Abstract

Machine interpretation of an image depends on the recognition of shapes and outlines within that image. One way to recognise a shape or outline is to measure how much distortion needs to be applied to a pre-existing template or model to get a good fit to the shape. Two shapes are generally considered the same if they differ only in their scale, rotation and translation. Other differences between shape and template can be quantified by measuring the Procrustes distance in shape space between them. Weights can be applied to the shape model to minimise this distance.

In this project we implement a 'Point Distribution Model' and use it to capture the variability of various test shapes. The shape models are then used to parameterise, by the use of an 'Active Shape Model', outlines both hand-generated and obtained from real images. We investigate the robustness of the algorithms in the face of sample images containing shapes which are scaled and rotated with respect to the model, and in which there is a degree of background noise or clutter.

Acknowledgements

I would like to thank my project supervisor Aleka Psarrou for suggesting this project and for her comments on an earlier draft of this report; Claire Neesham for her help in proof-reading and for her suggestions on writing style; Loot Ltd. for allowing me to use their computing resources at evenings and weekends; and finally David E. Stewart and Zbigniew Leyk for their ‘Meschach’ maths library.

Table of Contents

Modelling the Description of Shapes from Examples

Chapter 1. Introduction

1.1 Object Models

Machine interpretation of an image depends on the recognition of shapes and outlines within the image. There are two general approaches for representing a two-dimensional shape: region-based and contour-based. The region-based approach encodes the space occupied by an object. It suffers from two principal disadvantages: it is susceptible to noise and it cannot deal well with partly obscured objects. The contour-based approach models the boundary of an object as an outline. This approach can deal better with partially obscured objects. However, it can be difficult to compute.

A contour-based model can be constructed by placing landmark points on distinct object features and at points in between. To enable comparison between different instances, the models usually need to be normalised to a canonical scale, rotation and translation. For instance, the two points furthest apart within each model could be superimposed. The total distance between corresponding landmark points in the two normalised shapes provides a measure of the shapes' similarity.

Two shapes are generally considered the same if they differ only in their scale, rotation and translation – their *pose*.* In this case, after normalisation, the landmark points on the two shapes are all aligned and the distance between them is zero.

Deformation can lead to differences between similar objects. The ways in which an outline may deform can be modelled as a statistical variation in the location of the landmark points, and correlations between point movements. For instance, all the points corresponding to a single finger within a hand shape will always move together.

Cootes *et al*. (1991) [1] describe a method for modelling a two-dimensional, deformable, shape. The method models the statistical variation in chord lengths over a set of examples. Capturing the relative sizes of chord lengths within a set of shapes has the advantage that the model is independent of rotation, translation and scaling disparities between shapes, but has the significant disadvantage that it has an $O[n^2]$ computational complexity, where n is the number of points in a sample shape. In addition recreating a shape from a set of chord lengths is not a simple operation.

If instead the shapes are first normalised in size and orientation to a reference shape, variations in parameters due to differences in scaling, rotation and translation become irrelevant. Once this is done, the statistical variation in the coordinates of landmark points around the normalised shapes can be modelled directly. Cootes *et al*. (1992) [2] describe such a model. They extract an 'average' shape and a number of parameters, which quantify the main ways in which a set of training shapes vary from their

* The *shape* of a collection of points is defined as "whatever is left after the effects of rotation, translation and scaling have been ruled out." (Kendall, 1984.)

average. They note that ‘though the labelling of the training set is done manually, finding the mean shape and main modes of variation is automatic.’

1.2 Shape Recognition

Given a shape model and a metric for comparing similar shapes, we have the beginnings of a method for recognising an object in an image. One way to recognise an object using a shape model is to measure how much distortion the model requires to get a good fit to the object’s outline. This assumes that we have already located the object in the image and found its landmark points.

An alternative approach is to start with the mean shape, approximately aligned to the object, and distort it to obtain a better fit. Using this approach there is no need to detect the landmark points. This is the basis for the Active Shape Model (*smart snake*) mechanism first described by Cootes and Taylor [3].

1.3 Project Overview

In this project, we investigate methods for modelling shapes described by labelled points and implement algorithms, similar to that described by Cootes *et al.*, to model various hand-generated test shapes. The shape models are then used to parameterise, and recognise, shapes detected automatically within bit-mapped images using an active shape model. We also investigate the robustness of the algorithms in the face of sample images that contain shapes which are scaled and rotated with respect to the model, and in which there is a degree of background noise or clutter.

The remainder of this paper is organised as follows: section 2 describes the models and methods used in this project. Section 3 describes the programs that were written; section 4 goes on to look at the results that were obtained; and section 5 draws some conclusions. Appendix A contains fuller details of the mathematics involved.

A number of computer programs were written for this project; manual entries can be found in Appendix B and source text in Appendix C.

Chapter 2. Models and Methods

The first section of this chapter describes briefly the models and methods that were used in this project and the ways in which they interrelate. Subsequent sections describe each method in more detail.

2.1 Background

Principal Component Analysis (PCA) is an O[n] method for modelling the statistical variation of landmark points across a set of similar shapes or outlines. From a set of training shapes, it creates a parametric model consisting of a base or mean shape and a number of adjustment components. The adjustment components describe the training set's principal modes of variation from the mean. New shapes can be generated by varying the weights of the various components. The model – known as a *Point Distribution Model* (PDM) – is compact and easy to use. A PDM can approximate any of the shapes in the original training set, and it can be used to generate plausible – and sometimes implausible – new shapes.

The point distribution model can also be used in reverse. Given a sample outline, it is possible to infer the PDM component weights that would be required to generate that outline. The magnitude of the component weights that are required gives a feel for how similar the outline is to the mean. By limiting the weights to 'reasonable' values, one can ensure that the best-fit match to the outline is a feasible shape. This property is used to good effect in an iterative scheme for shape modelling: the *Active Shape Model* (ASM.)

The ASM requires an initial approximate match, so some pre-processing is required to search the image – globally – for likely candidates. Heap [4] suggests the use of a genetic algorithm for this purpose. Given a good initial match, an ASM can recognise and find shapes in 'real world' images, using only the information from a PDM model.

The following sections describe these methods in further detail.

2.2 The Point Distribution Model

The Point Distribution Model or PDM captures the variability of a set of training shapes or digitised outlines in a mathematical model. Normally the outlines are digitised by hand since it is important that they are labelled consistently, such that each vertex corresponds to the same landmark point in each image. This is necessary so that, when the outlines are aligned with each other, equivalent points on different images can be compared.

The outlines of the training shapes are aligned with each other by scaling, rotating and translating to minimise the weighted sum of the squares of distances between corresponding landmark points on the shapes and on a fixed target. This sum is the square of the Procrustes* distance, and corresponds to the *metric* in Kendall's shape space. [5] The result is a matrix that describes the *pose* of the candidate shape – its scale, rotation and translation – with respect to the target.

In aligning the outlines it is possible that some points may show much greater variability in position over the training set than others. Points with a large variability in position are of less value when aligning the outlines, since the purpose of the model is to capture the variation in shapes with respect to an essentially static base shape. A weight, inversely proportional to the variability, can be assigned to each point when computing the alignment metric. Appendix A describes the mathematics involved.

Given the ability to align one shape with another, how does one go about aligning a whole set ? Cootes *et al*. describe one method. To paraphrase:

> *Take the first shape in the set as a target. Then,* ***repeat****: align all the shapes to the target; find the mean aligned shape – make this the new target; normalise the target to a certain scale, orientation and origin;* ***until*** *convergence.*

The normalisation operation is required because the algorithm is ill conditioned: there are more variables than there are constraints. Without normalisation, any small numerical inaccuracies in the scale, orientation and translation of the mean could

* The adjective "Procrustes" refers to the Greek giant who would stretch or shorten victims to fit a bed.

accumulate to give an unpredictable result. Normalisation can be performed either by aligning the target to, say, the first shape in the training set, or by picking an arbitrary scale, orientation and origin. Normalising to a member of the training set has the advantage that the mean acquires a – hopefully – typical pose. One suggestion for an arbitrary scale is to set the centroid size (the square root of the sum of the squared distances of the landmark points from their mean) to unity.

Cootes *et al.* comment that they haven't formally proved that the scheme converges to the same solution whichever shape is taken as initial target, although this appears to be the case. In this project, shapes are normalised by aligning the average shape to a fixed circle of points. This has the advantage that the resulting model can be scaled to any size we wish, simply by changing the size of the circle. In addition, the question regarding which shape should be used for the initial target is avoided by aligning the samples to the circle. It doesn't really matter what shape is used as the initial reference since it is used only to get a rough, starting, alignment of the shapes. Likewise using the circle for normalisation is a straightforward way of aligning to a known orientation, scale and origin.

In order to get the best results, the points on the circle should be labelled clockwise or anti-clockwise according to the principal direction of the points in the training shape. A reversed labelling will still result in a normalised shape but it will be rather small, due to the need for many of the landmark points to be aligned with points on 'the other side' of the circle.

2.3 Principal Component Analysis

One can look on the *n* landmark (x, y) points that describe a normalised two dimensional shape as (just) a *2n*-long list of numbers. These in turn can be looked upon as the coordinates of a single point in *2n*-dimensional space. Similar shapes will be 'nearby' in that space: the points will form a 'cloud.' This is Kendall's shape space.

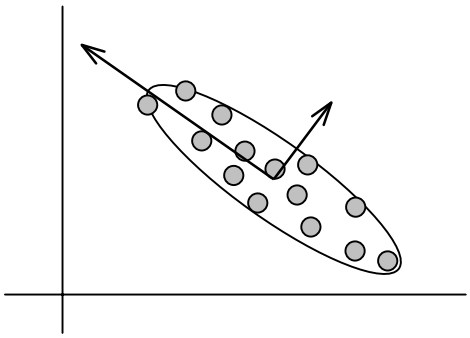

Figure 1: Principal Components of a Cloud of Points

If the positions of two or more points in the shape always vary together, the cloud will be extended into a hyperellipsoid.

Principal Component Analysis is a statistical technique that determines the principal axes of a cloud of points in *n*-dimensional space. PCA assumes that points are distributed with a gaussian probability density function in each dimension. However good results can be obtained even when this is not strictly the case. Figure [1] shows a simple example: applying PCA to a two-dimensional elliptical cloud. The arrows show the direction and magnitude of the cloud's principal axes.

Mathematically the principal components represent a least-squares approximation to the cloud's principal axes. The components are computed as the 2*n* eigenvectors of the covariance matrix $\mathbf{S} = \sum_{i=1}^{m} \partial\psi_i \partial\psi_i^T$. The covariance matrix is also known as the matrix of central moments. The relative weights for the various components are given by the eigenvalues $\lambda_k \geq \lambda_{k+1}$ of **S**; these are in fact the variances of the individual components across the sample data. Normally the weight of a component should be

limited to $\pm 3\sqrt{\lambda}$; i.e. within ±3σ of the mean.* Appendix A covers the mathematics in more detail.

The components are ordered so that the first represents the largest (most significant) variation, the second the next largest, and so on. In general, only the first few of the 2*n* components are useful for modelling; the rest represent noise in the sample data. The minimum number of components that need to be retained to create an accurate model can be estimated by accumulating the weights associated with those components and comparing that sum with the total for all the weights. For instance, if 95% of the variance is captured with the first *t* weights the rest of the weights can be set to zero.

In a perfect model, *t*, the number of components to keep, would equal the number of degrees of freedom in the original shape. Where there are more components than there are degrees of freedom in the model, there will be interdependencies between the components. For instance, errors generated by the first component might be corrected by the forth. In this case it is possible to generate impossible / implausible shapes by assigning weights in atypical proportions.

Note that the PCA scheme does not assign or require any inherent meaning to the data it is analysing; it simply finds correlations. It would be acceptable, for example, to mix Cartesian and polar coordinate information in the same data set. However, if polar and Cartesian coordinates *are* mixed in the same PDM, it is important that they are the same 'scale.' If angles were measured for instance in micro-radians, while

* The standard deviation σ is defined as the square root of the variance λ. A range of ±3σ covers more than 90% of the total variation.

distances were measured in meters, changes in angle would – for most models at least – dominate changes in position. Heap suggests [6] multiplying the angle by the radius as a normalisation step, converting it into a distance around an arc.

2.4 The Active Shape Model

Active Shape Models (ASMs or *smart snakes*) "were originally designed as a method for exactly locating a feature within a still image, given a good initial guess." (Heap, 1995.) An ASM iteratively refines the pose and PDM parameters of a shape – the 'guess' – to obtain the best fit to a shape or outline within an image, whilst ensuring that the guess remains a 'reasonable' shape.

The refinement process operates as follows. Based on the landmark points of the current guess the image is searched locally for object edges, and the landmark points are moved to those edges to generate a new – distorted – guess. The distorted guess is then 'unposed.' This is achieved by applying the inverse of the pose matrix which best aligned the base shape to the current (undistorted) guess.* The base shape is then subtracted from the unposed distorted guess to yield a set of difference vectors. The difference vectors are then fed 'backwards' through the principal component matrix – by multiplying by its inverse† – to obtain a set of PDM component weights.

---

* The least-squares alignment code written for the PCA was used for this purpose; Cootes *et al*. describe a fast approximation that is claimed to be more computationally efficient.

† Note that it is particularly simple to invert the principal component matrix; being a matrix of eigenvalues it is orthogonal, that is $\mathbf{P}^{T}\mathbf{P}=\boldsymbol{I}$; hence to find its inverse we have only to construct its transpose.

The component weights that are obtained are those required by the PDM to generate – exactly – the new, distorted, guess. The weights must be modified to ensure that the next guess's shape remains reasonable. This can be achieved by discarding all but the first few weights, the rest being set to zero. If necessary, the weights that are retained must be reduced pro-rata to ensure that the total distortion is within a 3σ hyperellipse of the mean.

We now have a weight set that can be used to generate a new guess. This will be a reasonable shape since the weights are appropriately small, but it will have been adjusted towards the edges detected in the previous iteration. After some number of iterations, the guess should tend towards the candidate shape.

The main problem with the scheme as described is its reliance on the initial guess. This must be sufficiently close to the outline being modelled that a local search for edges has some chance of success. It also needs to have approximately the correct pose.

