$\mathbf{s}_i$	A vector of k_s floating point numbers
$\mathbf{a}_i$	A vector of k_a floating point numbers
$\mathbf{b}_i$	A vector of k_b floating point numbers
$\mathbf{b}_i^{\mathbf{smooth}}$	A vector of k_b floating point numbers
$T_i^{\mathbf{birth}}$	An integer: The number of exemplars which had been stored in the tree before this one was added
M_i	A short integer: the mass of this exemplar
W_i	A floating point number: the denominator of the local average calculation of $\mathbf{b}_i^{\mathbf{smooth}}$, discussed below

Table 7.1: The fields of a *SA*-tree node

7.4.2 *SAB*-tree Representation: Details

An *SAB*-tree has two roots: an *SA*-tree root and an *SB*-tree root. The former is a *k*d-tree indexed on the $(\mathbf{s}, \mathbf{a})$ components, and the latter indexed on the $(\mathbf{s}, \mathbf{b})$ components. All the data is held in the nodes of the *SA*-tree, while the nodes of the *SB*-tree are simply pointers to *SA*-tree nodes which contain the same exemplar information. The difference between these *k*d-trees is that the hyperplanes splitting up the *SA*-tree space are in **State** $\times$ **Action** space, while the hyperplanes for the *SB*-tree are in **State** $\times$ **Behaviour** space. Thus, nearest neighbour (and related) searches can take place in either the $(\mathbf{s}, \mathbf{a})$ or $(\mathbf{s}, \mathbf{b})$ components by using the appropriate root. In either case the result is an *SA*-tree node.

Each node of the *SA*-tree contains the data shown in Table 7.1. The set of nodes are subject to the following invariants, which are maintained whenever a new exemplar is added:

$$W_i = \sum_{j \in \mathit{locals}(i)} \left(\mathit{weight}(\mid (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j) \mid) \times M_j \right) \tag{7.13}$$

$$\mathbf{b}_i^{\mathbf{smooth}} = \frac{\sum_{j \in \mathit{locals}(i)} \left(\mathit{weight}(\mid (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j) \mid) \times M_i \times \mathbf{b}_j \right)}{W_i} \tag{7.14}$$

where $j \in \mathit{locals}(i)$ is true if $\mid (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j) \mid < D_{\mathbf{range}}$ and the function $\mathit{weight} : \Re \rightarrow \Re$ is defined in Section 7.2.

7.4.3 Update

All the computation is performed when a point is added to the *SAB*-tree instead of when the *SAB*-tree is queried to make a prediction. This is useful because there will be a maximum of one update per control cycle, whereas there may be many predictions required in order to assess different possible actions. Table 7.2 illustrates the sequence of operations when a new point $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{b}_{n+1})$

Operation	Action	Order of Cost
find-locals	Find all local exemplars in range of smoothing kernel.	$k_{sa}(S \log N)$
maintain-bsmooth	Evaluate $\mathbf{b}_{n+1}{}^{\text{smooth}}$.	$k_{sa}S$
	Update $\mathbf{b}_i{}^{\text{smooth}}$ components of locals.	$k_{sa}S$
remove-bad	Check for elderly and inaccurate locals.	$k_{sa}S$
	Delete bad points.	(see text)
insert-new	Find insertion tree nodes.	$(k_{sa} + k_{sb}) \log N$

Table 7.2: When a new exemplar is added

is added. In this table, N is the number of exemplars in the SAB-tree and S is the number of exemplars found to lie within the smoothing range of $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1})$. $k_{sa} = k_s + k_a$ and $k_{sb} = k_s + k_b$.

The **find-locals** operation makes a range search (also described in Section 6.7) of the SAB-tree in the $(\mathbf{s}, \mathbf{a})$ components, finding all exemplars within distance $D_{\mathbf{range}}$ of $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1})$. Assume that S are found. It is hard to predict in advance how the expected value of S will depend on the dimensionality of the data and the size of the SAB-tree, and so this will be left to empirical observation.

maintain-bsmooth evaluates the $\mathbf{b}^{\text{smooth}}$ of the new exemplar and simultaneously updates the locals. The smoothed values of the locals could be readjusted naively by recursively recomputing the smoothed values for each local. This would involve a range search for each of the S locals, which would be expensive. Instead, the W_i field can be used to reduce the computation to $2k_b$ arithmetic operations for each local.

Let $\left[\mathbf{b}_i{}^{\text{smooth}}\right]_{\text{old}}$ be the smoothed value of the ith local exemplar before the new exemplar, $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{b}_{n+1})$, is added. Let $\left[\mathbf{b}_i{}^{\text{smooth}}\right]_{\text{new}}$ be the smoothed value after. Finally, let $\Delta W = \mathit{weight}(\mid (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_{n+1}, \mathbf{a}_{n+1}) \mid)$. Then from Equation 7.14

$$\left[\mathbf{b}_i{}^{\text{smooth}}\right]_{\text{old}} = \frac{\sum_{j \in \mathit{locals}(i)} (\mathit{weight}(\mid (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j) \mid) \times M_j \times \mathbf{b}_j)}{W_i} \tag{7.15}$$

and

$$\left[\mathbf{b}_i{}^{\text{smooth}}\right]_{\text{new}} = \frac{\Delta W \times \mathbf{b}_{n+1} + \sum_{j \in \mathit{locals}(i)} (\mathit{weight}(\mid (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j) \mid) \times M_j \times \mathbf{b}_j)}{\Delta W + W_i} \tag{7.16}$$

$$= \frac{\Delta W \times \mathbf{b}_{n+1} + \left[\mathbf{b}_i{}^{\text{smooth}}\right]_{\text{old}} \times W_i}{\Delta W + W_i} \tag{7.17}$$

Algorithm:	Maintaining $\mathbf{b}_i{}^{\mathbf{smooth}}$ components
Input:	$\mathbf{s}_{n+1}$, the new state vector $\mathbf{a}_{n+1}$, the new action vector $\mathbf{b}_{n+1}$, the new behaviour vector **locals**, the set of local vectors
Output:	$\mathbf{b}_{n+1}{}^{\mathbf{smooth}}$, the new smoothed vector W_{n+1} the new denominator value
Changed:	The $\mathbf{b}_i{}^{\mathbf{smooth}}$ and W_i components of each exemplar in **locals**
Preconditions:	Invariants of Equations 7.14 and 7.13
Postconditions:	$\mathbf{b}_{n+1}{}^{\mathbf{smooth}}$, W_{n+1}, and the $\mathbf{b}_i{}^{\mathbf{smooth}}$ component and W_i component of each exemplar in **locals** is updated to maintain the invariant upon the addition of the new exemplar.
Code:	
1.	$\mathbf{weighted\text{-}sum} := \mathbf{0}$
2.	$W_{n+1} := 0$
3.	for each exemplar $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i, \mathbf{b}_i{}^{\mathbf{smooth}}, M_i, W_i) \in$ **locals**
3.1	$\quad \Delta W := \mathit{weight}(\mid (\mathbf{s}_{n+1}, \mathbf{a}_{n+1}) - (\mathbf{s}_i, \mathbf{a}_i) \mid)$
3.2	$\quad \mathbf{b}_i{}^{\mathbf{smooth}} := \dfrac{\left[\mathbf{b}_i{}^{\mathbf{smooth}}\right]_{\mathrm{old}} \times W_i + \mathbf{b}_{n+1} \times \Delta W}{W_i + \Delta W}$
3.3	$\quad W_i := W_i + \Delta W$
3.4	$\quad \mathbf{weighted\text{-}sum} := \mathbf{weighted\text{-}sum} + \Delta W \times M_i \times \mathbf{b}_i$
3.5	$\quad W_{n+1} := W_{n+1} + \Delta W \times M_i$
4.	$\mathbf{b}_{n+1}{}^{\mathbf{smooth}} := \dfrac{\mathbf{weighted\text{-}sum}}{W_{n+1}}$

Table 7.3: Algorithm: The **maintain-bsmooth** operation. This evaluates the new smoothed behaviour component and adjusts the local smoothed components.

When $\left[\mathbf{b}_i{}^{\mathbf{smooth}}\right]_{\mathbf{new}}$ has been updated, W_i is then updated to $W_i + \Delta W$. The algorithm to simultaneously update locals and find $\mathbf{b}_{n+1}{}^{\mathbf{smooth}}$ is given in Table 7.3.

The **remove-bad** operation searches those local points which are older than the mean age and finds any which are deemed inaccurate by the measure of Section 7.3. These are deleted from the *SAB*-tree, which is achieved by simply marking the exemplar to be ignored in all future searches. Section 7.5 details how these dead nodes are occasionally garbage collected. When a point is deleted, all of its local neighbours must change their $\mathbf{b}^{\mathbf{smooth}}$ components to take account of the deletion. The local neighbours of the exemplar to be deleted are found by a range search, and their $\mathbf{b}_i{}^{\mathbf{smooth}}$ and W_i components are updated accordingly (by an algorithm analogous to the **maintain-bsmooth** algorithm).

The **insert-new** operation finds the correct node in each of the SA-tree and the SB-tree in which to store the new exemplar, using the incremental insertion operation from Section 6.7. During the search for the correct node it might find a near identical node, in which case it is not stored explicitly but instead implicitly, by increasing the mass of the near identical node.

7.5 SAB-tree Garbage Collection

The world model represented by a SAB-tree is updated and accessed *during* task execution. The local updates have the ideal abstract properties which allow us to consider the behaviour of the SAB-tree as a simple set of exemplars which can be accessed by nearest neighbour. When the task is not executing, there may be some desirable alterations which could be made to the SAB-tree. These are not intended to alter the meaning of it in any way, but to improve the expected access time for future nearest neighbour searches.

One exemplar set can be represented by many different SAB-trees, and some SAB-trees are more efficient than others. The tree that has been built incrementally may not be the most efficient for the following reasons:

1. **Poor Structure.** As discussed in Section 6.7, the shape of the hyperregions in a kd-tree can affect the nearest neighbour search. These can only be chosen well if the tree is built with knowledge of all the data which it is to contain: this is clearly not possible if points are added incrementally, when the kd-tree's structure will be determined entirely according to the order in which the points arrived.

2. **Imbalance.** A robotic system is likely to spend periods of time in very similar areas of the state space. This can mean very uniform data arriving which causes serious imbalance.

3. **Dead Points.** Points are marked as dead when they are old and inaccurate. From then on they remain in the tree, wasting memory and occasionally increasing search time. This is particularly serious after a sudden change which results in large regions of the SAB-tree being killed.

The garbage collection operation happens when the robotic system is not in use. It flattens the SAB-tree into a list of exemplars, removes the dead points and then rebuilds the SAB-tree using the balance-inducing and hyperrectangle-choosing algorithm of Section 6.7. The theoretical cost is

$$O((k_s + k_a + k_b)(N_D + N \log N)) \tag{7.18}$$

where N is the number of live points and N_D the number of dead points. There is an amusing analogy between this garbage collection and human sleep; one purpose of the latter is believed to be the removal of extraneous links between memories or knowledge acquired during the day.

Chapter 8

SAB Control

A computationally cheap method for learning world models has been proposed and developed in Chapters 5—7. This chapter shows how to use the world models to make control decisions using a probabilistic heuristic. It then analyses and illustrates the method.

8.1 Making a Control Decision

It is now time to review the control cycle first introduced in Section 4.4. This cycle controlled a dynamic task while maintaining and improving its *SAB* world model. Since then, Chapter 5 explained how and why the nearest neighbour generalization is to be used to represent the model. Chapters 6 and 7 showed to to update and access this world model, in the possible presence of disorder. The emphasis was on performing these operations efficiently. This chapter addresses the question of how the world model should be used. It will consider a general method to implement Step 3 of the *SAB* control cycle, repeated here for convenience in Table 8.1.

To motivate this section, I provide an illustration from the simple Mountain Car domain of Section 4.3. In this example, the state is the distance from the start, the action is the distance that the pedal is pressed and the behaviour is the perceived horizontal velocity. Imagine that learning has just begun, and the task is to guide the mountain car from state $s = 0.5$km to state $s = 9.5$km at a constant velocity $b = 5.6$ units. An error of ± 0.2 units is considered acceptable. So far, there have been four *SAB* learning cycles which have produced these four exemplars:

$$\mathbf{E} = \{(0.5, 4.8 \rightarrow 5.7), (1.8, 6.0 \rightarrow 6.3), (2.7, 7.5 \rightarrow 5.1), (1.3, 9.0 \rightarrow 3.5)\} \tag{8.1}$$

The situation is depicted in Figure 8.1, with the exemplars positioned in **State** $\times$ **Action** space. The polygons separate the various nearest neighbour regions. The vertical line denotes the current state $\mathbf{s}_{\mathbf{current}} = 3.4$.

At this point the controller must decide what to do. If it chooses a low-valued action then the

1.	Observe current perceived state $\mathbf{s}_{\mathbf{current}}$
2.	Receive task specific goal behaviour $\mathbf{b}_{\mathbf{goal}}$ from higher level of control. The requested behaviour might depend partially on $\mathbf{s}_{\mathbf{current}}$.
3.	**Access the current world model to obtain a raw action $\mathbf{a}_{\mathbf{raw}}$ which is predicted to be likely to achieve $\mathbf{b}_{\mathbf{goal}}$ acting in the current state.**
4.	Apply action $\mathbf{a}_{\mathbf{raw}}$.
5.	Observe actual behaviour $\mathbf{b}_{\mathbf{actual}}$.
6.	Update the world model with the information that $(\mathbf{s}_{\mathbf{current}}, \mathbf{a}_{\mathbf{raw}}) \rightarrow \mathbf{b}_{\mathbf{actual}}$.

Table 8.1: The *SAB* Control Cycle

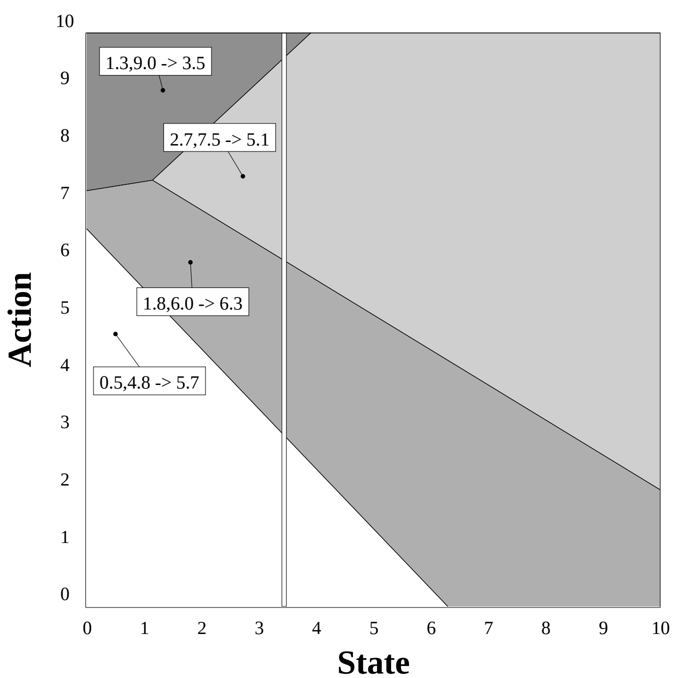

Figure 8.1

State $\times$ **Action** space with the regions denoting the behaviour predicted by the nearest neighbour. The brighter the region, the closer it is to the goal behaviour of 5.6. Now we want to find x to give $(3.4, x \rightarrow 5.6)$.

state-action pair it applies will be in the lower bright region, which is predicted to be very good, with behaviour 5.7. But in many ways, an action near the most recent action (which produced behaviour 5.1) has a prediction which looks more *reliable* because it is closer. As a result, despite the predicted behaviour being inferior, it might be preferable (we will see later that this is what partial inversion would recommend). But then, perhaps it would be a mistake to try the same action again: if the controller never tries anything new, how can it hope to improve?

8.1.1 The Utility of Information

There are two opposing aims in making a control choice. These are

1. **Perform.** We wish to perform as well as possible given the knowledge contained in the performance element.

2. **Experiment.** In order to perform better in future, it is worth trying actions with no guaranteed success, but with the chance of a valuable discovery.

Aim 1 represents doing the best for the system immediately and Aim 2 represents an investment in the future. This is the same decision as that of a corporate director deciding whether to pay the shareholders a dividend, or provide extra finances for the company's R & D division. The system must take into account the *utility of information.*

Given a *SAB*-tree representation of the PSTF, the **perform** mode of operation can be easily achieved by partial inversion. There might be some experience in the *SAB*-tree in which an action applied in a state close to the current state had produced behaviour close to that of the goal. If this is so, then partial inversion will obtain this action. If such previous experience is not available, partial inversion may suggest something sensible, but for reasons described in Section 5.1 is not reliable. Figure 8.2 shows the action recommended by partial inversion for the exemplar set of Equation 8.1.

It is immediately clear that it is not desirable for the learning controller to either perform only Aim 1 or Aim 2. If it only ever tried what it knew best there would be no diversity of experience and learning would not occur. If it only ever experimented there would be a large body of knowledge, but no improvement in performance.

8.1.2 Random Experimentation

One approach to this dilemma is to have two modes of operation. The first is experimental, where actions are chosen randomly with no reference to previous experience. The second is demonstration-mode in which no chances are taken, and the best known action is always used. These modes are altered by an intelligent supervisor which can judge when further experimentation is needed and when it is positively unwelcome. An effective example of these two modes of operation is Mel's

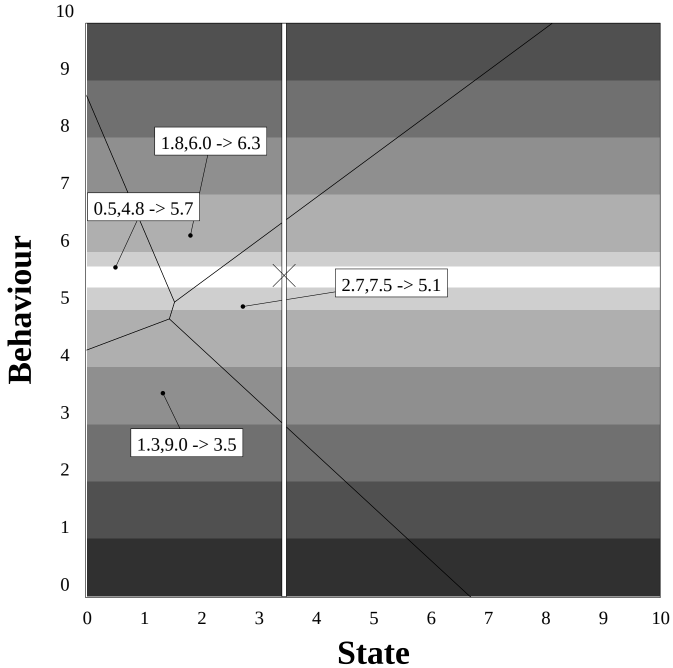

Figure 8.2

Exemplars in **State** × **Behaviour** space. The regions denote the action recommended by partial inversion. The shading denotes the desirability of various behaviours. Partial inversion to obtain behaviour 5.6 finds the nearest neighbour to the cross, and thus recommends action 7.5.

