
;; we're in tour-mode.
(define (tour-announce-place)
  (when (= global-mode tour-mode)
    (set! priority-message
          (frame-speech current-frame))))


;; Update the LCD display with position information, etc.
(define (update-display)
  (clear-line 10)
  (when (> (logand feature-bits #b10) 0)
    (pdisplay 10 0 "Left"))
  (when (> (logand feature-bits #b1) 0)
    (pdisplay 10 5 "Wall"))
  (when (> (logand feature-bits #b1000) 0)
    (pdisplay 10 10 "Dark"))
  (when (> (logand feature-bits #b100) 0)
    (pdisplay 10 15 "Right"))

  (display-digit 2 4 (quotient last-x 10))
  (display-digit 2 6 (quotient last-y 10))
  (display-digit 2 16 (quotient goal-x 10))
  (display-digit 2 18 (quotient goal-y 10)))
```

C.6 Voice

`Chatter` controls voice output. It takes two inputs, a pointer to an unimportant "chatter" message and a priority message. If there is a priority message, it sends it to the voice synthesizer. If not, and if there is a chatter message, it sends that to the voice synthesizer. Otherwise it is silent. If a priority message arrives in the middle of saying a chatter message, the `chatter` routine terminates the chatter message, says the priority message, and restarts the chatter message from the beginning.

Priority messages are generated by `tour-announce-place` when a landmark is recognized in tour mode. Chatter messages are generated by `try-chatter` when in tour mode and by `messages` when in patrol mode. Each has its own list of messages. `Try-chatter` works through them in order, while `messages` generates them randomly (and rarely) by looking at the low order bits of the odometer.

Implementation notes

The voice synthesizer is driven through the FEP. An escape code in the FEP output stream causes subsequent bytes to be diverted from the base to the voice synthesizer. A CR diverts output back to the base. To prevent a large block of speech output from starving the base, `chatter` only sends four bytes per clock tick.

There is no feedback channel from the voice synthesizer, so the robot doesn't know when the speech is done or when the synthesizer's input buffer is full. To deal with this problem, `chatter` waits between saying messages. The length of the wait is proportional to the length of the previous message.

To save memory, messages are stored as packed strings (4 8-bit bytes per 32-bit word).

C.6.1 chatter.lisp

```
(define priority-message null-vector)
(define chatter-message null-vector)
(define-constant chattering 0)
(define-constant priority 1)
(define talk-state chattering)
(define string-position 0)
(define chatter-delay 0)

(define-box (chatter :outputs (done-talking?))
  ;; Check if speaking is inhibited (because of previous utterance).
  (decf chatter-delay)
  (when (< chatter-delay 0)
    ;;; It's OK to talk.

    ;; If we were chattering, but a priority message comes in, then
    ;; break the chatter and start the priority message.
```

```
    (when (and (= talk-state chattering)
               (not (eq? priority-message null-vector)))
      ;; Interrupt the chatter and move to the priority message.
      (set! talk-state priority)
      (set! string-position 0)
      (buffered-write-char (char-code #\!))
      (buffered-write-char (char-code #\.))
      (buffered-write-char (char-code #\return)))

    (let ((string (if (= talk-state chattering)
                      chatter-message
                      priority-message)))
      (when (not (eq? string null-vector))
        ;; Send some more.
        (when (< string-position (vector-length string))
          (say-some string))
        ;; If done with this utterance, then set delay and setup for next
        ;; utterance.
        (when (not (< string-position (vector-length string)))
          (set! chatter-delay (* (vector-length string) 6))
          (set! string-position 0)
          (if (= talk-state chattering)
            (set! chatter-message null-vector)
            (set! priority-message null-vector))
          (set! talk-state chattering)))))
  (set! done-talking? (and (= talk-state chattering)
                           (eq? chatter-message null-vector))))