To improve the quality of the initial guess, it is evident that some kind of global image search needs to be performed. A good guess needs to have an accurate pose; that is, the scale, orientation and position of the guess should be reasonably accurate. In this project, a straightforward genetic algorithm was used to perform a global image search and find a plausible starting point.

2.5 Genetic Algorithms

In a 'classic' genetic algorithm, as described by Levy, [7], a pool of 'genes' is created; each gene contains a bit-vector encoding all of the parameters that need to be

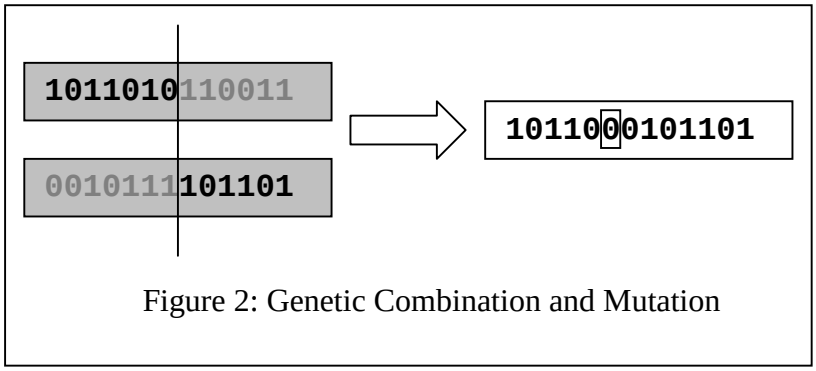

Figure 2: Genetic Combination and Mutation

optimised. The bit-vectors are initially assigned random values and a 'fitness' function is then applied to grade the quality of the genes' parameters.

Next the genes are sorted by fitness, and the best few are crossbred with each other, and possibly subjected to 'mutation', as illustrated in figure 2. These are then used to populate a new pool, and the entire operation is repeated. This continues until either the fitness reaches an appropriate limit, or a fixed number of iterations have passed.

Genetic algorithms are not deterministic – there is no guarantee that a good match will be found – but after many iterations a small number of genes is likely to dominate. The dominant genes will be those with a high fitness value; they have good parameter values that in our case represent good initial guesses for the active shape model's pose. For this application, the 'fittest' gene was taken to create the initial guess.

Chapter 3. Implementation

During the course of this project, programs were written to:

- build a point distribution model from example shapes
- exercise the PDM by generating shapes
- detect candidate shapes in a bitmap image
- adjust the PDM weights to get a best fit, using an active shape model

The programs were written in the C++ programming language [8]. To minimise any platform dependencies, all data visualisation was achieved by having the programs emit PostScript [9] data. This could then be viewed on screen,* printed out on a PostScript compatible printer or embedded directly – with minor editing – as illustrations in this paper.

PostScript's great advantage as a visualisation language is that it is simple and easy to use, with built in support for vectors and bitmaps. The downside is that visualisation is only possible on a batch basis: the programs have to complete before any results can be seen. However, to balance this, the programs are platform and display independent.

The programs were compiled with the Microsoft C++ (VC4.2) compiler on a 133 MHz 'Pentium' PC running the NT operating system. The code is reasonably portable and work is in hand to move the programs to a Unix-based system.

3.1 The Programs

pca.exe reads a text file containing the coordinates of sample shapes. Each point is encoded as a delta from the previous point, and each shape is terminated by a (0, 0) delta. The program performs a Principal Component Analysis of the shapes and writes a `.pdm` file containing the calculated base shape, the Eigen vectors and the Eigen

* using a PostScript viewer written previously by the author – see *http://www.opengate.co.uk/rops/*

values – the point distribution model. In addition, it generates a PostScript file containing various vector representations of the PCA in action. These files form the basis for many of the illustrations in this report.

gen.exe reads the base shape, principal components and component weights from a `.pdm` file generated by *pca.exe*, and generates PostScript drawings of the principal modes of variation, one per page, varying the weights by $\pm 3\sqrt{\lambda_i}$ for each mode.

asm.exe is an implementation of an Active Shape Model. The program reads the `.pdm` file generated by *pca.exe*, and a (binary) bitmap image in Windows `.bmp` format, and attempts to find a shape as described by the PDM data, within the bitmap image. It produces a PostScript file containing images of the ASM in action. Command line switches can be used to control the amount of detail that is drawn, and to configure various genetic algorithm parameters.

3.2 Coding notes

In general, the code is a straightforward interpretation of the underlying mathematics. The programs are necessarily vector- and matrix-arithmetic heavy, so to simplify coding a 3D-algebra C++ library was used [10]. Variable-sized arrays of points (also known as *shapes*) were implemented using the C++ standard template library (STL) and a number of utility functions were written to operate at the shape level. This 'syntactic sugar' allows, for instance, a shape to be scaled, rotated and translated simply by writing '`theShape *= theMatrix`' without worrying (or caring) about the low-level operations involved.

Coding was greatly facilitated by the use of the Meschach software library, which is freely available on the world-wide web as source code. [11] The library provides many common matrix and vector operations, including routines to extract Eigen vectors and variables. It is highly recommended.*

A homogeneous representation is used for points. As Newman and Sproull [12] explain, "A vector in the (n+1) homogeneous space can be viewed as an *n*-space vector with the addition of one more coordinate to the vector, a scale factor." A homogeneous representation allows any translation component to be encoded directly into a 3×3 (pose) matrix, together with the rotation and scale, which greatly simplifies transform handling. Cootes *et al*. had to retain information about the centres of participating shapes to track any translation component, which complicates the mathematics; using homogeneous coordinates means the translation aspect of the model 'comes out in the wash.'

Chapter 4. Experimental Results

4.1 Exercising the PCA Engine

In order to use the PCA program, properly labelled sample shapes are required. Earlier papers on the subject — Cootes *et al.*, 1991, 1992 — contain illustrations of various training sets, and the papers are available in a vector format – PostScript. Hence, in principle, example training sets can be obtained directly from the (PostScript versions

* Interfacing to this library efficiently required a small amount of programming trickery: an array of (x, y) coordinates has the same arrangement in memory as a vector of numbers ...

of) earlier papers. Using existing training data aids the testing process: if a problem is

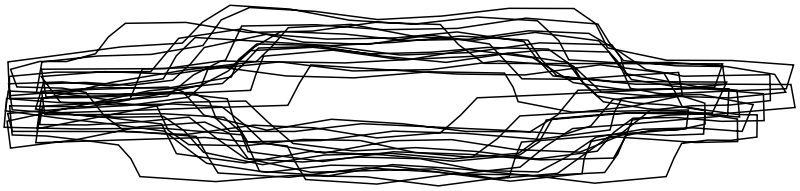

Figure 3: raw resistor data

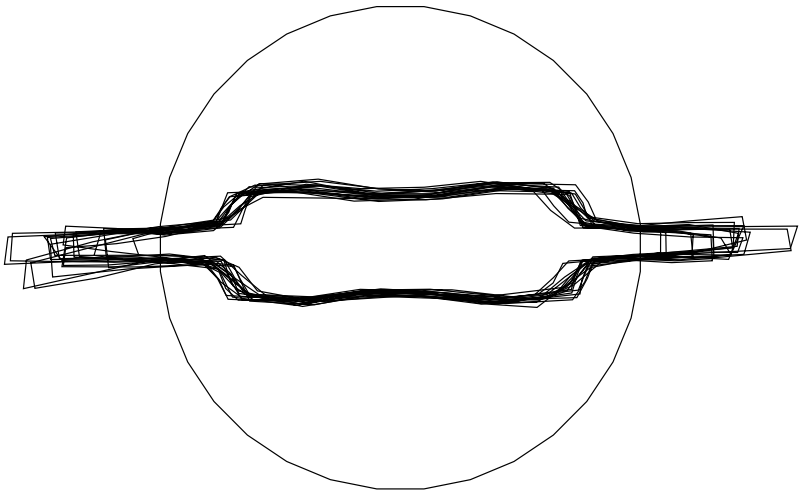

Figure 4: resistor data after normalisation to circle

found, it is unlikely to be due to errors in the input data.

Figure 3 shows the raw data for a resistor example used by Cootes *et al*.; figure 4 shows the same data after normalisation to a circle by the *pca.cpp* program. (The normalisation process was described earlier.) It is clear that the shapes have been scaled, rotated and zoomed to roughly align with each other. Evidence that the alignment is converging to a stable configuration can be found by monitoring the distance that the average shape moves as it is renormalised to the circle after each iteration. The ‘distance’ between two aligned shapes is estimated as the sum of the

squares of the distances between corresponding landmark points. In the resistor example, convergence was rapid; for instance, a typical result would be:

```
distance after pass 0:  9918.07
distance after pass 1:  0.000984473
distance after pass 2:  4.42042e-010
```

Once the shapes are aligned, the Eigen vectors and Eigen values can be calculated; these represent the principal components of the model and their relative contributions. Some confidence can be gained whether a mathematically correct solution has been found by verifying certain fundamental properties of the principal components. For instance $p_i \cdot p_i = 1$ and $p_i \cdot p_j (i \neq j) = 0$ because the components are orthogonal. One can also verify that $\mathbf{S}p_k = \lambda_k p_k$ for each component (where **S** is the covariance matrix, p_k is a component, and λ_k is that component's weight); i.e. each component really is an Eigen solution of the covariance matrix.

The final step in checking that the PCA program 'does the right thing' is to use the

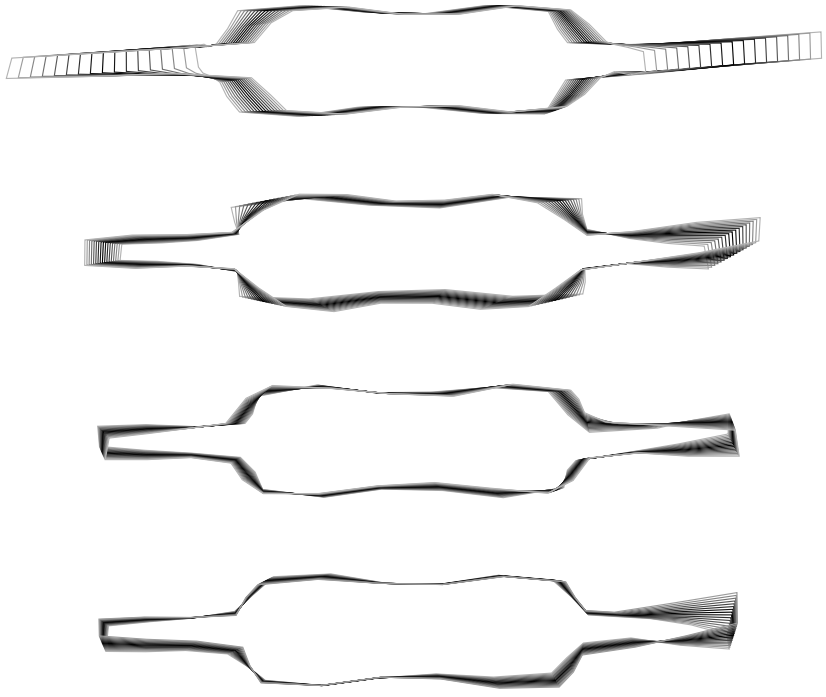

Figure 5: the first four modes for the resistor PDM

model to generate new shapes; *gen.cpp* was written for this purpose. This program varies each of the principal components in turn and draws the shapes so generated. Figure 5 shows the first four modes for the resistor model, with the weights being varied $\pm 3\sqrt{\lambda_i}$. It is evident that the principal variations have been captured, and the

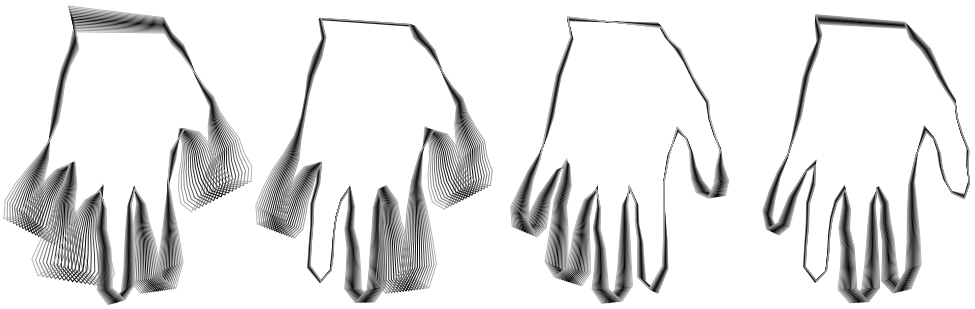

Figure 6: the first four modes for the hand PDM

results compare well to those obtained by Cootes *et al*.

By way of experiment, a PCA was performed on the hand samples from Cootes *et al.* (1992.) Figure 6 illustrates the first four modes of variation that were inferred from that set of outlines. Again, the results compare well with those of earlier researchers. Table 1 shows the weights assigned to each component. It is evident that 90% of the variation is captured in the first 4 modes of the model.

The point distribution model can also generate implausible shapes. Figure 7 shows the effect of varying the first mode of variation further than would normally be reasonable.