MURPHY [Mel, 1989]. This learns the inverse differential kinematics by firstly "flailing" the arm in front of its camera and then making plans based on the results of the observations.

Instead of requiring an intelligent supervisor to decide when it has finished learning, the controller can occasionally test itself.

Let us call this method 'R'-control (Random decisions). The mountain car learning task was run with alternate trials of perform and experiment. On every odd numbered trial, the controller simply chose random actions. On every even numbered trial, the controller, on each occasion, chose the best known action. The choice was by means of partial inversion.

The learning curve for this scheme is shown in Figure 8.3. The learning run was was repeated forty times. Each learning run started with no world knowledge, and then performed eighty trials. Each trial was an attempt at the mountain car task. The car was started at state 0.5km and actions were executed until the state was greater than 9.5km. An action was successful if it resulted in a speed between 5.4 units and 5.8 units. The correct trajectory is obtained by the set of actions which keep the state in the white zone of the diagram. On each odd trial, random actions were taken, and on each even trial the best known actions were taken. For each even trial, the number of unsuccessful action choices was recorded. For each odd, random, trial the results were discarded. For the nth trial, the graph shows the mean number of errors for the forty occasions of trial n. The vertical bars display sample standard deviations (for explanation of the confidence-level key, see Appendix A).

This graph indicates that the 'R'-controller does indeed learn, but that even after eighty trials the performance tends to be poor. The learning rate is slow because, for each area of the state space, the controller must wait until the random experiments happen to obtain a correct action.

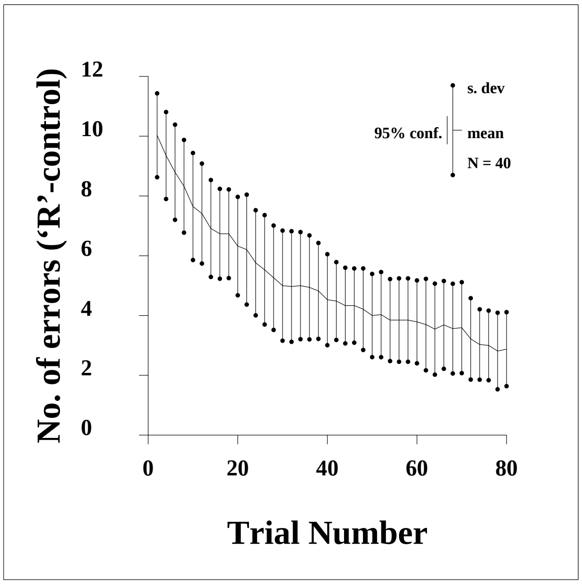

Figure 8.3

Performance against trial number for a controller which, on alternate trials, performs and well as possible and experiments randomly. This data is obtained from forty independent learning runs.

Some experiments at some states might be lucky early, others very unlucky, perhaps for all eighty trials. The slow learning is directly related to the large amount of useless data collected. Figure 8.4 shows the distribution of exemplars in the SAB-tree after one such run of eighty trials, and it can be seen to be almost uniform. As I explained in Chapter 5, a uniform distribution of exemplars can lead to a very slow rate of learning. Recalling from Chapter 5 how badly the nearest neighbour generalization performs in uniform distributions, it is perhaps surprising that learning occurs at all. The reason it does occur is that the mountain car uses only a two-dimensional control space. It seems unlikely, in a control space of higher dimension, that the method would learn to a tolerance as accurate as this within an attainable number of trials.

8.1.3 Local Random Experimentation

An improvement is the 'L'-controller. This makes local random perturbations of the best known action instead of choosing entirely random actions. The first action, in the first trial, is chosen entirely at random, because with no experience there can be no best known action. Figure 8.5 shows the performance, with identical experimental conditions to those of the 'R'-controller. Its learning is superior, and occasionally it achieves perfect behaviour within eighty trials. There is still great deviation in learning speeds. The speed depends on the very first randomly chosen action. If it happens to be very wrong, then it will take many trials until it is adjusted enough to be right. If it is lucky with the first random guess, then learning will proceed much more quickly.

There is also another danger with 'L'-control which was escaped here: that it might fall into a local minimum at some area of the state space. This would occur if the best known action is better than its surrounding actions, without being globally best. In a slightly more complex domain, such

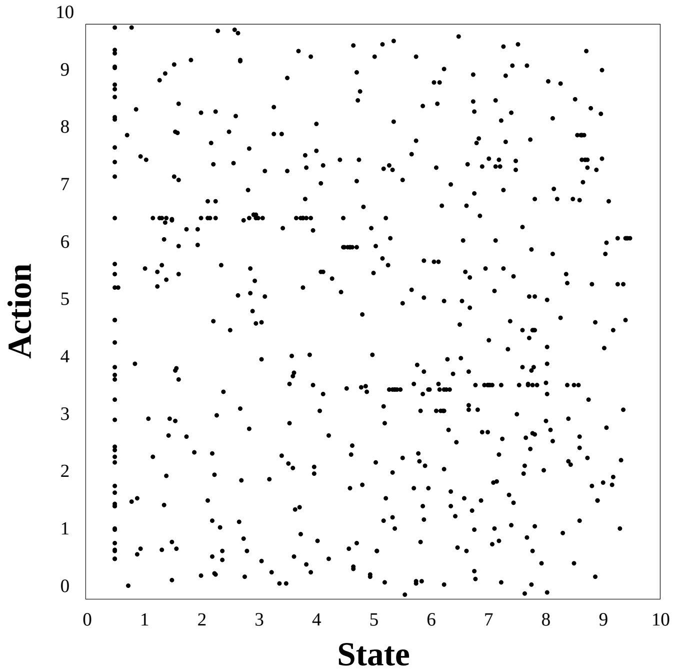

Figure 8.4

The distribution of exemplars in **State** $\times$ **Action** space after learning using the 'R'-controller.

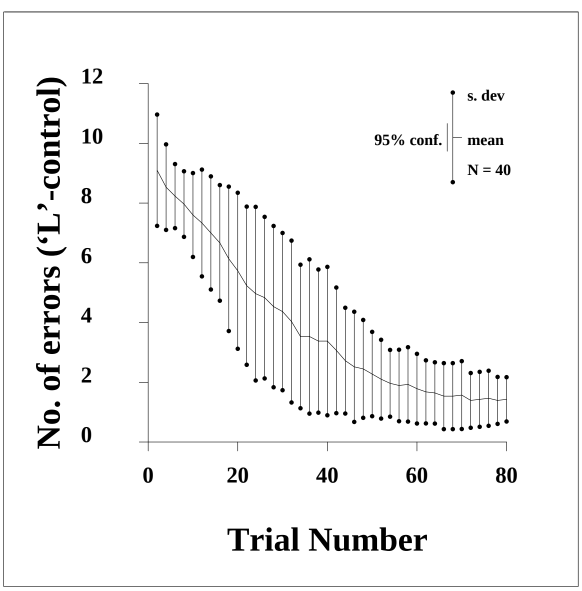

Figure 8.5

Performance against trial number for a controller which, on alternate trials, performs as well as possible and experiments with local perturbations. This data is obtained from forty independent learning runs.

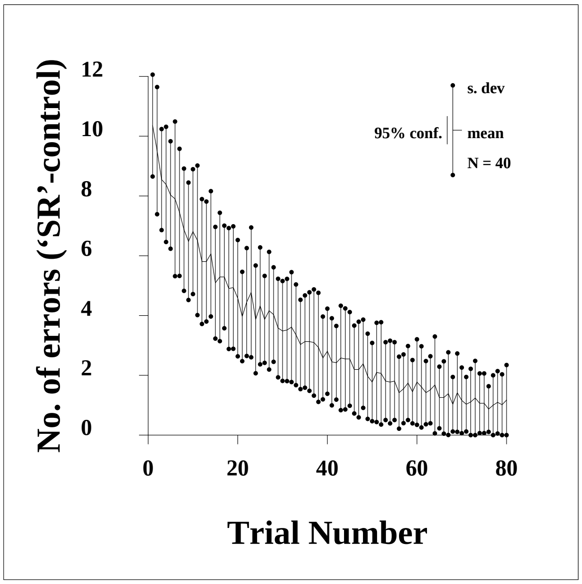

Figure 8.6

Performance against trial number for a sceptical controller ('SR'-control). This checks the best recommended action before use, and if success is not predicted, tries something random.

local minima are possible. A controller which has a danger of never escaping local minima is not robust.

A combination of 'R'-control and 'L'-control could be designed which, when experimenting, would occasionally choose locally and occasionally globally. This is similar to the solution which simulated annealing uses to avoid local minima in its hill climbing. The learning behaviour is likely to lie between that of 'R' and 'L'-control, with the danger of local minima removed.

8.1.4 Sceptical Experimentation

The control schemes mentioned up to now have the problem that learning does not occur *during* a trial. During odd-numbered trials there are occasions when the 'R' or 'L'-controller should know what should be done next but wastefully ignores it, instead trying some different random action. Conversely, on some even trials there are occasions when the controller should know that the action suggested by partial inversion is not good enough. On such occasions, to repeat the same action is to repeat a known error.

Instead, because the exemplar set can be used for prediction as well as for partial inversion, a candidate action can be *checked* before its application. Thus the use of two modes of operation can be dispensed with, and instead the best known action can, on all occasions be considered, but only as a *candidate* action. Such a sceptical controller only then experiments when the candidate action is predicted to be unsuccessful. Figures 8.6 and 8.7 shows the results of two sceptical controllers. 'SR'-control uses the best known action when it predicts success, and chooses a random experiment otherwise. 'SL' is similar except that it perturbs the best action when experimenting.

These methods perform better than 'R'-control and 'L'-control respectively. 'SL'-control achieved

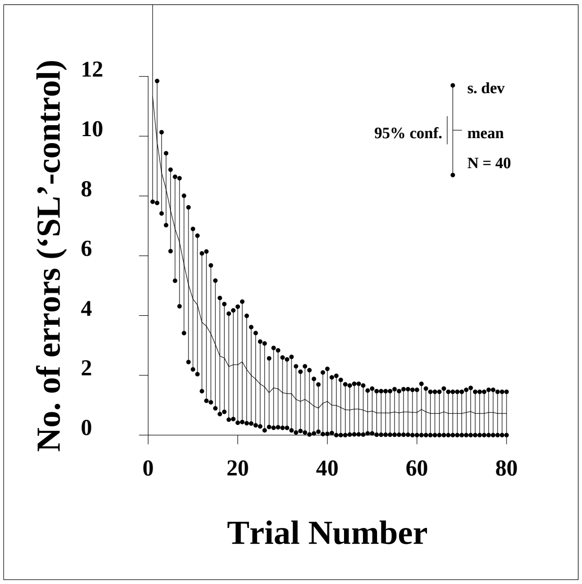

Figure 8.7

Performance against trial number for a sceptical controller ('SL'-control). This checks the best recommended action before use, and if success is not predicted, perturbs the action.

perfect performance within eighty trials on almost half of the learning runs. This success motivates an attempt to try to squeeze further information out of the exemplar set.

8.1.5 SAB Action Chooser

The new scheme will again be content if the predicted behaviour of the candidate action is successful. However, if not it will choose an alternative action with more care. Instead of choosing the first random action which it generates, it generates several random actions, and in turn uses its world model to predict the result of using them. How should it rank different candidate actions?

One idea is to choose the action with the best predicted behaviour. Upon inspection, this loses its appeal. If nothing has worked well in the controller's experience, then despite its inadequacy, the action with the best predicted behaviour is likely to be close to or at the initial candidate action obtained from partial inversion. Thus the controller will consistently choose something which it knows will perform mundanely in preference to actions which it, less reliably, predicts to be worse.

Fortunately, the *reliability* of predictions can also be estimated thanks to the explicit exemplar set representation. The closer the nearest neighbour which has been used to make the prediction, the more reliable it is. A desirable qualitative ranking of candidate actions is:

1. (Best) The candidate action is predicted to be successful, and is reliable.

2. The candidate action is predicted to be successful, but the prediction is unreliable.

3. The candidate action is not predicted to be successful, but this prediction is not reliable.

4. (Worst) The candidate action is reliably predicted to be unsuccessful.

I will now introduce a quantitative heuristic which achieves this qualitative ranking. First, it should be made clear what constitutes successful behaviour. The *SAB* cycle definition states that the higher level controller specifies a target behaviour $\mathbf{b}_{\text{goal}}$. If success is ever to be attainable, some small deviation from this goal must be deemed acceptable. Depending on the task, different accuracies might be adequate. Furthermore, different components of the goal behaviour may be needed to different accuracy. To specify this, the controller also supplies a $\mathit{Dim}(\mathbf{Behaviour})$-dimensional tolerance vector, $\boldsymbol{\tau}$. It is not necessary for the tolerance to be varied during learning—if high accuracy is required a high tolerance can be specified right from the start. Behaviour $\mathbf{b}$ *achieves behaviour* $\mathbf{b}_{\text{goal}}$ *to tolerance* $\boldsymbol{\tau}$ iff

$$\forall i \quad [\mathbf{b}_{\text{goal}}]_i - \boldsymbol{\tau}_i < \mathbf{b}_i < [\mathbf{b}_{\text{goal}}]_i + \boldsymbol{\tau}_i \tag{8.2}$$

where $\mathbf{b}_i$ is the ith component of vector $\mathbf{b}$.

The ranking heuristic can be achieved quantitatively by some elementary decision analysis. Actions are ranked by their estimated *probability of success.*

To make this estimate, we have at our disposal the nearest neighbour in $\mathbf{E}$ to $(\mathbf{s}_{\text{current}}, \mathbf{a})$. Call this exemplar $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}^{\text{smooth}}_{\text{near}})$. We model the unknown behaviour at the point $(\mathbf{s}_{\text{current}}, \mathbf{a})$ as a random variable influenced by this nearest neighbour. Call this random variable B. It varies over the space of real-valued $\mathit{Dim}(\mathbf{Behaviour})$-dimensional vectors. The expected behaviour is $\mathbf{b}^{\text{smooth}}_{\text{near}}$. The reliability of the expected behaviour is high if the exemplar is close, and it is low if the exemplar is distant. This is modelled by setting the standard deviation of the variable to be proportional to the distance to $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}})$. Call the constant of proportionality C. There is no other useful information which can be obtained from the nearest neighbour, and so to fully specify the random variable, this heuristic will assume it is distributed normally with independent vector components. We are now in a position to quantify the probability of success. Let us first consider the chance that the ith component is successful. This probability is denoted by

$$P^i_{\text{succ}}(\mathbf{a}, \mathbf{s}_{\text{current}}, [\mathbf{b}_{\text{goal}}]_i, \boldsymbol{\tau}_i, \mathbf{E}) \tag{8.3}$$

which we will define as the probability that applying action $\mathbf{a}$ in the current state $\mathbf{s}_{\text{current}}$ will cause the ith component of the resulting behaviour to be successful. Writing the ith component of the resulting behaviour as B_i, then "ith component success" means that B_i is within tolerance, or symbolically:

$$[\mathbf{b}_{\text{goal}}]_i - \boldsymbol{\tau}_i < B_i < [\mathbf{b}_{\text{goal}}]_i + \boldsymbol{\tau}_i \tag{8.4}$$

The probability of success will be computed solely in terms of the exemplar set $\mathbf{E}$. To summarize, we have defined

$$\begin{aligned} &P^i_{\text{succ}}(\mathbf{a}, \mathbf{s}_{\text{current}}, [\mathbf{b}_{\text{goal}}]_i, \boldsymbol{\tau}_i, \mathbf{E}) \\ &\qquad = \mathbf{Prob}([\mathbf{b}_{\text{goal}}]_i - \boldsymbol{\tau}_i < B_i < [\mathbf{b}_{\text{goal}}]_i + \boldsymbol{\tau}_i \;\Big|\; \mathbf{s}_{\text{current}}, a, \mathbf{E}) \end{aligned} \tag{8.5}$$

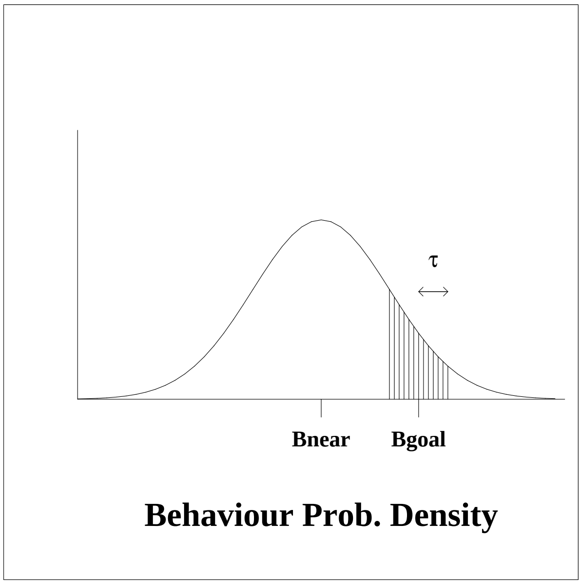

Figure 8.8

The P_{succ} estimate. The probability of success is the area of the shaded region.