(define (say-some string)
  (let ((word (vector-ref string string-position)))
    ;; Send the FEP escape code to initiate voice output.
    (buffered-write-char #.(char-code #\!))
    ;; Now send one word's worth (4 bytes) of the current speech.
    (buffered-write-char (logand word #xff))
    (buffered-write-char (logand (ash word -8) #xff))
    (buffered-write-char (logand (ash word -16) #xff))
    (buffered-write-char (logand (ash word -24) #xff))
    ;; Update pointers.
    (when (= string-position (- (vector-length string) 1))
      (buffered-write-char 0))
    (incf string-position)
    ;; Turn off speech output.
    (buffered-write-char #.(char-code #\return))))
```

```
(define message-counter 30)
(define (messages)
  (let ((rand (logand (ash odometer -1) 1023))
        (yow (external (! yow))))
    ;; This is to keep it from talking before the base is actually connected.
    (unless (and (= odometer 0)
                 (= odometer-offset 0))
      (when (and (eq? chatter-message null-vector)
                 (= global-mode 1)
                 (< rand (vector-length yow)))
        (decf message-counter)
        (when (= message-counter 0)
          (set! message-counter 10)
          (set! chatter-message
                (vector-ref yow rand)))))))

(define-constant tour-chatter-delay (secs->ticks 20.0))
(define tc-index 0)
(define tc-delay tour-chatter-delay)
(define (tour-chatter)
  (if (= global-mode 2)
    (try-chatter)
    (set! tc-index 0)))
(define (try-chatter)
  (decf tc-delay)
  (let ((tm (external (! tour-messages))))
    (when (and done-talking?
               (< tc-delay 0)
               (< tc-index (vector-length tm)))
      (set! chatter-message (vector-ref tm tc-index))
      (set! tc-delay tour-chatter-delay)
      (incf tc-index))))
```

C.6.2 pith.lisp

```
;;; This file creates a separate loader segment for Polly's
;;; phrases.  The phrases are stored as packed strings.

(eval-when (:load-toplevel :compile-toplevel :execute)
  ;; Define an assembler symbol NAME whose address is the start
  ;; of the packed string STRING.
  (defmacro define-phrase (name string)
    ;; Code-string converts a string to a packed string
```

```
    ;; (a list of 32-bit integers).
    (let ((code (code-string string)))
      `(defc30 (,name polly)
         ,(first code) ,name ,@(rest code))))

  ;; Define an assembler symbol NAME that points to a vector of
  ;; pointers to packed strings
  (defmacro define-phrases (name &rest phrases)
    (let ((code (list (length phrases) name)))
      (dolist (phrase phrases)
        (let ((sym (gensym))
              (c (code-string phrase)))
          (setf code
                (nconc (list* (first c) sym (rest c))
                       code
                       `((word ,sym))))))
      `(defc30 (,name polly)
         ,@code)))

;; These messages are randomly generated by MESSAGES in chatter.lisp
;; when the robot is in patrol mode.

;; Note that the misspellings here are deliberate - they're needed to fool
;; the voice synthesizer into saying the right phonemes.

(define-phrases yow
  "Yaaw.  Are we having fun yet?"
  "My hovercraft is full of eels."
  "I think you ought to know I'm feeling very depressed."
  "I have this terrible pain in all the daaiodes
   down my left side."
  "The playroom is full of jaient purple jello people."
  "Help me.  I'm having a zen experience."
  "Yaaw.  I think I'm experiencing natural stupidity."
  "I will not buy this tobaconist, it is scratched."
  "Hey buddy, can you spare some change?")

;; These messages are generated (in order) by CHATTER during tours.

