Journal of Mathematical Imaging and Vision 11, 65–90 (1999)

Madura: A Language for Learning Vision Programs from Examples

RHYS A. NEWMAN
Robotics Research Group, Dept. Engineering Science, Oxford University
newman@robots.ox.ac.uk

Abstract. Recently the idea of designing a computer system which automatically connects a number of independent vision modules together to solve a given computer vision problem has attracted significant interest. However the main assumption of this endeavour, namely that the modules used as the building blocks of the vision system are essentially fixed, is questionable in the light of previous experience. Therefore it is important to be able to modify even the detailed operation of the basic modules used, something which is not practical using conventional techniques.

This paper constructs a general method by which the computer code of a vision module can be altered automatically to make it mimic a desired behaviour. The system which does this, termed L_u, modifies a basic module template using interaction with an *Oracle* as a guide. The Oracle is an entity which, when given an input value, produces the corresponding output of the function which is to be mimicked. The system developed is based upon a new model of computation which endows it with the important properties that extracting the template (i.e. structure) of any module's computer code, as well as determining the best questions to pose to the Oracle are both performed automatically. Thus the L_u described has significant advantages over many other models which might be used (e.g. Neural Networks).

Dealing directly with this new model is not always convenient. Therefore a new computer language *Madura* is defined which provides a high-level interface to it. As Madura is syntactically similar to JAVA, it is simple to express the code of many basic vision modules in its terms and the results of L_u (the Madura code of a module which mimics the Oracle) are similarly simple to understand and use.

This paper shows a number of results which demonstrate how the L_u developed can learn many state-of-the-art initial vision algorithms in a matter of minutes. The current and future impact of this work is also examined.

Keywords: machine learning, theoretical computing, automatic program refinement, computer vision

1. Introduction

Despite the rapid improvements of computer hardware over the last few years, competent computer vision systems have not been forthcoming except in controlled or restricted applications [5, 10, 11, 12, 15, 17, 18, 19, 21, 26, 30, 32]. One of the major causes of these failures is the unreliability of the initial or basic processes which form the initial step of most current vision systems [1, 9, 17, 18]. However it is now believed that the shortcomings of any single basic process (or *module*) can be overcome by using a number of different modules in parallel and combining their results intelligently.

Unfortunately, the task of deciding which modules to use and how to connect them is still unclear. Thus the idea of designing a system to do this automatically is attractive. This approach introduces the concept of a *supervisory* computer system whose task it is to connect basic vision modules together to solve any particular vision problem. Recently, the problem of designing such a system has attracted significant interest [2, 4, 6, 9, 17, 27, 30] and has been done with some success in limited applications [22, 31].

The central assumption underlying this endeavour is the belief that the modules chosen as part of a computer vision system may be considered fixed. Hence it is implicit that conventional computer vision modules do represent a competent toolkit of modules which a supervisory system can (in principle) arrange to solve many vision problems. Where extra flexibility within a module is required one possibility is to supply extra parameters, for example the cutoff level in a thresholding module.

Although an attractive idea, previous experience is a warning against complacency, especially as supervisory systems which automatically define module interactions are still in their infancy. It is to be expected that the more modules available the more complex and time-consuming the supervisory system's task will be. Therefore it is likely to be advantageous to have a small set of modules which are inherently flexible. This is because various instantiations of any such basic templates can simulate a number of different algorithms. These may be termed *flexible modules*.

In order to exploit the flexibility of such a module three things must be defined:

1. The way the goal is presented: i.e. means of judging when a solution is acceptable.
2. How the details of the template are to be expressed: i.e. the language used to express it and the information it contains.
3. An algorithm which modifies the template in order to find a solution (as specified by the goal).

Considering the first point, the least structured way of expressing a desired solution is by providing the modification program (i.e. that which carries out Step 3) access to an *Oracle*. The oracle is a device which provides the desired output corresponding to any input given to it. This arrangement is illustrated in Fig. 1. Here,[1] L_u is a computer program which modifies a flexible module, the Oracle is part of the supervisory system and the *hint* is the basic template of the module. It is this template which defines the limits of the module's flexibility.

This arrangement has the advantage of allowing the supervisory system to be ignorant both of the details of the modules' operation and of the modification procedure. Thus from the supervisory point of view, the most useful flexible module is one which can be modified by some other program (L_u) in response to input/output pairs of desired behaviour. The supervisory system's task is made even easier if L_u determines automatically which questions to ask the Oracle.

To address the first point above the model of computation used to define a flexible template must be

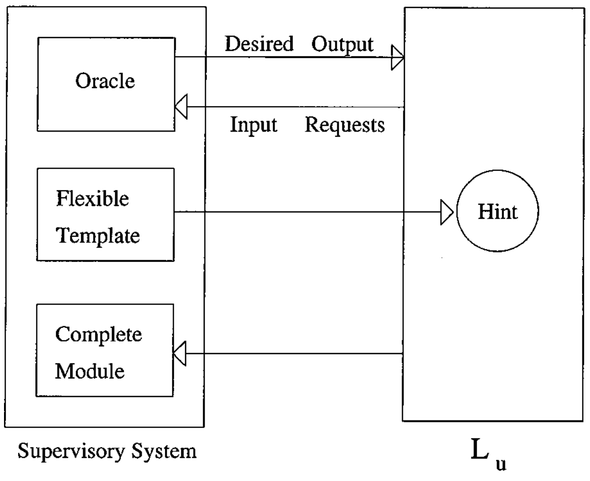

Figure 1. The mimicry model with L_u.

chosen. This must be a universal model to avoid inadvertently restricting L_u to a subset of computable functions which may not contain many useful vision algorithms. In [23, 25] Turing Machines, Recursive functions and more complex computer languages are assessed for their suitability in this application. As none of these models is ideal a new universal model is defined and an algorithm developed which allows L_u to target its questions to the Oracle efficiently. This model is summarised in Section 1.1.

The rest of this paper builds on the work in [25] to outline a complete algorithm for L_u: i.e. point 3 above. In the course of doing this, practical considerations require the development of a new computer language coined *Madura*. To use the complete system developed, a computer vision module must be written firstly in Madura (a simple task given the similarity of Madura to the JAVA language). Using the definition of a computer program's structure as defined in [25] and summarised below, the flexible template of this algorithm is then extracted automatically. It is this which the L_u developed in this paper can modify to create an algorithm which mimics the Oracle. As will be seen in Section 8 the final result can be expressed automatically in the convenient form of Madura source code.

1.1. *The Decision Tree Model of Computation*

This section summarises the *decision tree* model of universal computation [23, 25], while the next describes the way the structure of any computer program can be extracted automatically from this representation.[2]

A function $f : \mathbf{N}^n \to \mathbf{N}^m$ expressed in the decision tree model consists of a *state vector* and a *test tree*.

Define the r-dimensional state vector $\vec{x}$:

$$\vec{x}_0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ u_n \\ u_{n-1} \\ \vdots \\ u_1 \\ 1 \end{pmatrix} \quad \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} m \text{ Output components} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \text{Working variables} \\ \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} \text{Input components} \\ \text{Unit: Always 1} \end{matrix} \tag{1.1}$$

Note how the output and working variables are initially zero.

The test tree is defined as follows. Let q_f and p be row vectors $(1 \times r)$, where q_f is problem dependent. q_f is termed the *halt-test* vector and p the output projection matrix:

$$p = (\mathbf{I} \mid \mathbf{O})$$

where $\mathbf{I}$ is a $m \times m$ identity matrix and $\mathbf{O}$ a $m \times (r - m)$ zero matrix. Define a set of indices J and *test vectors* q_i, where $i \in J$.

A *matrix choice* function $P_i(\vec{x})$ $(i \in \mathbf{N})$ is constructed in the following recursive manner:

$$P_i(\vec{x}) = \begin{cases} \begin{cases} P_{2i}(\vec{x}) & q_i\vec{x} \le 0 \\ P_{2i+1}(\vec{x}) & q_i\vec{x} > 0 \end{cases} & \text{if } i \in J \\ \text{An integer matrix: } M_i & \text{if } i \notin J \end{cases} \tag{1.2}$$

Attaching an index to the state vector, the input being $\vec{x}_0$, computation proceeds as follows:

```
$j = 0$
while   $(q_f\vec{x}_j \le 0)$
{
        $\vec{x}_{j+1} = P_1(\vec{x}_j)\vec{x}_j$
        $j = j + 1$
}
Answer $= p \cdot \vec{x}_j$
```

Thus linear transformations are applied to the state vector produce (in order) $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_j$. Computation halts (at $\vec{x}_j$) if $q_f \cdot \vec{x}_j > 0$. The solution (result) is the first m elements of $\vec{x}_j$, i.e.

$$f(u_1, \ldots, u_n) = p \cdot \vec{x}_j$$

This model derives its name from the observation that the matrix choice function P_1 above is most conveniently expressed as a (binary) decision tree.

Definition 1.1 (*Promotion Map*). A $r \times n$ linear promotion map $\Pi : \mathbf{Z}^n \to \mathbf{Z}^r$ $(r \ge n + 1)$ is defined by the equation:

$$\vec{x} = \Pi' \cdot \vec{u} + \mathbf{1}$$

where $\vec{x} \in \mathbf{Z}^r$, $\vec{u} \in \mathbf{Z}^n$ and $\mathbf{1}$ is a r-vector whose components are all zero except the lowest which is 1. The matrix Π' is defined as:

$$\Pi' = \begin{pmatrix} \mathbf{0} \\ \hline \mathbf{I} \\ \hline 0^T \end{pmatrix}$$

where $\mathbf{0}$ is a $(r - n - 1) \times n$ zero matrix, $\mathbf{I}$ a $n \times n$ identity matrix and 0^T a $1 \times n$ zero (row) vector.

A vector $\vec{u} \in \mathbf{Z}^n$ is said to be *promoted* to $\mathbf{Z}^r$ when mapped in this way.

Definition 1.2 (*Input Vector*). Given a program $P : \mathbf{Z}^n \to \mathbf{Z}^m$, a valid input vector is an element of $\mathbf{Z}^n$ whose components are positive integers. It is this which is then promoted to form the initial state vector. Even though the input components of the state vector may become negative during computation, they must be initially positive. No generality is lost in this restriction (see [3, 7]).

1.2. The Structure of a Computer Program

The structural hint which is derived from the analysis in [23, 25] is now summarised. Firstly partition the test vectors and matrices into parts which act on the input and output variables in the state vector.[3]

$$q_i = (\mathbf{0} \mid q_i') \tag{1.3}$$

Here the first section multiplies the output variables and the second the input variables. Similarly:

$$M_i = \begin{pmatrix} A_i & B_i \\ 0 & C_i \end{pmatrix} \tag{1.4}$$

where A_i multiplies the m output variables in the state vector, and B_i and C_i are $m \times (n+1)$ and $(n+1) \times (n+1)$ respectively. Further restrictions are also placed on the A_i:

$$\begin{aligned} &A_i = \text{Diagonal}\{0, 1\} \quad \text{or} \\ &A_i \text{ is an } m\text{-dimensional permutation matrix.} \end{aligned} \tag{1.5}$$

It is then assumed that the following conditions apply to the program:

- The matrices M_i are block upper triangular as in Eq. (1.4);
- Condition (1.3) applies to all test vectors;
- Each A_i is restricted as in Eq. (1.5) (see Eq. (1.4)).

The structural hint consists of the following elements of a program:

- The number of input, working and output dimensions in the state vector
- The test-tree structure
- The test vectors q_i and q_f
- The bottom block of each M_i, i.e. C_i in Eq. (1.4).

Note that this definition permits the structure of any computer program (expressed in decision tree format) to be extracted automatically.

2. Constructing L_u

This section recalls the basic definitions which are used both in [25] as well as the further development of L_u presented here.

Definition 2.1 (*Path*). Given a program $P : \mathbf{N}^n \to \mathbf{N}^m$ expressed in terms of the decision tree model, the list of matrices P applies to a particular input vector $\vec{x}_0$ to compute the corresponding output is referred to as the *path* of $\vec{x}_0$. The number of matrices in this list is the *length* of its path. Thus points for which the computation never halts have infinitely long paths.

Note that each matrix in a path is identified by its index within the matrix choice function of P. Thus, for example, the two paths $M_1M_2M_3$ and $M_1M_2M_2$ are considered different even though M_2 may equal M_3.