The probability is computed with respect to the nearest neighbour of $(\mathbf{s}_{\text{current}}, \mathbf{a})$. Call this neighbour $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}_{\text{near}}^{\text{smooth}})$. Then, as described above, we assume the random variable B_i is distributed with expectation $\left[\mathbf{b}_{\text{near}}^{\text{smooth}}\right]_i$ and standard deviation $\sigma = C \mid (\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}) - (\mathbf{s}_{\text{current}}, \mathbf{a}) \mid$. Thus the probability that $[\mathbf{b}_{\text{goal}}]_i - \boldsymbol{\tau}_i < B_i < [\mathbf{b}_{\text{goal}}]_i + \boldsymbol{\tau}_i$ is the area under the appropriately transformed normal distribution curve in Figure 8.8, and can be computed as

$$P_{\text{succ}}^i(\mathbf{a}, \mathbf{s}_{\text{current}}, [\mathbf{b}_{\text{goal}}]_i, \boldsymbol{\tau}_i, \mathbf{E})$$

$$= \text{erf}\left(\frac{[\mathbf{b}_{\text{goal}}]_i + \boldsymbol{\tau}_i - \left[\mathbf{b}_{\text{near}}^{\text{smooth}}\right]_i}{\sigma}\right) - \text{erf}\left(\frac{[\mathbf{b}_{\text{goal}}]_i - \boldsymbol{\tau}_i - \left[\mathbf{b}_{\text{near}}^{\text{smooth}}\right]_i}{\sigma}\right) \tag{8.6}$$

Where $\sigma = C \mid (\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}) - (\mathbf{s}_{\text{current}}, \mathbf{a}) \mid$, and $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}_{\text{near}}^{\text{smooth}})$ is the nearest neighbour to $(\mathbf{s}_{\text{current}}, \mathbf{a})$ in the state and action components. The erf function is defined by

$$\text{erf}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2} dt \tag{8.7}$$

We have computed the chance that the ith component is successful, P_{succ}^i. Now, under the independence, assumption we can calculate the actual probability of success:

$$P_{\text{succ}}(\mathbf{a}, \mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}}, \boldsymbol{\tau}, \mathbf{E}) = \prod_{i=1}^{Dim(B)} P_{\text{succ}}^i(\mathbf{a}, \mathbf{s}_{\text{current}}, [\mathbf{b}_{\text{goal}}]_i, \boldsymbol{\tau}_i, \mathbf{E}) \tag{8.8}$$

To illustrate the use of the heuristic, imagine that **State**, **Action** and **Behaviour** are all 1-d spaces, and that we are in state $\mathbf{s}_{\text{current}} = 3.3$ and the desired $\mathbf{b}_{\text{goal}}$ is 11. Assume further that the only previous experience is that when $\mathbf{s}_1 = 3$ and $\mathbf{a}_1 = 7$ then $\mathbf{b}_1 = 10$. The use of the heuristic,

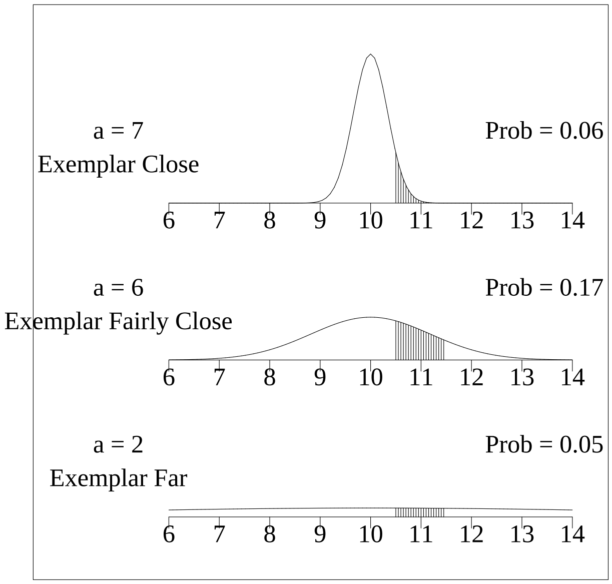

Figure 8.9

Choosing an action which is likely to cause behaviour in the range 10.5 to 11.5. Top: Action very close to earlier exemplar in which behaviour was 10. Middle: Action fairly close. Bottom: Action very different.

with a tolerance of 0.5, is demonstrated in Figure 8.9. It compares three possible actions and chooses one which is fairly near to 7, so that the behaviour will be fairly near to 10 (and thus might be close to the goal, 11), but not so near to 7 that the behaviour will surely be extremely close to 10.

We are now also able to inspect how the heuristic behaves for the control choice problem posed earlier in this section by Figure 8.1. The graph in Figure 8.10 shows how the probability of success varies with different candidate actions. The constant C is again chosen as 1. It shows that an action very close to the partial-inversion-recommended action is favoured ($\mathbf{a}_{\mathbf{raw}} = 7.5$). This is reasonable, because the *state* is fairly different from any state yet experienced, and so by repeating the same action, there is not a loss of diversity of experience. To show what would happen if the state *were* similar, I move the current state shown in the original example of Figure 8.1 back so it is very close to the exemplar $(2.7, 7.5 \rightarrow 5.1)$. I make the current state 2.9. The graph of the new probability of success, in Figure 8.11, indicates that the candidate action of 7.5 is now regarded very poorly. This is because it is so close to an earlier experience which did not achieve the required behaviour.

This new, probabilistic, control choice algorithm is summarized in Table 8.2. In future it will be called the *SAB* action chooser. Before performing empirical tests, let us consider some of its expected benefits.

- As we shall see later, the heuristic is not sensitive to the choice of C (the proportionality constant between nearest neighbour distance and standard deviation). The ranking between good and bad candidate actions is changed very little as C varies.

- Initially it is likely that for many states the only previous experience will be negative. Then

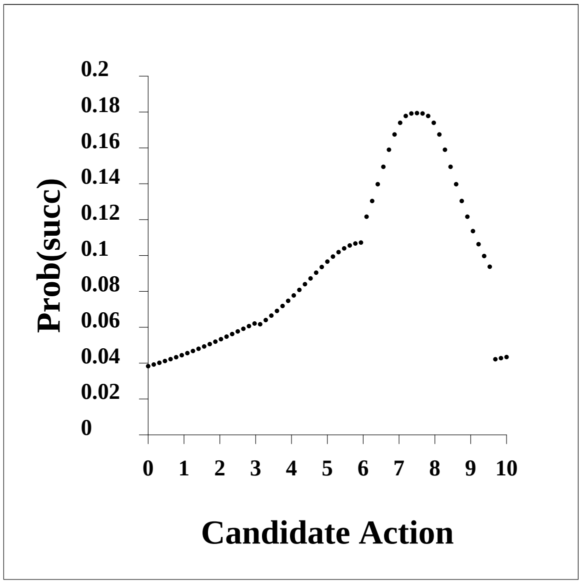

Figure 8.10

The probability of success heuristic judging the candidate actions for the situation depicted in Figure 8.1. The heuristic has discontinuities between nearest neighbour regions. For example, the state-action pair $(3.4, 9.6)$ has a different nearest neighbour than $(3.4, 9.2)$, resulting in a very different estimate.

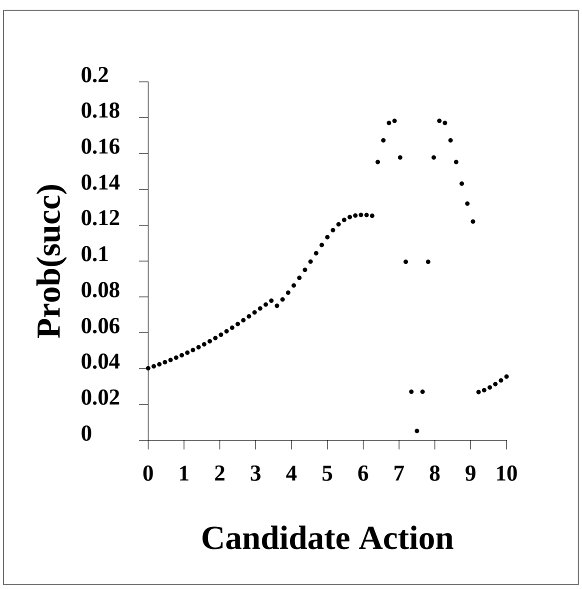

Figure 8.11

The probability of success heuristic judging the candidate actions for an altered version of the situation depicted in Figure 8.1. The alteration is that the current state is 2.9 instead of 3.4.

Algorithm:	*SAB* action chooser
Input:	$\mathbf{s}_{\text{current}}$, a *Dim*(**State**)-dimensional vector of real numbers $\mathbf{b}_{\text{goal}}$, a *Dim*(**Behaviour**)-dimensional vector of real numbers $\mathbf{E}$, of type **exemplar-set** $\boldsymbol{\tau}$, a *Dim*(**Behaviour**)-dimensional vector of real numbers **number-cands**, of type **integer**
Output:	$\mathbf{a}_{\text{raw}}$, a *Dim*(**Action**)-dimensional vector of real numbers **probsucc**, of type **real**
Pre:	$\mathbf{E} \neq \phi$
Post:	**probsucc** $= P_{\text{succ}}(\mathbf{a}_{\text{raw}}, \mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}}, \boldsymbol{\tau}, \mathbf{E})$ There is also the informal requirement that the probability of success be relatively high.
Code:	
1.	$(\mathbf{s}_{\text{pi}}, \mathbf{a}_{\text{raw}}, \mathbf{b}_{\text{pi}}, [\mathbf{b}_{\text{pi}}]_{\text{smooth}})$:= a nearest neighbour to $(\mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}})$ in (state,behaviour) space
2.	$(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}^{\text{smooth}}_{\text{near}})$:= a nearest neighbour to $(\mathbf{s}_{\text{current}}, \mathbf{a}_{\text{raw}})$ in (state,action) space
3.	**probsucc** $:= P_{\text{succ}}(\mathbf{a}_{\text{raw}}, \mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}}, \boldsymbol{\tau}, \mathbf{E})$
4.	If $\mathbf{b}^{\text{smooth}}_{\text{near}}$ is successful with tolerance $\boldsymbol{\tau}$ then return $\mathbf{a}_{\text{raw}}$ and **probsucc**
5.	for **i** := 1 until **number-cands** do
5.1	$\mathbf{a}_i$:= randomly generated action
5.2	$\mathbf{p}_i := P_{\text{succ}}(\mathbf{a}_i, \mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}}, \boldsymbol{\tau}, \mathbf{E})$
5.3	If $\mathbf{p}_i >$ **probsucc** then
5.3.1	**probsucc** $:= \mathbf{p}_i$
5.3.2	$\mathbf{a}_{\text{raw}} := \mathbf{a}_i$
6.	return $\mathbf{a}_{\text{raw}}$ and **probsucc**

Table 8.2: The *SAB* action chooser algorithm

this heuristic favours those actions which are as far away as possible from any yet applied. This is because the probability of success for actions which are close to a previous unsatisfactory behaviour is extremely low, while actions far away have a probability of success which is merely low. Thus a wide variety of actions will be generated until some relatively promising ones are discovered.

- If the promising actions were misleading, for example in a local minimum, then upon repeated trials, the heuristic would eventually once again favour further points. This is because the vicinity will gather many exemplars, and so any candidate action in the vicinity will be close to a promising, but unsuccessful, action. Because it is close, the reliability of the prediction will outweigh the favourable closeness to the goal behaviour.

- The partial inversion action can also be included among the candidate actions. If it is ranked as best, despite not having been predicted as successful, this merely indicates that the pessimistic prediction was unreliable. There is no danger of stuck states: if wrong, then next time we will know reliably that it is unsuccessful, meaning a very low estimated probability of success.

- The candidate actions need not be generated from a uniform distribution, provided that all actions have a finite probability of being generated. A non-uniform distribution, biased in favour of the actions close to the partial-inversion-recommended action, may in some circumstances be advantageous. This is analysed in the Section 8.2.

- If the task is in any way repetitive then the system will eventually return to the same neighbourhood of state space, and require the same goal behaviour. Let us assume that on the original occasion the action chooser had been successful. On the new occasion none of this computation is needed. Partial inversion immediately provides the action, which is then validated by a nearest neighbour prediction. So, for a repetitive task, the computational burden decreases as the behaviour improves.

- If there is known noise in the function, then even with the assumption that the $\mathbf{b}_{\text{near}}^{\text{smooth}}$ component is an accurate value of the expected behaviour, there is still some variation to be anticipated if the same state-action pair were to be repeated. This could be included in the P_{succ} heuristic by defining the standard deviation as

$$K + C \mid (\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}) - (\mathbf{s}_{\text{current}}, \mathbf{a}) \mid \tag{8.9}$$

  where K is the estimated standard deviation of noise. In this investigation this possibility has not been implemented because an explicit record of local variance is not stored.

The empirical trial of the mountain car task using the SAB action chooser, 'SAB'-control, was performed in the manner of the earlier trials. The results are given in Figure 8.12. On each cycle,

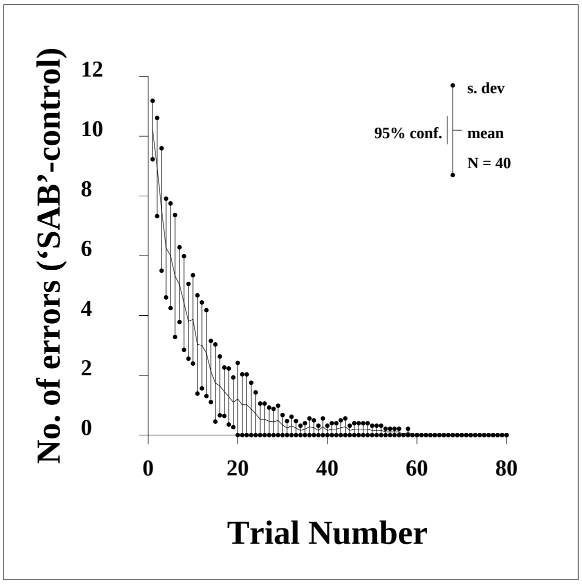

Figure 8.12

The performance against trial number for a sceptical controller, which when confronted with a predicted failure uses the best out of five randomly generated candidate actions.

the number of candidate actions was five. They were chosen randomly from a uniform distribution over the space of actions.

In all forty learning runs, this controller achieved perfect behaviour before the 80th trial. This is evidence of the power which can be obtained from the nearest neighbour generalization. Once again, there is initially wide variation in the performance, between trials 1–20. Figure 8.13 shows the distribution of exemplars after learning with 'SAB'-control. They are clustered around the solution. It is expected that this effect would be more pronounced and more important in a higher dimensional control space.

I have defined a measure for scoring candidate actions, but have not provided a method for obtaining that which has the highest score. This would require a search of all possible actions, which for a realistic space of possible actions (we are assuming a continuum of vectors), real time response does not permit. Instead, the favourite of a sample of randomly generated actions is used. A large sample can be expected to provide an action close to that which would be recommended by exhaustive search, but with the penalty of more computation per action. An important question is how does performance vary as the number of candidates used varies, and also as their distribution varies. This is difficult to analyse formally and so will be discussed later on the basis of empirical observations from the results chapter. Here we will be content to make a few informal observations.

Firstly, consider the case where the action is only ever chosen from a small number of candidates. Provided at least one candidate action is generated from a distribution which has non-zero density over the entire action space, then if one performs a series of trials all with the same $\mathbf{s}_{\mathbf{current}}$ and $\mathbf{b}_{\mathbf{goal}}$, then (subject to $\mathbf{b}_{\mathbf{goal}}$ being attainable) $\mathbf{a}_{\mathbf{raw}}$ will eventually converge to a value which achieves the tolerance. Informally the proof, which requires general continuity, would be similar to that in

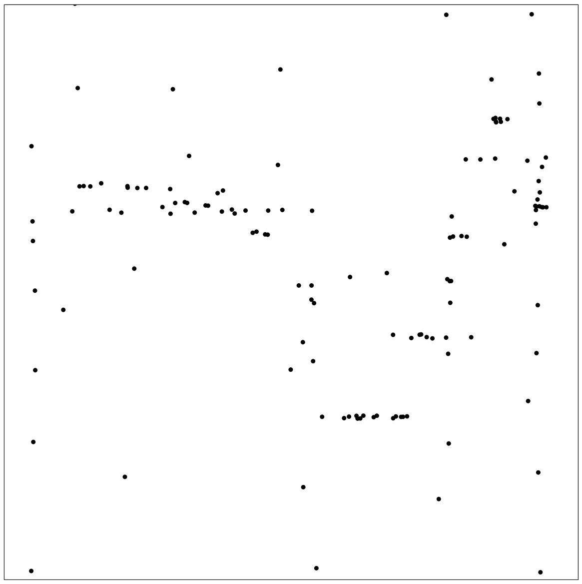

Figure 8.13

The distribution of exemplars in **State** $\times$ **Action** space after learning using the 'SAB'-controller.

Section 5.5, which assumes exemplars are drawn uniformly and randomly. Instead, the exemplar generation is biased so that, in the absence of success, they are positioned far apart from one another. This will not increase the time until the whole action space is covered with exemplars to distance $\epsilon = \mid \boldsymbol{\tau} \mid /G$, and so the time needed is, if anything less than that for a uniform distribution.

Secondly, in the limit we would not expect the learning performance to increase greatly with a large increase in the number of candidates. Consider using the best of a million candidate actions. This is likely to be very close to the best possible action to maximize the heuristic. But this heuristic is itself only an approximation, and a small change in, for example, the C parameter, or the assumed normal distribution, would change the best recommended action by more than one part in a million. The use of the heuristic is to choose actions significantly better than randomly. It is not expected to prescribe success.

At first it seems too good to be true that there exists an optimization procedure which is guaranteed to avoid local minima. The reason is that unlike most optimization procedures, this is using unbounded storage, and is thus modelling the whole function, rather than merely keeping a record of the local search state.

The previous few paragraphs have considered the case of repetitive occasions in which the same state and required behaviour occur. When we apply an action and discover it was unsuccessful we cannot immediately find out what would have happened had we tried something else—the state, $\mathbf{s}_{\mathbf{current}}$, of a dynamic manipulator cannot be expected to remain constant. The failure or success of the control choice of the previous time step may no longer be relevant. It is a result of the explicit storage of the exemplars that the performance can nevertheless be expected to improve, because information from all those occasions in the learning history which *are* currently relevant will still be available.

8.2 Control Decision Analysis

In this section I perform some elementary analysis to show, subject to some constraints on the continuity and slope of the PSTF, that the *SAB* action chooser method optimizes the chosen action very quickly. I will make some simplifying assumptions, then model the optimizing behaviour. I will then discuss the relationship between the analysis of the simplified system and the expected performance of the real action chooser.

The simplifications are

1. Assume the set of actions and the set of behaviours are both one-dimensional.
2. Assume that on each cycle we have the same state, and require the same behaviour.
3. Assume we only have a small memory: we recall only the most promising exemplar to date.
4. Assume we manage to choose a candidate action with the highest heuristic probability of success.

It will become apparent that given the pessimistic assumption 3, the optimistic assumption 4 is easy to satisfy.