(define-phrases tour-messages
  "I can avoid obstacles, follow corridors, recognize places,
   and navigate from point to point."
  "My vision system runs at 15 frames per second on a low-cost computer."
  "By the way, I don't understand anything I'm saying."
  "God, this place is such a dump.")
```

Bibliography

[1] Philip Agre and Ian Horswill. Cultural support for improvisation. In *Tenth National Conference on Artificial Intelligence*, Cambridge, MA, 1992. American Assoiciation for Artificial Intelligence, MIT Press.

[2] Philip E. Agre. The dynamic structure of everyday life. Technical Report 1085, Massachusetts Institute of Technology, Artificial Intelligence Lab, October 1988.

[3] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, 1987.

[4] John Aloimonos. Purposive and qualitative active vision. In *DARPA Image Understanding Workshop*, 1990.

[5] Yiannis Aloimonos and Azriel Rosenfeld. Computer vision. *Science*, 253:1249–1254, September 1991.

[6] Ronald C. Arkin. Motor schema based navigation for a mobile robot. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 264–271. IEEE, March 87.

[7] N. Ayache and O. D. Faugeras. Maintaining representations of the environment of a mobile robot. *IEEE Transactions on Robotics and Automation*, 5(6):804–819, 1989.

[8] Ruzena Bajcsy. Active perception vs. passive perception. In *Proc. Third IEEE Workshop on Computer Vision: Representation and Control*, pages 55–59. IEEE, October 1985.

[9] Dana H. Ballard. Animate vision. *Artificial Intelligence*, 48(1):57–86, 1991.

[10] A. Bandopadhay, B. Chandra, and D.H. Ballard. Egomotion using active vision. In *Proceedings, CVPR '86 (IEEE Computer Society Conference on Computer*

Vision and Pattern Recognition, Miami Beach, FL, June 22–26, 1986), IEEE Publ.86CH2290-5, pages 498–503. IEEE, 1986.

[11] Stephen T. Barnard and Martin A. Fischler. Computational and biological models of stereo vision. In *Proc. DARPA Image Understanding Workshop*, September 1990.

[12] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence.* William Kaufmann, Inc., 1981.

[13] Fred Bauer and John A. Nohel. *Ordinary Differential Equations: a First Course.* W. A. Benjamin, Inc., New York, 1967.

[14] King Beach. Becoming a bartender: The role of external memory cues in a work-directed educational activity. *Journal of Applied Cognitive Psychology*, 1992.

[15] Randall Beer. A dynamical systems perspective on autonomous agents. CES 92-11, Case Western Reserve University, Cleveland, Ohio, 1992.

[16] P. Bellutta, G. Collini, A. Verri, and V. Torre. Navigation by tracking vanishing points. In *AAAI Spring Symposium on Robot Navigation*, pages 6–10, Stanford University, March 1989. AAAI.

[17] Andrew Blake and Alan Yuille, editors. *Active Vision.* MIT Press, Cambridge, MA, 1992.

[18] David J. Braunegg. Marvel: A system for recognizing world locations with stereo vision. Technical report, MIT Artificial Intelligence Laboratory, 1990.

[19] J. Bresina and M. Drummond. Integrating planning and reaction. In J. Hendler, editor, *AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments.* AAAI, March 1990.

[20] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automoation*, 2(1):14–23, March 1986.

[21] Rodney A. Brooks. The behavior language; user's guide. AI Lab Memo 1227, MITAI, Apr 1990.

[22] Christopher Brown, David Coombs, and John Soong. Real-time smooth pursuit tracking. In Blake and Yuille [17], pages 126–136.

[23] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[24] David Chapman. Intermediate vision: Architecture, implementation, and use. TR 90-06, Teleos Research, 1990.

[25] David Chapman. Vision, instruction, and action. Technical Report 1204, Massachusetts Institute of Technology, Artificial Intelligence Lab, April 1990.

[26] David Chapman. *Vision, Instruction, and Action.* MIT Press, 1992.

[27] Noam Chomsky. *Aspects of the Theory of Syntax.* MIT Pres, Cambridge, MA, 1965.

[28] Jonathan H. Connell. *Minimalist Mobile Robotics.* Academic Press, 1990.