Definition 2.2 (*Cluster*). Given a program $P : \mathbf{N}^n \to \mathbf{N}^m$ expressed in terms of the decision tree model, a *cluster* is a set of points (in $\mathbf{N}^n$) which have the same path. The length of a cluster is the length of the paths which comprise it.

Theorem 2.3. *The output of any program is linear within any cluster.*

Proof: See [23] or [25]. □

It is shown in [23, 25] how the locations and paths of all clusters can be determined using the information in the structural hint (Section 1.2). With each matrix in the program constrained to be of the form in Eq. (1.4), L_u must determine each A_i and B_i using interaction with the Oracle as a guide.

To see how this can be done, consider the path of a certain cluster for a program described by[4] $P_1(\vec{x})$. For input $\vec{x}_0$, the sequence of linear transformations applied to it during computation of the corresponding output may be (for example): $M_2, M_0, M_8, M_3, M_1, M_2$. Partition the state vector into input $\vec{u} \in \mathbf{N}^n$ and output $\vec{y} \in \mathbf{N}^m$ components. For the purposes of simplicity, it is sufficient to consider any working variables part of the input components in the state vector.[5] Let $\vec{u}_j$ denote the input dimensions of the state vector after j transformations (M_i) have been applied (similarly for $\vec{y}_j$). Collect together (in order) the indices of any given product of several M_i into a set J. Thus $M_2M_0M_8M_3M_1M_2$ results in $J = \{2, 0, 8, 3, 1, 2\}$. If there are l matrices in the product, the result (M_T) has the form

$$M_T = \begin{pmatrix} \prod_{i=J_0}^{J_l} A_i & \sum_{i=J_0}^{J_l} \left(\prod_{k=J_0}^{i-1} A_k\right) B_i \left(\prod_{k=i+1}^{J_l} C_k\right) \\ 0 & \prod_{i=J_0}^{J_l} C_i \end{pmatrix} \tag{2.1}$$

which has a zero lower left-hand block, and simple diagonal elements. A general expression for the output $\vec{y}_n$ is therefore:

$$\vec{y}_j = \left(\sum_{i=J_0}^{J_l} \left(\prod_{k=J_0}^{i-1} A_k \right) B_i \left(\prod_{k=i+1}^{J_l} C_k \right) \right) \vec{u}_0 \tag{2.2}$$

Each cluster gives rise to at most $n+1$ linearly independent equations of this form[6] and these must be solved to find the A_i and B_i. To find the B_i, observe that they enter linearly in Eq. (2.2). So if the A_i can be found, the B_i follow from a simple matrix inversion. However the A_i enter non-linearly and so a method is required which can solve polynomial equations. This problem is too difficult in general, especially as the A_i are likely to be large.[7] It is for this reason that the A_i are constrained as in Eq. (1.5).

3. A_i is a Permutation Matrix

Requiring each A_i in the program to be a permutation matrix drastically simplifies L_u's task without making it trivial. The next two sections develop a way to exploit the properties of such matrices to discover additional constraints on the exact permutations each A_i may represent. Including the alternative possibility (A_i is diagonal) is deferred until Section 7.3.

3.1. Quartets

The cluster search algorithm (see [23, 25]) finds a number of input points (up to $n + 1$ for an n dimensional input space) which belong to a cluster and the Oracle supplies the corresponding correct output. Note that the algorithm also finds the list of matrices in a cluster's path. Suppose four clusters (C_1, C_2, C_3, C_4) can be found whose paths are generated from four collections of transformations (R_1, R_2, Q_1, Q_2) in the following way:

$$\begin{aligned} C_1 &= Q_1 R_1 \\ C_2 &= Q_1 R_2 \\ C_3 &= Q_2 R_1 \\ C_4 &= Q_2 R_2 \end{aligned}$$

Definition 3.1 (*Quartet*). A *quartet* is a set of four clusters whose paths are generated from four transformations in the above manner.

Note that the transformations R_1, R_2, Q_1, Q_2 need not be single matrices M_i (taken from the program structure). They could be comprised of several (or none) of the M_i collected together. For example, the following constitute a quartet:

$$\begin{aligned} C_1 &= M_1 M_2 M_3 M_5 M_4 \\ C_2 &= M_1 M_2 M_3 M_4 M_5 \\ C_3 &= M_0 M_1 M_3 M_5 M_4 \\ C_4 &= M_0 M_1 M_3 M_4 M_5 \end{aligned}$$

where R_1, R_2, Q_1, Q_2 in this case are:

$$\begin{aligned} R_1 &= M_5 M_4 \\ R_2 &= M_4 M_5 \\ Q_1 &= M_1 M_2 M_3 \\ Q_2 &= M_0 M_1 M_3 \end{aligned}$$

Let the number of linearly independent points found by the cluster search algorithm from within each C_i be N_i ($i \in [1, 4]$). Construct the matrices X_i as follows:

$$\begin{pmatrix} \mathbf{0} \\ X_i \end{pmatrix} = \begin{pmatrix} | & | & \dots & | \\ \Pi(\vec{u}_0) & \Pi(\vec{u}_1) & \dots & \Pi(\vec{u}_{N_i}) \\ | & | & \dots & | \end{pmatrix}$$

The zero block on the left hand side is, if there are m output dimensions, a $m \times N_i$ matrix. Similarly, collect the corresponding output vectors together in a matrix Y_i.

From Eq. (2.2) R_1 and R_2 can be expressed as upper triangular block matrices

$$R_1 = \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{13} \end{pmatrix} \quad R_2 = \begin{pmatrix} R_{21} & R_{22} \\ 0 & R_{23} \end{pmatrix}$$

and similarly for Q_1 and Q_2. The input/output relation of each cluster in the quartet can now be expressed simply using these observations. For the first, C_1, whose path is of the form $Q_1 R_1$:

$$Y_1 = (\mathbf{I} \mid \mathbf{0}) \begin{pmatrix} Q_{11} & Q_{12} \\ 0 & Q_{13} \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{13} \end{pmatrix} \begin{pmatrix} \mathbf{0} \\ X_1 \end{pmatrix}$$

which simplifies to

$$Y_1 = (Q_{11} R_{12} + Q_{12} R_{13}) X_1$$

The other three clusters in the quartet similarly give

$$\begin{aligned} Y_2 &= (Q_{11} R_{22} + Q_{12} R_{23}) X_2 \\ Y_3 &= (Q_{21} R_{12} + Q_{22} R_{13}) X_3 \\ Y_4 &= (Q_{21} R_{22} + Q_{21} R_{23}) X_4 \end{aligned}$$

Recall that as part of the structural hint, each C_i within M_i is given. Thus the matrices R_{13}, R_{23}, Q_{13}, Q_{23} (from Eq. (2.1)) are also known. Suppose matrices

$\alpha_1, \alpha_2, \alpha_3, \alpha_4$ can be found which satisfy

$$R_{13}X_1\alpha_1 + R_{23}X_2\alpha_2 = 0 \qquad R_{13}X_3\alpha_3 + R_{23}X_4\alpha_4 = 0$$

then the outputs can be combined to form

$$\begin{aligned} Y_1\alpha_1 + Y_2\alpha_2 &= Q_{11}(R_{12}X_1\alpha_1 + R_{22}X_2\alpha_2) \\ Y_3\alpha_3 + Y_4\alpha_4 &= Q_{21}(R_{12}X_3\alpha_3 + R_{22}X_4\alpha_4) \end{aligned} \tag{3.1}$$

From Eq. (2.1), Q_{11} and Q_{21} are products made up entirely of the A_i, which by assumption are all permutation matrices. Therefore Q_{11} and Q_{21} are simply permutation matrices too and so are invertible. If $X_1\alpha_1 = X_3\alpha_3$ and $X_2\alpha_2 = X_4\alpha_4$ the above two equations combine together to yield

$$Q_{11}^{-1}(Y_1\alpha_1 + Y_2\alpha_2) = Q_{21}^{-1}(Y_3\alpha_3 + Y_4\alpha_4) \tag{3.2}$$

Expressed in matrix form, the complete set of assumptions for $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ become

$$\begin{pmatrix} X_1 & 0 & X_3 & 0 \\ 0 & X_2 & 0 & X_4 \\ 0 & 0 & R_{13}X_3 & R_{23}X_4 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{pmatrix} = 0 \tag{3.3}$$

so that if a non-trivial solution exists for Eq. (3.3) then a non-trivial relationship may be found between Q_{11} and Q_{21} via Eq. (3.2).

Recall that the blocks C_i in Eq. (1.4) multiply both the input and working variables. Therefore each matrix of X_1, X_2, X_3, X_4 can be divided into two blocks corresponding to the working variables (W) and input components (U):

$$X_i = \begin{pmatrix} W_i \\ U_i \end{pmatrix} \quad i \in [1, 4]$$

Divide R_{13} and R_{23} into corresponding blocks:

$$R_{13} = \begin{pmatrix} R_{13}^a & R_{13}^b \\ R_{13}^c & R_{13}^d \end{pmatrix} \quad R_{23} = \begin{pmatrix} R_{23}^a & R_{23}^b \\ R_{23}^c & R_{23}^d \end{pmatrix}$$

Because all working variables are initially zero, the blocks W_i ($i \in [1, 4]$) are zero and Eq. (3.3) simplifies to:

$$\begin{pmatrix} U_1 & 0 & U_3 & 0 \\ 0 & U_2 & 0 & U_4 \\ 0 & 0 & R_{13}^b U_3 & R_{23}^b U_4 \\ 0 & 0 & R_{13}^d U_3 & R_{23}^d U_4 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{pmatrix} = 0 \tag{3.4}$$

Of course there may well be no non-trivial solutions to Eq. (3.4), in which case no useful relationship between the A_i in the clusters in the quartet can be found. This general procedure which derives quartets can be carried out for many other groups of clusters whose paths are related in some manner. Each group potentially finds a relationship between the matrices A_i under differing conditions, although the basic quartet is sufficient in the examples shown in Section 8.

3.2. *Processing into Equations*

Given a list of paths as found by the cluster search algorithm [23, 25], it is relatively simple and efficient to find all possible quartets. Using these the matrices α_1, $\alpha_2, \alpha_3, \alpha_4$ (Eq. (3.1)) are found simply using standard Gaussian elimination in order to obtain an equation like (3.1) for each.

Equation (3.2) assumes each A_i is a permutation matrix (so that the inverses of Q_{11} and Q_{21} exist), but note that Eq. (3.1) does not. Therefore a more general constraint can be obtained by examining Eq. (3.1) directly. Define L, R, and T as follows:

$$\begin{aligned} L &= Y_1\alpha_1 + Y_2\alpha_2 \quad R = Y_3\alpha_3 + Y_4\alpha_4 \\ T &= R_{12}X_1\alpha_1 + R_{22}X_2\alpha_2 \\ &= R_{12}X_3\alpha_3 + R_{22}X_4\alpha_4 \end{aligned}$$

and so Eq. (3.1) becomes:

$$L = Q_{11}T \quad R = Q_{21}T$$

Let each A_i in the program be either a permutation matrix, or a diagonal matrix with entries 0 or 1. Note the following:

1. The product of permutation matrices is a permutation matrix.
2. The product of diagonal matrices is a diagonal matrix.

3. For any permutation matrix P and diagonal matrix D, there exists a permutation matrix P' and diagonal matrix D' such that $PD = D'P'$.

Consequently, the products Q_{11} and Q_{21} can be expressed as

$$Q_{11} = \mathcal{Q}_{11}\Lambda_1 \quad Q_{11} = \mathcal{Q}_{21}\Lambda_2$$

where $\mathcal{Q}_{11}$ and $\mathcal{Q}_{11}$ are permutation matrices, and Λ_1 and Λ_2 are diagonal matrices (with entries 0 or 1). As the inverse of a permutation matrix always exists (and is, in fact, its transpose), the above equations can be combined to form:

$$\Lambda_2\mathcal{Q}_{11}^T L = \Lambda_1\mathcal{Q}_{21}^T R \tag{3.5}$$

Because the structural constraint supplies the paths which comprise the quartet, it is known which matrices A_i combine to form both Q_{11} and Q_{21}. Therefore the matrices Λ_2, Λ_1, $\mathcal{Q}_{11}$, $\mathcal{Q}_{21}$ in Eq. (3.5) can be produced from any hypothesised assignments to the A_i in the program as permutation or diagonal matrices.