Eigen values	Percentage of Total	Cumulative Percentage
125.97	49.66	49.66
74.15	29.23	78.89
19.66	07.75	86.64
13.28	05.23	91.87
8.37	03.30	95.17
4.15	01.63	96.79
2.63	01.03	97.82
2.03	00.80	98.61
1.01	00.39	99.00
0.75	00.29	99.29
0.58	00.22	99.51
0.37	00.14	99.65
0.07	00.03	99.68

Table 1: hand model Eigen values / weights

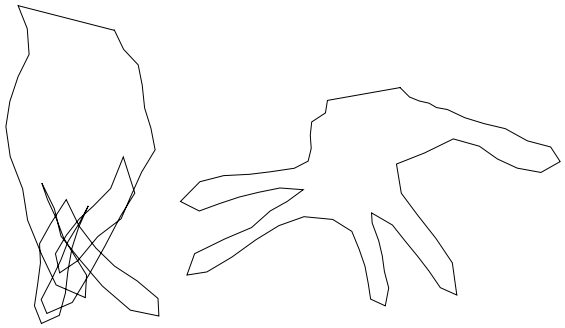

Figure 7: unreasonable weights (±6σ) for first mode in hand model

4.2 Capturing Rotation

A set of training data, representing an Anglepoise lamp, was created using a computer program – *drawlamp.cpp* – the source code for this is listed in Appendix B, together with the source for all the other programs written for this project. Figure 8 shows the basic shape; the angles at the joints marked A, B, and C were varied randomly –

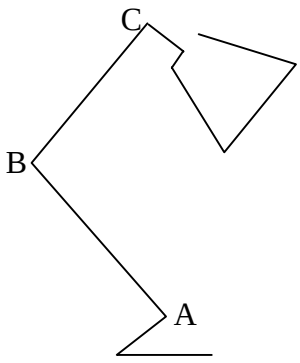

Figure 8: Anglepoise Labelling

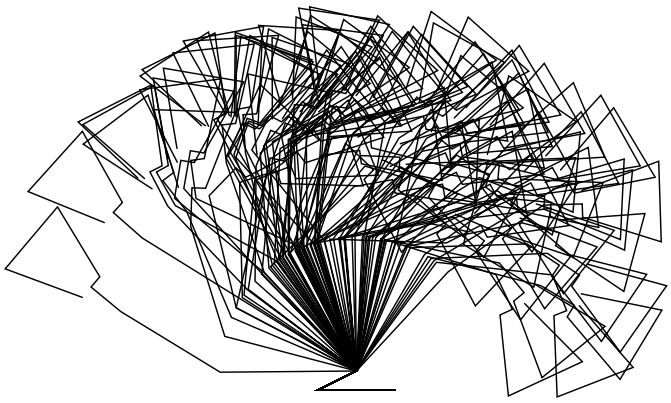

Figure 9: Anglepoise training set

within certain limits – to create a set of 100 training examples. The raw training data is illustrated in figure 9.

The ‘lamp’ training data was passed through the PCA engine; figure 10 shows the resulting primary modes of variation detected within the model. A magnified portion of the first mode is shown in figure 11. Notice that the points in the model move linearly from shape to shape, and that the ‘head’ of the lamp shows significant variation in size as the parameters are varied.

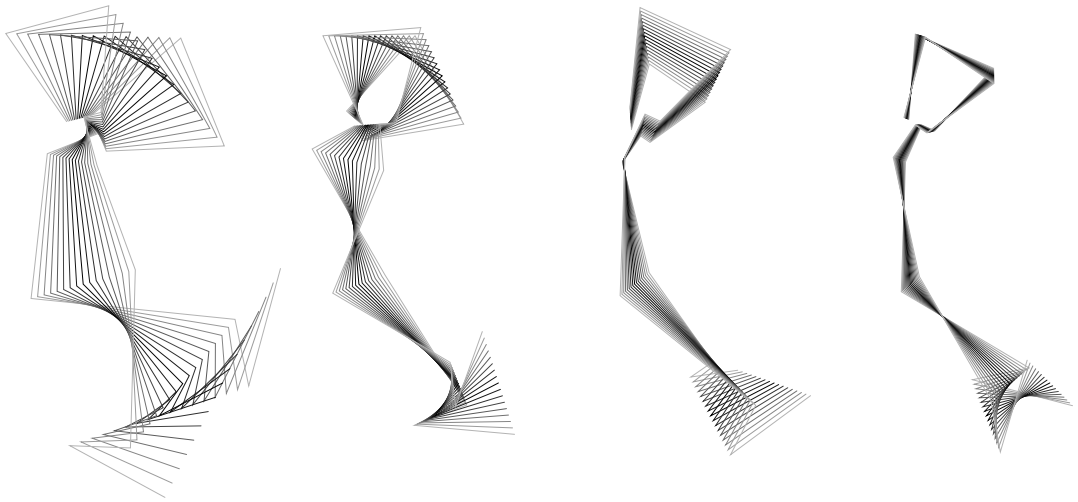

Figure 10: principal components of Anglepoise training set

This linearity is a significant flaw in the model, resulting in many more modes of variation than are strictly necessary. It is clear, for instance, that the primary purpose of much of mode 3 is to compensate for errors in the size of the lamp head caused by inaccuracies in earlier modes.

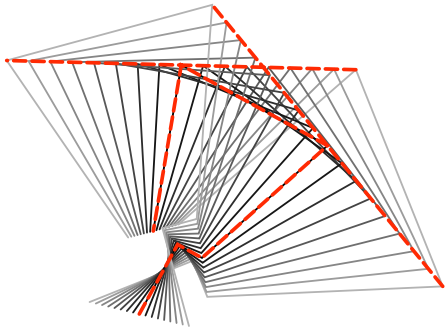

Figure 11: enlargement of part of Figure 10

Heap (1995) describes an extension to the linear PDM in which some parameters are measured in polar coordinates. This allows the modes of variation to describe circular arcs rather than lines. Existing points within the model are used as centres of rotation. As a further extension, Heap suggests a scheme [13] for inferring the centres of rotation for groups of points within the model. These centres are then added as new points in the model.

It is interesting to note that in Heap's work he did not apply an initial alignment process to the training data; the base of the 'lamp' in the models he generated remained stationary. To see what effect this has on a linear model of a shape

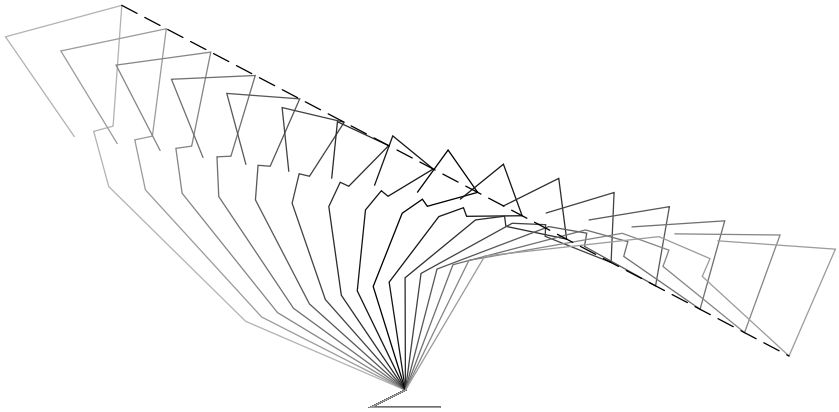

Figure 12: generated Anglepoise lamp, PDM created without initial alignment operation.

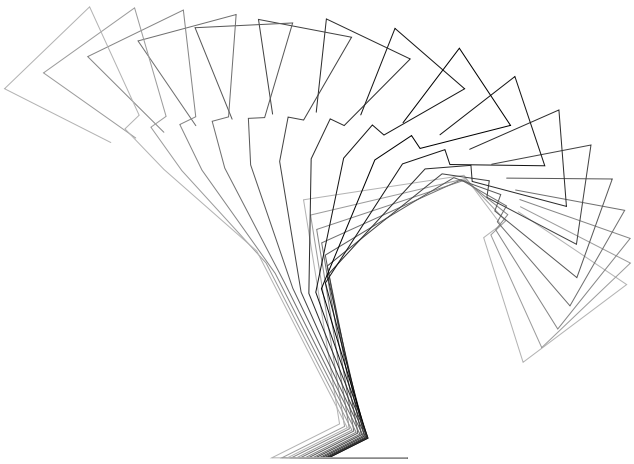

Figure 13: generated Anglepoise lamp, PDM created with aligned shapes.

containing rotation components, consider figures 12 and 13. Both figures show the principal mode of variation of the ‘lamp’ model, but in figure 12 no initial alignment step was performed on the training data: the linear nature of the model is obvious. (Figure 13 contains precisely the same shapes as those illustrated in the first column of figure 10, but in figure 13 they have been rotated and translated to align their bases, to aid visualisation.)

It is evident that the errors in sizing of the lamp head are less pronounced in the second figure. The aligned model has factored the average rotation of the various components of the lamp into the alignment process, hence the need to detect centres of rotation has been reduced.

4.3 Using the PDM to Create an Active Shape Model

One of the most useful properties of the point distribution model is that it can be run backwards: given an outline, weights can be determined which when applied to the model will generate that outline. This property can be used to iteratively refine the fit of a model to a distorted shape. The general approach – the *Active Shape Model* – was described by Cootes and Taylor (1992.)

The active shape model program implemented herein fits outlines to a (bitmap) image. Two separate bitmap formats are used in parallel: Microsoft Windows Bitmap (`.bmp`) and Encapsulated PostScript (`.eps`). The program scans the bitmap image to detect shapes, but the (identical data, different format) EPS image is used to generate displays. This is done for efficiency: the PostScript `imagemask` operator can be used

to draw a reference bitmap in a single operation. The alternative is to illustrate the bitmap by a matrix of tick-marks, which is hugely inefficient. The *Paint Shop Pro** program was used to convert from one bitmap format to another.

To optimise the starting point, the ASM searches the neighbourhood of the current guess to find nearby edges in the image. The search proceeds along vectors normal to the shape outline, from each vertex in turn. Figure 14 shows typical search vectors. When a search vector intersects an edge, the corresponding vertex is moved to the

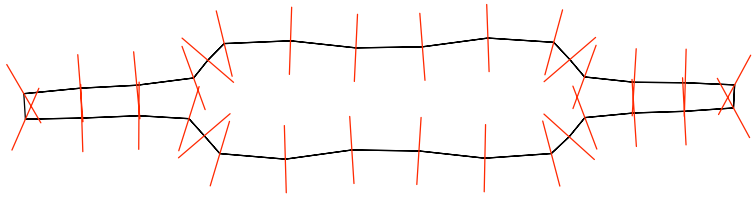

Figure 14: search vectors at landmark points

point of intersection. When all the points have been updated, The pose of the base shape is recalculated to best fit the new set of points. This is repeated some number of times: currently this is fixed at 15 iterations.

To see the iterative pose improver in action, a bitmap image was generated from the outline data describing one of the resistors in the original training set.† Figure 15 shows the result; as can be seen, a good match was obtained. This is not altogether surprising because the resistor was in the initial training set, but it gives confidence that the iterative alignment mechanism is operating correctly.

---

* A bitmap manipulation package — see *www.jasc.com*

† The outline data file was edited to create a PostScript program that drew then filled the outline.

Finally, the PDM parameters can be modified to get a best fit between the approximately correctly posed shape and a bitmap image. Figure 16 shows the result.

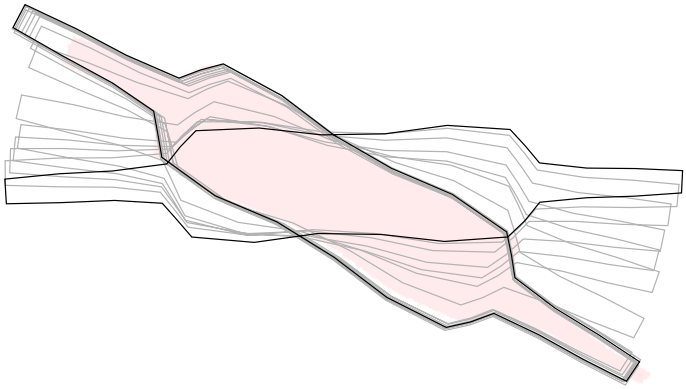

Figure 15: the ASM homes in

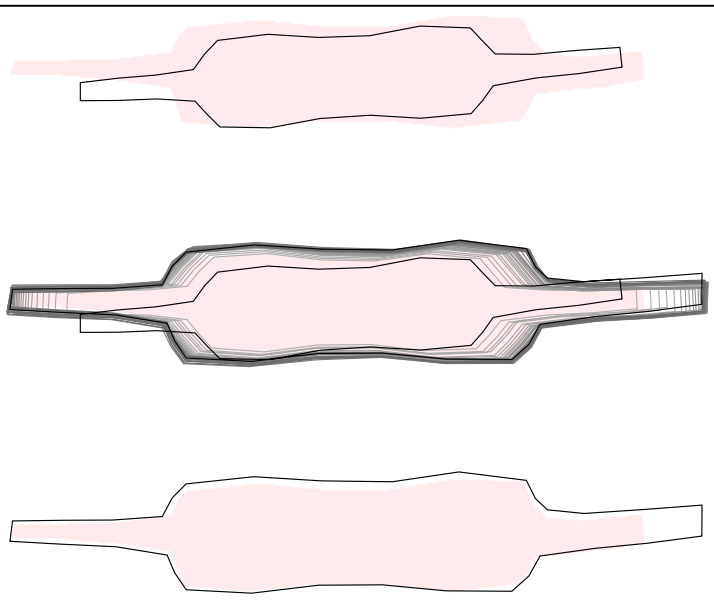

Figure 16: the ASM optimises PDM weights

4.4 Applying the ASM

To test the ASM with a real world example, a number of images of various hands were captured using an *Axis NetEye 200* digital camera. The images were reduced to silhouettes* and saved as bitmaps. Figure 17 shows typical raw images. A PDM was

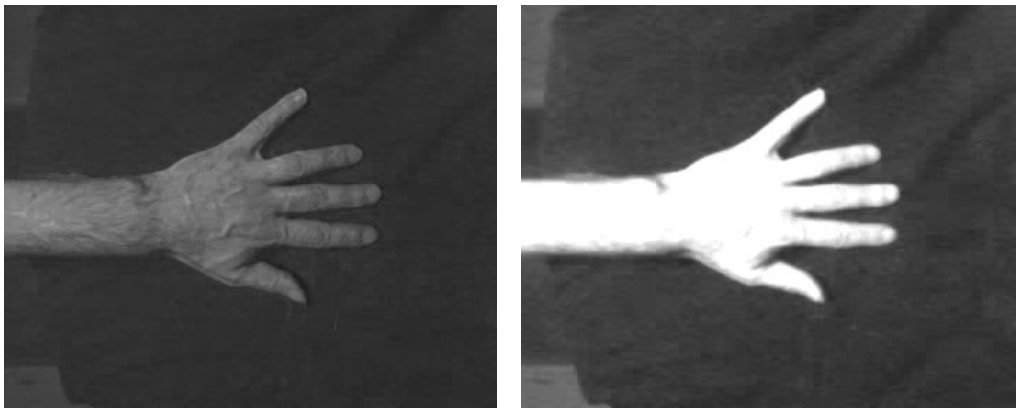

Figure 17: raw captured hand image: greyscale and red separation

constructed from Cootes' hand examples, and various experiments were performed to evaluate the speed and accuracy of the active shape model.