The *SAB* action chooser is now posed as an optimization method, and we can analyse how quickly this optimization converges. The exemplar set $\mathbf{E}$ contains only the exemplar $(s_{\mathbf{current}}, a_i, b_i, b_i^{\mathrm{smooth}})$ which has had the most successful behaviour to date. We can dispense with the b_i^{smooth} component because, if there is only one exemplar, $b_i^{\mathrm{smooth}} = b_i$. Given a candidate action we can use the probability of success estimate of Section 8.1.

$$\begin{aligned} & P_{\mathrm{succ}}(a, s_{\mathbf{current}}, b_{\mathbf{goal}}, \tau, \{(s_i, a_i, b_i)\}) \\ = \quad & \mathrm{erf}\left(\frac{b_i + \tau - b_{\mathbf{goal}}}{Cx}\right) - \mathrm{erf}\left(\frac{b_i - \tau - b_{\mathbf{goal}}}{Cx}\right) \qquad (8.10) \\ \text{where} \quad & x = \mid a - a_i \mid \end{aligned}$$

In this equation, x is the distance of the candidate action from the best action yet seen. If b_i had been successful, then the maximum probability of success occurs with this distance set to zero and thus with a repetition of the previous action a_i. More rigorously, as $x \to 0$, then $P_{\mathrm{succ}} \to \mathrm{erf}(\infty) - \mathrm{erf}(-\infty) = 1 - 0$.

If b_i is not successful then the probability of success of repeating a_i is zero. For example, if $b_i > b_{\mathbf{goal}} + \tau$, then $P_{\mathrm{succ}} \to \mathrm{erf}(\infty) - \mathrm{erf}(\infty)$ as $x \to 0$. Thus

$$x = 0 \Rightarrow P_{\mathrm{succ}} = 0 \qquad (8.11)$$

For a distance greater than zero, the probability of success estimate will be positive, but as the distance gets very large, then the probability of success will again tend to zero because as $x \to \infty$, $P_{\text{succ}} \to \text{erf}(0) - \text{erf}(0)$. Thus

$$x = \infty \Rightarrow P_{\text{succ}} = 0 \tag{8.12}$$

The P_{succ} estimate is a smooth function of x, and so there must be an intermediate optimum distance. Writing $e_i = b_i - b_{\text{goal}}$ as the behaviour error of the current exemplar, then the optimum occurs when

$$\begin{aligned}
& \frac{\partial P_{\text{succ}}}{\partial x} = 0 \\
\Rightarrow \quad & \frac{\partial}{\partial x}\left[\text{erf}\left(\frac{e_i+\tau}{Cx}\right) - \text{erf}\left(\frac{e_i-\tau}{Cx}\right)\right] = 0 \\
\Rightarrow \quad & -\left(\frac{e_i+\tau}{Cx^2}\right)\exp\left[-\left(\frac{e_i+\tau}{Cx}\right)^2\right] + \left(\frac{e_i-\tau}{Cx^2}\right)\exp\left[-\left(\frac{e_i-\tau}{Cx}\right)^2\right] = 0 \\
\Rightarrow \quad & \ln\left(\frac{e_i+\tau}{e_i-\tau}\right) = \frac{4e_i\tau}{C^2x^2} \\
\Rightarrow \quad & x = \frac{2}{C}\sqrt{\frac{e_i\tau}{\ln\left(\frac{e_i+\tau}{e_i-\tau}\right)}}
\end{aligned} \tag{8.13}$$

So the ideal distance, or *step size*, at which to try the next action can be computed. This recommended step size is a function of the current error in observed behaviour, and of the tolerance. If the error is large with respect to the tolerance, then

$$\ln\left(\frac{e_i+\tau}{e_i-\tau}\right) = \ln\left(1 + \frac{2\tau}{e_i-\tau}\right) \approx \frac{2\tau}{e_i-\tau} \approx \frac{2\tau}{e_i} \tag{8.14}$$

Thus, using Equation 8.13, the ideal distance is

$$x \approx \frac{2}{C}\sqrt{\frac{{e_i}^2\tau}{2\tau}} = \frac{\sqrt{2} \mid e_i \mid}{C} \tag{8.15}$$

Thus, if the current error is large compared with the tolerance, the ideal optimization step is proportional to the current error. At each cycle there will be two possible actions to try next, one distance x to the left of a_i and one distance x to the right. The actual action tried is selected at random. If there is improvement then the current best action is updated, else it remains the same. If either $a_i - x$ or $a_i + x$ leads to improved behaviour then within an expected two time steps, improved behaviour will be obtained.

If neither $a_i - x$ or $a_i + x$ give an improvement then either (i) the algorithm has used too large a step size or (ii) the algorithm is in a local minimum. This simplified *SAB* action chooser would then never progress. We will discuss later the behaviour of the real algorithm in this circumstance.

If, however, progress is made, how quick is it? It would be useful to derive the expected increase in accuracy per step. Unfortunately, to obtain this, a further assumption is needed, that the PSTF has locally a strictly positive (or strictly negative) slope, bounded below by $G_{\mathbf{min}} > 0$ and above by $G_{\mathbf{max}} > 0$.

$$\forall a_1 > a_2 \quad (a_1 - a_2)G_{\mathbf{min}} < b_1 - b_2 < (a_1 - a_2)G_{\mathbf{max}} \tag{8.16}$$

where $b_1 = \mathrm{PSTF}(a_1)$ and $b_2 = \mathrm{PSTF}(a_2)$. Assume without loss of generality that b_i, the behaviour of the most promising exemplar, is less than $b_{\mathbf{goal}}$. Then, unless one of the above problems occur, the addition of the computed step length to a_i will result in a chosen action that gives improved behaviour. Using the approximation of Equation 8.15, we have

$$a_{i+1} = a_i + \frac{\sqrt{2}}{C} \mid e_i \mid \tag{8.17}$$

But by the assumption of bounded slope, we can deduce that

$$b_i + (a_{i+1} - a_i)G_{\mathbf{min}} < b_{i+1} < b_i + (a_{i+1} - a_i)G_{\mathbf{max}} \tag{8.18}$$

where $b_{i+1} = \mathrm{PSTF}(a_{i+1})$. Therefore, from Equation 8.17,

$$b_i + G_{\mathbf{min}}\frac{\sqrt{2}}{C} \mid e_i \mid < b_{i+1} < b_i + G_{\mathbf{max}}\frac{\sqrt{2}}{C} \mid e_i \mid \tag{8.19}$$

We are interested in $e_{i+1} = b_{i+1} - b_{\mathbf{goal}}$, the error of the next time step. It is not known whether it will be positive or negative, but in either case its magnitude is bounded above. In the first case, where $b_{i+1} \leq b_{\mathbf{goal}}$,

$$\mid e_{i+1} \mid = b_{\mathbf{goal}} - b_{i+1} < b_{\mathbf{goal}} - b_i - G_{\mathbf{min}}\frac{\sqrt{2}}{C}(b_{\mathbf{goal}} - b_i) = (1 - G_{\mathbf{min}}\frac{\sqrt{2}}{C}) \mid e_i \mid \tag{8.20}$$

In the second case, where $b_{i+1} > b_{\mathbf{goal}}$,

$$\mid e_{i+1} \mid = b_{i+1} - b_{\mathbf{goal}} < b_i - b_{\mathbf{goal}} + G_{\mathbf{max}}\frac{\sqrt{2}}{C}(b_{\mathbf{goal}} - b_i) = (G_{\mathbf{max}}\frac{\sqrt{2}}{C} - 1) \mid e_i \mid \tag{8.21}$$

In either case $\mid e_{i+1} \mid < \mid Ke_i \mid$ where $K = \max\{G_{\mathbf{max}}\frac{\sqrt{2}}{C} - 1, 1 - G_{\mathbf{min}}\frac{\sqrt{2}}{C}\}$. The error $\mid e_{i+1} \mid$ is a constant factor less than $\mid e_i \mid$, provided that both $(G_{\mathbf{max}}\frac{\sqrt{2}}{C} - 1) < 1$ and $(1 - G_{\mathbf{min}}\frac{\sqrt{2}}{C}) < 1$. This implies that C must be greater than $G_{\mathbf{max}}/\sqrt{2}$. This, in turn agrees with the informal statement of the previous section that the value of C should reflect the expected slope of the function. If C is suitable then the error decreases exponentially with the number of steps of optimization.

In summary, if the function is well behaved in the region of interest, then if the initial error is large with respect to the tolerance, then each time progress is made

$$\mid e_{i+1} \mid < K \mid e_i \mid \tag{8.22}$$

where K is defined above in terms of the function's slope and the system parameter C. If the choice whether to increase or decrease the current best known action is made randomly then progress can

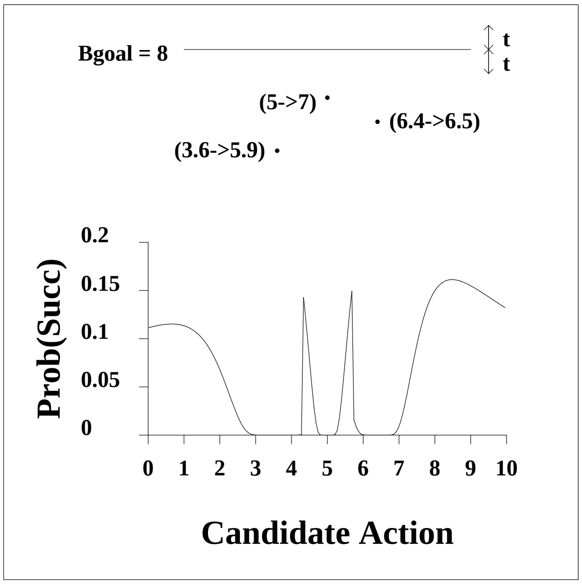

Figure 8.14

Three state-free exemplars have been experienced, none achieving within tolerance 0.5 of the target behaviour of 8. The action component of each exemplar is aligned with the action axis of the graph. There are now four local maxima in the P_{succ} estimate.

be expected once every two time steps. Thus after a number of time steps logarithmic in e_0/τ (the ratio between the initial error and the required error) the behaviour is expected to be successful.

Now let us consider what would happen to the algorithm if it *does* remember and reuse all its experience. It will be shown that the optimization can still achieve an exponential decrease in the error, even when the forgetful method would have stopped improving. Improvement would have stopped if the two actions $a_{f1} = a_i - \frac{\sqrt{2}}{C}e_i$ and $a_{f2} = a_i + \frac{\sqrt{2}}{C}e_i$ had both produced behaviour inferior to that of a_i. Let us consider the case where three exemplars are available for use by the P_{succ} estimate. An illustration is given in in Figure 8.14, in which the initial action $(5 \to 7)$ was not successful, but the two exemplars at the computed ideal distance were also unsuccessful, and both worse. After this situation, the probability of success for all other candidate actions is plotted in the same figure. As is exemplified here, there are precisely four local maxima in the P_{succ} estimate. These occur at the following points:

$$a_{\text{c}1} = a_{f1} - \frac{\sqrt{2}}{C}\sqrt{\frac{(b_{f1} - b_{\text{goal}})\tau}{\ln\left(\frac{b_{f1} - b_{\text{goal}} + \tau}{b_{f1} - b_{\text{goal}} - \tau}\right)}}$$

$$a_{\text{c}2} = (a_{f1} + a_i)/2$$

$$a_{\text{c}3} = (a_{f2} + a_i)/2$$

$$a_{c4} = a_{f2} + \frac{\sqrt{2}}{C}\sqrt{\frac{(b_{f2} - b_{\text{goal}})\tau}{\ln\left(\frac{b_{f2} - b_{\text{goal}} + \tau}{b_{f2} - b_{\text{goal}} - \tau}\right)}}$$

The best of these four actions will be attempted. Candidates 2 or 3 are chosen if the P_{succ} estimate suspects that there is some successful behaviour between a_i and either a_{f1} or a_{f2} respectively. Thus, if the step size had been too big, the step size would be reduced to half its previous value. If the resulting behaviour looked promising, search would continue, either as in the original linear search, or if step lengths were continually overestimated, then as a binary chop. In either case the error is reduced by a constant multiplicative factor at each step.

Candidates 1 or 4 are chosen if the behaviour at a_i is so unpromising that is is preferable to be a large distance from the inferior a_{f1} or a_{f2} to remaining in the presence of a_i. This can be viewed as an escape from a local minima.

Thus, for one dimension, the behaviour of this optimization technique is robust: it does not get stuck in local minima, but when the function is well behaved, the number of bits of accuracy increases linearly. The actual error thus decreases exponentially. Unfortunately, this has assumed a dimensionality of only 1 and has used the simplification that the chosen action is always the most promising according to the P_{succ} estimate. I shall now give informal explanations as to why the desirable behaviour can be expected to scale up.

- **Many Dimensions.** The action space is not as large as the control space, but it can be expected to be up to six-dimensional for a fully orientable robot arm. With more than one dimension there is a continuum of candidate actions at the ideal distance, instead of merely the two in the previous analysis. However, if the function is well behaved in the sense of Equation 8.16 then half the candidate actions can be expected to improve behaviour and half to degrade it. The expected number of actions needed to improve behaviour is still only two. For a high dimensional space of behaviours the expected decrease in error is, however, less than for one dimension.

- **Maximizing the P_{succ} estimate is too expensive.** This analysis has assumed that of those points which maximize the P_{succ} estimate, one was chosen as the next action. In practice, this can only be approximated. The approximation consists of the best of a number of randomly chosen actions. The distribution is biased in favour of actions which have the computed ideal distance. Thus it can be hoped that the best action approximates one of the actions with a relatively high probability of success and so in turn it can be hoped that the local search may still be linear in the number of bits of accuracy—this hope will be tested in the results chapter.

The reasoning of this section has lead to the following heuristic for choosing candidate actions. Some are generated locally, and some globally. The global actions are generated from a uniform distribution over the whole action space. Equation 8.15 indicated that a good place to look for candidates was within a distance proportional to the current error. There are two system parameters for the random candidate generator: P_{loc}, the probability of choosing a candidate action from the local distribution, and C_{loc}, the constant of proportionality used when an action *is* generated locally.

8.3 Learning One's Own Strength

The *SAB* action chooser does not get stuck in local minima, and thus, on repeated occasions, will eventually search the entire action space to any granularity until the behaviour is found. In some respects this is an advantage, but there is also a significant drawback. If the behaviour is not attainable then an enormous variety of actions will be tried unnecessarily. Each time one of these hopeless attempts is made, it is not guaranteed that the behaviour which is produced is as close to the goal as it can get. This is precisely because the *SAB* action chooser is avoiding local minima.

This section does not offer a general solution to this difficult problem. Instead it is suggested that for many tasks one of the two following proposals can be used.

- **Extra domain knowledge.** The designer can use their own knowledge of the system to ensure that all behaviours which will be requested are indeed attainable.
- **Heuristic estimate of attainability.** The controller can estimate from the world model whether it believes the proposed behaviour is possible.

One primitive mechanism for achieving the second solution is considered in Appendix C. If the behaviours have a *direction* then given a requested behaviour direction and a behaviour magnitude, the heuristic computes an estimated probability that the magnitude can be achieved. The higher level controller can use the heuristic to decide whether to request the behaviour.

This is related to a difficult problem of statistical decision theory: the "two armed bandit", reviewed in [Kaelbling, 1990a]. If one is told one may have a large number of goes on a two armed bandit in which each arm pays out with different probability, what should be done to maximize the expected payoff? Continue pulling the arm which first pays out? Perform three hundred trials of each and then always pull the one which performs best? The current consideration is similar. At what point should the *SAB*-control choice mechanism resign itself to never being able to supply the requested behaviour, and instead always supply the best known? If it resigns too early then it might miss trying an action which would have actually achieved the behaviour. Thus learning will have become "stuck" at an error. If it resigns too late then during the search the experimental actions are likely to degrade task performance far more severely than if the best known action were used.

Kaelbling, in [Kaelbling, 1990b], deals with this problem for the extreme case in which the controller has no world model at all, and is instead trying to maximize a *payoff* signal. Her work suggests some algorithms which are effective for worlds specified by binary-valued input vectors, and which empirically tend to find high payoffs. Although the learning algorithms are computationally expensive and the assumption is of a small finite space of actions, it would certainly be interesting to apply similar techniques to this problem.

Chapter 9

Learning to Perform a Task

This chapter discusses a scheme by which SAB world models and the SAB action chooser can be integrated into a variety of tasks. A naive control method called "ice puck control" is introduced. It is then explained (i) how low level tasks can be controlled directly by the SAB action chooser and (ii) how some medium level tasks can be indirectly controlled using "ice puck control" and then (iii) how a large compound task can be modularized into a hierarchy of learning subtasks.

9.1 Where do Goal Behaviours come from?

The previous chapters have introduced a robust and efficient method to quickly learn the dynamics of a possibly changing world. They have also shown how to use this model: when a goal behaviour is required, a raw action signal is obtained which will either achieve the behaviour or else gain useful information.

Let us reconsider Step 2 in the original learning controller scheme, shown again in Table 9.1. There is no general way to achieve this step because there is no generic task. In this chapter I will propose that for many of the common tasks which we might expect a robot to perform, the generation of goal behaviours is easy.

The specification of robotic tasks is similar to the specification of software systems. It occurs, at least initially, in an abstract and rather informal form, perhaps as a piece of text, or simply an idea. The design of a method to execute the task is again similar to the design of a program: it is broken up into smaller subcomponents, with correspondingly less abstraction. Different programs often share some features, such as graphical data display, and similarly many robot tasks have common subcomponents. Some of these were mentioned in Chapter 2. They include components such as sensing, trajectory following, and holding still. At the lower levels, the design of software and robot task design differ: at an intermediate level of abstraction, the human programmer can write down the program, and then the *compiler* converts these intermediate notations to the entirely

1.	Observe current perceived state $\mathbf{s}_{\text{current}}$
2.	**Receive task specific goal behaviour $\mathbf{b}_{\text{goal}}$ from higher level of control. The requested behaviour might depend partially on $\mathbf{s}_{\text{current}}$.**
3.	Access the current world model to obtain a raw action $\mathbf{a}_{\text{raw}}$ which is predicted to be likely to achieve $\mathbf{b}_{\text{goal}}$ acting in the current state.
4.	Apply action $\mathbf{a}_{\text{raw}}$.
5.	Observe actual behaviour $\mathbf{b}_{\text{actual}}$.
6.	Update the world model with the information that $(\mathbf{s}_{\text{current}}, \mathbf{a}_{\text{raw}}) \rightarrow \mathbf{b}_{\text{actual}}$.

Table 9.1: The *SAB* Control Cycle

concrete form of executable machine code. The designer of a robotic system has to go down to the concrete level manually. This work has been investigating the possibility of automating the very bottom level, world modelling, but I wish to show here that it can make higher levels easier as well. To motivate this, I will describe how to connect the *SAB* control cycle to various tasks. I will demonstrate that many aspects of an abstract task definition can remain abstract.