Equation (3.5) expresses the fact that the rows of L and R are simply rearranged versions of each other, with the added possibility that some of these have been set to zero (by Λ_1 or Λ_2). Therefore the information supplied can be compressed into a format which records only whether each row in L and R is entirely zero, is identical to another row in either L or R, or is unique amongst these rows. This can be expressed by constructing two m dimensional vectors $\vec{l}$ and $\vec{r}$ (assuming m output dimensions) which contain labels identifying which rows are identical, zero, or unique. A simple algorithm for doing this is outlined in [23].

The labels in $\vec{l}$ and $\vec{r}$ supply necessary conditions which any hypothesised assignment to Q_{11} and Q_{21} must satisfy. To see this, consider such an assignment and note that because of Eq. (3.5):

$$\Lambda_2\mathcal{Q}_{11}^T\vec{l} = \Lambda_1\mathcal{Q}_{21}^T\vec{r} \tag{3.6}$$

Should this fail for any component, then the chosen assignment to Q_{11} and Q_{21} is incorrect. However, it is possible for a row in L or R to be zero because the corresponding row in T is zero, rather than because of one of Λ_1 or Λ_2. In addition the possibility that rows (and therefore labels) are repeated means that some ambiguity may be present. Hence it is possible for the above equation to hold even though the chosen assignment to Q_{11} and Q_{21} is incorrect.

Therefore the Eq. (3.5) produced by processing quartets provide necessary rather than sufficient conditions for any assignment to the matrices A_i in the program to be correct. An illustrative example of this process is found in [23].

4. The Madura Programming Language

Even with all the information supplied within the structural hint[8] and the extra constraints obtained from quartets, the search L_u has to perform is still too extensive to be practical in all but the simplest problems [23]. Thus additional constraints must be imposed upon the allowed programs to make the search practical.

A related problem is the usability of the decision tree model. It is, as intended, a low-level description and so only simple programs can be expressed concisely in its format. Constructing computable functions in the notation of the decision tree model is analogous to writing programs for conventional computers in assembly code. Such a feat is possible, but becomes rapidly tedious as more complex programs are constructed.[9]

To address both these problems a programming language, coined *Madura*, is developed.[10] Madura is based upon the sequential programming language *Java* [14, 16] and therefore imposes a syntactical structure on a program. Consequently, the task facing L_u is simplified as its search can be restricted to include only syntactically valid Madura programs. Although this restriction reduces the number of possibilities L_u must consider in its search, it does not render the mimicry problem trivial (see Section 8).

The translation of Madura source code into the decision tree format is carried out automatically by a Madura compiler.[11] Hence another advantage of Madura is that it facilitates the automatic translation of computer code written in a concise and familiar format into a function expressed in terms of the decision tree model (Section 1.1). The way to achieve this translation is not obvious given the differences between the decision tree model and sequential languages, so the next section examines how the most difficult technical barriers can be overcome.

5. Conventional Sequential Computer Languages

Programs written in sequential computer languages comprise a list of instructions that a computer executes in strict sequential order unless specifically redirected.

The main features of such languages are:[12]

1. Referencing memory locations by name (i.e. variable names).
2. Allowing arrays of variables: i.e. accessing a variable relative to another variable, the offset determined during execution.
3. Functions: re-using code by defining mini-programs which operate on variables defined relative to a "Stack Pointer" variable.
4. Instructions in the code are executed in sequence unless specifically redirected.

To approach the functionality of languages such as Java, Madura must implement at least these features. The decision tree model does not inherently contain them and so it is the task of the Madura compiler to introduce them during compilation. The two essential elements that must be introduced by the compiler to provide this functionality are the simulations of a *Program Stack* and a *Program Counter*.

5.1. Program Counter

To emulate a program counter the compiler can simply reserve a component of the state vector and use it to keep track of the instruction/matrix currently being executed/applied. This is similar to the reserved component of the state vector which is required to contain the unit element during all stages of computation.

To see this process more clearly, examine the following code fragment:

```
Var1 = 7;
Var2 = Var1 - 5*Var3 + Var4;
Var3 = 6*Var5;
```

In a conventional computer, each variable (`Var1` etc.) represents a location in memory allocated by the compiler (for example *javac*). Each of these statements replaces the contents of the left-hand side memory location with the result of the calculation on the right.

In the decision tree model conventional computer memory is analogous to the state vector, thus each variable can be allocated a component (or components in the case of arrays) by the Madura compiler. Given that the last component of the state vector always contains a 1, each of these statements can be implemented as a matrix multiplication of the state vector.

Assume the compiler constructs one matrix for each of the above instructions: M_1, M_2, and M_3. To achieve the effect of these three instructions the state vector must have each of these matrices applied to it in order. If the state vector is $\vec{x}_0$ just before these statements are executed, and $\vec{x}_4$ afterwards, the following must hold:

$$\vec{x}_4 = M_3 \cdot M_2 \cdot M_1 \cdot \vec{x}_0$$

Let the second last component of the state vector be reserved by the compiler to act as the program counter and let each matrix (M_1, M_2, M_3) increment this component by one. In the notation of Section 1.1 the function P_1 becomes:

$$P_1(\vec{x}) = \begin{cases} P_2(\vec{x}) & \text{if } (\ldots, 1, -2) \cdot \vec{x} \leq 0 \\ P_3(\vec{x}) & \text{otherwise} \end{cases}$$

$$P_2(\vec{x}) = \begin{cases} P_4(\vec{x}) & \text{if } (\ldots, 1, -1) \cdot \vec{x} \leq 0 \\ M_3 & \text{otherwise} \end{cases}$$

$$P_4(\vec{x}) = \begin{cases} M_1 & \text{if } (\ldots, 1, 0) \cdot \vec{x} \leq 0 \\ M_2 & \text{otherwise} \end{cases}$$

where P_3 provides a path leading to other instructions in the program. This process can be extended in a simple manner to any number of sequential instructions. Thus the compiler can construct a program tree (P_1) in which the matrix applied is determined purely by the value of the program counter.

If each instruction in the sequential Madura code can be transformed into a matrix multiplication of the state vector, this shows how to emulate sequential computations. As in conventional machines, while most instructions will simply increment the program counter by one, instructions which redirect the flow of control can do so by setting the program counter directly.

5.2. Program Stack

Maintaining the analogy between the state vector and the memory of a conventional computer, the *stack* is simply a contiguous block of components in the state vector. Normally, a pre-defined variable termed the *Stack Pointer* contains the address of the top of the Stack, i.e. the first component in this contiguous block.

The main use of the stack is to facilitate function calling. To execute a function call, the function's arguments and the address of the current instruction are *pushed* onto the top of the stack, and the stack pointer increased by the number pushed on. Control is then set to the first instruction of the function called and execution continues. While executing, the function accesses its arguments by referring to the current value of the stack pointer. When complete, the control is returned to the place (address) from which the call was made, this having been stored on the stack previously. Note that this is only possible if the concept of a program counter exists (see previous section).

Conventionally, pushing values onto the stack affects the state vector (memory contents) as follows:

$$\begin{array}{r} \\ \\ \\ \\ \text{Stack Pointer} \rightarrow \\ \\ \text{Top of Stack} \rightarrow \\ \\ \text{Unit: Always 1} \rightarrow \end{array} \left(\begin{array}{c} 0\\0\\0\\0\\2\\1\\4\\3\\1 \end{array}\right) \begin{array}{l} \\ \\ \\ \\ \text{Push 9 and 7 to Stack} \\ \\ \\ \\ \\ \end{array}$$

$$\Rightarrow \quad \begin{array}{r} \\ \\ \text{Stack Pointer} \rightarrow \\ \\ \\ \\ \text{Top of Stack} \rightarrow \\ \\ \text{Unit: Always 1} \rightarrow \end{array} \left(\begin{array}{c} 0\\0\\7\\9\\2\\1\\4\\3\\1 \end{array}\right)$$

This process cannot be translated directly into the decision tree model, because a function's arguments may be contained in any number of components of the state vector. This is because they are located relative to the value of the stack pointer and so their exact position is known only during execution. However, in the new model, all matrices in a program are fixed and so always access and alter the same elements in the state vector. Thus the components they affect cannot be modified during execution by the current value of the stack pointer.

Fortunately, although the matrices cannot be altered during execution, the state vector contents can. Instead of letting the top of the stack change as the program executes, it can be fixed during compilation to some known location. Now when a value is pushed onto the stack, all the values in the state vector can be shifted up by one and the new value inserted at the (set) location of the top of the stack. This procedure is shown below, where a separate stack pointer is no longer needed:

$$\begin{array}{r} \\ \\ \\ \\ \\ \\ \text{Top of Stack} \rightarrow \\ \\ \text{Unit: Always 1} \rightarrow \end{array} \left(\begin{array}{c} 0\\0\\0\\0\\2\\1\\3\\4\\1 \end{array}\right) \begin{array}{l} \\ \\ \\ \\ \text{Push 9 and 7 to Stack} \\ \\ \\ \\ \\ \end{array}$$

$$\Rightarrow \quad \begin{array}{r} \\ \\ \\ \\ \\ \\ \text{Top of Stack} \rightarrow \\ \\ \text{Unit: Always 1} \rightarrow \end{array} \left(\begin{array}{c} 0\\0\\2\\1\\3\\9\\7\\4\\1 \end{array}\right)$$

The current values in the state vector are *rolled* up by the required amount and the new values inserted. This can be achieved by a single matrix multiplication. When a function is complete, the inverse operation restores the state vector to its original arrangement. This latter operation also sets the program counter component to the return address value stored on the stack.

Note that if the top element of the state vector is a meaningful value, i.e. its contents are the value of a variable in the program, a "Push" operation will overwrite its value (it will be pushed over the top of the state vector). This is an example of stack overflow which is a common problem in stack-based languages. The standard solution is to make the stack so large that it is extremely unlikely.

5.3. *Variable Arrays*

Another important feature Madura must allow is the use of arrays of variables. This is essentially a way to refer to a memory location as an offset from another reference location. In "C" for example, arrays are implemented in exactly this way. As seen above however, because the matrix entries are fixed, only set[13] components of the state vector (memory) may be accessed/modified. It may be imagined that swapping elements of the state vector may avoid this problem as it did when implementing the stack. However this situation is more difficult as the position with which to swap must be determined during execution.

Examine, for example, how the following section of code might be processed:

```
Vehicle[4*i - j*j] = 5;
```

Before the assignment is made, the element which is to be set must be determined. This can be done by performing the calculation in the square brackets and storing the result in a temporary location: t. The array *Vehicle* must then be *rolled t* times so that the appropriate element becomes the first element The assignment can then be made by a fixed matrix which alters this first element, `Vehicle[0]=5`, before the reverse *rolling* operation is performed to restore the array to its original order.

The difficulty in performing this procedure is not theoretical, but rather practical. The code to roll the arrays as above requires on average n matrix applications (n being the size of the array). As the arrays in vision are likely to be large,[14] the number of matrices which must be applied to perform even simple array computations is too large to be practical.

The problem can be minimised by observing that in most programs, including vision algorithms, the most common way to access elements in an array is in sequence:

```
for (i=0; i<10; i++)
      Vehicle[i] = 5;
```

This functionality can be provided in Madura by giving the programmer access to a `roll` function. This function is modelled as a hard-coded Madura language call (such as "System.out.println" in Java). To see how this function operates, let an array `VA` occupy components 2 to 5 in the state vector. The effect of `roll` is as follows:

$$\vec{x}_0 = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 0 \\ 1 \end{pmatrix} \quad \vec{x}_1 = \begin{pmatrix} 2 \\ 5 \\ 4 \\ 3 \\ 0 \\ 1 \end{pmatrix} \quad \vec{x}_2 = \begin{pmatrix} 4 \\ 3 \\ 2 \\ 5 \\ 0 \\ 1 \end{pmatrix}$$

where $\vec{x}_0$ is the initial vector, $\vec{x}_1$ the result of applying `roll(VA,1)` and $\vec{x}_2$ the result after `roll(VA,3)` (or equivalently `roll(VA,-1)`).