Given a good starting point, the program performed extremely well; but small errors in the initial guess resulted in large errors in the final parameterisation. Figures 18, 19 and 20 show typical results. It appears that the ASM's shape fitting method is too local: all the ASM can do is optimise an existing match.

Notice however that the fitted shape is always a 'reasonable' hand. Only the first ten modes were used to model the shape. These were constrained so that the total

* Using *Paint Shop Pro*. The camera was adjusted for daylight colour balance and hand images were taken on a black background under artificial light. The red channel of the resulting image was reduced to single bit format by a simple thresholding operation.

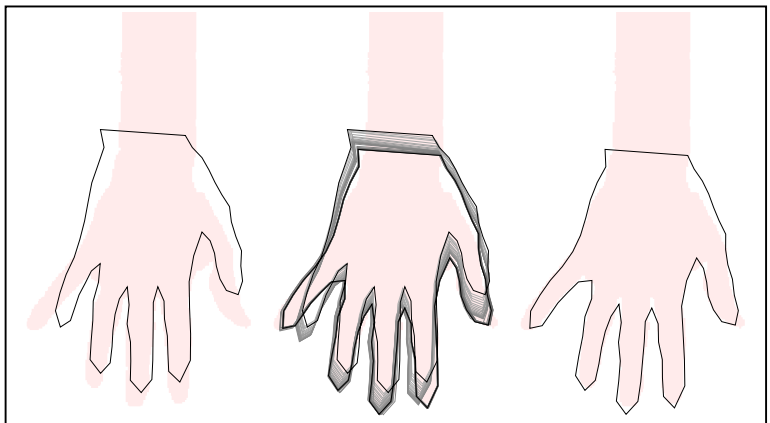

Figure 18: the ASM operating correctly

Figure 19: an extremely bad initial position

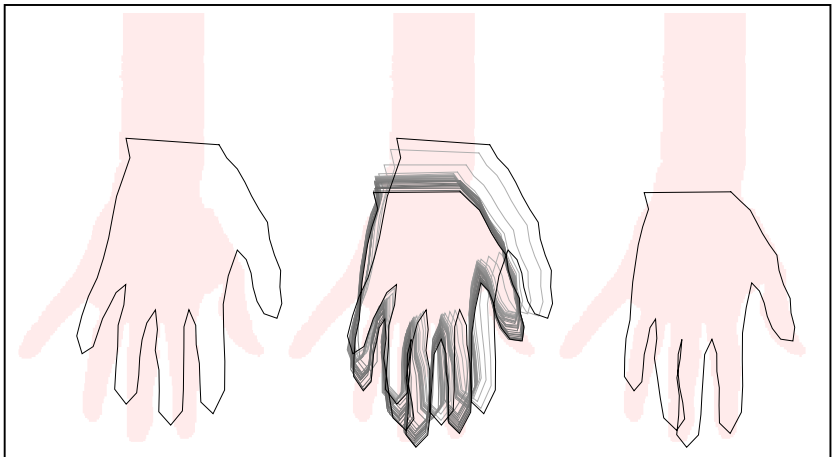

Figure 20: a poor initial position

distortion was within $\pm 3\sigma$ of the base shape. Weights were reduced pro-rata so that $\sqrt{\sum_{i=1}^{n}\left(w_i^2 / \lambda_i\right)} \leq 3$.

4.5 Given a Good Initial Guess

The active shape model iteratively refines an existing guess at a shape. It is, therefore, very dependent on the quality of that initial guess. From the results above with a badly aligned starting position, it was clear that additional programming effort would be required to create a robust recogniser.

A ‘classic’ genetic algorithm was implemented; these are straightforward to code and have a strong probability of Just Working. One alternative approach considered was a variation on the Hough transform; effectively a brute force search for candidate shapes at all likely sizes and orientations throughout the image. This was not implemented because the genetic algorithm turned out to be fast and effective.

The genetic algorithm is encapsulated as the C++ class *dna* in *asm.cpp*. Four 8-bit variables encode the translation, rotation and scale. For ‘breeding’ purposes they are viewed as a single 32-member bit vector. The fitness function is simply the number of points within the shape being evaluated that are within a specified distance of an edge within the image, measured perpendicular to the shape. Many other fitness functions are possible, for instance the distance between points and edges – closer being better – could be used.

4.6 Cross breeders

Initially the gene pool is populated randomly; but subsequently the ‘fittest’ genes must be combined, in pairs, to refill the pool. The obvious scheme, for a pool sized *N*, is to take the first $\sqrt{N}$ fittest genes and create all possible combinations. However this

would mean, for instance, that the combination 2×3 would appear twice while 1×1 – the most successful gene to date – would appear only once. Instead, for a 25-member gene pool, the following combinations were generated (with some proportion of the bits in the vector 'flipped' according to the mutation rate):

```
1×1 1×1 1×1 1×1 1×1
1×2 2×2 2×2 2×2 2×2
1×3 2×3 3×3 3×3 3×3
1×4 2×4 3×4 4×4 4×4
1×5 2×5 3×5 4×5 5×5
```

Hence the more successful a gene, the more times it appears in the next generation.

The genetic algorithm's speed is inversely proportional to the product of the pool size and the number of generations. A number of combinations were tried, with the best cost / benefit ratio – for the hand model on a clean background – being obtained with about fifty genes and about ten iterations. A high mutation rate – 10% – seemed to help; one might expect this to corrupt too many of the genes, but the technique of retaining several combinations of the 'best' genes into the next generation means at least a few of them are likely to live on uncorrupted.

4.7 Getting a Good Spread

The initial, random set of genes should generate a wide variety of plausible guesses for shapes within the image. Shapes which are vanishingly small, or much larger than the image are 'bad' guesses.

In the first incarnation of the C++ *dna* class, the byte variables in the genes were used to encode the scaling factor and rotation angle together as $a_x = s\cos\theta$ and $a_y = s\sin\theta$ avoiding the need for trigonometric functions that are computationally expensive. Values were limited to ± 1.28. However, when both a_x and a_y were very small this resulted in improbably small sizes for guessed images. The problem occurs because although each variable can be small (due to the *sin* and *cos* components) they should not be simultaneously small. An improved spread was obtained when the gene was modified to encode rotation and scale independently, with the rotation 'limited' to 0..360 degrees and the scale to 1 ± 0.25. The overhead of finding the sine and cosine of the rotation angle – required for the rotation matrix – was avoided by creating a lookup table.

Note that there is an implied correspondence in the size of the shapes being generated here and the size of the image being scanned. The average size of the generated shapes is controlled by the size of the normalising circle in the original PDM. The PDM circle has a diameter of 60 units, hence the gene's scaling and translation parameters

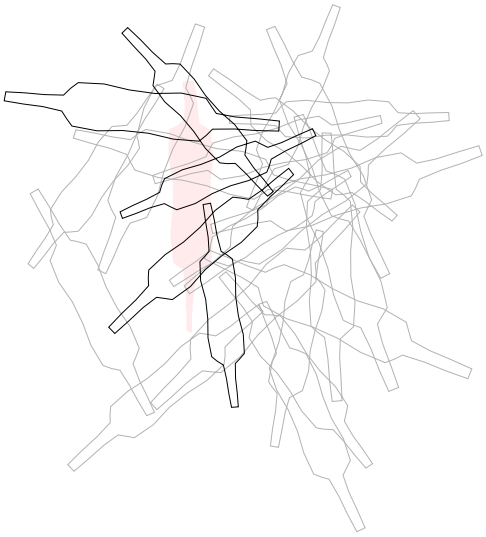

Figure 21: the initial random 'genes'

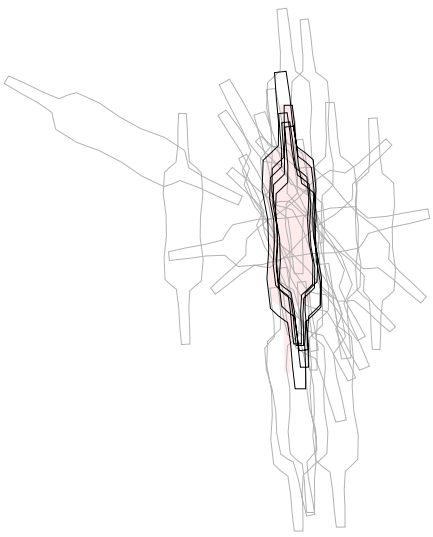

Figure 22: after ten iterations

correspond to a nominal work space (image size) of 120×120 pixels, with a notional feature (object to search for) diameter of 60 pixels ± 25%.

Figure 21 illustrates the initial – random – guesses generated using the resistor PDM. The best five (from a pool of 25) are emboldened. The emboldened few live on to be 'parents' in the next generation. Figure 22 shows the situation after ten generations; it is clear that the algorithm has successfully 'homed in' on the resistor image.

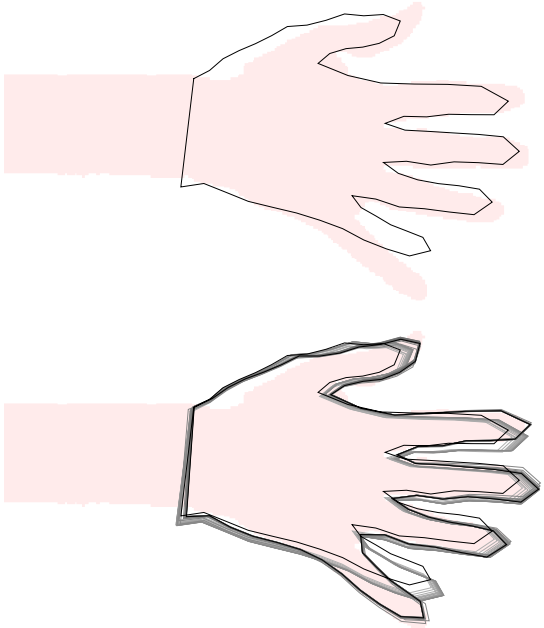

Figure 23: Genetic Algorithm + ASM

4.8 Combining the Genetic Algorithm and the ASM

An earlier illustration (figure 19) showed the poor performance of the ASM given a poor initial guess. Figure 23 shows, for the same image, the starting point found using the genetic algorithm, and the result of using the genetic algorithm and the ASM together. It is clear that the programs are 'working as advertised,' at least when they are presented with a clean image on a clean background.

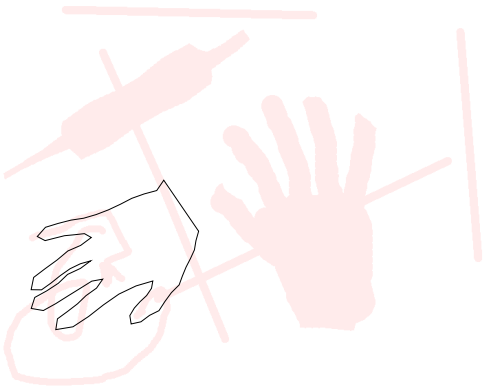

Figure 24: ASM matching to clutter

Further tests were performed using ‘real world’ pictures and hand-drawn approximations. The effect of noise, in the form of lines across the image, was also investigated.

Figure 24 shows a typical test image; both resistor and hand model candidates are present, and there is a fair amount of background clutter, particularly in the bottom left of the image. The resistor is properly detected every time without problem but the hand is not (usually) found. Instead, the clutter provides a better match.

The problem occurs because if the model is positioned in or near to the cluttered area, an edge can always to be found within a reasonable distance. Performance might be improved by taking into account the orientation of the edges that are found, but this has not been investigated to date.

Occasionally the genetic algorithm ‘strikes lucky’ and gets a good fit to the hand within the cluttered image, but, more often than not, it does not. It appears that any early good matches that are found dominate the rest of the search scheme, and hence any early variability is quickly lost. Even with more iterations and a lower mutation rate, the genetic algorithm still suffers the problem; presumably because the pool is repopulated using a large number of the ‘best’ gene. When the *n* best are skimmed off for the next generation they are all (too) closely related to the previous winner.

To investigate this, the ‘cross-breeding’ code was rewritten to use straightforward random combinations of the best genes, with no preference being given to the ‘best of the best.’ With a small pool – 50 genes – and a high mutation rate – 10% – this had no apparent effect. With a large pool – 400 genes – and a low mutation rate – 1%, the hand image was detected even in the presence of background clutter. However, the program took much longer – about thirty seconds – to complete. Figures 25 and 26 show the resulting fit.

There is a danger here: fiddling with parameters to make the program work well with a small number of test images can result in an inadvertent tuning of the program for just those images. It is all too easy to lose generality this way.

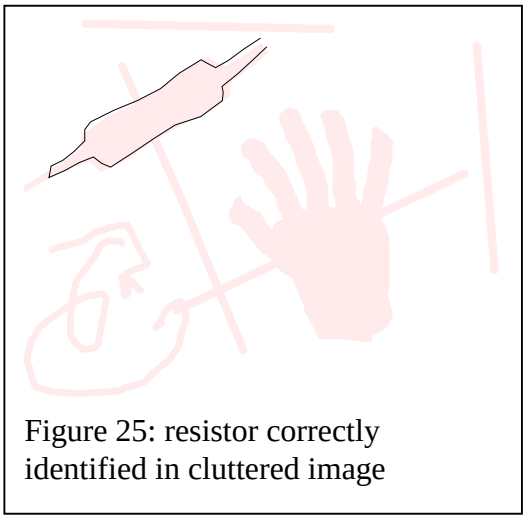

Figure 25: resistor correctly identified in cluttered image

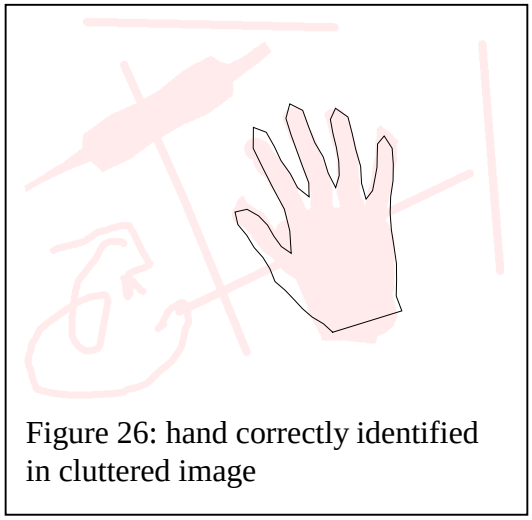

Figure 26: hand correctly identified in cluttered image

4.9 Performance

Performance was not considered a major issue during the creation of the programs described herein; the author’s philosophy has always been (1) make the program work; and then, if necessary, (2) make it work faster. As Bentley [14] explains, it is

notoriously difficult to guess where a program's hot spots are going to be. Speeding up a program may require coding 'hacks' or a whole new algorithm, but in many cases the performance is already 'good enough.' Hence the code described herein has been written for simplicity rather than speed.