The chapter will begin with a diversion—the control of an extremely elementary dynamic system.

9.2 Controlling an Ice Puck is Easy

The argument that is it easy to control an ice puck makes a substantial jump in the train of thought of the dissertation. The reason for the effort expended here will become clear in the next section, which will coalesce the ideas of *SAB* control choice, and ice puck control.

9.2.1 An Idealized Ice puck

Figure 9.1 shows an ice puck: a small one kilogram mass sitting on a frictionless surface. It has two forces acting on it, F_1 and F_2, F_1 shown acting horizontally and F_2 vertically on the diagram. Because it is idealized, its equations of motion are very simple:

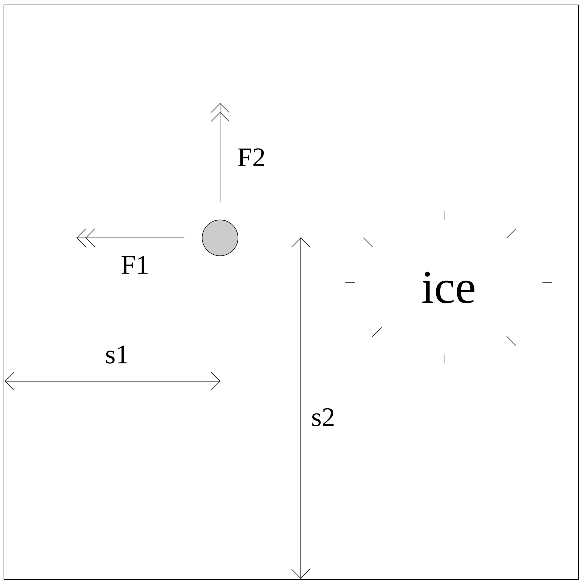

Figure 9.1

The ice puck world. The puck moves across an idealized frictionless surface, controlled by two orthogonal force vectors.

$$\begin{aligned} a_1(t) &= F_1(t)/M \\ a_2(t) &= F_2(t)/M \end{aligned} \tag{9.1}$$

where $M = 1$, a_1 is the horizontal acceleration and a_2 is the vertical acceleration. Accelerations and forces are thus equivalent, so ice puck control is merely the application of accelerations. Let us write the state of the ice puck as $\mathbf{s}(t) = (s_1(t), v_1(t), s_2(t), v_2(t))$, where s_i is the distance from the origin in the ith direction, and v_i is the ith component of velocity.

From the definitions of position, velocity and acceleration, it is simple to write down the behaviour of the puck at any future time as a function of all the future accelerations. We can do this for each component independently.

$$\begin{aligned} v_i(T) &= v_i(0) + \int_0^T a_i(t)dt \\ s_i(T) &= s_i(0) + \int_0^T v_i(t)dt \end{aligned} \tag{9.2}$$

The ice puck is controlled at discrete time steps of length h. During each time step a constant acceleration is supplied. Write $s_i[n]$ as the position at the start of the nth time step, $v_i[n]$ as the velocity, and $a_i[n]$ as the acceleration applied during the nth time step. Then

$$\begin{aligned} v_i[n] &= v_i[n-1] + ha_i[n-1] \\ &= v_i[0] + h\sum_{j=0}^{n-1} a_i[j] \end{aligned} \tag{9.3}$$

and

$$
\begin{aligned}
s_i[n] &= s_i[n-1] + \int_0^h (v_i[n-1] + a_i[n-1]\,t)\,dt \\
&= s_i[n-1] + hv_i[n-1] + \tfrac{1}{2}h^2 a_i[n-1] \\
&= s_i[0] + h\sum_{j=0}^{n-1}\left(v_i[j] + \frac{1}{2}ha_i[j]\right) \\
&= s_i[0] + h\sum_{j=0}^{n-1}\left(v_i[0] + h\sum_{k=0}^{j-1}a_i[k] + \frac{1}{2}ha_i[j]\right) \\
&= s_i[0] + nhv_i[0] + h^2\sum_{j=0}^{n-1}(n-j-\frac{1}{2})a_i[j]
\end{aligned}
\tag{9.4}
$$

After n steps of constant acceleration α, the state of the ith variable is

$$
\begin{aligned}
v_i[n] &= v_i[0] + nh\alpha \\
s_i[n] &= s_i[0] + nhv_i[0] + \tfrac{1}{2}\alpha n^2 h^2
\end{aligned}
\tag{9.5}
$$

It is thus not difficult to predict future behaviour of the idealized ice puck. It is also elementary algebra to control the ice puck in order to make it reach a goal position and goal velocity at a goal time step, t_g. Call this combination of objectives a *puck goal.* All that is necessary is, for each direction of the puck world, to invent a trajectory which will achieve the appropriate component of the puck goal.

One such simple trajectory is bang-bang control, shown in Figure 9.2. It consists of applying a constant acceleration α for N time steps and then acceleration $-\alpha$ for $t_g - N$ time steps. There are two values to choose, and two constraints to meet: the values of $s_i[t_g]$ and $v_i[t_g]$. Thus the values can be derived by elementary algebra. Interested readers may inspect the results at the end of this section.

Instead of generating the trajectory and then tracking it, the controller computes the initial recommended acceleration, applies it, and then discards it. On the next time step it implements the same procedure. This is possible, because the trajectory is cheap to compute: one square root and approximately ten multiplications are required per puck component. In this manner the trajectory can be automatically adjusted to take account of earlier tracking errors.

There are many other ways of choosing accelerations. For example, one could find accelerations α and β, such that applying α for time $\frac{1}{2}t_g$ followed by β for time $\frac{1}{2}t_g$ achieves the desired state. The bang-bang method has two particularly desirable properties.

- The maximum magnitude of any acceleration on the trajectory is as low as possible. For example, imagine that there was a constraint that no component of the acceleration applied

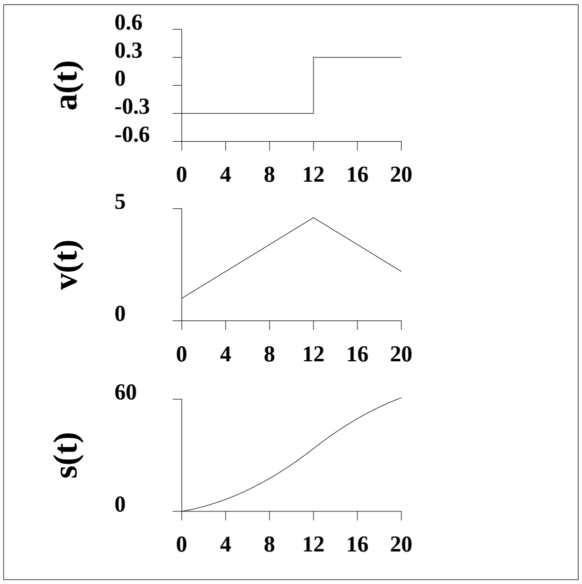

Figure 9.2

The puck starts in state $s = 0, v = 1$ and wishes to achieve state $s = 60.8, v = 2.2$ in twenty time steps of one second. The solution is acceleration $\alpha = 0.3$ for 12 time steps and acceleration $-\alpha = -0.3$ for eight steps.

to the puck could have magnitude greater than $A_{\max}$. If the controller receives a puck goal, and the bang-bang computation produces an acceleration which is too large, then no other trajectory would have managed either.

- The controller dynamically recomputes the trajectory at each time step, and this computation is affordable. If the computed acceleration is applied accurately then the next dynamically computed trajectory will match the first. This property does not generally hold for other kinds of trajectory.

There is also a potential disadvantage. The sudden switch in requested accelerations would, in a non-ideal world, be hard to achieve and difficult to control. In practice, many robot trajectory generators are designed to produce paths which in terms of position are smooth to five levels of differentiation.

The conclusion of this section is that given a goal state and time, it is easy to achieve the goal for an idealized ice puck. In Sections 9.3 and 9.4 I explain why the SAB model of the world permits other dynamic systems to be controlled in the same manner.

9.2.2 Generating the Bang-Bang Trajectory

All control variables are treated independently. Let the current position and velocity be (s_0, v_0), and the goal be (s_g, v_g) after t_g time steps. This analysis finds the values N, α and β such that when acceleration α is performed for N time steps and acceleration β for $N - t_g$ time steps, the goal state will be obtained. These values will be chosen to make α and β close in magnitude while ensuring that N is integral.

First we find X, possibly non-integral, and γ, such that accelerating with γ for Xh seconds and then accelerating with $-\gamma$ for $(t_g - X)h$ seconds will achieve the goal. Such values would satisfy the two equations

$$v_g = v_0 + h\gamma(2X - t_g) \tag{9.6}$$

$$s_g = s_0 + hv_0t_g + \frac{1}{2}h^2\gamma(-t_g^2 + 4Xt_g - 2X^2) \tag{9.7}$$

Thus, γ in terms of X is, from Equation 9.6, $\gamma = (v_g - v_0)/(h(2X - t_g))$. Substituting into Equation 9.7 gives

$$(2X - t_g)(s_g - s_0 - hv_0t_g) = \frac{1}{2}h(v_g - v_0)(-t_g^2 + 4t_gX - 2X^2) \tag{9.8}$$

This quadratic equation is solved by

$$X = t_g - \frac{R}{h} \pm \frac{1}{2h}\sqrt{4R^2 - 4Rht_g + 2h^2{t_g}^2} \tag{9.9}$$

where

$$R = \frac{s_g - s_0 - hv_0t_g}{v_g - v_0} \tag{9.10}$$

N is then chosen as the closest integer to X, and then the accelerations α and β are computed to solve the goal position, time and velocity. The solution for α, which is then used as the recommended acceleration, is:

$$\alpha = \frac{2(s_g - s_0)}{h^2Nt_g} - \frac{v_0(t_g + N) + v_g(t_g - N)}{hNt_g} \tag{9.11}$$

9.3 Low Abstraction Tasks

A "low abstraction task" is defined here to be a task which has been specified to sufficient detail that the controller is able to compute the goal behaviour directly from the task description. This level is the bottom level that the system designer must consider—this is a considerable improvement on the conventional bottom level of abstraction.

9.3.1 Achieving a Fixed Behaviour

The Mountain Car domain provided an example of this kind of task in Chapter 8, where the goal was to continue moving at a constant velocity. The domain was constructed so that the velocity was the behaviour being modelled. There is no difficulty integrating the *SAB* action chooser into such a task. At each control cycle, the requested behaviour remains fixed. This continues until some higher level of control, perhaps a human operator, is satisfied. The success of the task can be judged as either the sum of the deviations from the goal behaviour, or the number of cycles which were in error.

9.3.2 Following a Perceived Trajectory

For a dynamic system, in which the Perceived State Transition Function is being learned, the behaviour vectors are perceived accelerations. The task can be controlled as if accelerations were being applied directly to the perceived state. In fact they will be applied indirectly, by requesting the *SAB* action chooser to try to achieve them.

The trajectory task can thus pretend that it is an ice puck controller. Put more formally, it can imagine it is an entirely linear and decoupled system. On each cycle, and for each perceived position component, it considers the current position and velocity. It also considers the desired position and velocity specified by the trajectory for the next time step. Can this one-time-step puck goal be achieved? If there is a solution to the one-step puck goal, then the component of the goal acceleration has been obtained. It might, however, not be possible because the current velocity might not allow positions to be matched on the next cycle if velocities are also to be matched. In that case the puck goal of two steps in the future is used. The required acceleration is then handed to the *SAB* action chooser.

If the current position and velocity were to mismatch badly with those desired then the requested accelerations would be unattainably large. The action chooser would on repetitions of the same situation try a wide variety of actions. This is because all previous occurrences would have failed, and thus the highest probability of success would be achieved by trying something as different as possible. The large variety of low probability actions might degrade performance, and could not be expected to bring the state any closer to the trajectory. In this case it might be preferable that the *SAB* action chooser always produced accelerations as close as it could to those requested. Empirically, this problem was not observed in the experiments detailed in Chapter 10. However, if it were likely to be a problem it could be avoided by use of domain knowledge to specify limits on accelerations. A request would then never be made to the *SAB* action chooser for any larger acceleration. Alternatively, the maximum accelerations could be estimated by a heuristic such as that described in Appendix C.

9.4 Middle Abstraction Tasks

Some middle abstraction tasks can be immediately transformed into low abstraction tasks. These include the "hold still" task, the "move here" task and, at a slightly more abstract level, the "move here quickly" task.

9.4.1 Hold Still

This can be implemented as a request to follow the stationary trajectory. The position at which to remain can be specified as the position when the "hold still" task was first activated.

9.4.2 Move Here

In the "move here" task, the position, velocity and time to achieve the goal are specified. This is then an ice puck goal of Section 9.2. The correct accelerations can be derived accordingly.

9.4.3 Move Here Quickly

Commonly, one does not wish to specify the time the robot should take to get to a particular configuration, but instead would prefer it to arrive reasonably quickly. This could be done by simply guessing a time or having a default time for movements, but this could lead to inefficiencies, either because the robot will have moved unnecessarily slowly, or because it tried to achieve too fast a trajectory and overshot the target, because of inadequate deceleration. A naive controller which consistently made this mistake could oscillate unstably if it persisted in trying to get to the goal at a speed which exceeded the capabilities of the actuators.

Instead, the controller uses its knowledge about its strength. The algorithm performs a binary chop to find a suitable movement time. When a movement time is considered, the ice puck controller is consulted to determine the accelerations which would be necessary to achieve the movement time. Then the *SAB* world model is consulted to determine the probability that the necessary accelerations can be attained. The shortest time with a sufficiently high probability of success is used. This decision could be formalized if it were provided with (i) the expected cost of trying a trajectory which is not attainable and (ii) the expected payoff from using short times.

Experiments with such a controller are documented in Section 10.4. The advantage of the autonomous time choice controller is that it has some abstractness: the detail of how long the movement should take is computed automatically. Its performance on repeated trials can be expected to find a trajectory which is not unacceptably slow. However, it would certainly be unlikely to find the theoretically optimal trajectory. If such a trajectory were required then a more computationally expensive controller would be necessary.

9.5 Compound Tasks

Many interesting tasks have several components. In this section I discuss, by means of an example, how *SAB* learning is integrated with multiple-component tasks. The example is volleying a ball. The physical situation is depicted in Figure 9.3.

The ball on the right of the diagram is fired towards the two-jointed arm on the left of the diagram. The arm must hit the ball so that it lands in the bucket. The arm and ball are both acting under gravity. The controller can sense the following data (in perceived coordinates):

- The position and velocity of the hand.

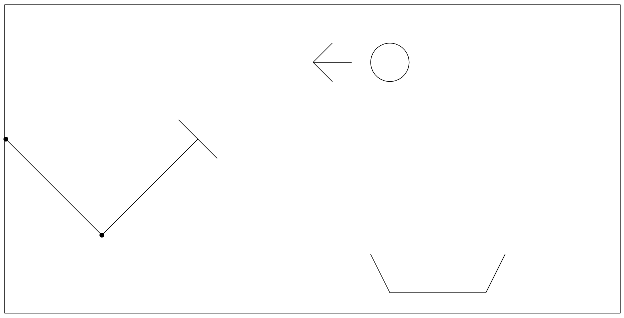

Figure 9.3

The volley task: physical situation.

- The position and velocity of the ball on the following occasions: (i) when it is fired, (ii) when it is hit and (iii) when it lands.
- The position of the bucket.

The control consists of the torques sent to the shoulder and elbow joint of the arm. The bat is fixed at right-angles to the forearm. The details of the simulation parameters are given in the results chapter.

One approach to using *SAB* learning to control this task would be to specify it as a nine-dimensional state space. The state vector has four components for the arm state, four components for the ball state and one component for the position of the bucket. The action is the two-component vector of the joint torques supplied, and the behaviour is the change in arm state and in ball state.

The PSTF of such a system would have the strange property that four components of the state, namely those of the ball state, are almost never affected by the action. If the *SAB*-controller were given the goal system-state of "anywhere" for the arm and "at rest, y = 0, x = bucket position" for the ball, then generally there would be absolutely no action which could achieve it and very occasionally all actions would achieve it. Thus local control, specified by local goal behaviours, to send the ball to the bucket would be entirely useless. Instead, there would have to be some global level to the controller, much as was used for ice puck goals, to derive the final ball position. Such a global level could, for example, search a large number of time steps ahead to find if any of the actions currently available could, with adequate subsequent actions, achieve the final goal. A more computationally tractable alternative would be a dynamic programming approach. In principle, if the PSTF is learned to sufficient accuracy, then the controller will succeed at the task. A large amount of data would be needed because the controller would not know in advance which parts of the control space would be important. In the absence of other domain knowledge, no data would be useful until the goal had been achieved at least once—for the volley problem this would almost certainly take a very long time.

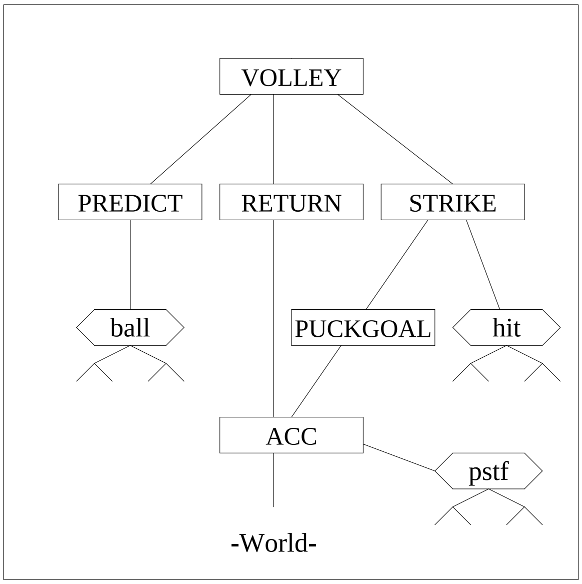

Figure 9.4

The volley task decomposed into a small hierarchy of subtasks.

The problem with the formulation as a nine-dimensional state space system was that the structure inherent in the problem was flattened away. It is clear that a human given the volley task would make some abstract, symbolic organization of the task into components. In particular they would notice that the final behaviour of the ball depends on whether it contacts the bat, and that if it does contact the bat, the bat's behaviour at the time will affect the ball's subsequent behaviour.

If a robot is to achieve the task, it would be an interesting exercise to let it try to deduce this relationship from the observed data. Some recent work by [Sammut and Michie, 1989] investigates discovering such qualitative descriptions of dynamic systems from raw behavioural data. However, for a large class of cases, such as this volley example, domain knowledge such as this is easy for a human designer to express. It is similarly easy to give an abstract, qualitative description of how to perform the entire task.