The effect of `roll` on multi-dimensional arrays is similar, but applies to the dimension indicated. For example if the variable `V` is declared as:

```
int V[2][3];
```

then `roll(V[0],1)` alters the state vector in the following way:

$$\vec{x}_0 = \begin{pmatrix} V[1][2] \\ V[1][1] \\ V[1][0] \\ V[0][2] \\ V[0][1] \\ V[0][0] \end{pmatrix} \quad \texttt{roll(V[0],1)} \Rightarrow \begin{pmatrix} V[1][2] \\ V[1][1] \\ V[1][0] \\ V[0][0] \\ V[0][2] \\ V[0][1] \end{pmatrix}$$

whereas `roll(V,1)` does:

$$\vec{x}_0 = \begin{pmatrix} V[1][2] \\ V[1][1] \\ V[1][0] \\ V[0][2] \\ V[0][1] \\ V[0][0] \end{pmatrix} \quad \texttt{roll(V,1)} \Rightarrow \begin{pmatrix} V[0][2] \\ V[0][1] \\ V[0][0] \\ V[1][2] \\ V[1][1] \\ V[1][0] \end{pmatrix}$$

Each call to the `roll` function translates simply into a single matrix in the decision tree implementation, as each effectively permutes the contents of the state vector (computer memory). The above loop can now be written efficiently as follows:

```
for (i=0; i<10; i++)
{
     Vehicle[0] = 5;
     roll(Vehicle, 1);
}
```

This construction will be used extensively.

6. A Brief Madura Outline

As Madura is a subset of Java, a Madura program should be mostly familiar to the reader.[15] The reader unfamiliar with programming concepts such as *Declarations*, *Variables*, *Functions*, *Function arguments*, *Statements*, *Keywords* is referred to [14, 16] for an appropriate introduction. A Madura program structure is essentially that of a single Java class.

Madura currently supports the following Java programming statements: `if -then -else`, assignments, addition, subtraction, multiplication by a constant as well as three types of loops. The latter are `while-do`, `do-while` and `for` loops. Function calls are possible with all parameters called by reference, each function returning `void` type. Note that the indices of an array access must be constants (set during compilation).

Because of the latter restriction, Madura supports three built-in functions: `roll`, `zero` and `swap`. The `roll` function is explained above, the `zero` function zeros its argument (in the case of arrays, it zeros all elements of the array) and the `swap` function exchanges the values of its arguments.

In practise the compilation of Madura code is fast and convenient.

7. Consequences of Madura for the Structural Hint

This section describes how the syntax of the Madura language translates into further restrictions on the matrices M_i in the program. L_u can then exploit these extra constraints to restrict its search and consider only valid Madura programs. An important consequence of this is that any solution to the mimicry problem can then be expressed automatically as a Madura program (rather than in decision tree format).

7.1. The A_i Blocks

The structures of the Madura language which control, via the compiler, the format of the A_i blocks are:

1. Any `roll` statements which act on an output variable or array.
2. Any `swap` statements which act on output variables.
3. Any call to a function which takes an output variable (or array) as an argument.
4. Any `zero` statements which act on an output variable or array.

The first three of these translate into permutation matrices, while the last becomes a diagonal matrix with zeros in the appropriate locations.

The simplest way to reduce the possibilities is to consider all blocks A_i fixed and known unless they have been produced by one of the statements above. Even in this case, the possibilities can be reduced sensibly to the following:

1. A `roll` statement: Only those permutations which can be generated by varying the second argument (through all its legal values) are possible.
2. A `swap` statement: Only those permutations which result from replacing either argument with any of the output variables in the current variable context are possible (i.e. in conventional computer language terminology: in *scope*).
3. A function call with an output argument: If the argument is a single output variable, then only the stated possibility is allowed. If the argument is an array, then only those permutations generated by varying the index (or indices in the case of a multi–dimensional array) through its legal values are allowed.
4. A `zero` statement: This is considered fixed and known by L_u.

For example, if `y` is defined as an array of 4 output variables the following Madura statement:

```
roll(y, 1);
```

is compiled into a matrix whose A_i is a permutation matrix. The permutation matrices which could replace this A_i during the search procedure[16] are those which correspond to the Madura statements `roll(y,0)`, `roll(y,1)`, `roll(y,2)` or `roll(y,3)`.

In the case of a `swap` statement, let `y1` be another single output variable currently defined (in scope). The Madura statement:

```
swap(y[1], y[2]);
```

also compiles into a matrix whose A_i is not the default identity matrix. The permutation matrices which could replace this A_i during the search are those corresponding to the following Madura statements:

```
swap(y[0],y[0])  swap(y[0],y[1])
swap(y[0],y[2])  swap(y[0],y[3])
swap(y[1],y[0])  swap(y[1],y[1])
swap(y[1],y[2])  swap(y[1],y[3])
swap(y[2],y[0])  swap(y[2],y[1])
swap(y[2],y[2])  swap(y[2],y[3])
swap(y[3],y[0])  swap(y[3],y[1])
swap(y[3],y[2])  swap(y[3],y[3])
swap(y[0],y1)    swap(y[1],y1)
swap(y[2],y1)    swap(y[3],y1)
swap(y1,y[0])  swap(y1,y[1])
swap(y1,y[2])  swap(y1,y[3])
swap(y1,y1)
```

The final example examines the possibilities arising out of the function call:

```
SomeFunction(y[1])
```

The permutation matrix A_i produced is one which rearranges the output variables so that the arguments are the lowest in the block of output components (in the state vector). This is how the compiler simulates pushing values onto the output stack. The permutation matrices which may replace this A_i are those which would be produced by the following Madura statements:

```
SomeFunction(y[0])  SomeFunction(y[1])
SomeFunction(y[2])  SomeFunction(y[3])
```

Were there two output variables as arguments, the permutation matrix produced by every possible pair of valid indices would be considered for the A_i in question.

It is immediately apparent that the number of possible permutation matrices is considerably less when controlled in this way than when all possible permutations are allowed. In fact, the programmer has some control over the number of possible programs within the structural hint. This number is reduced by limiting function calls with output arguments and the use of `swap` statements. The search is most likely faster than when unnecessary function calls, `swap` and `roll` statements are present in the Madura code.

7.2. The B_i Blocks

The syntax of Madura can also limit the number of non-zero entries in the blocks B_i in each of the matrices (M_i) produced by the compiler. Although once the A_i blocks have been chosen finding the B_i blocks is a linear problem, the size of the matrix is prohibitive if all entries in each block B_i are potentially non-zero (see [23] for an example).

The sensible solution is to assume that all entries in all B_i are zero except those generated from an assignment to an output variable (in the Madura source code). For example, consider the following Madura function:

```
void SomeFunction(in x[4], out y)
{
     int t;
     y = 4*x[1] - t;
}
```

Assuming `y` is stored in the ith output component of the state vector, the assignment to `y` is compiled into a matrix which only has non-zero entries in its ith row. These entries are those multiplying the components of the state vector which correspond to `x[1]` and `t`. When L_u constructs the linear problem to find the B_i, it assumes that all entries in this nth row are potentially non-zero. All other assignments to output variables in the program are similarly treated while all remaining entries in the B_i are considered zero. This restriction again implies that any solution can be expressed in Madura source code automatically.

Thus, in terms of Madura, the freedom assumed by L_u corresponds to determining `A0..A4` in the following:

```
void SomeFunction(in x[4], out y)
{
    int t;
    y = A0*x[0] + A1*x[1] + A2*x[2]
        + A3*x[3] + A4*t;
}
```

i.e. the right-hand side of each assignment is replaced by a linear combination of all input variables currently defined (i.e. in scope).

7.3. Additional Cluster Constraints

Recall how in Section 3 information from the Oracle is combined with the known paths of points to form quartets. The equations produced (3.1) assist L_u by reducing the number of possible assignments to the A_i in the program. This section describes how using the `zero` function can produce additional constraints which assist L_u further.

Suppose the cluster search algorithm finds t clusters $(C_1, C_2, \ldots, C_t)$ whose paths can be described in the following way:

$$\begin{aligned} C_1 = QZR_1 \quad C_2 = QZR_2 \\ C_3 = QZR_3 \ldots C_t = QZR_t \end{aligned} \tag{7.1}$$

Here Z is a matrix whose A_i block is diagonal with entries either 0 or 1, while Q and R_i ($i \in [1, t]$) are collections of matrices as defined by the paths of the clusters. For example, clusters with the following paths conform to this condition:

$$\begin{aligned} C_1 &= M_1M_2M_3M_8M_9M_2 \\ C_2 &= M_1M_2M_3M_8M_9M_4M_5M_4M_3 \\ C_3 &= M_1M_2M_3M_3M_9M_2M_8M_6M_2 \\ C_4 &= M_1M_2M_3M_8M_9 \\ C_5 &= M_1M_2M_3M_7M_9M_6 \end{aligned}$$

where $Q = M_1M_2$, $Z = M_3$ and

$$\begin{aligned} R_1 &= M_8M_9M_2 \\ R_2 &= M_8M_9M_4M_5M_4M_3 \\ R_3 &= M_3M_9M_2M_8M_6M_2 \\ R_4 &= M_8M_9 \\ R_5 &= M_7M_9M_6 \end{aligned}$$

Referring to Section 3.1, assume the cluster search produces N_i linearly independent points $\vec{u}_j$ ($j \in [0, N_i]$) from within C_i and construct the matrices X_i as follows:[17]

$$\begin{pmatrix} \mathbf{0} \\ X_i \end{pmatrix} = \begin{pmatrix} | & | & \dots & | \\ \Pi(\vec{u}_0) & \Pi(\vec{u}_1) & \dots & \Pi(\vec{u}_{N_i}) \\ | & | & \dots & | \end{pmatrix}$$

Here, denoting the output dimension of the program m, the block $\mathbf{0}$ is a $m \times N_i$ block of zeros. Similarly, collect the corresponding output vectors together to form the matrices Y_i.

Recall (Section 3.1) that all matrices in the program are block upper triangular and so the products Q and R_i are also. Hence partitioning Q, Z and R_i into input and output components (working variables considered input components):

$$\begin{aligned} Y_i = (\mathbf{I} \mid \mathbf{0}) &\begin{pmatrix} Q_{11} & Q_{12} \\ 0 & Q_{13} \end{pmatrix} \begin{pmatrix} Z_{11} & Z_{12} \\ 0 & Z_{13} \end{pmatrix} \\ &\times \begin{pmatrix} R_{i11} & R_{i12} \\ 0 & R_{i13} \end{pmatrix} \begin{pmatrix} \mathbf{0} \\ X_i \end{pmatrix} \end{aligned}$$

This simplifies to

$$Y_i = (Q_{11}Z_{11}R_{i12} + Q_{11}Z_{12}R_{i13} + Q_{12}Z_{13}R_{i13})X_i$$

Wherever a Madura program uses the `zero` function on an output variable, the compiler translates it into a simple matrix Z:

$$Z = \begin{pmatrix} \Lambda & 0 \\ 0 & \mathbf{I} \end{pmatrix}$$

where Λ is a diagonal matrix with zeros in the required components. Additionally, the constraints proposed in Section 7.1 require the diagonal matrix Z_{11} to be supplied in the structural hint. Therefore the above equation simplifies to:

$$Y_i = (Q_{11}\Lambda R_{i12} + Q_{12}R_{i13})X_i$$

Suppose now that t matrices α_i can be found which satisfy the following equation non-trivially:

$$\sum_{i=0}^{t} R_{i13}X_i\alpha_i = 0$$

Since R_{i13} is the product of the lower blocks C in the matrices in the program (Section 1.2), every R_{i13} can be found from the structural constraint supplied. Since a non-trivial solution of the above equation exists, the equation below may represent a non-trivial relationship between the output and the unknown elements of the matrices:

$$\sum_{i=0}^{t} Y_i\alpha_i = Q_{11}\Lambda\left(\sum_{i=0}^{t} R_{i12}X_i\alpha_i\right) \tag{7.2}$$

Because R_{i12} is not known, the bracketed term on right hand side of this equation cannot be computed. However the zero components of Λ are known and for these the term on the right is irrelevant. More precisely, if Λ zeros the ith component then the product of Λ with the term on its right will also be zero in the ith component.

This property can be used to obtain information about the matrix Q_{11}. Denote the left hand side of Eq. (7.2) F, and construct a vector $\vec{z}$ whose ith component is zero if the ith row of F is zero and 1 otherwise. Similarly, define a vector $\vec{\lambda}$ whose ith component is the ith diagonal element of Λ.