[29] David Coombs and Karen Roberts. "bee-bot": using peripheral optical flow to avoid obstacles. In *Proc. of the SPIE Conf. on Intelligent Robots and Computer Vision XI: Algorithms, Techniques, and Active Vision, (Boston, MA, November 15–20, 1992)*, 1992.

[30] Jill D. Crisman. Color region tracking for vehicle guidance. In Blake and Yuille [17], chapter 7.

[31] James E. Cutting. *Perception with an Eye for Motion.* MIT Press, 1986.

[32] DARPA SISTO. *Proceedings of the 1993 DARPA Image Understanding Workshop*, Washington, D.C., 1993. Morgan Kaufman.

[33] D. DeMenthon. A zero-bank algorithm for inverse perspective of a road from a single image. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 258–263. IEEE, March 1987.

[34] Ernst D. Dickmanns. Expectation-based dynamic scene understanding. In Blake and Yuille [17], chapter 18.

[35] Michael Dixon. Embedded computation and the semantics of programs. TR SSL-91-1, Xerox Palo Alto Research Center, Palo Alto, CA, September 1991.

[36] Bruce Randall Donald and James Jennings. Constructive recognizability for task-directed robot programming. *Robotics and Autonomous Systems*, 9:41–74, 1992.

[37] Sean P. Engelson and Drew McDermott. Image signatures for place recognition and map construction. In *Proceedings SPIE Symposium on Intelligent Robotic Systems, Sensor Fusion IV*, November 1991.

[38] Jerome A. Feldman. Four frames suffice: A provisionary model of vision and space. TR 99, Computer Science Department, University of Rochester, Rochester, NY 14627, September 1982.

[39] Margaret M. Fleck. Boundaries and topological algorithms. TR 1065, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[40] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings, AAAI-92*, 1992.

[41] J. J. Gibson. *The Senses Considered as Perceptual Systems.* Houghton-Mifflin, Boston, 1966.

[42] J. J. Gibson. *The Ecological Approach to Perception.* Houghton-Mifflin, Boston, 1979.

[43] Richard W. Hamming. *Coding and Information Theory.* Prentice Hall, Englewood Cliffs, N.J. 07632, 1980.

[44] Kristian J. Hammond and Timothy M. Converse. Stabilizing environments to facilitate planning and activity: An engineering argument. In *Ninth National Conference on Artificial Intelligence*, pages 787–793, Menlo Park, CA, July 1991. American Association for Artificial Intelligence, AAAI Press.

[45] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[46] B. K. P. Horn. *Robot Vision.* MIT Press, 1986.

[47] Ian Horswill. The senselisp programmer's manual. Unpublished technical note, MIT Artificial Intelligence Laboratory, 1989.

[48] Ian Horswill. How to hack yourself senselisp. Unpublished technical note, MIT Artificial Intelligence Laboratory, March 1990.

[49] Ian Horswill. Proximity detection using a spatial filter tuned in three-space. In *Proceedings of the 1991 AAAI Fall Symposium on Sensory Aspects of Robotic Intelligence*, 1991.

[50] Ian Horswill. *Specialization of perceptual processes.* PhD thesis, Massachusetts Institute of Technology, Cambridge, May 1993.

[51] Ian Horswill and Rodney Brooks. Situated vision in a dynamic environment: Chasing objects. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, August 1988.

[52] Ian D. Horswill. Reactive navigation for mobile robots. Master's thesis, Massachusetts Institute of Technology, June 1988.

[53] Katsushi Ikeuchi and Martial Herbert. Task oriented vision. In *DARPA Image Understanding Workshop*, 1990.

[54] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, 1986.

[55] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. A spectrum of abstraction hierarchies for planning. In *Proceedings of AAAI-90*, 1990.

[56] David Kortencamp. Applying computational theories of cognitive mapping to mobile robots. In Marc Slack and Erann Gat, editors, *Working notes of the AAAI Spring Symposium on Control of Selective Perception*, pages 83–89. AAAI, Cambridge, Massachusetts, 1992.

[57] A. Kosaka and A. C. Kak. Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties. *Computer Vision, Graphics, and Image Processing*, 56(3), September 1992.

[58] Jana Košecká. Control of discrete event systems. GRASP LAB report 313, University of Pennsylvania Computer and Information Science Department, Philadelphia, PA, April 1992.