> Predict where the ball will be when it comes into convenient hitting range. Make sure the hand is in the same position at the same time, and moving at such a speed as to cause the ball to land in the bucket. After the ball and bat have collided, move the arm to a waiting position.

This abstract volley task can be separated into three slightly less abstract subtasks, indicated in Figure 9.4. `Predict` estimates the ball's behaviour. `Strike` brings the bat to contact the ball at a controlled position and speed. `Return` holds the bat steady after the strike.

The `Predict` task models the perceived behaviour of the ball prior to being hit. This model is learned by a *SAB*-tree, which estimates the time until the ball arrives within range of the arm and the state of the ball at this point. The ball is defined to be in range of the arm when it reaches a distance of 1.5m from the shoulder (the stretched arm is 2m long). The *SAB*-tree is a mapping

$$\underbrace{\text{Ballstart}}_{S} \times \underbrace{()}_{A} \rightarrow \underbrace{\text{Ballhit} \times \text{Timehit}}_{B}$$

which is updated once every trial. There is no control over this aspect of the ball's behaviour so **Action**, the space of control actions, is empty.

The `Return` task simply computes a perceived acceleration to thrust the endpoint towards the stationary waiting position. This is an instantiation of the "hold still" task described in the previous section. The torques to achieve this acceleration are computed and executed by the low level `Acc` task, which is a SAB action chooser.

The `Strike` task controls the ball indirectly by means of a collision between the ball and the bat. It requires a model of the real world:

$$\begin{gathered}\underbrace{\text{Ball at Hit} \times \text{Bat position and direction}}_{S} \times \\ \underbrace{\text{Bat speed}}_{A} \longrightarrow \\ \underbrace{\text{X coordinate of Landed Ball}}_{B}\end{gathered} \tag{9.12}$$

This too can be learned using a SAB-tree. From trial to trial, the `Strike` task attempts to always position the bat to contact the oncoming ball at the bat's centre, and to have it moving in the same relative direction at impact. The speed of the bat at impact is varied. This speed affects the landing position of the ball, and so can be used to indirectly control the landing position. At the trial start the SAB action chooser is given $\mathbf{s}_{\text{current}}$ as the estimated ball state when it arrives in range and the bat position and direction during impact. The search produces a recommended speed for the bat during impact.

It should be noted that this SAB-tree, like the others, is simply a set of objective observations about the world, and its own accuracy does not depend on the performance of the subtasks which are being learned. There is no blame or credit assignment problem. For example, suppose that we believed that hitting speed S_1 at position P_1 would ensure that the ball landed at X_1, but due to a low level error the ball was volleyed with the correct speed S_1, but at the wrong position P_2. The ball then lands at X_2. The speed S_1 will not now be wrongly associated with landing at X_2: the SAB-tree will simply contain an observation that hitting with speed S_1 at the wrong position, P_2, results in a landing at X_2, and will contain no explicit prediction as to what would happen were the ball hit at the correct position.

The `PuckGoal` task guides the initial perceived state to the target impact state. This is ice puck control, again dealt with in the previous section.

9.5.1 Discussion

The volley example has demonstrated the general principle that an apparently complex dynamic control problem can be broken down using naive, qualitative reasoning. The resulting plan uses middle abstraction *SAB*-tasks which can be automated. The interesting observation is that whenever a quantitative relationship is required, *SAB*-learning can be used, and if necessary, the control choice mechanism can be used to invert and explore these other world models.

The state for the lowest-level model, the arm's Perceived State Transition Function, is the raw action for the higher level `Strike` model. This is an entirely natural way to decompose a hierarchical task. An abstract task decides it needs a certain action performed which is delegated to a lower level task. For the lower task, this then becomes the goal state. A particularly pleasing feature of a hierarchy of *SAB*-learners is that they can usefully learn simultaneously, without there being any danger of a credit assignment problem.

9.6 The Benefits of Learning

This chapter has shown how a variety of complexities of robot task can be interfaced with *SAB*-learning. It has discussed the fact that the *SAB* action chooser cannot be used for all aspects of task achievement, but instead some other means must be used to obtain the abstract plans for achieving abstract tasks. However, there are some benefits of *SAB* learning, derived from the raised level of abstraction below which task automation can take place. There are several ways this is achieved:

- It is unnecessary to work in any coordinate system other than that which the robot perceives. Thus, for example, the output of a planner to move a hand towards an observed goal is simply to thrust the observed hand position in the observed direction.

- If it is suspected that the environment has some noise, then it may be unnecessary to take it into account in the generation of the plan. Noise tolerance can be dealt with at the modelling level.

- An environment which might change, either gradually or suddenly, is again dealt with at the modelling level. This is provided that there is no qualitative change, such as part of the robot arm actually disintegrating, in which case a new abstract strategy might be needed.

- If the raw actions can supply sufficiently powerful actions (for example, large enough torques), then the middle levels of control—trajectory following or atomic movements—can be controlled with trivial "ice puck" control.

- World models can be used at multiple locations in the subtask hierarchy. Even if these world models depend upon each other, learning can take place simultaneously with no danger of errors made by one level resulting in other levels learning incorrect models.

There are, however, some extra requirements on the designer.

- It is necessary to know, for each world model, that it *can* be usefully learned as a deterministic, generally continuous function. This does not eliminate noisy functions, but it makes the assumption that the noise distribution is useless and is to be removed.

- It is necessary to supply, along with each requested behaviour, a tolerance of acceptable behaviour. However, no difficulty is foreseen should it be required that these values are generated automatically. Chapter 10 demonstrates that within a wide range, performance is not sensitive to these values.

- In the current implementation, for each *SAB*-tree, it is necessary to provide a small set of parameters. These consists of the constant C used by the P_{succ} estimate, (from Section 8.1) the expected noise level (specified indirectly by the D_{range} search width value of Section 7.2), the number of candidate actions to be considered should the partial inversion action be inadequate, and the spread of values within each dimension of the state, action and behaviour spaces (see Section 5.1). Empirically, we will see that the performance is not sensitive to these parameter choices, nor are they difficult to estimate. It would also be possible to estimate them from the data.

- The current implementation also requires that the *SAB*-tree be occasionally garbage collected (see Section 7.5). The plan must set aside computation time for this purpose. This could also be automated by performing the garbage collection incrementally, or as a background process.

The design and analysis of plans to achieve high level robotic tasks are generally not complex. They are significantly easier than the design and analysis of software systems. Thus, we may hope that robot control based on learned world models makes robot system design a cheap and easy process.

A more interesting alternative is that the planning can be automated. Many researchers have investigated the use of planning for robot tasks. An objection to some of this research has been the *micro-world* problem—that the investigations factored out too much disorder from the environment in order to be able to work at a logical, abstract, level instead of a numeric, concrete level. But it is apparent that, with a learning modelling system at the low levels of the task structure, such disorder *can* be factored out.

Thus, by learning the world models we have automated the low level part of robot planning. An analogy with biological systems would be to suggest that the *SAB* models are the equivalent of the low level, subconscious processes of the motor control system, while the task planning is equivalent to a higher level process.

Chapter 10

Experimental Results

This chapter describes the methods, results and conclusions for a variety of learning control experiments. These consist of: learning a hand-eye coordination; learning to follow a perceived trajectory of a simulated torque-controlled arm; learning movement control for non-repetitive tasks; learning to keep a ball bouncing; and learning the volley task described in Section 9.5.

10.1 AB Learning

This section details the results of an experiment performed early on during the research described in this dissertation. As a result, many features of the SAB learning system described in this dissertation are missing. However, it is an interesting experiment worth examining because it works with a real robot instead of a simulation and it uses a fairly high dimensional mapping. It is also described in [Clocksin and Moore, 1989].

The experiment learns the combined perspective transformation and the kinematics of a five-jointed robot arm. The robot (an RTX arm manufactured by UMI Inc.) is driven by specifying joint positions, which are then attained by independently driven, servo-controlled actuators. The experimental set-up is shown in Figure 10.1. A camera is pointed at the robot which holds a small pen light in its gripper. The camera is connected to a frame-grabber in which the image is thresholded, distinguishing the bright spot. Some image processing obtained the location of the centre of the bright spot as image x and y coordinates. It was not necessary for the image processing to be sophisticated and so the simple implementation took less than quarter of a second, running on an IBM PC.

The world was learned as a SAB-tree:

$$\underbrace{\{\}}_{\textbf{State}} \times \underbrace{\text{Joint Positions}}_{\textbf{Action}} \rightarrow \underbrace{\text{Image Position} \times \text{Comfort}}_{\textbf{Behaviour}} \tag{10.1}$$

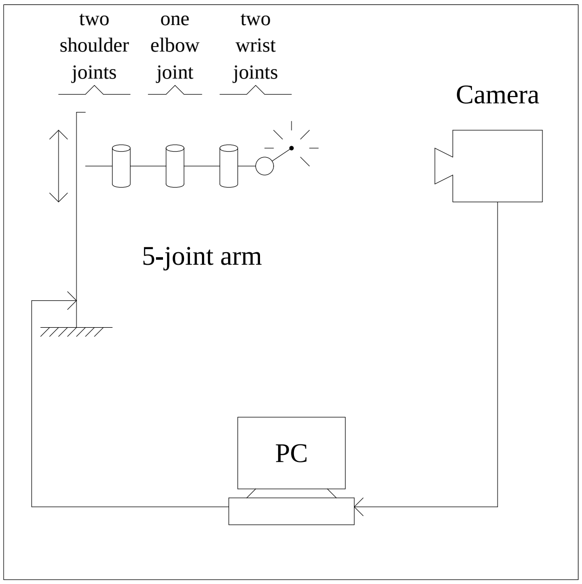

Figure 10.1

The hand-eye coordination experiment.

The **State** component was empty because the configuration of the system prior to requesting a set of joint positions had no effect on the configuration afterwards. The joint positions were a vector of five values: shoulder height, shoulder angle, elbow angle, wrist pitch and wrist yaw. Wrist roll was held constant, because some roll configurations obscured the light from the camera. The image position were x and y values specified in pixels. The comfort value was a measure, depending on the configuration of the joints. The closer each joint position was to the centre of its range, the more "comfortable" the joint position. The ranges of all variables were explicitly known. The comfort value was used to choose between alternative joint angle configurations which achieved the same image position. The best possible comfort value would be zero.

The use of "comfort" was a primitive example of task dimension reduction, described also in the experiments of Section 10.4.

In this experiment the SAB-tree did no smoothing or adaptation to environmental changes. The SAB action chooser was also in a more primitive form, described below.

To test the system a simple game was played in which the robot started with an empty SAB-tree. On each trial a random image location was chosen and the system recorded the number of movements which need to be made before the observed image location agreed with the goal image location (to within 12 pixels on a 512×512 pixel display).

The prototype action chooser which was used began by using partial inversion to obtain the exemplar which had its behaviour vector closest to the goal behaviour (goal x position , goal y position , comfort $= 0$). If it was predicted to be good enough then the action was used. Else, a small random perturbation was applied to the recommended joint angles. The result was inspected and stored in the SAB-tree. If it had improved upon the predicted behaviour it was re-

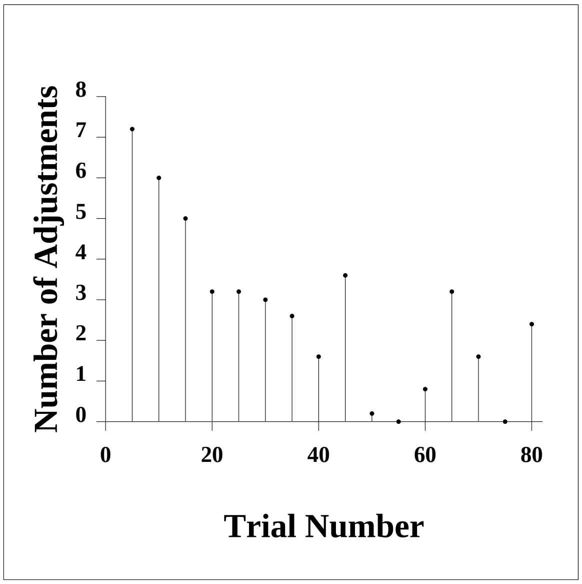

Figure 10.2

Number of extra arm movements required to find the goal image position. The statistics are averages of buckets of five trials.

peated, else a different random perturbation was selected. This process was repeated until the goal location was obtained. This technique is similar to the Hooke and Jeeves multivariate optimization method [Murray, 1972].

Figure 10.2 shows the performance of the robot for the first eighty such trials, measured as the number of perturbations above the initial movement required to reach each randomly generated target position. The whole experiment, including the rather slow movements of the robot and the image processing, took only 18 minutes.

10.1.1 Local Interpolation

This experiment also used local regression to improve the nearest neighbour generalization. The regression was used in the **Behaviour** $\rightarrow$ **Action** direction of the mapping, by means of the nearest 5 neighbours which surrounded the goal behaviour. It was recognised that this was generally too expensive to be able to perform on-line, anticipating future application of *SAB*-trees to dynamic tasks. To solve this, the interpolation was performed off-line in what were termed "dreaming" periods.

During a "dreaming" period, the software links to the robot and sensors are cut, and replaced by links to the local regresser. The controller continues to learn as if the task were still happening for real, but instead of gaining observations from the real world it gains them from the regresser. The regressed observations are added to the *SAB*-tree, but are tagged as approximations. Once real world learning is switched on again, the controller ensures it verifies any tagged point prior to its use.

Figure 10.3 shows how the performance improved when a dreaming period took place just after

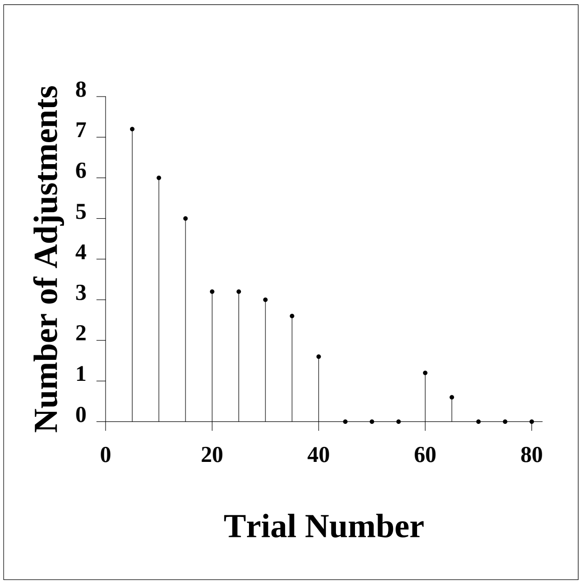

Figure 10.3

Number of extra arm movements required to find the goal image position, with a "dreaming session" just after trial 40. The statistics are averages of buckets of five trials.

the fortieth trial.

The next experiment was to learn to move in sequence through 100 points uniformly spaced on an ellipse specified on the image. First, the robot practised moving to each point in turn for a total of ten minutes. Figure 10.4 shows the results of a "test" given after the practice session: moving as close as it can (according to the SAB-tree) to each point. The target sequence of points is between the two white bands shown on the image.

Figure 10.5 shows how it performed on the test with a dreaming period after 10 minutes of practice, but before it verified the memory values regressed during this time. Figure 10.6 shows the results after a further fifteen minutes of practice. The times for these results are relatively short in a field where learning can often take hours or even days.

In the subsequent experiments of this chapter, dynamic control is learned. The RTX robot cannot be controlled in a realistic dynamic fashion and it was therefore necessary to abandon the use of a real robot in favour of simulation.

10.2 SAB Learning—The Two Joint Arm

This section provides a very detailed investigation of one application of SAB learning. It is applied to a simulation of the two-jointed dynamic manipulator depicted in Figure 10.7. The arm has the following features:

- **Geometry.** The arm is constrained so that it may never straighten: the elbow angle, θ_1, is always between 0 and 180 degrees. The hand may not move outside the frame of the diagram in Figure 10.7.

Figure 10.4

The ellipse tracing test immediately after 10 minutes of practice. The bright spots are the positions to which the light was moved. The image of the arm can be seen in the background.

Figure 10.5

The ellipse tracing test after 10 minutes of practice, followed by a period of dreaming but no verification.

Figure 10.6

The ellipse tracing test after 10 minutes of practice, followed by a dreaming period and followed in turn by 15 minutes of verification and practice.

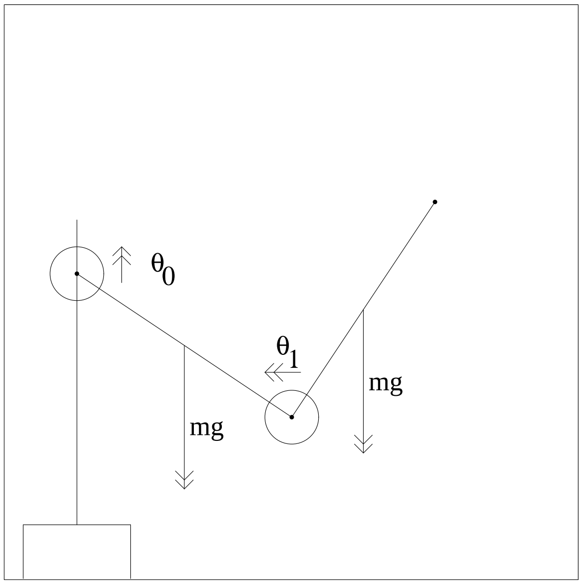

Figure 10.7

A two-jointed dynamic manipulator acting under gravity.

- **Perception.** The perception is visual. The simulation can, at each time step, view the x-coordinate and y-coordinate positions of the hand. It can also sense the velocity of the hand in both the x and y directions. Because there are only two joints, and because of the constraint that the elbow may not straighten, this is sufficient to define the current perceived state.

- **Dynamics.** It is controlled by two motors: Motor 1 at the shoulder and Motor 2 at the elbow. These each provide a torque. At each time step the torques may be changed.

In this domain the **Perceived State** consists of the observed position and velocity of the hand. The two-dimensional **Action** is the pair of applied joint torques and the two-dimensional **Behaviour** is the perceived acceleration of the hand's position. This is not observed directly, but is estimated as

$$\mathbf{b}_i = \frac{\mathbf{v}_{i+1} - \mathbf{v}_i}{h} \tag{10.2}$$

where $\mathbf{v}_i$ is the perceived velocity on the ith time step, and h is the length of a time step.

The two-jointed manipulator was chosen because (i) it has a six-dimensional control space, which is of non-trivial size (ii) the equations of motion of an idealized two-jointed manipulator are well known, and cheap to compute and (iii) manipulators are a particularly useful piece of robot technology. The dynamic equations of motion are from [Fu *et al.*, 1987].