To assess the validity of a particular assignment to the A_i which form Q_{11} (this list is known from the cluster search), compare $\vec{z}$ to $\vec{w}$:

$$\vec{w} = Q_{11}\vec{\lambda}$$

If a component of $\vec{w}$ is zero and the corresponding component of $\vec{z}$ is not, then the current assignment to the matrices which form Q_{11} is incorrect. However, if a component of $\vec{w}$ is non-zero and the corresponding component of $\vec{z}$ is, then this is not a contradiction. This is because the product Q_{11} may contain yet more matrices produced from other `zero` statements, or because the rightmost term in Eq. (7.2) actually has a zero component.[18]

Nevertheless, this procedure can obtain useful information about the matrices in the program. Furthermore, because the condition the clusters' paths have to satisfy in order to use this procedure is weaker than for quartets (Eq. (7.1)), it is expected that such constraints will be readily found (Eq. (7.2)).

8. Learning to Solve Vision Problems

This section demonstrates the effectiveness of the L_u developed by showing how a number of basic vision processes can be learnt (mimicked) in a matter of seconds. Three classes are considered in the following sections:

1. thresholding
2. edge detection
3. corner detection

Following these, a number of additional algorithms and techniques are examined to show the power of the Madura-based structural constraint and associated L_u. In particular some flexible modules are designed and it is shown how L_u can modify these automatically to mimic the required function.

The results in this chapter are produced using a Madura compiler/debugger/runner environment written in Java. This is run on an *ultra-sparc* computer with a Java JIT (just-in-time) compiler enabled. All the following example images and program searches (by L_u) are each produced in at most a few minutes.

In the examples which follow the Madura source code shown is used as the Oracle as well as the basis of the structural hint supplied to L_u (as augmented with Madura). The Oracle's program is the decision tree representation of the Madura source code as constructed by the Madura compiler. The structural constraint (hint) is produced by firstly using the Madura compiler again to create a separate copy of this decision tree representation. This second copy then has all entries of each matrix which are not part of the structural constraint[19] tagged as unknown. The cluster search as described in [23, 25] is then performed using this incomplete copy as the hint, the correct answers being supplied by the complete copy. Finally, the program search uses this information to reconstruct the incomplete copy of the decision tree implementation.

Note that a link is maintained between those entries of the matrices which are tagged as unknown in the incomplete copy and the Madura source code which produces them. Thus when the program search finds a program which mimics the Oracle over a specified number of clusters, the solution can be expressed automatically in terms of the original Madura source code. A simple check of the overall success is therefore to confirm that this source code is equivalent to the original code compiled to produce the Oracle's program.

8.1. Thresholding

Image thresholding algorithms attempt to quantise each pixel in an image into either 1 or 0. Pixels set to 1 are intended for further processing, while the others are considered background. The result of thresholding is termed a *binary* image [29].

For a 10×10 image, the general outline of a Madura program which performs thresholding is:

```
void main(in x[10][10], out y[10][10])
{
        int i, j, Threshold;

        for (i=0; i<10; i += 1)
        {
                for (j=0; j<10; j += 1)
                {
                        Calculate_Threshold(...);

                        if (x[0][0] > Threshold)
                             y[0][0] = 1;
                        else
                             y[0][0] = 0;

                        roll(x[0], 1);
                        roll(y[0], 1);
                }
                roll(x, 1);
                roll(y, 1);
        }
}
```

where x and y are arrays containing the input and output images respectively.

8.1.1. Bernsen's Method. As outlined in [29] this method initially examines a $r \times r$ locality around each pixel I_{xy} to find the maximum and minimum grey level. The threshold level is then:

$$T = \frac{I_{\max} - I_{\min}}{2}$$

The pixel I_{xy} is then set as follows:

$$I_{xy} = \begin{cases} 1 & \text{if } \|I_{\max} - I_{\min}\| > l \text{ and } I_{xy} > T \\ 0 & \text{otherwise} \end{cases}$$

where l is a contrast measure. In [29] $r = 15$ and $l = 75$ is used. For the purposes here $r = 3$ and $l = 5$ is sufficient.

The Madura code which computes this threshold is:

```
Min = x[0][0];
Max = x[0][0];

for (k=0; k<3; k += 1)
{
     for (l=0; l<3; l+=1)
     {
          if (x[0][0] < Min)
               Min = x[0][0];
          if (x[0][0] > Max)
               Max = x[0][0];

          roll(x[0], 1);
     }
     roll(x, 1);
}
```

where the image is contained in the two-dimensional array `x`, and `Max` and `Min` are local variables.

This implementation is inefficient because most of the computation deals with executing the loops, rather than computing the maximum and minimum. Because the locality is only 3×3 a better version is to write each test explicitly for the whole neighbourhood:

```
Min = x[0][0];
Max = x[0][0];

if (x[0][1] > Max)
     Max = x[0][1];
if (x[0][2] > Max)
     Max = x[0][2];
if (x[1][0] > Max)
     Max = x[1][0];
if (x[1][1] > Max)
     ⋮
```

This version minimises the number of matrices which must be applied (the lengths of computational paths) and so both execution and processing (by L_u) is faster.

An example of this method on a typical optical character recognition image is shown in Fig. 2. Both images are 50×50 pixels and the thresholding takes about 20 s.

The following table shows a summary of L_u processing the structure of the Bernsen detector above. The first column contains the size of the image in pixels, the second shows the time taken (in seconds) for the cluster search [25] to generate 12 clusters and the third the time the program search takes to identify the algorithm.

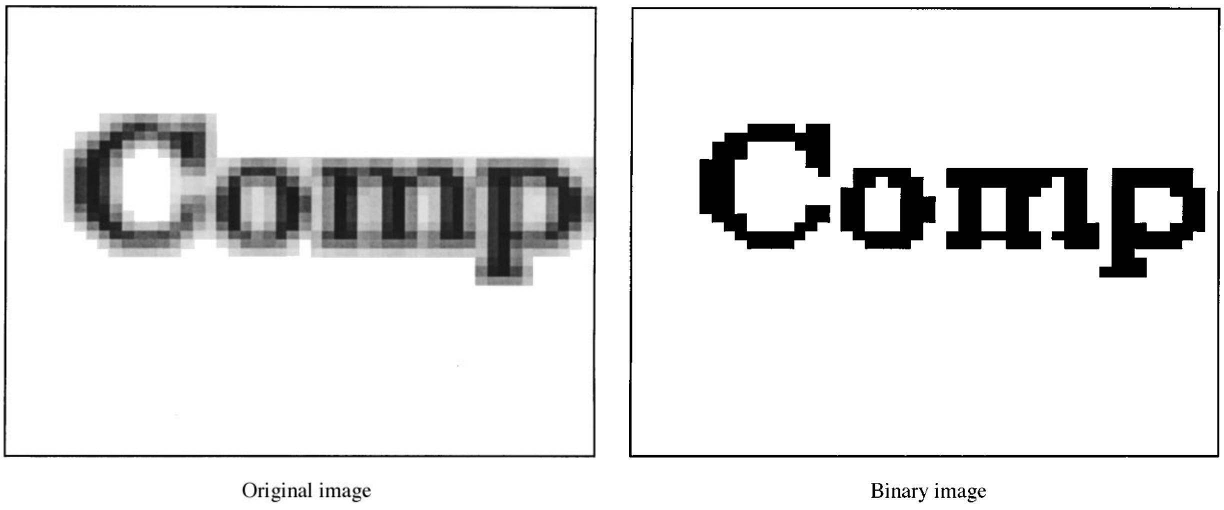

Figure 2. Example of Bernsen's thresholding.

The fourth column contains the total number of possible assignments to the A_i in the program, while the fifth lists the number of quartet constraints found.

Size	Cluster search	Program search	Total	Quartets
3	120	15	81	0
4	300	130	256	2
5	600	300	625	0

Although the main delay is caused by the cluster search, the program search is not efficient as there are rarely any constraints found[20] (Section 3). This can be solved by including `zero` statements in the code. In the threshold test, if the input pixel is below the threshold explicitly call the `zero` function to set the output to zero (rather than simply leaving it unchanged). For example if `y[0][0]` is the output pixel, `x[0][0]` the input pixel and `t` the threshold write:

```
if (x[0][0] > t)
     y[0][0] = 1;
else
     zero(y[0][0]);
```

Now L_u can determine 3 *zero* constraints from the first 8 clusters obtained, and these reduce (in the 4×4 case) the number of assignments to the A_i which need to be processed further to 6 out of the original 256. This reduces the program search time to a couple of seconds. Therefore without altering the program's behaviour, extra constraints can be induced by using the `zero` function.

8.1.2. Niblack's Method. A locality (15×15 in [29]) is examined around each pixel (I_{xy}) and the threshold value is:

$$T = m_{xy} - k \cdot s_{xy} \tag{8.1}$$

where m_{xy} is the mean and s_{xy} the standard deviation of the pixel values in the locality around the centre pixel: I_{xy}.

This is more difficult to translate into Madura because the standard deviation (s) of a series of values I_i ($i = 0..n$) requires numerous products of variables:

$$s^2 = \frac{\sum_{i=0}^{n} I_i^2}{n} - \left(\frac{\sum_{i=0}^{n} I_i}{n}\right)$$

A much simpler alternative to Niblack's method is often sufficient for obtaining a threshold value [8]. This is motivated by the fact that the threshold's value is not important provided it separates the pixels correctly.[21] One simple but popular expression for the threshold is:

$$T = \mu - k(M - \mu)$$

Here k is a parameter (set by experiment), μ the mean value of the pixels in a local neighbourhood around the pixel being thresholded and M their maximum.

This is much simpler to implement in Madura. For example when the neighbourhood is 3×3 and the image is contained in the two dimensional variable `x`, the following code fragment computes the required quantities:

```
Mean = x[0][0] + x[0][1] + x[0][2] +
       x[1][0] + x[1][1] + x[1][2] +
       x[2][0] + x[2][1] + x[2][2];
Max = x[0][0];

for (k=0; k<3; k += 1)
{
       for (l=0; l<3; l+=1)
       {
            if (x[0][0] > Max)
                 Max = x[0][0];

            roll(x[0], 1);
       }
       roll(x, 1);
}
```

Because the variable `Mean` above cannot be divided by n (in this case $n = 25$), the thresholding comparison must be altered to:

```
if (250 * x[0][0] >= 10 * Mean - (25 * Max - Mean))
     y[0][0] += 1;
```

where the parameter k has been set to 0.1. An example of this algorithm is shown in Fig. 3.

The following table shows a summary of L_u processing the structure of the simplified Niblack algorithm above. The first column contains the size of the image in pixels, the second shows the time taken (in seconds) for the cluster search [25] to generate 12 clusters and the third the time the program search takes to identify the algorithm. The fourth column contains the total number of possible assignments to the A_i in the program, while the fifth lists the number of quartet constraints

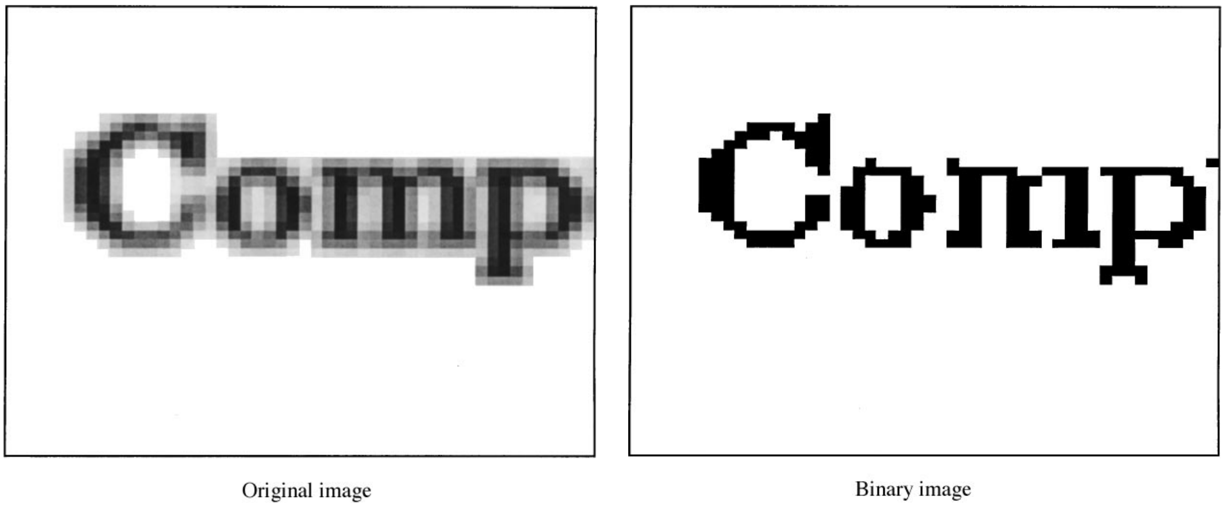

Figure 3. Example of the simplified Niblack thresholding.

found.