[59] David J. Kriegman and Ernst Triendl. Stereo vision and navigation within buildings. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 402–408. IEEE, March 87.

[60] David J. Kriegman, Ernst Triendl, and Tomas O. Binford. A mobile robot: Sensing, planning and locomotion. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 402–408. IEEE, March 87.

[61] Benjamin J. Kuipers and Yung-Tai Byun. A robust, qualitative approach to a spatial learning mobile robot. In *SPIE Advances in Intelligent Robotics Systems*, November 1988.

[62] Jean-Claude Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, 1991.

[63] Hector J. Levesque and Ronald J. Brachman. A fundamental tradeoff in knowledge representation and reasoning (revised edition). In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*, pages 42–70. Morgan Kaufman, Los Altos, CA, 1985.

[64] Michael L. Littman. An optimization-based categorization of reinforcement learning environments. In Meyer and Wilson [73], pages 262–270.

[65] David G. Luenberger. *Introduction to Dynamic Systems: Theory, Models, and Applications*. John Wiley and Sons, 1979.

[66] Kevin Lynch. *The Image of the City*. MIT Press, 1960.

[67] D. M. Lyons and A. J. Hendriks. Exploiting patterns of interaction to achieve reactive behavior. *in submission*, 1993.

[68] David Marr. *Vision*. W. H. Freeman and Co., 1982.

[69] Maja J. Mataric. Minimizing complexity in controlling a collection of mobile robots. In *IEEE International Conference on Robotics and Automation*, pages 830–835, Nice, France, May 1992.

[70] David McFarland. What it means for robot behavior to be adaptive. In Meyer and Wilson [72], pages 22–28.

[71] Jean-Arcady Meyer and Agnes Guillot. Simulation of adaptive behavior in animats: Review and prospect. In Meyer and Wilson [72], pages 2–14.

[72] Jean-Arcady Meyer and Stewart W. Wilson, editors. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*. MIT Press, Cambridge, Massachusetts, 1991.

[73] Jean-Arcady Meyer and Stewart W. Wilson, editors. *From Animals to Animats: The Second International Conference on Simulation of Adaptive Behavior*. MIT Press, Cambridge, Massachusetts, 1993.

[74] Hans P. Moravec. The stanford cart and cmu rover. Technical report, Robotics Institute, Carnegie-Mellon University, February 1983.

[75] Randal C. Nelson. Visual homing using an associative memory. In *Proceedings of the DARPA Image Understanding Workshop*, pages 245–262, 1989.

[76] ed. Nils J. Nilsson. Shakey the robot. Technical Report 323, SRI International, April 1984.

[77] H. Keith Nishihara. Minimal meaningful measurement tools. TR 91-01, Teleos Research, 1991.

[78] R. C. Patton, H. S. Nwana, M. J. R. Shave, and T. J. M. Bench-Capon. Computing at the tissue/organ level (with particular reference to the liver). In Varela and Bourgine [107], pages 411–420.

[79] Roger Penrose. *The Emperor's New Mind.* Oxford University Press, 1989.

[80] T. Poggio and V. Torre. Ill-posed problems and regularization analysis in early vision. In Lee S. Baumann, editor, *Image Understanding Workshop (New Orleans, LA, October 3-4, 1984)*, pages 257–263. Defense Advanced Research Projects Agency, Science Applications International Corp., 1984.

[81] Dean A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1), 1991.

[82] Louise Pryor and Gregg Collins. Planning to perceive: a utilitarian approach. In Simmons [91], pages 113–122.

[83] Douglas A. Reece and Steven Shafer. Active vision at the system level for robot driving. In Simmons [91], pages 70–77.

[84] Daniel Reisfeld, Haim Wolfson, and Yehezkel Yeshurun. Detection of interest points using symmetry. In *Proceedings of the Third International Conference on Computer Vision*, pages 62–65, Osaka, Japan, December 1990. IEEE Computer Society.

[85] Herbert L. Roitblat. *Introduction to Comparitive Cognition.* W. H. Freeman and Company, 1987.

[86] Stanley J. Rosenschein. Formal theories of knowledge in ai and robotics. report CSLI-87-84, Center for the Study of Language and Information, Stanford, CA, 1987.