In any case the time/space requirements of the PDM/PCA are not particularly important: the programs need only be run once for any training set. The execution time is likely to be insignificant, in comparison to the time taken to construct – by hand – training examples. The ASM code is a different matter: ideally a shape should be recognised – or not – as quickly as possible, which typically means in as few machine instructions as possible. (The genetic algorithms would presumably benefit greatly from a multi-processor architecture since each crossover combination can be tested simultaneously.)

Running the ASM program through a profiler* reveals that during a typical six second run – producing the sequence of images in figure 20 – approximately 55% of the total processing time is spent in reading the PDM data. 13% is spent in the genetic processing portion of the program and 4% in optimising the shape with the ASM. The remainder of the time – 29% – is spent writing the (PostScript) output files.

In terms of individual functions, 28% of the total processing time is spent in the C++ runtime routine which converts a string to a floating point variable; and 14% is spent in the *bitmap::scanForEdge* routine. *dnaSort* consumes rather less than 1% of the total processing time.

---

* The profiler supplied with the Microsoft VC4.2 compiler.

It is clear that the most effective way of speeding up the existing program would be to optimise the input routines. Another option would be to amortise the large start-up cost by reading the PDM information once only and holding it in memory for future recognition tasks.

The *scanForEdge* routine would also be a good candidate for optimisation. At present it makes a (relatively expensive) subroutine call to the *getBit* subroutine for each image pixel that it needs to examine, and uses floating point arithmetic to compute the pixels to check. An integer-based approach would be more efficient, and eliminating the subroutines calls – perhaps by marking *getBit* as `inline` – would also improve things.

scanForEdge calls *getBit* an average of 13.6 times per incarnation, i.e. an edge is on average found within 6 or 7 pixels of a starting point (the checks alternate sides.) The higher the 'hit rate' (i.e. the smaller the average) the better; if no edge is found the scanning route will keep checking until it reaches a pre-set pixel count. (For the statistics given above, the limit was ten pixels.)

In the worst case, there would be no shapes at all within the image being checked. In that case, all scans would miss and a maximum number of program cycles would be consumed for a minimum return. Unfortunately this would be the most common occurrence in a shape recognition system, where a large number of models would need to be applied to the same image to determine which of the shapes appeared within the image – if any.

One specific performance related observation relates to the use of an exact least-squares alignment routine in the ASM code. The routine, which was written for the PDM, finds the best alignment by computing the inverse of a 4×4 matrix. Cootes and Taylor (1992) describe an approximation that instead requires the calculation of a square root for each point in the shape. This 'optimisation' is likely to be slower than the exact method. In any case, according to program profile information, the time spent aligning shapes is completely negligible; the majority of time is spent searching for edges.

Chapter 5. Discussion and Conclusions

The focus of much of the work described herein has been to investigate the properties of the genetic algorithm used to generate good guesses for the active shape model. Both the PDM and the ASM programs appear to be fast, efficient and effective: Once a reasonable guess has been found the ASM very quickly optimises it. The genetic algorithm has turned out to be the weak link in the system for reasons discussed earlier. However, the PDM and ASM also have their weaknesses.

5.1 Weaknesses in the Point Distribution Model

A major disadvantage of the PDM is its need for hand-generated training samples. A great deal of work is required 'up front' to create the model. For clearly delineated shapes, such as hands, this isn't so much of a problem, but for shapes with less obvious landmarks some kind of automatic labelling scheme would be A Good Thing.

One approach to automatic labelling, described by Heap (1995), which could be used if the shapes are sufficiently similar, is simply to position the landmark points at regular intervals around the outline. However as boundary lengths change on different images the points would tend to ‘slide’ around the boundary. Better perhaps would be to position key points at positions of maximum curvature; in a hand model, these would be at the fingertips; one could then place a fixed number of auxiliary points in between. A third option might be to approximate the outline with a cubic spline and use the spline knots as landmarks. Graphics Gems [15] describes a suitable algorithm.

It is important that the landmark points that are used to represent a shape are, in some manner, ‘equally important’ in their contribution to the shape’s parameterisation. McFarlane and Tillett [16] describe fitting a PDM to images of fish; their model had difficulty in ‘sliding along’ the length of the fish once an initial match had been found. They state that ‘despite locating the correct edges ... the small number of tail points could not move the PDM against the resistance of large numbers of landmarks on the body.’ The obvious solution would be to use more points at the tail. However work is required to develop a heuristic that quantifies how many points are necessary to represent a particular outline, and how they should be distributed.

5.2 Weaknesses in the Active Shape Model

The active shape model provides an effective method for fitting shapes to images but it assumes that exactly one shape is visible somewhere in the image. It is ‘looking’ for shapes rather than ‘classifying’ existing shapes. A classification scheme would require multiple ASMs running simultaneously. The existence of a particular shape within an

image would be signalled by its ASM getting a particularly good fit. The problem with there being multiple candidate shapes has yet to be tackled.

To determine the accuracy of fit of an ASM it is not sufficient just to look at the resulting PDM weights. The weights have been limited to ensure that the generated shape is (likely to be) reasonable. Instead, it is necessary to check how close the discovered shape is to actual edges in the image. If the shape model is sufficiently flexible, every point on the outline should correspond to an edge within the image.

5.3 Future Work

Point Distribution Models, Principal Component Analysis and the Active Shape Model are all open-ended topics for research. The straightforward implementations described herein demonstrate that the methods fundamentally work, but to increase their applicability to real world problems additional work would be required. Below, in no particular order, are a few ideas for future work.

- The ASM is highly reliant on a good initial hint. The genetic algorithm used herein is reasonably effective, but it is not guaranteed to find the best result. A multi-resolution search using a variation of the Hough transform could be an alternative approach.

- The PDM depends on hand-generated training shapes. Techniques for automatically identifying landmark points were described in the previous section.

- The images worked with herein were thresholded into silhouettes, and the programs simply scanned for the nearest edge – of the correct sense – that they could find. In general this would not be an effective approach, particularly with a grey-scale or colour image, because the edges are not usually so obvious. Some pre-processing of the image could help to expose the edges: for instance, gaussian blurring followed by a Laplacian edge-detection operation. However it is not clear whether a near, weak edge should be preferred over a far strong one or vice versa. A more complex model could take into account the density of the training images along vectors normal to the landmark vertices.

- The linear nature of the standard Point Distribution Model prevents it from capturing the rotation of a point or points with respect to others within the same model. Compensatory modes are required to fix up errors due to earlier modes having modelled curved movement with lines. Heap and Hogg (1995) describe a method for mixing polar and Cartesian relations between points within the same model, and in a later paper show how centres of rotation can be determined automatically. Other authors have suggested the use of arbitrary polynomials to parameterise point motion.

Another interesting experiment would be to assign weights to the ASM during the genetic algorithm phase, so that the GA captures the essentials of the shape as well as its pose. It would probably take many more iterations to find a good match, because there are many more variables to optimise.

5.4 Summary

The initial aim of this project was to create a Principal Component Analysis system that could be used to capture a distortable object's variability in shape. it would read a set of hand-generated training examples and create a Point Distribution Model of the object.

This aim was accomplished; a flexible PCA was constructed and it was shown to capture the principal variations in shape of both real-world and synthetic data, without any specific 'built in' knowledge of the shapes that it was analysing. One novel aspect of the approach that was taken was to align example shapes to a fixed circle of points rather than to a randomly chosen member of the sample set; this proved to be both simple and effective.

The PDM was used to generate both plausible and implausible variations of the original shape, and we looked at ways of capturing rotation within the constraints of a linear PDM. It had been hoped that the programs could be tested using data extracted from sample *Historia* images; however there were not sufficient recurrent features within the available images to allow for the generation of a meaningful shape model.

The project's aims were then extended somewhat to test the PDM as the input to a shape recognition system. An Active Shape Model (*smart snake*) system was implemented and it quickly became apparent that the ASM is very sensitive to the quality of the initial guess. In an attempt to generate good guesses, a genetic algorithm was implemented. This proved accurate and efficient when used within a controlled environment.

The acid test of a shape recognition system is whether it can recognise shapes. The system described herein performs a global search across a bitmap image for a likely candidate and, in most cases, it successfully captures the nuances of the shape as weights in a parametric model. However, the initial alignment scheme – the genetic algorithm – is prohibitively expensive in terms of computer time when used in a real-world environment; this could be ameliorated by the use of a parallel processor.

There are many applications where a global image search would be unnecessary, or a simpler searching scheme would suffice. For example when tracking an object in a video sequence, the position found in a previous frame would constitute a very good starting point for the current frame. As another example, a quality control system that checked the shape of moulded items could be arranged so the item was always central in the image frame. The programs developed herein would in those cases be useful as they stand.

References

[1] T.F.Cootes, D.H.Cooper, C.J.Taylor and J.Graham, "A Trainable Method of Parametric Shape Description", *Proc. British Machine Vision Conference*, Springer-Verlag, 1991, pp. 54-61.

[2] T.F.Cootes, C.J.Taylor, D.H.Cooper and J.Graham, "Training Models of Shape from Sets of Examples", *Proc. British Machine Vision Conference*, Springer-Verlag, 1992, pp. 9-18.

[3] T.F.Cootes, C.J.Taylor, "Active Shape Models – 'Smart Snakes'", *Proc. British Machine Vision Conference*, Springer-Verlag, 1992, pp. 266-275.

[4] A.J.Heap, "Real-Time Hand Tracking and Gesture Recognition using Smart Snakes", *Proc. Interface to Human and Virtual Worlds*, Montpellier, France, June 1995.

[5] D.E.Slice, F.L.Bookstein, L.F.Marcus and F.J.Rohlf, "A Glossary for Geometric Morphometrics", *http://life.bio.sunysb.edu/morph/glossary/gloss1.html*

[6] A.J.Heap and D.C.Hogg, "Extending the Point Distribution Method Using Polar Coordinates", *Proc. Computer Analysis of Images and Patterns*, Prague, September 1995.

[7] S.Levy, "Artificial Life: The Quest for a New Creation", Random House Inc., New York, 1992

[8] B.Stroustrup, "The C++ Programming Language", Addison-Wesley, 1986.

[9] "PostScript Language Reference Manual", Adobe Systems Inc., (2nd edition) 1990.

[10] J.F.Doué, "C++ Vector and Matrix Algebra", *Graphics Gems IV*, ed. P.S.Heckbert, Academic Press Inc., 1994, pp. 534-557.

[11] Meschach numerical library, *ftp://ftpmaths.anu.edu.au/pub/meschach/*

[12] W.M.Newman and R.F.Sproull, "Principles of Interactive Computer Graphics", McGraw-Hill, (2nd edition) 1979.

]13] A.J.Heap and D.C.Hogg, "Automated Pivot Location for the Cartesian-Polar Hybrid Point Distribution Model", *Proc. British Machine Vision Conference*, Birmingham, September 1995.

[14] J.Bentley, "Programming Pearls", Addison-Wesley, 1986.

[15] P.J.Schneider, "An Algorithm for Automatically Fitting Digitized Curves", Graphics Gems, ed. A.S.Glassner, 1990.

[16] N.J.B.McFarlane and R.D.Tillett, "Fitting 3D point distribution models of fish to stereo images", *html://peipa.essex.ac.uk/paper.html*

Appendix A — Mathematical Details

A.1 Principal Component Analysis

We can describe a shape ψ_i as an *n*-element array of (x, y) coordinates, or, more simply, as a 2*n*-element array of ordinates:

$$\boldsymbol{\psi}_i = (x_{i1}, x_{i2}, \mathrm{K}\ , x_{in}, y_{i1}, y_{i2}, \mathrm{K}\ , y_{in})^T$$

The mean shape is then obtained by averaging the ordinates over all (*m*) candidate shapes:

$$\overline{\boldsymbol{\psi}} = \frac{1}{m}\sum_{i=1}^{m}\boldsymbol{\psi}_i$$

The difference of each shape with respect to the average can be obtained by subtraction:

$$\partial\boldsymbol{\psi}_i = \boldsymbol{\psi}_i - \overline{\boldsymbol{\psi}}$$

The covariance matrix **S** is then given by:

$$\mathbf{S} = \sum_{i=1}^{m}\partial\boldsymbol{\psi}_i \partial\boldsymbol{\psi}_i{}^T$$

We compute the 2*n* orthogonal Eigen vectors $p_k\,(k = 1, \mathrm{K}\ , 2n)$ and the corresponding Eigen values λ_k of the covariance matrix such that

$$\mathbf{S}p_k = \lambda_k p_k \qquad \lambda_k \geq \lambda_{k+1}$$

The Eigen vectors are an orthogonal basis set ordered according to their significance. Any trial shape can be approximated as a combination of the mean shape and a weighted set of the first $t \leq 2n$ Eigen vectors:

$$\boldsymbol{\psi} = \overline{\boldsymbol{\psi}} + \mathbf{Pb}$$

where $\mathbf{P} = (p_1, p_2, \mathrm{K}\ , p_t)$ is the array of Eigen vectors and $\mathbf{b} = (b_1, b_2, \mathrm{K}\ , b_t)$ the required set of weights. The relative magnitude of each weight with respect to the corresponding Eigen value gives a measure of the distortion that must be applied to the base shape to generate the trial shape. In general $b_i \leq \pm 3\sqrt{\lambda_i}$ is a reasonable limit.