10.2.1 Experimental environment

The simulation and SAB-learning were performed in an environment in which the system parameters were all adjustable for experimentation purposes. They are listed in Table 10.1.

SAB-Tree Parameters		*SAB*-Control Parameters		Simulation Parameters	
`Statemax0`	1	`Tolerance`	0.10	`Max-torque`	6
`Statemax1`	1	`Num-cands`	10	`Time-step`	0.02
`Statemax2`	10	`Prob-const`	1	`Noise-level`	0.002
`Statemax3`	10	`Locality`	0.67		
`Actionmax0`	6	`Local-const`	1		
`Actionmax1`	6	`Delta-old`	0.05		
`Bhvmax0`	100				
`Bhvmax1`	100				
`Statemin0`	-1				
⋮					
`Bhvmin1`	-100				
`Search-width`	0.02				

Table 10.1: Arm experimental system parameters

`Statemax0` ... `Bhvmin1` are the sixteen values required to scale the vectors in the *SAB*-tree. The need to know the maximum ranges of **State**, **Action** and **Behaviour** components was discussed in Section 5.1. All vectors in the tree are scaled to the range $[0, 1]$. State variables 0 and 1 are the x and y positions respectively. These position maxima were easy to estimate, because the hand may not move outside the coordinates $[-1, 1]^2$. The range of observed speeds (state components 2 and 3) were less obvious but were estimated as 10ms^{-1} from observations of the simulation. The action ranges were again known explicitly, as part of the definition: they are the maximum permissible torques. The behaviour ranges were estimated from observation. The `Search-width` parameter is the range width D_{range} of the smoothing kernel discussed in Section 7.2.

The *SAB*-control parameters were discussed in Chapter 8. `Tolerance` is the deviation in the behaviour which is deemed acceptable when the *SAB*-controller is requested to generate an action $\mathbf{a}_{\text{raw}}$ to achieve a goal behaviour $\mathbf{b}_{\text{goal}}$. It is scaled to the range $[0, 1]$. `Num-cands` (see Section 8.1) is the number of candidate actions considered when the value recommeded by partial inversion is predicted as inadequate. `Locality` and `Local-const` are the P_{loc} and C_{loc} parameters of Section 8.2, which bias the generation of candidate actions. `Prob-const` (again, see Section 8.1) is the scaling factor C used in the probability of success estimate, P_{succ}. Its value reflects the expected slope of the function being learned, but its accuracy is not expected to be important. `Delta-old` is the Δ_{old} parameter used in the detection of old, inaccurate, exemplars (see Section 7.3).

The simulation parameters `Max-torque` and `Time-step` are self explanatory. The arm is given simulated noise defined by the `Noise-level` parameter. At each stage, the requested torques are corrupted before being applied. The corruption is uniform in each action component, up to a

```
       Start State: <  0.00,  0.00,  0.00,  0.00 >

Minimal behaviour:    <-10.00,-10.00 >
Maximal behaviour:    < 10.00, 10.00 >
(Empty tree. Random action.)
            Action: < -4.15,  0.37 >
Predicted Behaviour: < NULL VECTOR >
   Actual Behaviour: <-13.39,-37.33 >
```

Table 10.2: Repetitive Control: The first cycle

maximum value of `Max-torque` $\times$ `Noise-level`.

10.2.2 Convergence of Actions

The first experiment examines how the *SAB* action chooser performs on repeated occasions of the same state and requested behaviour. In this experiment, the state is reset after every control cycle. This hypothetical situation was discussed in Section 8.2, but is unrealistic—normally in subsequent experiments the state, and often the goal behaviour, would vary between cycles. Here, the start state is position and velocity both zero—the hand stationary in the center of the display. The goal behaviour is zero acceleration, which can be achieved by action $(0.98, 0.25)$. The action is non-zero because the torques have to counteract gravity.

Tables 10.2–10.5 consist of *SAB* action chooser debug output. This shows in extreme detail, the behaviour of the algorithm. Table 10.2 shows the performance on the very first cycle. Here, there is no previous experience whatsoever in the *SAB*-tree, and so a uniformly distributed random action is chosen. The minimal and maximal behaviours mentioned in the table are those generated by the default `Tolerance` parameter setting of 0.10 (see above for a description of `Tolerance`).

Table 10.3 shows the performance on the second control cycle. There is one exemplar in the *SAB*-tree, obtained from the first cycle. This is, of course, the exemplar suggested by partial inversion. It is not suitable, because is does not achieve the `Tolerance`. Instead, ten candidate actions are generated (ten, because the `Num-cands` parameter is set to 10 by default). These all predict the same behaviour because they all have the same nearest neighbour. The estimated probabilities of success vary because the reliability of the nearest neighbour prediction varies with distance. Notice that the first four candidate actions are the extreme actions. For this controller they are always generated because they might be needed for the estimation of strength (see Appendix C). The quoted nearest neighbour distances are in the uniformly scaled *SAB*-tree units. The best candidate action is, unfortunately, not good enough, and the actual behaviour is further from goal than the original.

The third cycle is displayed in Table 10.4. There are now two exemplars, but the original was more promising. As a result it is again returned by partial inversion, but is again, of course,

```
        Start State: <  0.00,  0.00,  0.00,  0.00 >

Minimal behaviour:       <-10.00,-10.00 >
Maximal behaviour:       <10.00,10.00 >
Partial Invert Action: <-4.15, 0.37 >
Nearest BSmooth:         <-13.39,-37.33 > at distance 0.000. Psucc = 0
Cand 0: <-6.00,-6.00 > BSmooth: <-13.39,-37.33 > at distance 0.553. Psucc = 0.00910982 (Best yet)
Cand 1: < 6.00,-6.00 > BSmooth: <-13.39,-37.33 > at distance 0.999. Psucc = 0.00306144
Cand 2: <-6.00, 6.00 > BSmooth: <-13.39,-37.33 > at distance 0.493. Psucc = 0.0110591 (Best yet)
Cand 3: < 6.00, 6.00 > BSmooth: <-13.39,-37.33 > at distance 0.967. Psucc = 0.00325637
Cand 4: < 1.63,-2.01 > BSmooth: <-13.39,-37.33 > at distance 0.521. Psucc = 0.0100832
Cand 5: < 0.56, 2.34 > BSmooth: <-13.39,-37.33 > at distance 0.426. Psucc = 0.0140408 (Best yet)
Cand 6: < 4.76,-0.07 > BSmooth: <-13.39,-37.33 > at distance 0.744. Psucc = 0.00534009
Cand 7: <-5.60, 0.06 > BSmooth: <-13.39,-37.33 > at distance 0.124. Psucc = 0.0185096 (Best yet)
Cand 8: <-3.34, 3.13 > BSmooth: <-13.39,-37.33 > at distance 0.239. Psucc = 0.0277313 (Best yet)
Cand 9: < 2.49, 1.73 > BSmooth: <-13.39,-37.33 > at distance 0.565. Psucc = 0.0087751
Probability of Success is   0.0277

             Action: <-3.34, 3.13 >
Predicted Behaviour: <-13.39,-37.33 >
   Actual Behaviour: <-84.87, 4.54 >
```

Table 10.3: Repetitive Control: The second cycle

deemed inadequate. Ten further candidate actions are again generated. This time there is variety in the nearest neighbour prediction. Not surprisingly, the most promising candidate has the original exemplar as its nearest neighbour. This time the choice is lucky, and achieves a good behaviour, though it still does not meet the tolerance.

The action discovered by cycle 3 remained the most promising for the next 7 cycles. On trial 10 it was replaced by action $(-1.01, 0.35)$, which produced behaviour $(-6.68, -13.71)$. This was in turn replaced by a slightly closer action on cycle 15. On trial 19, an action was discoved which produced behaviour $(-6.23, 2.40)$—within the tolerance. Cycle 20 is displayed in Table 10.5. All subsequent cycles used the same action. The small mismatch between the predicted and actual behaviour was due to the simulated noise. If there had been no noise the prediction would have been entirely accurate, because the state and action were the same as before.

The whole experiment was repeated for thirty runs. The graph in Figure 10.8 shows, for each cycle number n, the proportion of occasions for which the tolerance not had been achieved by cycle n. Thus, for example, the value is 1 for cycle number 1, because on every run, the action was unsuccessful on trial 1. By cycle 13, 50% of runs had succeeded. This is encouraging, because it agrees with the prediction of Section 8.2, that the correct behaviour is converged to rapidly using the SAB action chooser, even with only a few candidate actions. The tolerance was, however, quite large, meaning that approximately 1% of possible behaviours would succeed. How does the time to achieve goal behaviour increase as the tolerance becomes more strict?

```
        Start State: <  0.00,  0.00,  0.00,  0.00 >
Minimal behaviour:     <-10.00,-10.00 >
Maximal behaviour:     <10.00,10.00 >
Partial Invert Action: <-4.15, 0.37 >
Nearest BSmooth:       <-13.39,-37.33 > at distance 0.000. Psucc = 0
Cand 0: <-6.00,-6.00 > BSmooth: <-13.39,-37.33 > at distance 0.553. Psucc = 0.00910982 (Best yet)
Cand 1: < 6.00,-6.00 > BSmooth: <-13.39,-37.33 > at distance 0.999. Psucc = 0.00306144
Cand 2: <-6.00, 6.00 > BSmooth: <-84.87, 4.54 > at distance 0.326. Psucc = 0.00553616
Cand 3: < 6.00, 6.00 > BSmooth: <-84.87, 4.54 > at distance 0.815. Psucc = 0.00364744
Cand 4: <-1.40, 0.57 > BSmooth: <-13.39,-37.33 > at distance 0.230. Psucc = 0.028409 (Best yet)
Cand 5: <-2.79, 3.06 > BSmooth: <-84.87, 4.54 > at distance 0.046. Psucc = 0
Cand 6: <-0.33, 1.07 > BSmooth: <-84.87, 4.54 > at distance 0.304. Psucc = 0.00495881
Cand 7: <-3.23, 3.54 > BSmooth: <-84.87, 4.54 > at distance 0.035. Psucc = 0
Cand 8: <-4.00, 2.94 > BSmooth: <-84.87, 4.54 > at distance 0.057. Psucc = 0
Cand 9: <-5.75, 4.13 > BSmooth: <-84.87, 4.54 > at distance 0.218. Psucc = 0.00162928
Probability of Success is   0.0284
             Action: <-1.40, 0.57 >
     Goal Behaviour:  [ 0 , 0 ]
Predicted Behaviour: <-13.39,-37.33 >
   Actual Behaviour: <-13.02,-13.96 >
```

Table 10.4: Repetitive Control: The third cycle

```
        Start State: <  0.00,  0.00,  0.00,  0.00 >
Minimal behaviour:     <-10.00,-10.00 >
Maximal behaviour:     <10.00,10.00 >
Partial Invert Action: < 0.90, 0.47 >
Nearest BSmooth:       <-6.23, 2.40 > at distance 0.000. Psucc = 1
Partial Inversion action is tolerable.
Probability of Success is   1.0000
             Action: < 0.90, 0.47 >
Predicted Behaviour: <-6.23, 2.40 >
   Actual Behaviour: <-6.13, 2.08 >
```

Table 10.5: Repetitive Control: The twentieth cycle

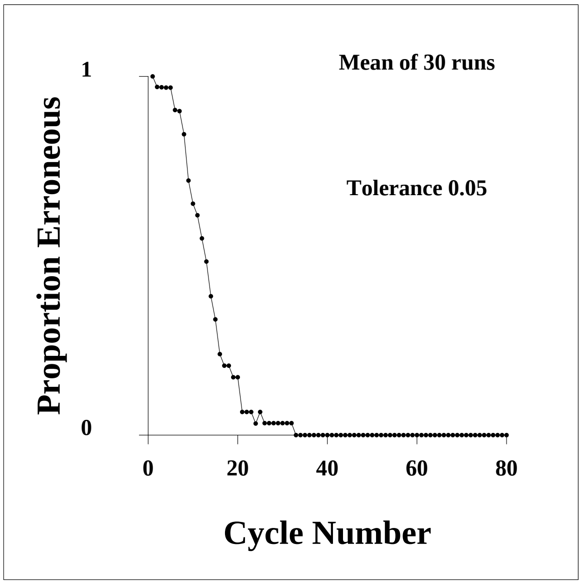

Figure 10.8

Proportion of runs of the repetitive control experiment which met tolerance against cycle number. The tolerance was 10%.

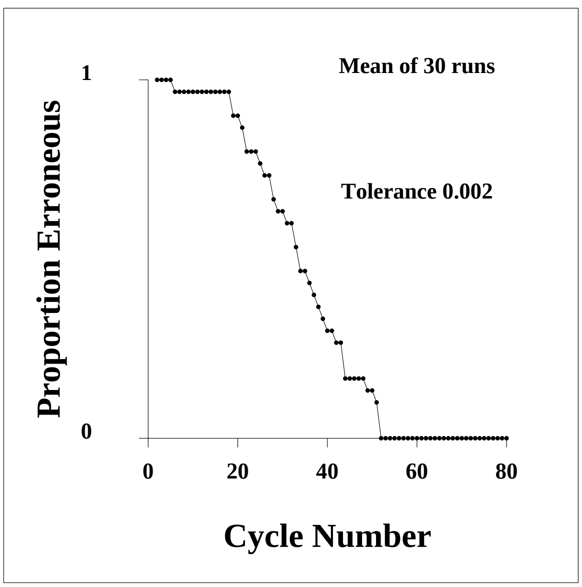

Figure 10.9

Proportion of runs of the repetitive control experiment which met tolerance against cycle number. The tolerance was 0.4%.

A second experiment was performed in which the tolerance was only 0.004, a 25-fold decrease over the previous experiment. The experimental conditions were identical, except that the `Noise-level` parameter was set to zero. This was because the tolerance would not have been possible (except occasionally by luck) with the noise level of the previous experiment. The behaviour is shown in Figure 10.9, and it can be seen to be no more than a small factor worse than the previous experiment. The median number of cycles before achieving tolerance 0.004 was 35.

Further experiments were performed with varying tolerance levels. The results are tabulated in

Tolerance	Median time to success
0.2	6
0.1	13
0.04	15
0.02	25
0.01	24
0.004	35
0.002	40
0.0004	48
0.0001	58

Table 10.6: Time to achieve tolerance for the simple "holding still" task.

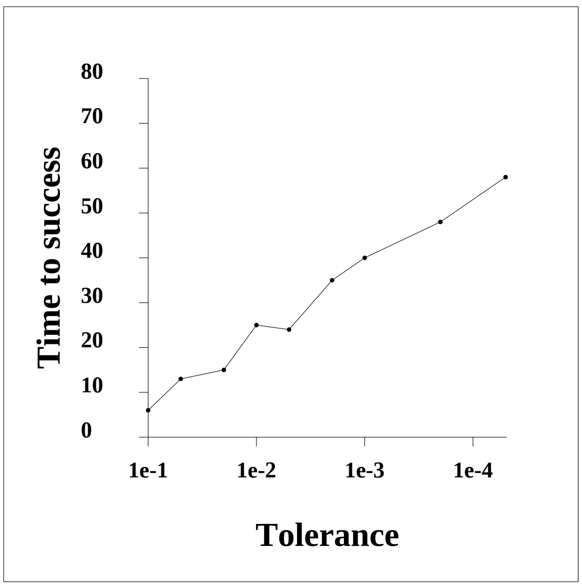

Figure 10.10

A plot of the data from Table 10.6: Median time to find the correct set of torques to achieve the goal tolerance. The horizontal axis is logarithmic.

Table 10.6 and plotted in Figure 10.10.

10.3 Trajectory Tracking Experiments

The trajectory following task is a realistic robot control problem, which will be used to empirically explore the detailed behaviour of the *SAB* control cycle. I will begin this section by describing the experiment in detail, and will then explore the effects of varying the following features:

- Number of candidate actions.
- Tolerance level supplied to the *SAB* action chooser.

- Locality C_{loc}, and smoothness estimate C.
- Weighting of variables in the *SAB*-tree.
- World noise.
- Environmental changes.

10.3.1 The Trajectory Tracking Task

The experiment can be viewed at the *trial* level, the *learning run* level and the *experiment* level.

- **Trajectory Tracking Trial.** The arm must follow a perceived circle at a specified speed. The circle is shown in Figure 10.11. The goal speed takes fifty simulated time steps to travel around the circle. The abstract level controller uses the ice puck trajectory tracking mechanism of Section 9.3. The hand is initialized at the right-most point of the circle. Then on each cycle, the goal behaviour is the ice puck acceleration required to stay as close to the current trajectory as possible. One trial consists of one attempt to follow the circle.
- **Trajectory Tracking Learning Run.** A run consists of thirty-two trials. Before the initial trial, the *SAB*-tree is empty—the controller begins with no world knowledge. Between trials the world knowledge is maintained, and so if learning occurs, performance should tend to improve during a learning run.
- **Trajectory Tracking Experiment.** The experiment consists of executing a number of independent learning runs. The purpose of running a number of times is to obtain more reliable statistics about the learning performance. There are typically 20 runs in each experiment.

10.3.2 Format of Results

The results for the first experiment are shown in Figure 10.12. However, before they can be discussed it is necessary to explain how they were obtained and what information they are intended to convey.

The **Mean Tolerance** graph plots the behaviour of the *SAB* action chooser against trial number. The *tolerance error* is the amount by which the actual behaviour differs from the tolerance band defined by the task. Each trial has a mean tolerance error. Thus for each learning run a graph of mean tolerance error against trial number can be obtained. The graph in the figure shows the average of 23 such graphs, obtained from 23 learning runs. The vertical bars display the standard deviations of the statistics. Notice that an experiment in which the mean tolerance error is low is not necessarily performing the task well—it is simply achieving the tolerance. If a large tolerance were specified then it would be easy to achieve a zero mean tolerance error. The axis is labelled as a percentage of the maximum possible tolerance error.

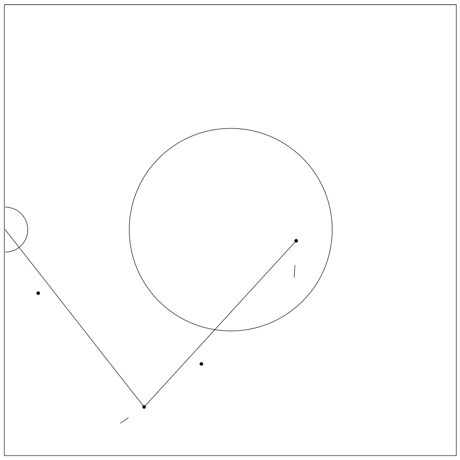

Figure 10.11

The simulated arm must follow the circular trajectory in an anticlockwise direction. It starts from the rightmost point. Here, it is shown during the end of an initial, unsuccessful, learning trial.