[87] Stanley J. Rosenschein. Synthesizing information-tracking automata from environment descriptions. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 386–393, May 1989.

[88] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of machines with provable epistemic properties. In Joseph Halpern, editor, *Proc. Conf. on Theoretical Aspects of Reasoning about Knowledge*, pages 83–98. Morgan Kaufmann, 1986.

[89] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2), 1974.

[90] Roger C. Schank. *Tell Me a Story.* Charles Scribner's Sons, 1990.

[91] Reid Simmons, editor. *Working notes of the AAAI Spring Symposium on Control of Selective Perception*, Stanford, California, 1992. American Association for Artificial Intelligence.

[92] Herbert A. Simon. *Sciences of the Artificial.* MIT Press, Cambridge, Massachusetts, 1970.

[93] Lee Spector and James Hendler. The supervenience architecture. In Avi Kak, editor, *Working notes of the AAAI Fall Symposium on Sensory Aspects of Robotic Intelligence*, pages 93–100. AAAI Press, Asilomar, California, 1991.

[94] Anselm Spoerri. The early detection of motion boundaries. Technical Report 1275, MIT Artificial Intelligence Laboratory, 1991.

[95] K. Storjohann, T. Zeilke, H. A. Mallot, and W. von Seelen. Visual obstacle detection for automatically guided vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 761–766, May 1990.

[96] Michael J. Swain. Color indexing. Technical Report 390, University of Rochester Computer Science Department, November 1990.

[97] Michael J. Swain. Active visual routines. In Simmons [91], pages 147–149.

[98] W.B. Thompson and J.K. Kearney. Inexact vision. In *Workshop on Motion: Representation and Analysis*, 1986.

[99] W.B. Thompson, K.M. Mutch, and V.A. Berzins. Dynamic occlusion analysis in optical flow fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7:374–383, 1985.

[100] C. E. Thorpe. *FIDO: Vision and Navigation for a Robot Rover.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, December 1984.

[101] Where to Look Next Using a Bayes Net: The TEA-1 System and Future Directions. Raymond d. rimey. In Simmons [91], pages 118–122.

[102] Peter M. Todd and Stewart W. Wilson. Environment structure and adaptive behavior from the ground up. In Meyer and Wilson [73], pages 11–20.

[103] Peng-Seng Toh and Andrew K. Forrest. Occlusion detection in early vision. In *Proceedings of the International Conference on Computer Vision*, 1990.

[104] John K. Tsotsos. Analyzing vision at the complexity level. *Behavioral and Brain Sciences*, 13(3):423–469, 1990.

[105] Matthew A. Turk, David G. Morgenthaler, Keith Gremban, and Martin Marra. Video road following for the autonomous land vehicle. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 273–280. IEEE, March 1987.

[106] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984.

[107] F. J. Varela and P. Bourgine, editors. *Toward a Practice of Autonomous Systems: the Proceedings of the First European Conference on Artificial Life*. MIT Press, Cambridge, MA, 1992.

[108] R. Wallace. Robot road following by adaptive color classification and shape tracking. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 258–263. IEEE, March 1987.

[109] R. Wallace, K. Matsuzaki, Y. Goto, J. Crisman, J. Webb, and T. Kanade. Progress in robot road-following. In *1986 IEEE Internation Conference on Robotics and Automation*, pages 1615–1621, April 1986.

[110] R. Wallace, A. Stenz, C. Thorpe, H. Moravec, W. Whittaker, and T. Kanade. First results in robot road-following. In *IJCAI-85*, 1985.

[111] A. M. Waxman, J. LeMoinge, and B. Srinivasan. Visual navigation of roadways. In *1985 IEEE Internation Conference on Robotics and Automation*, April 1985.

[112] Norbert Wiener. *Cybernetics*. MIT Press, Cambridge, 1961.

[113] Stewart W. Wilson. The animat path to ai. In Meyer and Wilson [72], pages 15–21.

[114] Lambert E. Wixson. Detecting occluding edges without computing dense correspondence. In IU93 [32], pages 933–938.

[115] John Woodfill and Ramin Zabih. Using motion vision for a simple robotic task. In *AAAI Fall Symposium on Sensory Aspects of Robotic Intelligence*, 1991.