Size	Cluster search	Program search	Total	Quartets
4	90	15	16	5
5	210	120	25	0

Again, adding `zero` function calls can induce additional constraints.

8.2. Edge Detection

This is possibly the most popular initial vision processing algorithm, as edges can provide much of the salient information required to compute symbolic information. Many schemes are based upon an ideal edge, where the image intensity is changing rapidly in some direction. Consequently many detectors use some approximation to the local image gradient.

The basic outline of the local edge detectors considered is similar to that of thresholding algorithm in Section 8.1. Each pixel is examined in turn by two nested loops, but as most detectors cannot compute a response on the edge of the image (as local differences are impossible), the `zero` function is used to remove these. Therefore, for a 10 × 10 image:

```
void main(in x[10][10], out y[10][10])
{
      int i, j, Response;
      for (i=0; i<10; i += 1)
      {
         for (j=0; j<10; j += 1)
         {
              Calculate_Local_Edge_Response(...);
              y[0][0] = Response;
              roll(x[0], 1);
              roll(y[0], 1);
         }
         zero(y[0][0]);
         zero(y[0][9]);
         roll(x, 1);
         roll(y, 1);
      }
      zero(y[0]);
      zero(y[9]);
}
```

8.2.1. Simple Edge Detection.

This section examines a Madura implementation of a simple edge detector described in [8]. The response is determined from the first order differences in image intensity `dx` and `dy`:

```
dx = x[1][0] - x[0][0];
dy = x[0][1] - x[0][0];
```

The result, Max$\{\|dx\|, \|dy\|\}$, is computed using the following Madura code fragment:

```
if (dx < 0)
    dx = -dx;
if (dy < 0)
```

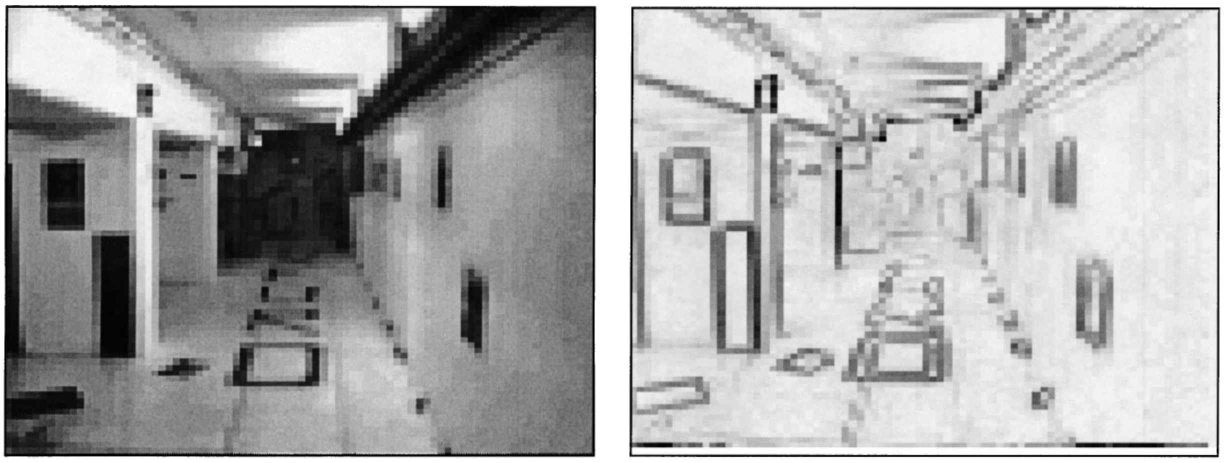

Figure 4. Example of the simple edge detector.

```
        dy = -dy;
    if (dx > dy)
        y[1][1] += dx;
    else
        y[1][1] += dy;
```

An example of this edge detector on an 80 $\times$ 80 image is shown in Fig. 4.

The following table lists the execution times (in seconds) of the cluster search [25] (time taken to get 12 clusters) and the time the program search takes to examine all the possibilities (the total number being in column four). The latter search is simplified if any constraints generated as described in Sections 3 and 7.3 are present. The fifth column lists the number of quartet constraints found (Section 3) and the sixth the number of zero constraints (Section 7.3). The first column indicates the size of the image being processed.

Size	Cluster search	Program search	Total	Quartets	Zeros
3	40	5	9	2	25
4	120	10	16	0	24
5	900	120	25	0	22
6	1200	120	36	0	23

It is clear that the cluster search is the slowest part of the learning process in this case. Another point of interest is that the constraints supplied by the calls to `zero` reduce the number of consistent assignments to the matrices A_i to less than 3 (in all cases above). This explains why the program search is so fast.

An interesting alternative can be explored if the `zero` function is not used. To implement the edge detector without it alter the main loop so that the last pixel on each line, as well as the entire last row are not processed at all. Their value is therefore unchanged from the original of zero. The main loop of the code now becomes in the case of a 5 $\times$ 5 detector:

```
void main(in x[5][5], out y[5][5])
{

    int i, j, Response;
    for (i=0; i<4; i += 1)
    {
        for (j=0; j<4; j += 1)
        {
            Calculate_Local_Edge_Response(...);
            y[0][0] = Response;
            roll(x[0], 1);
            roll(y[0], 1);
        }
        roll(y[0], 1);
        roll(x, 1);
        roll(y, 1);
    }
    roll(y, 1);
}
```

Unfortunately in this case the quartet constraints do not restrict the allowable assignments to the matrices.

The solution is found eventually by counting through all possible assignments to the A_i (in brute-force fashion) and attempting to find the B_i in each case. This is still reasonably fast because the number of possibilities is not vast. The results are summarised in the table below.

Size	Cluster search	Program search	Total	Quartets
4	60	150	256	27
5	120	1750	625	15
6	240	5000	1296	23

This structure has the unexpected property that several programs which mimic the output can be written based on it. In the 5 × 5 case L_u takes up to 5 min to find successive programs which do this, one example being:

```
void main(in x[5][5], out y[5][5])
{
    int i, j, Response;

    for (i=0; i<4; i += 1)
    {
        for (j=0; j<4; j += 1)
        {
            Calculate_Local_Edge_Response(...);
            y[0][0] = Response;
            roll(x[0], 1);
            roll(y[0], 2);
        }
        roll(y[0], 4);
        roll(x, 1);
        roll(y, 4);
    }
    roll(y, 0);
}
```

It is not immediately obvious how this code mimics the required output, but because L_u is not constrained by conventional programming it can find such unintuitive programs. Note in particular that, because the last `roll` function has a zero second argument, it can be removed from the code without altering the behaviour. Thus there may well be prospects for L_u in program optimisation.

The table above also shows that the program search eventually overtakes the cluster search as the number of possibilities increase. One of the main causes is that, in the code above, L_u assumes the right hand side of the assignment to `y[0][0]` can be a combination of all input variables currently in scope. As the whole image is in scope in the function `main`, this number grows quadratically with image size. This problem can be avoided by passing the output variable `y[0][0]` to the function `Calculate_Local_Edge_Response` as an argument. The assignment to `y[0][0]` now occurs inside this function and because only a few input variables are in scope (i.e. only global variables and input arguments to the function), the task of finding the right hand side is much simpler (see also Section 7.2).

The results of L_u processing the edge detector written using the above alteration are now summarised. Note that the `zero` function is not used. For a 4×4 image, the cluster search algorithm takes approximately 20 s to find 12 clusters and 19 quartets. The total number of possibilities (for the A_i) is 4096 and the program search takes approximately 30 s to discover the first identical program shown below:

```
void main(in x[4][4], out y[4][4])
{
    int i, j;

    for (i=0; i<3; i += 1)
    {
        for (j=0; j<3; j += 1)
        {
            Response(x[0][0], x[1][0],
                     x[0][1], y[0][1]);
            roll(x[0], 1);
            roll(x[1], 1);
            roll(y[0], 1);
        }
        roll(y[0], 2);
        roll(x, 1);
        roll(y, 1);
    }
    roll(y, 1);
}
```

A number of alternative solutions exist and L_u discovers them every 30 s. Importantly, the quartets in this example drastically reduce the search time. Every possible assignment to the A_i which passes the conditions imposed by the quartets found is actually a solution to the problem. The constraints obtained by the quartet algorithm therefore save large numbers of unnecessary matrix inversions in futile attempts to find the B_i. The improvement factor, well over 100 in this example, increases dramatically as the complexity of the program increases. The same results for a 5×5 algorithm take only a few more seconds to produce, while a 6×6 version takes about twice as long.

This last version of the edge detector clearly demonstrates the usefulness of the quartet constraints derived by L_u. Their presence makes the search much faster than brute-force enumeration, justifying the background work required to extract them.

8.2.2. Sobel's Method. Sobel's edge detector, one of the first, simply computes the magnitude of the (smoothed) discrete image gradient [8]. A 3×3 neighbourhood of pixels is examined to determine the detector's response at the central pixel. Firstly, the image gradient in the horizontal (S_x) and vertical (S_y) directions are approximated by applying the following convolution masks:

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

The response R at the centre pixel is most often determined by one of the following magnitude operations [8]:

1. $R = |S_x| + |S_y|$
2. $R = \text{Max}\{|S_x|, |S_y|\}$

The Madura code listed in the previous section can compute either of these two.

If the two dimensional array x contains the input image, and y the output image, the Sobel convolutions in Madura are:

```
S_x = x[2][0] + 2*x[2][1] + x[2][2]
        - x[0][0] - 2*x[0][1] - x[0][2];
S_y = x[0][0] + 2*x[1][0] + x[2][0]
        - x[0][2] - 2*x[1][2] - x[2][2];
```

An example is shown in Fig. 5, where the images are both 80×80 pixels large.

Unsurprisingly, the performance of L_u in this case is similar to the previous section. When examining a 6×6 detector the cluster search takes about 400 s to find 12 clusters and 21 quartet constraints can be constructed from these. The result is that only very few of the possible assignments to the A_i in the program are valid. In fact every valid assignment actually represents the Madura code of a program which mimics the required response. The result is that, while searching the 4096 possibilities, valid programs are discovered by L_u once every 120 s.

8.2.3. Smith Edge Detector. This edge detector is again based on a local calculation and a detailed analysis is found in [28]. The value of centre pixel I_{xy} is compared to all others in a locality and an edge response is based upon the number of pixels whose values are greater I_{xy}. In detail, let t be a preset threshold (usually

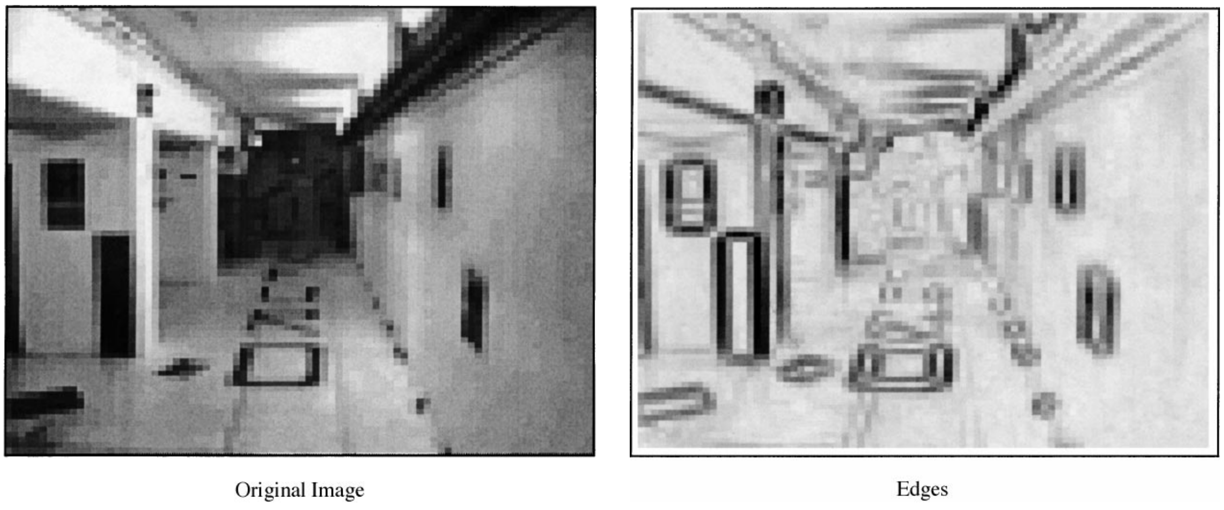

Figure 5. Example of the Sobel edge detector.