A.2 Aligning Shapes

The following is a expanded version of Appendix A from Cootes *et al.* (1992.) We wish to scale, rotate and translate one shape to best align it with another. We define an alignment metric that is the sum of the squares of the distances between

corresponding landmark points on the two shapes. The optimum scaling, rotation and translation parameters will be those values that minimise the alignment metric.

To align shape $\boldsymbol{\psi}_2$ to shape $\boldsymbol{\psi}_1$ we need to find the matrix *M* such that the error term

$$E = \left(M\left(\boldsymbol{\psi}_2\right) - \boldsymbol{\psi}_1\right)^T \mathbf{W}\left(M\left(\boldsymbol{\psi}_2\right) - \boldsymbol{\psi}_1\right) \qquad (1)$$

is minimised. **W** is an diagonal matrix of weights $(w_1, w_2,\ldots,w_n)$ specifying which points are most significant when calculating the error.

Rewriting *M* as a standard scale, rotation and translation matrix,

$$M\left(\boldsymbol{\psi}_{jk}\right) \equiv \begin{pmatrix} s\cos\theta & -s\sin\theta & t_x \\ s\sin\theta & s\cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{jk} \\ y_{jk} \\ 1 \end{pmatrix}$$

and substituting $a_x = s\cos\theta$, $a_y = s\sin\theta$, we can rewrite (1) as

$$E = \sum_{k=1}^{n} \left(\left(a_x x_{2k} - a_y y_{2k} + t_x - x_{1k}\right)^2 + \left(a_y x_{2k} + a_x y_{2k} + t_y - y_{1k}\right)^2 \right) \cdot w_k$$

The error will be minimised when all the partial derivatives $\frac{\partial E}{\partial a_x}, \frac{\partial E}{\partial a_y}, \frac{\partial E}{\partial t_x}, \frac{\partial E}{\partial t_y}$ are simultaneously zero. For instance differentiating w.r.t. a_x we have

$$\frac{\partial E}{\partial a_x} = \sum_{k=1}^{n} 2w_k \left(\left(a_x x_{2k} - a_y y_{2k} + t_x - x_{1k}\right) \cdot x_{2k} + \left(a_y x_{2k} + a_x y_{2k} + t_y - y_{1k}\right) \cdot y_{2k} \right) = 0$$

and similarly for the other derivatives. Collecting terms in a_x, a_y, t_x and t_y we obtain

$$\sum_{k=1}^{n} w_k \left(a_x \left(x_{2k}^2 + y_{2k}^2\right) + a_y (0) + t_x \left(x_{2k}\right) + t_y \left(y_{2k}\right) - \left(x_{1k} x_{2k} + y_{1k} y_{2k}\right) \right) = 0$$

Expanding the other derivatives similarly we can create a matrix

$$\begin{pmatrix} X_2 & -Y_2 & W & 0 \\ Y_2 & X_2 & 0 & W \\ Z & 0 & X_2 & Y_2 \\ 0 & Z & -Y_2 & X_2 \end{pmatrix} \begin{pmatrix} a_x \\ a_y \\ t_x \\ t_y \end{pmatrix} = \begin{pmatrix} X_1 \\ Y_1 \\ C_1 \\ C_2 \end{pmatrix}$$

where

$$X_i = \sum_{k=1}^{n} w_k x_{ik} \qquad Y_i = \sum_{k=1}^{n} w_k y_{ik}$$

$$Z = \sum_{k=1}^{n} w_k \left(x_{2k}^2 + y_{2k}^2\right) \qquad W = \sum_{k=1}^{n} w_k$$

$$C_1 = \sum_{k=1}^{n} w_k \left(x_{1k} x_{2k} + y_{1k} y_{2k}\right) \quad C_2 = \sum_{k=1}^{n} w_k \left(y_{1k} x_{2k} - x_{1k} y_{2k}\right)$$

Then by inverting the matrix we can solve for the a_x, a_y, t_x and t_y which best align the shapes.

Appendix B — Program Manual Entries

Program: **asm**

Purpose: Use an active shape model to detect a shape within an image.

Usage: asm -f [-g*N* [-p*N*]] [-v*N*] [-m] < x.pdm *image-name* > x.psr

asm [-g*N* [-p*N*]] [-m] [-v*N*] *base-name*

Switches:

-g*N*	set number of genetic algorithm iterations
-p*N*	set genetic algorithm pool size
-m	set genetic algorithm mutation rate
-v*N*	set verboseness level
-f	filter mode

Files:

stdin	Point Distribution Model data file
stdout	PostScript file showing program in action

Description:

The asm program reads a bitmap image and a PDM data file and attempts to find the shape described by the PDM within the image. A genetic algorithm is used to find an initial guess. Switches can be used to set the genetic algorithm parameters.

A PostScript file is generated illustrating the program in action. The '-v' switch can be used to change the amount of detail that is drawn.

If the –f switch is not specified, data is read from a file named *base-name*.pdm and written to a file named *base-name*.psr.

Program: **gen**

Purpose: generate shapes from Point Distribution Model

Usage: gen < x.pdm > x.psg

Switches:

none

Files:

stdin Point Distribution Model data file

stdout PostScript file illustrating the model

Description:

The gen program reads a PDM data file and generates PostScript illustrations of the principal modes of variation of the model. The first ten modes are drawn, one per page, with each mode's weight being varied $\pm 3\sigma$.

Program: **pca**

Purpose: Use Principal Component Analysis to create a Point Distribution Model from a set of training shapes.

Usage: pca < samples > x.ps 2> x.pdm

Switches:

none

Files:

stdin data file containing training shapes to be modelled

stdout PostScript file illustrating program in action

stderr data file containing generated Point Distribution Model

Description:

The pca program reads a data file containing a set of sample shapes, and creates a Point Distribution Model that models those shapes.

The sample shapes are contained in an ASCII data file, encoded as a list of coordinate deltas, with each shape being separated by a delta value of 0, 0.

The PDM data file is also ASCII. It contains a list of decimal values. The first is the number of points – n – in the model, the next n are the Eigen values (the model weights) and the next $n \times n$ are the Eigen vectors (the principal components.)

The output PostScript file illustrates the raw training data and the data after alignment to a circle.

Appendix C — Program Text

The following files comprise the programs written for this project:

• makefile	Master make file for project
• rand.h, rand.cpp	Random number generation
• winbmp.h, winbmp.cpp	Windows bitmap handling
• pcalib.h, pcalib.cpp	Helper functions
• asm.cpp	Active shape model
• gen.cpp	Shape generator
• pca.cpp	Principal component analyser
• drawlamp.cpp	Anglepoise lamp generator

The source text for these files appears in this appendix. In addition, machine-readable copies of the files can be downloaded from *http://www.centprod.demon.co.uk/msc*.

The files algebra3.cpp and algebra3.h are not listed here: they are taken from Graphics Gems, see reference [10].

---

makefile

```
# .pdm = point distribution model
# .dat = raw (training) shape data file
# .psg = PostScript containing generated shapes
# .psr = PostScript during shape recognition (ASM)
# .psa = PostScript during model extraction (PCA)

.SUFFIXES: .pdm .dat .psg .psr .psa

.cpp.obj:
	cl -c -GX -Zi -Imes -Ie:\msdev\include $<

# build acm from data file
.dat.pdm:
	pca < $*.dat > $*.psa 2> $*.pdm

# generate shapes from acm
.pdm.psg:
	gen < $*.pdm > $*.psg

#recognise bitmap
.pdm.psr:
	asm $*

target: pca.exe gen.exe asm.exe hand.pdm hand.psg hand.psr resistor.pdm resistor.psg resistor.psr lamp.pdm lamp.psg

others:
	asm -p15 -g5 -f claire < hand.pdm > hand.psr
```

```
clean:
	del *.obj
	del *.exe
	del *.pdm
	del *.psg
	del *.psa
	del *.psr
	del lamp.dat

hand.psr: asm.exe hand.pdm hand.bmp

resistor.psr:	asm.exe resistor.pdm resistor.bmp

lamp.dat: drawlamp.exe
	drawlamp > lamp.dat

pca.exe:  pca.obj pcalib.obj algebra3.obj
	cl -GX -Zi pca.obj pcalib.obj algebra3.obj mes/meschach.lib /link /libpath:e:\msdev\lib

gen.exe:  gen.obj pcalib.obj algebra3.obj
	cl -GX -Zi gen.obj pcalib.obj algebra3.obj mes/meschach.lib /link /libpath:e:\msdev\lib

asm.exe:  asm.obj pcalib.obj algebra3.obj rand.obj winbmp.obj
	cl -GX -Zi asm.obj pcalib.obj algebra3.obj rand.obj winbmp.obj \
	mes/meschach.lib /link /libpath:e:\msdev\lib

drawlamp.exe:	drawlamp.obj rand.obj
	cl -GX -Zi drawlamp.obj rand.obj

winbmp.obj:	winbmp.cpp winbmp.h
```

---

rand.h

```
/* rand.h */

unsigned long nextLong();
double nextDouble();
double nextGaussian();
```

---

rand.cpp

```
#include <math.h>
#include "rand.h"

double nextDouble();
double nextGaussian();

unsigned long randdata[] = {
  0x3b781c0a, 0x782778c0, 0x1e0e3b56, 0xcda7f8d, 0x7b27071d,
  0x7eb5d7bb, 0x5d9cdff0, 0x472eb2e6, 0x32d759da, 0x5eb07091,
  0x5e4f414a, 0x7f2b53ef, 0xc52c5877, 0x539809b6, 0x3dc07dcb,
  0x4a1015d0, 0x604e5d24,
};
static int rp1 = 0;
static int rp2 = 12;

// random between zero and one (32 bit only)

unsigned long nextLong()
{
   randdata[rp1] += randdata[rp2];

   if (rp1 == 0) rp1 = 17;
   if (rp2 == 0) rp2 = 17;
   rp1--;
   rp2--;

   return randdata[rp1];
}
```

```
double nextDouble()
{
   return ((double)nextLong()) / 4294967296.0;
}



// Gaussian with mean 0.0 and SD 1.0
// courtesy Java.util.Random

double nextGaussian()
{
   // See Knuth, ACP, Section 3.4.1 Algorithm C.

   static double nextNextGaussian;
   static int haveNextNextGaussian = 0;

   if (haveNextNextGaussian) {
           haveNextNextGaussian = 0;
           return nextNextGaussian;
   } else {
           double v1, v2, s;
           do {
              v1 = 2 * nextDouble() - 1; // between -1 and 1
              v2 = 2 * nextDouble() - 1; // between -1 and 1
              s = v1 * v1 + v2 * v2;
           } while (s >= 1);
           double multiplier = sqrt(-2 * log(s)/s);
           nextNextGaussian = v2 * multiplier;
           haveNextNextGaussian = 1;
           return v1 * multiplier;
   }
}
```

---

winbmp.h

```
#pragma pack(push)
#pragma pack(2)

/* format of Windows bitmap on disk */

typedef struct tagBITMAP
 {
      unsigned short    bfType;
      long   bfSize;
      unsigned short    bfReserved1;
      unsigned short    bfReserved2;
      long   bfOffBits;

      long      biSize;
      long       biWidth;
      long       biHeight;
      unsigned short       biPlanes;
      unsigned short       biBitCount;
      long      biCompression;
      long      biSizeImage;
      long       biXPelsPerMeter;
      long       biYPelsPerMeter;
      long      biClrUsed;
      long      biClrImportant;
 } winbmp;

#pragma pack(pop)

extern unsigned char bits[8];

class bitmap { /* one bit deep pc .bmp format */

   char rootname[20];
   winbmp raw;
```

```
    int stride;
    unsigned char *p;
    int sense;

public:
    bitmap(char *_rootname);
    ~bitmap();

    int getBit(int x, int y);

    int scanForEdge(double x, double y, double& dx, double& dy, int limit);
    void draw(ostream& s);
};

/* end */
```

---

winbmp.cpp

```
/*
 * winbmp.cpp: Windows bitmap handling
 */

#include <iostream>
#include <io.h>
#include <stdio.h>
#include <fcntl.h>

#include <math.h>

unsigned char bits[8] = { 128,64,32,16,8,4,2,1 };

#include "winbmp.h"

bitmap::bitmap(char *_rootname)
{
    strcpy(rootname, _rootname);

    char temp[30];
    strcpy(temp, rootname);
    strcat(temp, ".bmp");

    int i = open(temp, _O_RDONLY | _O_BINARY);

    fprintf(stderr, "open '%s' returned %d\n", temp, i);

    int len = lseek(i, 0L, SEEK_END);
    lseek(i, 0L, SEEK_SET);

    fprintf(stderr, "extent is %d\n", len);

    fprintf(stderr, "read returned %d\n", read(i, (char *)&raw, sizeof(raw)));

    fprintf(stderr, "type: %d width %d height %d planes %d bits %d ptr %d\n",

                raw.bfType, raw.biWidth, raw.biHeight, raw.biPlanes,
                raw.biBitCount, raw.bfOffBits);

    p = new unsigned char [len - raw.bfOffBits];
    lseek(i, raw.bfOffBits, SEEK_SET);

    read(i, (char *)p, len - raw.bfOffBits);
    close(i);

    stride = ((raw.biWidth + 31) & ~31) >> 3;
    sense = (p[0] & 1);
}


bitmap::~bitmap()
{
```

```
    if (p)
            delete [] p;
    p = 0;
}


int bitmap::getBit(int x, int y)
{
    x *= 5; /* compensate for to-screen scaling */
    y *= 5;

    x += raw.biWidth / 2;
    y += raw.biHeight / 2;

    if (x < 0 || x >= raw.biWidth || y < 0 || y >= raw.biHeight)
            return 0;

    return ((p[(y * stride) + (x >> 3)] & bits[x & 7]) != 0) ^ sense;
}


int bitmap::scanForEdge(double x, double y, double& dx, double& dy, int limit)
{
    // normalise so we take single pixel steps

    if (abs(dx) >= abs(dy)) {
            dy /= abs(dx);
            dx /= abs(dx);
    }
    else {
            dx /= abs(dy);
            dy /= abs(dy);
    }

    int senseP = getBit(x, y);
    int senseM = senseP;

    for (int i = 1; i < limit; i++) {

            double _dx = dx * i;
            double _dy = dy * i;

            /* scanning inwards - want a 0->1 transition */

            if (getBit(x + _dx, y + _dy) != senseP) {
                if (senseP == 0) {
                        dx = _dx;
                        dy = _dy;
                        return i;
                }
                else senseP = 0;
            }

            /* scanning outwards - want a 1->0 transition */

            if (getBit(x - _dx, y - _dy) != senseM) {
                if (senseM == 1) {
                        dx = -_dx;
                        dy = -_dy;
                        return -i;
                }
                else senseM = 1;
            }
    }

    dx = 0;
    dy = 0;
    return limit;
}
```

```
void bitmap::draw(ostream& s)
{
#if 1
    /*
     * patch the 'image' operator to perform imagemask and execute eps version of
     * bitmap image. (Use PaintShop Pro, open bitmap, save as mono EPS / no preview)
     */
    s << "save 0.2 0.2 scale " << -raw.biWidth / 2 << " " << -raw.biHeight / 2;
    s << " translate 1 0.9 0.9 setrgbcolor\n";
    s << "/image { pop pop pop true [ width 0 0 height 0 height neg ]\n";
    s << "{ currentfile picstr readhexstring pop } imagemask } def";
    s << "(" << rootname <<".eps) run restore\n";

#else
    /*
     * paint tick marks to indicate image position (very slow ...)
     */
    s << "1 0 0 setrgbcolor\n";

    for (int y = -84; y < 84; y++) {
            for (int x = -84; x < 84; x++) {
                if (getBit(x, y)) {
                        s << x << " " << y << " moveto 0.25 0.25 rlineto stroke\n";
                }
            }
    }
#endif
}

/* end */
```