The **Mean Position** graph plots the average position error against trial number. At each time step the position error is computed as the Euclidian distance between where the hand is, and where it should be. The average error for each trial is recorded. The graph is constructed as the average of the 23 observations for each trial. This is a more direct measure of the performance of the task—it would only ever be zero if the trajectory were followed precisely. It is labelled as a percentage of the maximum possible position error, which would occur if the desired position were in the bottom right of the display and the hand were in the top left.

A learning system which learned to perform 98% of the trajectory perfectly, but consistently failed badly at the remaining 2% would be judged well by the Mean Position estimate. For some uses of trajectory tracking however, such performance would be inadequate. This undesirable behaviour would be detected by the **Worst Position** graph which plots the average of the worst position errors which occur on each trial. The *worst position error* of a trial is the furthest Euclidian distance which the hand reaches from the desired point on the trajectory.

The graphics on the right of the diagram display six examples of the trajectories followed by the arm during learning. These are all taken from the same randomly chosen experiment. The edge of the gray disc indicates the desired trajectory. The graphics display the performance on the 1st, 2nd, 4th, 8th, 16th and 32nd trial. They are shown in order to provide a more intuitive feel of the performance of the arm, in the absence of an actual animation of its behaviour. They cannot be used reliably to compare between different experiments because (i) they are only random examples, and so may not be typical of the performance of the experiment, particularly when there is high deviation between runs and (ii) they are only shown with limited resolution, and so apparently circular motion on the diagram might still contain tracking errors. A particular kind of undetectable error would be the case where the hand lags behind the desired position.

The small table of data in the bottom right hand corner gives the following information:

- The experiment name, "BASIC", named as such because it is this basic experiment to which others will be compared.
- The number of runs in the experiment. The mean behaviours displayed by the graphs on the right become more reliable as indicators of the true mean behaviours, the greater the number of runs. For 23, the mean behaviour is, with confidence 95%, estimated to lie within 0.5 sample standard deviations of the sample mean (as mentioned earlier the standard deviations are represented by the vertical bars—see Appendix A for further details).
- The tolerance level used by the *SAB* action chooser. The units are as a proportion of the estimated minimal and maximal ranges of behaviour. A high value is easier to achieve, but may lead to reduced task performance.
- The *Learned By* statistic. This figure is a score which will be used to quantitatively compare different experiments. It is the number of the final trial upon which the mean tolerance error exceeds a small fixed threshold (4%). This threshold is indicated near the bottom of the y-axis of the Mean Tolerance graph. The *Learned By* statistic will be used to compare the performance of the *SAB* action chooser with different parameter sets.
- The *Final Position Error* statistic. This is another score, used to quantitatively compare the overall task performance of different experiments. It consists of the mean position error during the last six trials of each run: it is thus the mean of the final six values of the Position Error graph.

Neither the *Learned By* statistic, nor the *Final Position Error* statistic are particularly useful in isolation, but they will prove useful as comparators between experiments.

10.3.3 Discussion of the BASIC experiment results

In this experiment all the learning parameters were the default values given in Section 10.2. The results demonstrate that *SAB* learning does indeed learn to control this task. By the standards set above it took only eight trials to, on average, achieve adequate performance, and after sixteen trials the tolerance error was always almost zero, indicating that on all trials, the tolerance was almost always achieved.

The task was also learned accurately. In the later trials, the position was typically at worst no more than 0.5% away from the desired distance. Good positional tracking was obtained early on–after only the 4 trials the average position error was only 2%. The sample standard deviations after the first 10 trials are low, reflecting the confidence that the learning runs were not simply lucky.

`Tolerance`	Learned By	End Position Err
50%	3	2.84%
30%	5	2.17%
20%	6	0.73%
10% (BASIC)	8	0.41%
4%	21	0.35%
2%	25	0.33%
0.2%	Didn't Learn	0.64%

Table 10.7: Trajectory tracking: Performance variation with `Tolerance`

Figures 10.13–10.16 show the distribution of exemplars in the *SAB*-tree after a learning run. Some components, particularly the position and velocity components are clustered around the important areas of state space. This is a desirable result, demonstrating the variable resolution available from nearest neighbour learning.

Each trial took only 20 seconds to simulate, learn and display graphically. The parameter values used in this experiment were not chosen with great care—they were the *SAB* learning system defaults. The exception to this were the state, action and behaviour minimum and maximum ranges which were estimated as what seemed reasonable after playing with the simulation for a short time. It is nevertheless important to determine empirically whether the system is brittle as the values change. It would also be interesting to see if the performance improves with other parameters. Such experiments are documented in the following subsections.

10.3.4 Tolerance

Making the `Tolerance` parameter larger makes it easier to learn, but means that the accelerations (behaviour) which are supplied might be further from those recommended by ice puck control. An extreme example of this is the TOL.50, experiment displayed in Figure 10.17 in which any behaviour within half the maximum range of behaviours to the goal was considered acceptable. Unsurprisingly this learns to achieve the requested tolerance very quickly, but the performance is poor. The final mean position error of 2.84% corresponds to an average 8cm deviation from the trajectory. There is very little diversity in the actions—in practice the same action is repeated many times for many parts of the state space. This results in the smooth, but wrong, curves of the top half of the trajectory. Very little changes between trials 4–32 because there is no need to refine the behaviour. The results for a variety of experiments with other tolerances are summarized in Table 10.7. Each experiment had identical parameters to the BASIC experiment, except for the `Tolerance`.

The pattern is clear that with decreasing tolerance size the time to learn increases and the final performance accuracy improves. The final experiment, with a tolerance of 0.2% was too accurate

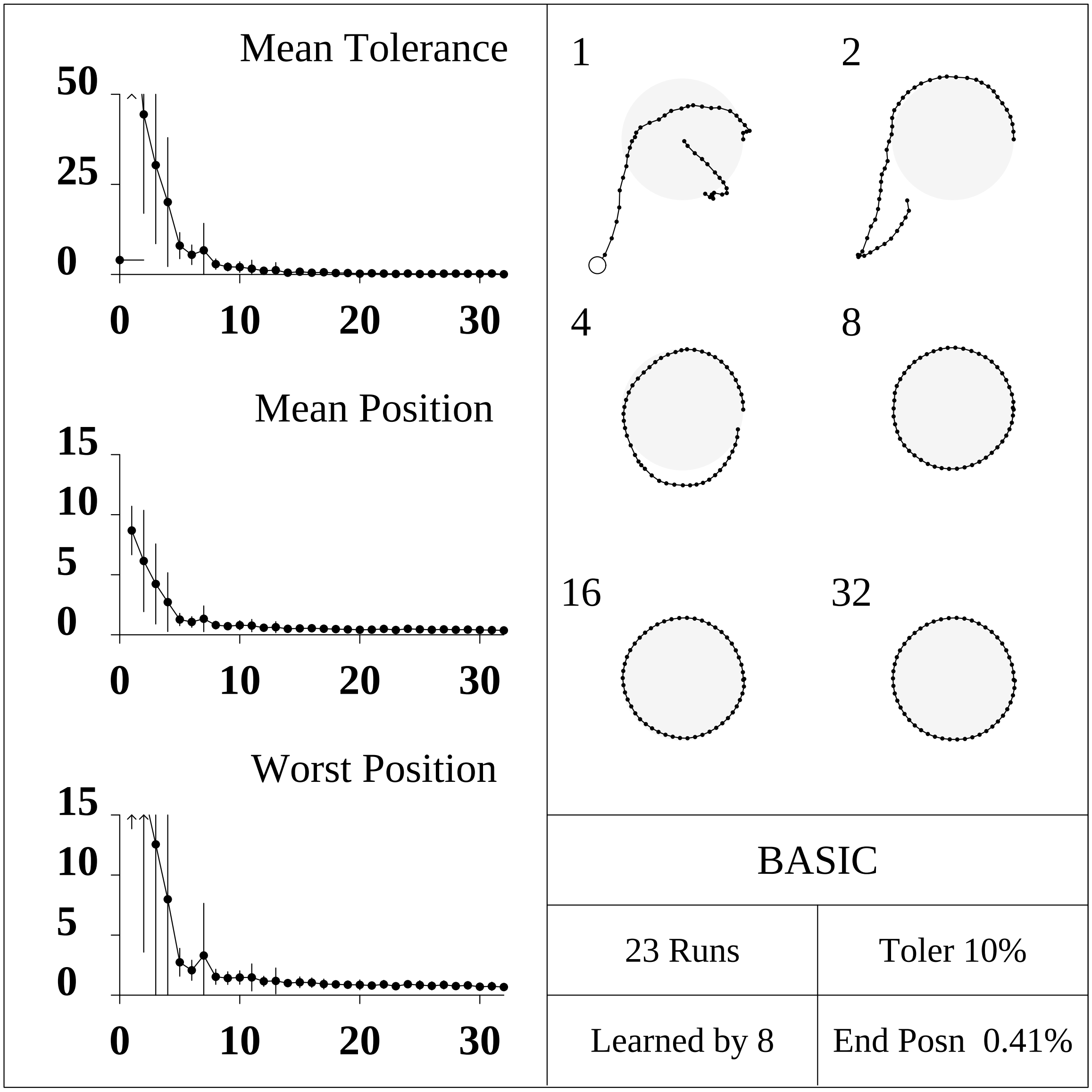

Figure 10.12

Experimental method described in the text. All parameters were default.

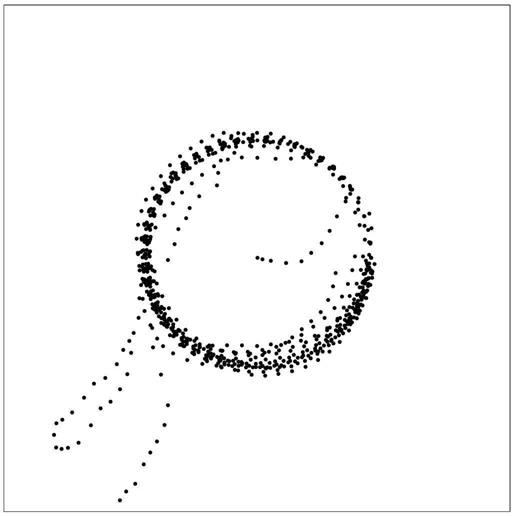

Figure 10.13: The distribution of exemplars in "position" space after a learning run of the BASIC Experiment.

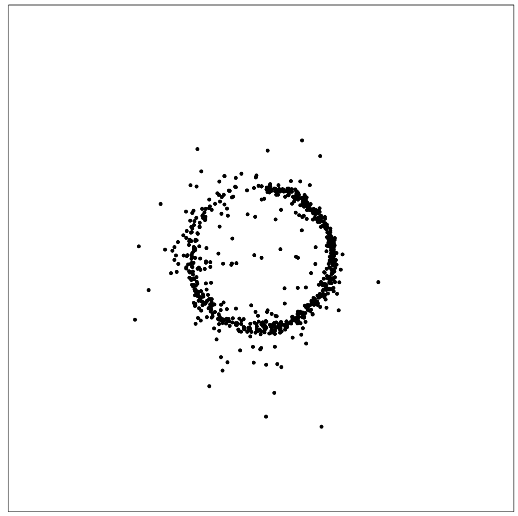

Figure 10.14: The distribution of exemplars in "velocity" space after a learning run of the BASIC Experiment.

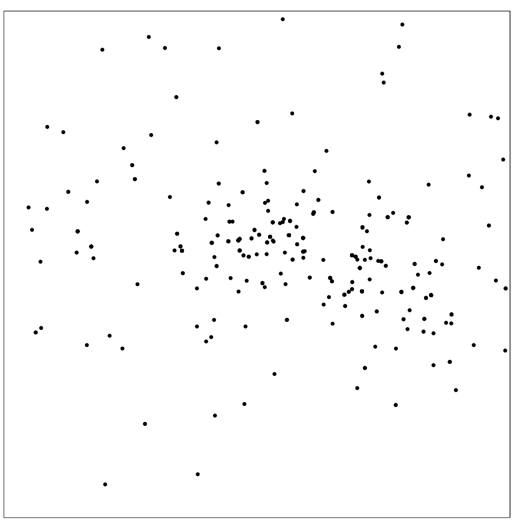

Figure 10.15: The distribution of exemplars in "action" space after a learning run of the BASIC Experiment.

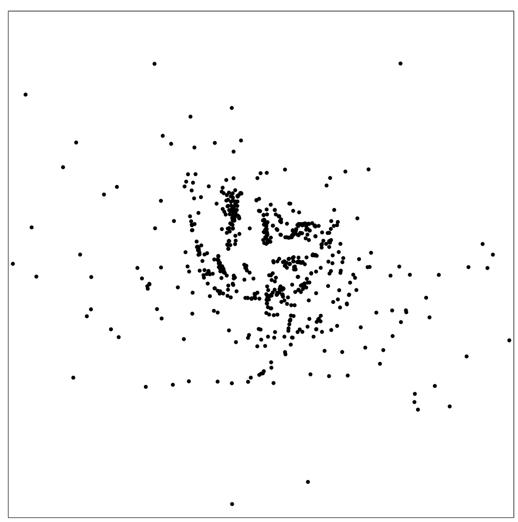

Figure 10.16: The distribution of exemplars in "behaviour" (that is, acceleration) space after a learning run of the BASIC Experiment.

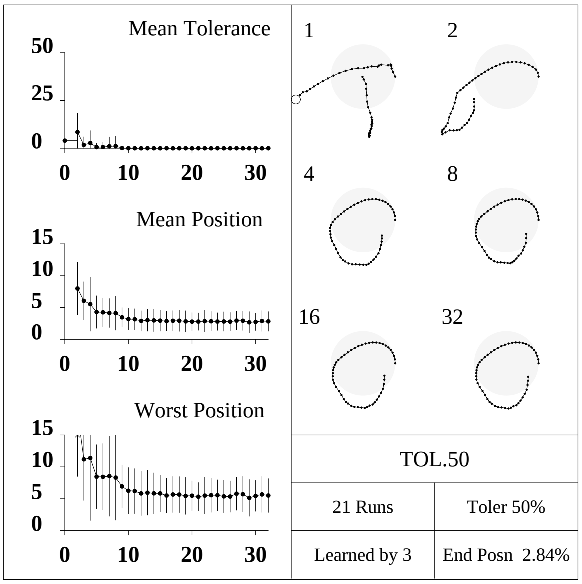

Figure 10.17

Experimental method described in the text. All parameters were default with the exception of the **Tolerance** parameter, which had value 50%.

to be achieved in only thirty-two trials. It is possible that because of the simulated noise it would never have been achieved. However, there is a large range of **Tolerance** values for which the learning is both fast and accurate, and so it has, fortunately, proved not to be a critical parameter. It is notable that its ideal value depends on the abstract task being performed. Many tasks do not require high accuracy in their trajectory tracking, and so tolerances as large as 50%, with their impressive rate of learning, might be desirable.

10.3.5 Number of Candidates

At each control cycle, in the absence of a known best action, a number of candidate actions are considered. The BASIC experiment used 10 candidates. At least four candidates, the extreme actions are always considered. Figure 10.18 shows the result when these are the *only* candidates considered. The performance is consistently bad, indicating that the task cannot be achieved by extreme actions alone.

Figure 10.19 displays an experiment with five candidate actions. The performance is considerably improved, but learning is still much slower than that of the BASIC experiment. The final position error is also inferior. This experiment is equivalent to a combination of 'SR'- and 'SL'-control discussed in Section 8.1 in which in the absence of known success a random action is taken.

The other extreme is shown in Figure 10.20 in which 100 candidate actions are considered on each cycle. The learning speed is faster than for the BASIC experiment. After four experiments, when the tolerance is achieved, no further performance improvement takes place because the *SAB* action chooser needs not experiment any further. The real time to run the 100-candidate experiment was about five times greater at the start than the BASIC experiment, but the total time was

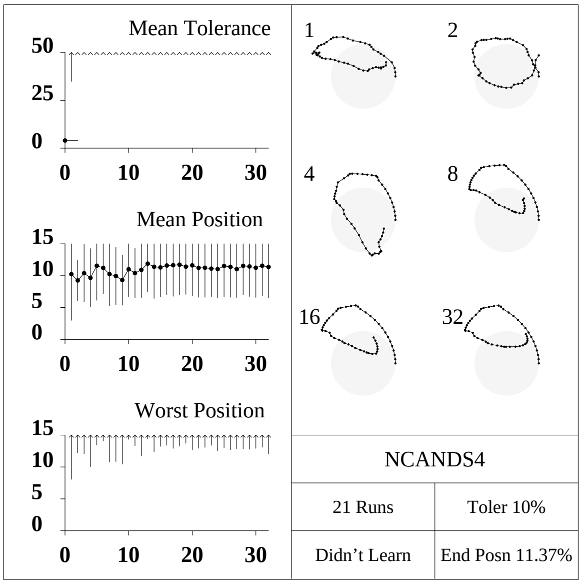

Figure 10.18

Experimental method described in the text. All parameters were default with the exception of the **Num-cands** parameter, which had value 4.

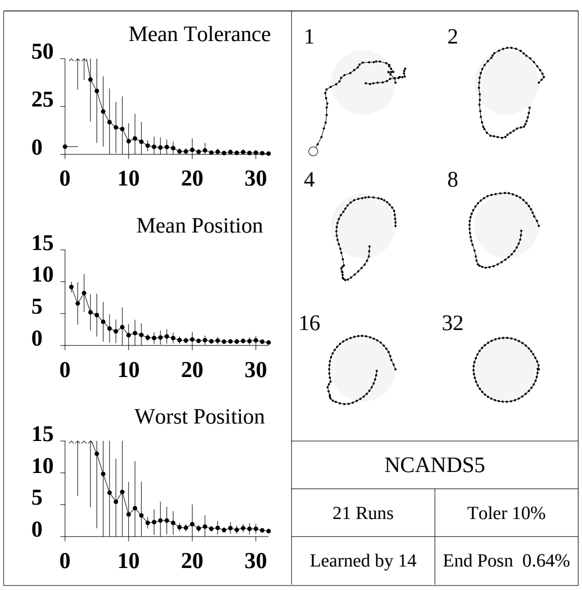

Figure 10.19

Experimental method described in the text. All parameters were default with the exception of the **Num-cands** parameter, which had value 5.