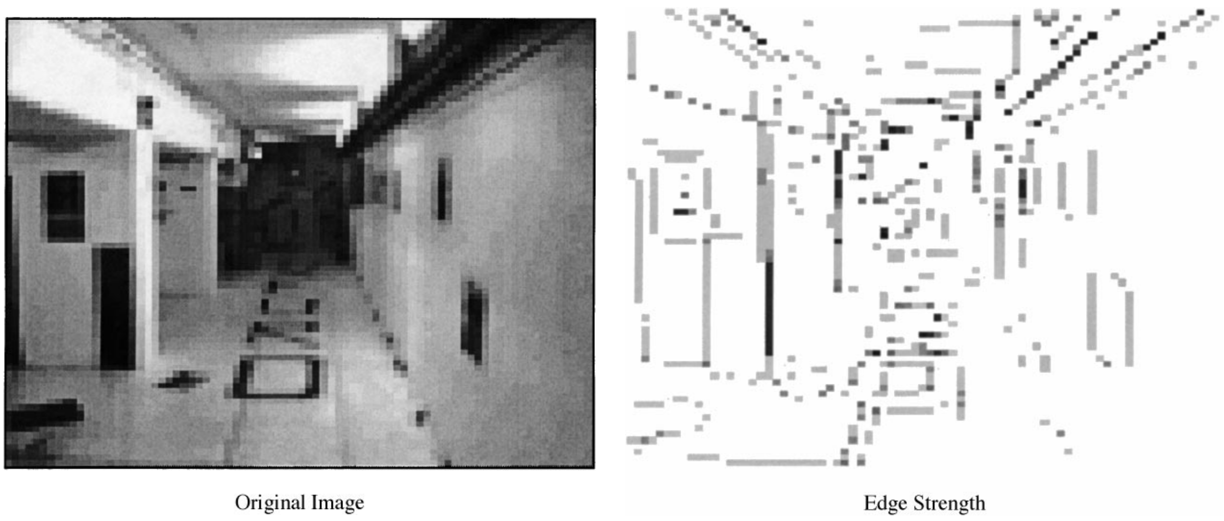

Figure 6. Example of the Smith edge detector.

about 15 for 8-bit images) and define the edge response R_{xy}

$$R_{xy} = \text{Max}\left\{\frac{3}{4}n - \sum_{uv} H(|I_{x+u,y+v} - I_{xy}|), 0\right\}$$
$$H(x) = \begin{cases} 1 & x \geq t \\ 0 & x < t \end{cases} \qquad (8.2)$$

Here the sum over u and v extends throughout the locality considered, usually a circular region of about 3.4 pixels in radius [28]. The parameter n is the total number of pixels in the locality.

A simpler version of this detector uses only a 3 × 3 neighbourhood. The Madura code for comparing the centre pixel to another is:

```
if (x[0][0] - x[1][1] > t)
      C += 1;
if (x[1][1] - x[0][0] < t)
      C += 1;
```

where `C` is a local variable in which the value of the sum in Eq. (8.2) is stored. For the 3 × 3 locality, seven other such comparisons must be made and the edge response computed by:

```
if (4*C < 27)
      y[1][1] += 27 - 4*C;
else
      y[1][1] += 0;
```

An example of this edge detector is shown in Fig. 6. This is produced using the above Madura code, where the input and output images are both 80 × 80 pixels large and the threshold t is 15. When examining a 4 × 4 version of this detector the cluster search [25] generates 12 clusters in approximately 15 min. In the particular Madura implementation used there are 2 valid programs (16 total), both of which L_u finds in about 40 s.

8.3. *Corner Detection*

As summarised in [24], more complex models of local image structure are sometimes preferred to basic edges. The simplest example is a corner detector, where sharp junctions of edges are sought. This section examines the Smith and Moravec corner detectors.

8.3.1. Smith Corner Detector. This detector is a simple extension of the edge detector of Section 8.2.3. The only change is that the response function R_{xy} is now:

$$R_{xy} = \begin{cases} 1 & \text{if } \sum_{uv} H(|\ I_{x+u,y+v} - I_{xy}\ |) < \frac{n}{2} \\ 0 & \text{otherwise} \end{cases}$$

It's implementation in Madura is a trivial change from the Smith edge detector in Section 8.2.3. An example of this Madura implementation is shown in Fig. 7. Both images are 80 × 80 pixels large and the result takes a few seconds to produce. The performance of L_u in this example is almost identical to that when processing the Smith edge detector.

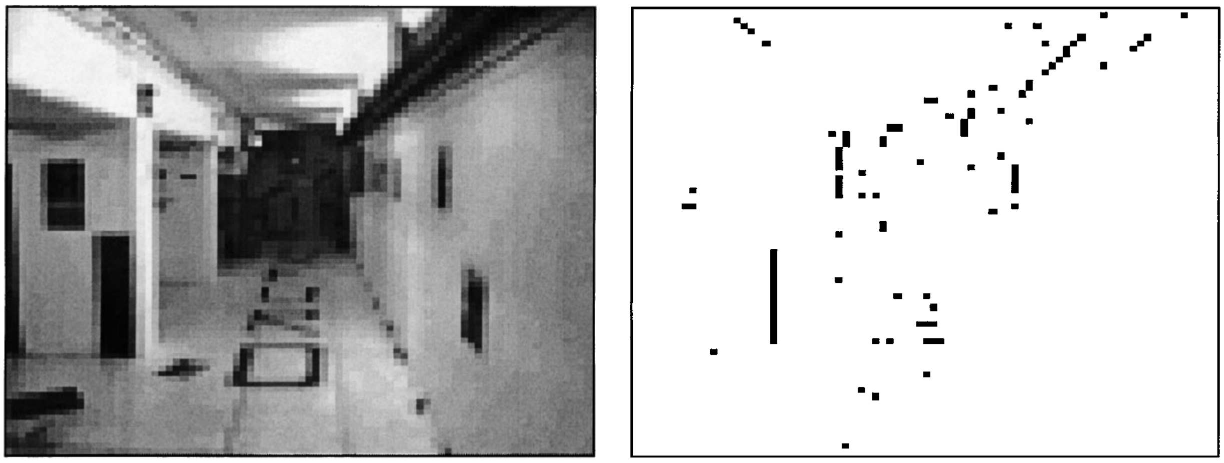

Figure 7. Example of the Smith corner detector.

8.3.2. Moravec Corner Detector. Moravec's corner detector [13] considers a local window around a central pixel I_{xy} and examines the changes in average image intensity that result from shifting the window by a small amount in various directions. If centre of the window is on a corner, then all shifts will result in a large change in this average. Define the average intensity change due to a small shift (x, y) to be E_{xy}:

$$E_{xy} = \sum_{uv} |I_{x+u,y+v} - I_{xy}|^2$$

where u and v range over the locality examined. The only shifts considered are (1, 0), (1, 1), (0, 1), (−1, 1). The central pixel I_{xy} is considered a corner if the minimum of all the E_{xy} calculated for it is above a preset threshold. In fact this detector is the predecessor of the more accurate (but slower) Plessey corner detector [13].

This section presents a simple version of the Moravec detector. The local window is 3 × 3 and the following Madura code computes the intensity changes above (the input image is contained in the array x):

```
E10 = 0;
if (x[2][1] > x[1][1])
     E10 = x[2][1] - x[1][1];
else
     E10 = x[1][1] - x[2][1];
E01 = 0;
if (x[1][0] > x[1][1])
     E01 = x[1][0] - x[1][1];
else
     E01 = x[1][1] - x[1][0];
E11 = 0;
if (x[2][0] > x[1][1])
     E11 = x[2][0] - x[1][1];
else
     E11 = x[1][1] - x[2][0];
E111 = 0;
if (x[0][0] > x[1][1])
     E111 = x[0][0] - x[1][1];
else
     E111 = x[1][1] - x[0][0];
```

Note that E111 denotes $E_{-1,1}$, thus if the minimum of E10, E01, E11, E111 is above the threshold then x[0][0] is a corner.

An example of this Madura implementation is shown in Fig. 8. Both images are 80 × 80 pixels large and the output is produced in about 20 s. Again a the effect of this detector on smaller images can be learnt by L_u in only minutes.

8.4. *Creating Multiply Flexible Modules*

The previous sections demonstrate how a representative set of state-of-the-art vision algorithms can be written in Madura and learnt by L_u effectively. The basic structure of any of these algorithms can be extracted automatically once the Madura compiler has translated it into the decision tree representation. This can then be passed to L_u as demonstrated in the previous sections.

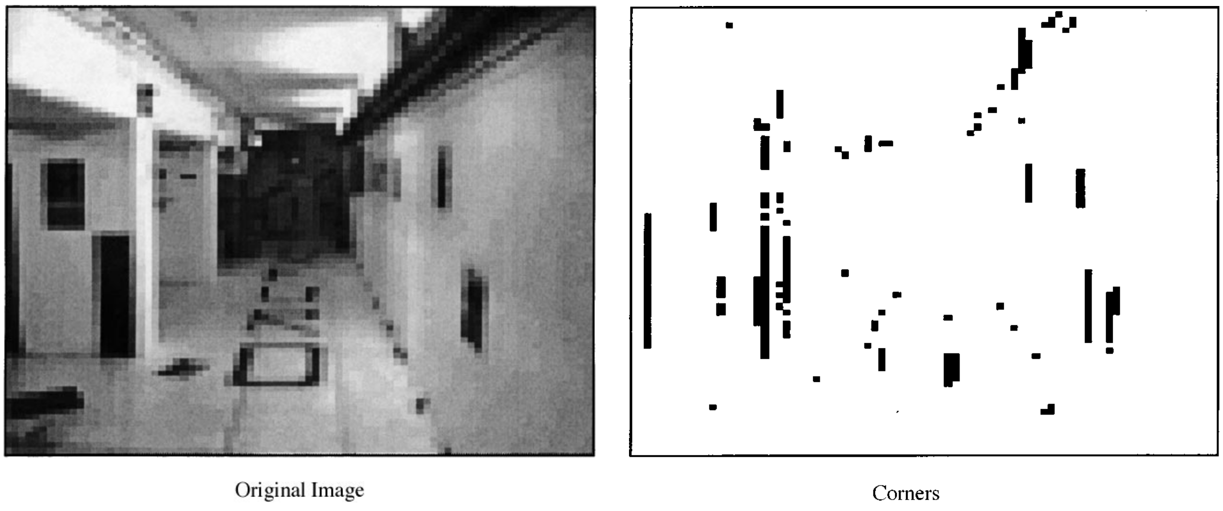

Figure 8. Example of the simple Moravec corner detector.

As outlined in [24] this inherent flexibility may be extremely helpful in creating computer vision systems automatically (via a supervisory system).

The flexibility of the modules constructed previously have relied on that intrinsically introduced by the fact that the source code is written in Madura. However, it is possible to construct a program whose structure can be altered by L_u into a number of known algorithms. Thus these modules may be termed *multiply flexible*. Three such modules are now outlined.

8.4.1. A Flexible Threshold Module. Examine the thresholding algorithms described in Sections 8.1.1 and 8.1.2. Apart from the raster-scan progress through the image they all perform, each bases its threshold on only a few values calculated from a locality around the pixel in question.

Consider a thresholding program which calculates the quantities required by both of these methods (for a 3×3 implementation):

1. The mean of the locality (required by simplified Niblack).
2. The maximum pixel value (required by Bernsen, and simplified Niblack).
3. The minimum pixel value (required by Bernsen).