---

pcalib.h

```
#include <vector>
#include "algebra3.h"

extern "C" {
#include "matrix2.h"
};

#include <math.h>

#define _(t) <t,allocator<t> > /* hack because MS' STL isn't */

void assfail(char *, char *, int);
void showpage(ostream&);

#define ASSERT(EX) do { if (!(EX))assfail(#EX, __FILE__, __LINE__); } while(0)

typedef vector _(double) weights;

// a shape is a vector of points

class shape : public vector _(vec2) {
public:
    shape();
    ~shape();

    shape(const shape& v);                    // copy constructor
    shape& operator = ( const shape& v );     // assignment

    shape& operator += ( const shape& v );
    shape& operator -= ( const shape& v );
    shape& operator += ( const vec2& v );
    shape& operator -= ( const vec2& v );
    shape& operator /= ( const double d );
    shape& operator *= ( const mat3& mat );

    int import(istream& s); // read from stdin
    void export(ostream &s); // write to sdout
    void display(ostream &s); // PostScript stream
```

```
    vec2 centre();
    double distance2( shape& s );
};

// a training set is a vector of shapes

class trainingSet : public vector _(shape)
{
public:
    trainingSet();
    ~trainingSet();

    int import(istream& s);
};


// encapsulate operations on eigen vectors

class eigenData {

public:
    int size;
    MAT *vectors;
    VEC *values;

    eigenData();
    ~eigenData();

    void export(ostream& s);
    void import(istream& s);

    void calculate(trainingSet&);
    void generate(const shape& aveShape, ostream& s, double limit = 3.0, int count = 16);
};

// align shape to target (weighted least squares)

mat3 align(shape& target, shape& anchor, const weights& wt);

/* end */
```

---

pcalib.cpp

```
#include "pcalib.h"

void showpage(ostream& s)
{
    s << "showpage grestore gsave\n";
}

shape::shape() {}
shape::~shape() {}

shape::shape(const shape& v)                    // copy constructor
{
    int size = v.size();
    for (int i = 0; i < size; i++) {
            this->push_back(v[i]);
    }
}

shape& shape::operator = ( const shape& v )
{
    int size = this->size();

    ASSERT(size == v.size());

    while (size--)
            (*this)[size] = v[size];
```

```
    return *this;
}


shape& shape::operator += (const shape& v)
{
    int size = this->size();

    while (size--)
            (*this)[size] += v[size];

    return *this;
}


shape& shape::operator -= (const vec2& v)
{
    int size = this->size();

    while (size--)
            (*this)[size] -= v;

    return *this;
}


shape& shape::operator += (const vec2& v)
{
    int size = this->size();

    while (size--)
            (*this)[size] += v;

    return *this;
}


shape& shape::operator -= (const shape& v)
{
    int size = this->size();

    while (size--)
            (*this)[size] -= v[size];

    return *this;
}


shape& shape::operator /= (const double d)
{
    int size = this->size();

    while (size--)
            (*this)[size] /= d;

    return *this;
}


shape& shape::operator *= (const mat3& mat)
{
    int k = this->size();

    while (k--)
            (*this)[k] = mat * (*this)[k];

    return *this;
}


int shape::import(istream& s)
```

```
{
    vec2 v(0.0, 0.0);
    vec2 d(0.0, 0.0);

    while (s >> d) {
            if (d[VX] == 0.0 && d[VY] == 0.0)
                break;
            v += d;
            this->push_back(v);
    }
    return this->size();
}


void shape::export(ostream& s)
{
    int size = this->size();
    vec2 v(0.0, 0.0);

    for (int i = 0; i < size; i++) {
    v = (*this)[i] - v;
            s << v << "\n";
            v = (*this)[i];
    }

    v[VX] = v[VY] = 0.0;
    s << v << "\n";
}


void shape::display(ostream& s)
{
    int size = this->size();

    for (int i = 0; i < size; i++)
            s << (*this)[i][VX] << " " << (*this)[i][VY] << ((i == 0) ? " moveto\n" : " lineto\n");

#if 0 /* if shapes are closed ... */
    s << "closepath stroke\n";
#else
    s << "stroke\n";
#endif
}


vec2 shape::centre()
{
    vec2 res(0.0, 0.0);

    int size = this->size();

    while (size--)
            res += (*this)[size];

    return res / this->size();
}


double shape::distance2( shape& v )
{
    double res = 0.0;

    int size = this->size();

    while (size--) {
            vec2 t = (*this)[size] - v[size];
            res += t.length2();
    }

    return res;
}
```

```
trainingSet::trainingSet() {}
trainingSet::~trainingSet() {}


int trainingSet::import(istream& s)
{
    int s1 = 0; // size of first shape
    while (! s.eof()) {
            shape  *aShape = new shape;
            int t = aShape->import(s);

            if (s1 == 0)
                s1 = t;

            if (s1 != t)
                fprintf(stderr, "warning: irregular shape size (%d != %d)\n", s1, t);
            else
                this->push_back(*aShape);
            delete aShape;
    }
    return this->size();
}


eigenData::eigenData()
{
    size = 0;
    vectors = 0;
    values = 0;
}


eigenData::~eigenData()
{
    if (size) {
            M_FREE(vectors);
            V_FREE(values);
    }
    vectors = 0;
    values = 0;
    size = 0;
}


void exportVec(ostream& s, const int n, VEC *vec)
{
    for (int i = 0; i < n; i++)
            s << vec->ve[i] << ((i == n-1) ? "\n" : " ");
}


void importVec(istream& s, const int n, VEC *vec)
{
    for (int i = 0; i < n; i++)
            s >> vec->ve[i];
}


void eigenData::export(ostream& s)
{
    s << size << "\n";

    exportVec(s, size, values);
    s << "\n";

    VEC *vec = v_get(size);
    for (int i = 0; i < size; i++) {
            (void)get_row(vectors, i, vec);
            exportVec(s, size, vec);
```

```
    }

    V_FREE(vec);
}


void eigenData::import(istream& s)
{
    if (size) {
            M_FREE(vectors);
            V_FREE(values);
    }

    s >> size;

    fprintf(stderr, "importing %d vectors\n", size);

    vectors = m_get(size, size);
    values = v_get(size);

    importVec(s, size, values);

    VEC *vec = v_get(size);
    for (int i = 0; i < size; i++) {
            importVec(s, size, vec);
            (void)_set_row(vectors, i, vec, 0);
    }

    V_FREE(vec);
}


void eigenData::calculate(trainingSet& tSet)
{
    if (size) {
            M_FREE(vectors);
            V_FREE(values);
    }

    size = 2 * tSet[0].size();

    vectors = m_get(size, size);
    values = v_get(size);

    MAT *covarianceMatrix = m_get(size, size);
    m_zero(covarianceMatrix);

    MAT rawShape; rawShape.m = 1; rawShape.n = size;

    MAT *tmat = m_get(size, size);
    for (int i = 0; i < tSet.size(); i++) {

            // type punning: array of vec2 -> array of double
            double *hack = &tSet[i][0][0];
            rawShape.me = &hack;

            mtrm_mlt(&rawShape, &rawShape, tmat);
            m_add(covarianceMatrix, tmat, covarianceMatrix);
    }
    M_FREE(tmat);

    sm_mlt(1.0 / tSet.size(), covarianceMatrix, covarianceMatrix);
    (void)symmeig(covarianceMatrix, vectors, values);

#if 0
    /* verify eigenvector dot eigenvector == 1 and covmat . evector == evalue . evector */

    for (i = 0; i < size; i++)
            fprintf(stderr, "eigenvalue %d = %g\n", i, values->ve[i]);

    VEC *ev = get_col(vectors, 0, (VEC*)0);
```

```
    VEC *rr = mv_mlt(covarianceMatrix, ev, (VEC*)0);

    double lam = values->ve[0];

    double dd = 0.0;
    for (i = 0; i < n; i++)
            dd += ev->ve[i] * ev->ve[i];

    fprintf(stderr, "VEC2 = %g\n", dd);

    for (i = 0; i < n; i++)
            fprintf(stderr, "%g %g\n", rr->ve[i], ev->ve[i] * lam);

    V_FREE(ev);
    V_FREE(rr);
#endif

    M_FREE(covarianceMatrix); // don't need this any more ...
}


void eigenData::generate(const shape& aveShape, ostream& s, double limit, int count)
{
    VEC *baseVector = v_get(size);
    VEC *weightedVector = v_get(size);

    for (int component = 0; component < 10; component++) {
            if (values->ve[component] < 0.001)
                continue; // stationary points ...

            (void)get_col(vectors, component, baseVector);

            double lambda = limit * sqrt(values->ve[component]);

            count /= 2;
            for (int i = -count; i <= count; i++) {
                double weight = (lambda * i) / count;

                sv_mlt(weight, baseVector, weightedVector);

                VEC hack; hack.dim = size;
                shape testShape(aveShape);
                hack.ve = &testShape[0][0];

                v_add(weightedVector, &hack, &hack);

#if 0
                /* rotate around origin so point 1 is left of point 0 */

                double hx = hack.ve[2] - hack.ve[0];
                double hy = hack.ve[3] - hack.ve[1];

                double del = sqrt(hx * hx + hy * hy);

                double cc = -hx / del;
                double ss =  hy / del;

                for (int t = 0; t < size; t += 2) {
                        double x, y;
                        x = hack.ve[t] * cc - hack.ve[t+1] * ss;
                        y = hack.ve[t] * ss + hack.ve[t+1] * cc;
                        hack.ve[t] = x;
                        hack.ve[t+1] = y;
                }
                // translate point 0,0 to origin

                vec2 adjust(hack.ve[0], hack.ve[1]);
                testShape -= adjust;
#endif
                s << 0.7 * abs(weight / lambda) << " setgray\n";
                // s << 0.15 - 0.1 * abs(weight / lambda) << " setlinewidth\n";
```

```
            testShape.display(s);
        }
        showpage(s);
    }

    V_FREE(weightedVector);
    V_FREE(baseVector);
}


// scale and rotate target shape to be 'most similar' to anchor;
// overwrite target shape. Straightforward coding of algorithm in
// appendix A of Cootes et al., 1992

mat3 align(shape& target, shape& anchor, const weights& wt)
{
    double X1, Y1, X2, Y2, W, Z, C1, C2;

    int k = target.size();
    ASSERT(k == anchor.size() && k == wt.size());

    X1 = Y1 = X2 = Y2 = W = Z = C1 = C2 = 0;

    while (k--) {
        double x1 = anchor[k][VX];
        double y1 = anchor[k][VY];
        double x2 = target[k][VX];
        double y2 = target[k][VY];

        double w = wt[k];

        W += w;
        Z += w * (x2 * x2 + y2 * y2);

        X1 += w * x1; Y1 += w * y1;
        X2 += w * x2; Y2 += w * y2;
        C1 += w * (x1 * x2 + y1 * y2);
        C2 += w * (y1 * x2 - x1 * y2);
    }

    vec4 soln = mat4(vec4(X2, -Y2,   W,  0),
                     vec4(Y2,  X2,   0,  W),
                     vec4(Z,    0,  X2, Y2),
                     vec4(0,    Z, -Y2, X2)).inverse() * vec4(X1, Y1, C1, C2);

    mat3 trx(vec3(soln[0], -soln[1], soln[2]),
             vec3(soln[1],  soln[0], soln[3]),
             vec3(0, 0, 1));

    target *= trx;
    return trx;
}


void assfail(char *what, char *file, int line)
{
    fprintf(stderr, "assertion failed: %s ('%s' line %d)\n", what, file, line);
    exit(4);
}

/* end */
```