Instead of a single condition which determines whether a pixel should be set to 1 or 0, imagine a set of nested conditions so that any condition of the three methods can be accommodated. For example the following code can emulate both the simplified Niblack and the Bernsen method:

```
if (Max - Min > 5)
{
    if (2 * (255 - x[1][1]) >= Max - Min)
    {
        if (90 * x[1][1] >= 10 * Mean
            - (9 * Max - Mean))
            y[1][1] += 255;
        else
            y[1][1] += 0;
    }
    else
    {
        if (90 * x[1][1] >= 10 * Mean
            - (9 * Max - Mean))
            y[1][1] += 255;
        else
            y[1][1] += 255;
    }
}
else
{
    if (x[1][1] < 128)
    {
        if (90 * x[1][1] >= 10 * Mean
            - (9 * Max - Mean))
            y[1][1] += 255;
        else
            y[1][1] += 0;
    }
```

```
        else
        {
            if (90 * x[1][1] >= 10 * Mean
                - (9 * Max - Mean))
                 y[1][1] += 255;
            else
                 y[1][1] += 255;
        }
    }
```

The outer two levels of conditionals derive from Bernsen's method, while the inner conditions come from simplified Niblack. Because L_u is able to determine the right hand side of any assignment to an output variable (in this case `y[1][1]`), it can modify this structure to mimic whichever method the Oracle is executing. Importantly, L_u is also capable of constructing novel blends of these methods in an attempt to mimic the Oracle.

8.4.2. A Flexible Edge Detector. This section describes a flexible module which L_u can alter to create the following edge detectors:

1. The simple detector outlined in Section 8.2.1.
2. The Sobel detector in Section 8.2.2, using either of the last two magnitude operations.
3. A linear convolution with a set mask.

The similarity between these detectors can be exploited to define a structure capable of modelling all three. The first in the above list requires a simple local pixel difference (in the *x* and *y* directions, while the Sobel detector requires a more complex version. These can be provided by four local variables:

```
Sobeldx = x[2][0] + 2*x[2][1] + x[2][2]
        - x[0][0] - 2*x[0][1] - x[0][2];
Sobeldy = x[0][0] + 2*x[1][0] + x[2][0]
        - x[0][2] - 2*x[1][2] - x[2][2];
Simpledx = x[2][1] - x[1][1];
Simpledy = x[1][2] - x[1][1];
```

The edge response is set by a call to the function `Response`, which has these variables passed to it as parameters (together with `x`, the central pixel of the 3×3 neighbourhood).

```
    void Response(in x, out y, in dx1,
      in dx2, in dy1, in dy2)
    {
        if (dx1 < 0)
            dx1 = -dx1;
        if (dx2 < 0)
            dx2 = -dx2;

        if (dy1 < 0)
            dy1 = -dy1;
        if (dy2 < 0)
            dy2 = -dy2;

        if (dx1 > dy1)
        {
            if (dx2 > dy2)
               y += dx2;
            else
               y += dy2;
        }
        else
        {
            if (dx2 > dy2)
               y += dx2;
            else
               y += dy2;
        }
    }
```

Because L_u can determine the right hand sides of all the above assignments to `y`, it can alter this code to behave like any of the edge detectors above. In addition, a vast number of (potentially) non-linear combinations of these algorithms can also be found by L_u based on the above structure.

8.4.3. Flexible Morphology. This section develops a flexible module which is capable of performing a number of morphological processes. The reader is referred to [20] for an introduction to the morphological operators used in computer vision.

A general morphological structure can be constructed based on the fact that dilation and erosion operations are based on the maximum and minimum respectively of pixels in a small locality. To perform morphological operations using a 2×2 square structuring element, define a local array of 4 values `Values` and assume the input image is contained in the array `x`. The following code orders the values of the image contained within the structuring element using a *bubble-sort* method:

```
Values[0] = x[0][0];
Values[1] = x[1][0];
```

```
Values[2] = x[0][1];
Values[3] = x[1][1];

for (Swapped = 1; Swapped > 0; )
{
     Swapped = -1;

     for (k=0; k<3; k += 1)
     {
          if (Values[0] < Values[1])
          {
              Swapped = 1;
              swap(Values[1], Values[0]);
          }
          roll(Values, 1);
     }
     roll(Values, 1);
}
```

The following single assignments to an output pixel (y) can then achieve the associated operations listed:

1. Dilation: `y+=Values[3]`
2. Erosion: `y+=Values[0]`
3. Simple morphological edge detection [20]: `y+= x[0][0]-Values[0]`
4. Better morphological edge detection [20]: `y+= Values[3]-Values[0]`
5. Median Filtering (replace each pixel with the median of the pixels in the locality): `y+=Values[1]`

Note that median filtering is not possible using only dilations and erosions.

Importantly, because each of the above assignments are to the output variable y, L_u is able to determine the right hand side from the examples it requests. Therefore depending on the behaviour of the Oracle, L_u can alter this code into that of an erosion, dilation, median filter, either morphological edge detector shown, or a novel combination of these.

9. Conclusion

This paper addresses the important problem of automatically modifying the basic structure of computer program so that it mimics the behaviour of an Oracle. The Oracle is an entity which, given an input request, responds with the desired output. The formal definition of a program's structure or template is extracted by examining the decision tree model of computation [23, 25] and an algorithm (L_u) is developed which can modify this to mimic the Oracle. The decision tree model allows L_u to determine the most useful questions to pose to the Oracle, the answers to which L_u combines automatically into additional restrictions on the possible solutions it must consider. Although theoretically L_u must search through the whole space of computer programs spanned by the given template, its search is generally more efficient due to these additional constraints (Sections 3 and 7.3).

The new computer language Madura is also outlined to facilitate the translation of familiar computer code into the decision tree format required by the L_u developed. Another advantage of Madura is that, because of its natural syntactical structure, even stronger constraints can be imposed on the possible solutions L_u must consider when attempting to mimic the Oracle. The result is that L_u can mimic several state-of-the art basic vision algorithms in a matter of minutes.

All the examples examined possess inherent flexibility as a result of their translation into Madura, but not all programs L_u can explore using this represent algorithms whose function is well understood by a human programmer. It is programs at these limits which may represent interesting modifications to well-known algorithms, or indeed exotic blends of more than one. Consequently there is good reason to suppose that L_u and the theory behind its design will be of significant service to the continuing efforts of constructing computer vision systems, especially those systems which attempt to solve vision problems automatically [2, 4, 6, 9, 17, 27, 30].

Acknowledgments

Supported by a Hackett Scholarship from the University of Western Australia, and an ORS award from the British Government.

Notes

1. The term L_u is derived from “Universal Learner”.
2. Details of the motivation and the theory behind the work here are found in [23, 25].
3. In this section working variables are considered part of the input components.
4. P_1 is the matrix choice function of the program once translated into decision tree format.
5. The unit element is also considered part of $\vec{u}$.
6. As the output within any cluster is linear and the input space is n dimensional.
7. There is a large number of output components in most vision algorithms.
8. Including the restriction that the matrices A_i in the program be as in Eq. (1.5).

9. Note that constructing programs by hand using this new model is still far more tractable than using Turing machine code or recursive functions, especially for multi-dimensional functions.
10. See [23] for a detailed description of Madura.
11. The detailed development of a Madura compiler is found in [23], to which the interested reader is referred.
12. There are clearly other important features, this list being those of most concern here.
13. Set during compilation rather than execution.
14. Typically an image is an array of pixels in vision problems: most likely over 100 elements.
15. Java is, in turn, similar to "C".
16. Performed by L_u.
17. Refer to Section 1.1 for a definition of the promotion map Π.
18. This would propagate through Q_{11} and imitate an extra zero in Λ.
19. As developed in Section 1.2 and augmented with Madura in Section 7.
20. There are no zero constraints because the `zero` statement is not used in the program.
21. i.e. those of interest from the background.

References

1. A.K. Bhattacharjya and B. Roysam, "Joint solution of low, intermediate, and high-level vision tasks by evolutionary optimisation: Application to computer vision at low SNR," *Neural Networks*, Vol. 5, No. 1, pp. 83–95, 1994.
2. M. Brady, "Forms of knowledge in some machine vision systems," *Philosophical Transactions of the Royal Society: London Series B*, Vol. 352, No. 1358, pp. 1241–1248, 1997.
3. D.S. Bridges, *Computability: A Mathematical Sketchbook*, Springer-Verlag, 1994.
4. K. Cho and P. Meer, "Image segmentation from consensus information," *CVGIP: Image Understanding*, Vol. 68, No. 1, pp. 72–89, 1997.
5. R.I.D. Cowie, "Understanding shape: Perspectives from natural and machine vision," *Image and Vision Computing*, Vol. 11, No. 6, pp. 307–308, 1993.
6. D. Crevier and R. Lepage, "Knowledge-based image understanding systems," *CVGIP: Image Understanding*, Vol. 67, No. 2, pp. 161–185, 1997.
7. N.J. Cutland, *Computability*, Cambridge University Press, 1989.
8. E.R. Davies, *Machine Vision*, Academic Press, 1997.
9. B. Draper, A. Hanson, and E. Riseman, "Knowledge-directed vision: Control learning and integration," *Proceedings of the IEEE*, Vol. 84, No. 11, pp. 1625–1637, 1996.
10. W.E.L. Grimson, *Object Recognition by Computer*, MIT Press, 1990.
11. W.E.L. Grimson, "The intelligent camera—Images of computer vision," *Proceedings of the National Academy of Sciences of the USA*, Vol. 90, No. 21, pp. 9791–9794, 1993.
12. W.E.L. Grimson and J.L. Mundy, "Computer vision applications," *Communications of the ACM*, Vol. 37, No. 3, pp. 45–51, 1994.
13. C. Harris and M. Stephens, "A combined corner and edge detector," in *AVC4*, 1988, pp. 147–151.
14. P. Heller, S. Roberts, P. Seymour, and T. McGin, *Java 1.1 Devlopers Handbook*, SYBEX, 1997.
15. B.K.P. Horn, *Robot Vision*, MIT Press, 1986.
16. J. Bishop, *Java Gently*, Addison Wesley, 1997.
17. A. Jain and C. Dorai, "Practising vision: Integration, evaluation and applications," *Pattern Recognition*, Vol. 30, No. 2, pp. 183–196, 1997.
18. R.C. Jain and T.O. Binford, "Ignorance, myopia, and naiveté in computer vision systems," *CVGIP—Image Understanding*, Vol. 53, No. 1, pp. 112–117, 1991.
19. A. Kak, "Editorial," *CVGIP: Image Understanding*, Vol. 61, No. 2, p. 153, 1995.
20. P. Maragos, "Tutorial on advances in morphological image processing and analysis," *Optical Engineering*, Vol. 26, No. 7, pp. 623–632, 1987.
21. J.D. McCafferty, *Human and Machine Vision*, Chichester, West Sussex, England: Ellis Horwood, 1990.
22. M. Mirmehdi, P. Palmer, and J. Kittler, "Genetic optimisation of the image feature extraction process," *Pattern Recognition Letters*, Vol. 18, No. 4, pp. 355–365, 1997.
23. R.A. Newman, "Automatic Learning in Computer Vision," Ph.D. thesis, Oxford University, 1998. Available online at: ftp://ftp.robots.ox.ac.uk/pub/outgoing/newman/thesis.ps.gz.
24. R.A. Newman, "Automatic learning in computer vision," 1998, in progress.
25. R.A. Newman, "A new model of computation for learning from examples," *Journal of Mathematical Imaging and Vision*, Vol. 11, No. 1, pp. 45–63, 1999.
26. T. Pavlidis, "Why progress in computer vision is so slow," *Pattern Recognition Letters*, Vol. 13, pp. 221–225, 1992.
27. P. Robertson and M. Brady, "Adaptive image analysis for aerial surveillance," *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999.
28. S. Smith and M. Brady, "SUSAN—A new approach to low level image processing," *International Journal of Computer Vision*, Vol. 23, No. 1, pp. 45–78, 1987.
29. Ø. Trier and A. Jain, "Goal-directed evaluation of binarisation methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 17, No. 12, pp. 1191–1201, 1995.
30. Y. Venkatesh, "Some aspects of information processing in biological vision," *Current Science*, Vol. 68, No. 2, pp. 168–184, 1995.
31. R. Vogt, *Automatic Generation of Morphological Set Recognition Algorithms*, Springer-Verlag, 1989.
32. H. Wechsler, *Computational Vision*, Academic Press, 1990.

Rhys Newman completed his Honours and Masters degrees in Applied Mathematics at the University of Western Australia in 1992 and 1994 respectively. He then moved to Oxford where he completed his Doctorate of Philosophy in Robotics/Computer Vision in 1998. He is currently working at the Research School of Information Sciences and Engineering at the Australian National University.