---

asm.cpp

```
/*
 * asm.cpp: Active Shape Model
 */

#include <fstream>
#include <math.h>
```

```
#include "pcalib.h"
#include "rand.h"
#include "winbmp.h"


// switches

int verbose = 1;
int GeneticIterations = 12;
int GeneticPool = 10;

// shape data
eigenData eigen;
shape baseShape;

double MutationRate = 0.1;

//--------------------------------------------------------------
// support routines
//--------------------------------------------------------------

/*
 * given 3 (absolute) ordered points on an outline, find
 * a vector bisector and normalise it. (See 'The pleasures of
 * "Perp Dot" Products", Graphic Gems IV)
 */

vec2 vNormal(vec2& p0, vec2& p1, vec2& p2)
{
   vec2 p, q, d;

   d = p1 - p0;
   p[VX] =  d[VY]; /* rotate 90 degrees */
   p[VY] = -d[VX];

   d = p2 - p1;
   q[VX] =  d[VY];
   q[VY] = -d[VX];

   p += q;
   return p / sqrt(p[VX] * p[VX] + p[VY] * p[VY]);
}


/*
 * scan vector normals to current shape and construct new
 * shape according to where we hit ...
 */

int edgeSearch(bitmap& image, shape& currentShape, shape& newShape, int limit)
{
   int size = currentShape.size();
   int retval = 0;

   for (int i = 0; i < size; i++) {
           newShape[i] = vNormal(currentShape[(i + size - 1) % size],
                                 currentShape[i],
                                 currentShape[(i + 1) % size]);

           if (verbose > 2) {
               cout << "0.7 setgray\n";
               cout << currentShape[i][VX] << " " << currentShape[i][VY] << " moveto\n";
               cout << -4 * newShape[i][VX] << " " << -4 * newShape[i][VY] << " rmoveto\n";
               cout << 8 * newShape[i][VX] << " " << 8 * newShape[i][VY] << " rlineto stroke\n";
           }

           // overwrites newShape vars.
           int x = image.scanForEdge(currentShape[i][VX], currentShape[i][VY],
                                     newShape[i][VX], newShape[i][VY], limit);

           // make absolute
```

```
            newShape[i] += currentShape[i];

            /* display snap lines */

            if (verbose > 3) {
                if (x >= 0)
                        cout << "gray gray 1 setrgbcolor\n";
                else
                        cout << "gray 1 gray setrgbcolor\n";

                cout << currentShape[i][VX] << " " << currentShape[i][VY] << " moveto\n";
                cout << newShape[i][VX] << " " << newShape[i][VY] << " lineto stroke\n";
            }

            // scoring function (fitness)
#if 1
            if (x > -limit && x < limit)
                retval += (limit - 1 - ((x > 0) ? x : -x)) / 3 + 1;
            else
                retval -= 3;
#else
            retval += (x > -limit && x < limit) ? 1 : 0; // == limit -> failed to detect
#endif
    }

    return retval; /* number of edges within 'limit' pixels of shape points */
}


//--------------------------------------------------------------
// 'classic' genetic algorithm - find scale, rotate, zoom
//--------------------------------------------------------------

static double _axsin[256];
static double _scale[256];

class dna
{
    union {
            struct {
                unsigned char theta;
                char tx;
                unsigned char scale;
                char ty;
            };
            unsigned long bitvector;
    };

public:
    int fitness;

    dna() {
            bitvector = nextLong();
            fitness = 1L << 30;

            if (_axsin[0] == 0) {
                for (int i = 0; i < 256; i++) {
                        _axsin[i] = sin((i * 3.1415926) / 128.0);
                        _scale[i] = 1.0 + (i - 128.0) / 640.0;
                }
            }
    };

    ~dna() {};

    /* crossover operator */
    dna& operator += ( const dna& v ) {
            int crossover = (nextLong() >> 10) & 31;
            unsigned long m = (1L << crossover) - 1;
            bitvector = (bitvector & m) | (v.bitvector & ~m);
            for (int i = 0; i < 32; i++)
```

```
            if (nextDouble() < MutationRate)
                bitvector ^= (1L << i);
        return *this;
    };

    mat3 matrix() {
        double axsin = _axsin[theta] * _scale[scale];
        double axcos = _axsin[(theta + 64) & 255] * _scale[scale];

        mat3 trx(vec3(axcos, -axsin, tx * 0.4),
                 vec3(axsin, axcos, ty * 0.4),
                 vec3(0, 0, 1));
        return trx;
    }
};


// shellsort an array of dna instances according to fitness

void dnaSort(dna *v, int count)
{
    int i, j, h;
    dna d;

    for (h = 1; h <= count/9; h = 3*h+1)
        ;

    for ( ; h > 0; h /= 3) {
        for (i = h; i < count; i += h) {
            d = v[i]; j = i;
            while (j >= h && v[j - h].fitness < d.fitness) {
                v[j] = v[j - h];
                j -= h;
            }
            v[j] = d;
        }
    }
}


//---------------------------------------------------------------

/*
 * calculate a good pose for the base shape using a genetic algorithm
 */
void scanGenetic(bitmap& image, shape& baseShape, shape& currentShape)
{
    if (verbose)
        image.draw(cout);

    shape tempShape(currentShape);

#define VPOP GeneticPool
    dna *trial = new dna[VPOP*VPOP]; /* random scale, rotate, zoom */

    for (int g = 0; g < GeneticIterations; g++) {

        // MutationRate = 0.01 + 0.09 * (GeneticIterations - g) / GeneticIterations;

        if (g && verbose > 1)
            image.draw(cout);

        for (int t = 0; t < VPOP*VPOP; t++) {
            currentShape = baseShape;
            currentShape *= trial[t].matrix();

            // fitness is number of vertices within <number> pixels of an edge
            trial[t].fitness = edgeSearch(image, currentShape, tempShape, 8);
        }
        dnaSort(trial, VPOP*VPOP);
```

```
            //--- visualization

            if (verbose > 1) {
               for (t = VPOP*VPOP-1; t >= 0; t--) {
                        currentShape = baseShape;
                        currentShape *= trial[t].matrix();
                        cout << ((t < VPOP) ? 0.0 : 0.7) << " setgray\n";
                        currentShape.display(cout);
               }
               showpage(cout);
            }
            else if (verbose > 0) {
               currentShape = baseShape;
               currentShape *= trial[0].matrix();
               cout << "/gray gray 0.01 sub def gray setgray\n";
               currentShape.display(cout);
            }
            for (t = 0; t < VPOP; t++)
               fprintf(stderr, "%d ", trial[t].fitness);
            fprintf(stderr, "\n");

            if (g == GeneticIterations - 1)
               break; // want current best, not mutation ... */

#if 0
            //--- cross breed

            /*
             * keep VPOP instances of best of show; VPOP-1 of next etc. + one
             * instance each of every cross-combination. For vpop = 5, matrix
             * would be (give or take mutations):
             *
             *   1x1 1x1 1x1 1x1 1x1
             *   1x2 2x2 2x2 2x2 2x2
             *   1x3 2x3 3x3 3x3 3x3
             *   1x4 2x4 3x4 4x4 4x4
             *   1x5 2x5 3x5 4x5 5x5
             *
             */

            for (t = VPOP; t < VPOP*VPOP; t++)
               trial[t] = trial[t % VPOP]; // dup best candidates over rows

            for (t = VPOP*VPOP-1; t >= 0; t--) {
               if (t / VPOP > t % VPOP)
                        trial[t] += trial[t % VPOP]; // above diagonal
               else
                        trial[t] += trial[t / VPOP]; // crossbreed
            }
#else

            dna *fittest = new dna[VPOP];

            for (t = 0; t < VPOP; t++)
               fittest[t] = trial[t];

            for (t = 0; t < VPOP*VPOP; t++) {
               trial[t]  = fittest[nextLong() % VPOP];
               trial[t] += fittest[nextLong() % VPOP];
            }

            delete [] fittest;

#endif

         } /* iterate loop */

         currentShape = baseShape;
         currentShape *= trial[0].matrix();

         delete [] trial;
```

```
}


void optimiseShape(bitmap& image, shape& baseShape, shape& currentShape)
{
    image.draw(cout);
    cout << "1 0 0 setrgbcolor\n";
    currentShape.display(cout);

    // transpose the eigen vector matrix (get its inverse.)
    MAT *vPrime = m_transp(eigen.vectors, (MAT*)0);

    // initialize variables for iter loop

    shape newShape(baseShape);

    weights weightSet; // sanity: all weights are unity
    for (int i = 0; i < baseShape.size(); i++)
            weightSet.push_back(1.0);

    int size = baseShape.size();

    VEC *bvals = v_get(eigen.size);
    for (i = 0; i < eigen.size; i++)
            bvals->ve[i] = 0.0;

    for (int iter = 0; iter < 30; iter++) {

            // scan around the shape and find edges nearby if we can ...
            // overwrite newShape with suggested new shape

            edgeSearch(image, currentShape, newShape, 10);

            // find pose that best aligns baseShape to newShape; store result in
            // currentShape [[ look at this - inefficient can use just an approximation ]]

            currentShape = baseShape;
            mat3 pose = align(currentShape, newShape, weightSet);

            /* n.b.: baseShape * pose == currentShape */

            if (iter > 15) {

                /* okay, residual errors can only be sorted by bending the base shape */

                newShape *= pose.inverse();
                newShape -= baseShape;

                VEC rawNew; rawNew.dim = size + size;
                rawNew.ve = &newShape[0][0];

                (void)mv_mlt(vPrime, &rawNew, bvals);

                /* limit the bent shape to a 'reasonable' extent - should move to 'nearest'
                   point in N-space on hypersphere, which is not the same as moving toward
                   the origin - see [Improving specificity ... Heap] - we'll ignore that for
                   now */

                double bend = 0.0;
                for (i = 0; i < 6; i++) // just look at the 1st 6 modes
                        bend += (bvals->ve[i] * bvals->ve[i]) / eigen.values->ve[i];

                fprintf(stderr, "bend2 = %g (%g)\n", bend, sqrt(bend));
                bend = sqrt(bend);

#if 1
                if (bend > 3.0) {
                        for (i = 0; i < 6; i++)
                            bvals->ve[i] *= 3.0 / bend;
                }
```

```
                bend = 0.0;
                for (i = 0; i < 6; i++) // just look at the 1st 6 modes
                        bend += (bvals->ve[i] * bvals->ve[i]) / eigen.values->ve[i];

                fprintf(stderr, "new bend2 = %g (%g)\n", bend, sqrt(bend));
                bend = sqrt(bend);
#endif

                // in any case don't want these residual corrections

                for (i = 7; i < eigen.size; i++)
                        bvals->ve[i] = 0.0;

                // generate delta values (wrt baseShape)

                mv_mlt(eigen.vectors, bvals, &rawNew);

                newShape += baseShape;
                newShape *= pose;

                currentShape = newShape; // iterate ...
            }

            /* around the loop ... */

            cout << "/gray gray 0.01 sub def gray setgray\n";
            currentShape.display(cout);
    }

    showpage(cout);

    fprintf(stderr, "factors: ");
    for (i = 0; i < 6; i++)
            fprintf(stderr, "%g ", bvals->ve[i]);
    fprintf(stderr, "\n");

    fprintf(stderr, "limits: ");
    for (i = 0; i < 6; i++)
            fprintf(stderr, "%g ", 3 * sqrt(eigen.values->ve[i]));
    fprintf(stderr, "\n");

    M_FREE(vPrime);
    V_FREE(bvals);
}


//-------------------------------------------------------------
// main program
//-------------------------------------------------------------

int main(int argc, char *argv[])
{
    int filter = 0;

    argv++;
    argc--;

    while (argc && argv[0][0] == '-') {
            switch (argv[0][1]) {
            case 'g':
                GeneticIterations = atoi(&argv[0][2]);
                break;
            case 'p':
                GeneticPool = atoi(&argv[0][2]);
                break;
            case 'v':
                verbose = atoi(&argv[0][2]);
                break;
            case 'f':
                filter = 1;
                break;
```

```
            case 'm':
                MutationRate = atoi(&argv[0][2]) / 100.0;
                break;
            }
            argv++;
            argc--;
    }
    if (argc == 0) {
            fprintf(stderr, "usage: asm [-gN [-pN]] [-vN] < x.eig imgname > x.psr\n");
            return -4;
    }

    fprintf(stderr, "started\n");

    ifstream tin;
    ofstream tout;

    if (! filter) {
            char temp[60];

            strcpy(temp, argv[0]);
            strcat(temp, ".pdm");
            tin.open(temp);
            cin = tin; // reassign input

            strcpy(temp, argv[0]);
            strcat(temp, ".psr");
            tout.open(temp);
            cout = tout; // reassign output
    }

    eigen.import(cin);
    baseShape.import(cin);

    ASSERT(eigen.size == 2 * baseShape.size());

    // read binary image (n.b. use artificial light, sunlight colour balance, red channel)
    bitmap image(argv[0]);

    // set default graphics context
    cout << "300 420 translate 5 5 scale 0.01 setlinewidth 0 setgray /gray 0.7 def gsave\n";

    image.draw(cout);
    baseShape.display(cout);
    showpage(cout);

    //---

    shape currentShape(baseShape);

    // adjust currentShape to optimise pose w.r.t. baseShape

    if (GeneticIterations > 0)
            scanGenetic(image, baseShape, currentShape);

    optimiseShape(image, baseShape, currentShape);

    image.draw(cout);
    cout << "0 setgray\n";
    currentShape.display(cout);
    showpage(cout);

    return 0;
}

/* end */
```

---

gen.cpp

```
/*
 * gen.cpp: generate shapes from pca model
```

```
 */

#include "pcalib.h"

int main(int argc, char *argv[])
{
    eigenData eigen;
    shape baseShape;

    cerr << "import eigen data\n";

    eigen.import(cin);

    cerr << "import base shape data\n";

    baseShape.import(cin);

    cerr << "generating shapes ... \n";

    ASSERT(eigen.size == 2 * baseShape.size());

    // set default graphics context
    cout << "300 420 translate 5 5 scale 0.01 setlinewidth 0 setgray gsave\n";

    eigen.generate(baseShape, cout);

    cerr << "done\n";

    return 0;
}

/* end */
```

---

pca.cpp

```
/*
 * pca.cpp: Principal Component Analysis
 */

#include "pcalib.h"

// main program
// pca < samples > out.ps

int main(int argc, char *argv[])
{
    trainingSet tSet;
    weights weightSet;

    int ttt = tSet.import(cin);

    // set default graphics context
    cout << "300 420 translate 5 5 scale 0.01 setlinewidth 0 setgray gsave\n";

    for (int i = 1; i < tSet.size(); i++)
            tSet[i].display(cout);

    showpage(cout);

#if 1 /* translate, rotate scale to average shape */

#if 1
    // create a circle of points

    shape roundShape(tSet[0]);
    for (i = 1; i < 2 * roundShape.size(); i += 2) {
            double angle = (i * 3.1415926) / roundShape.size();
            roundShape[i / 2] = vec2(30 * sin(angle), 30 * cos(angle));
    }

    shape aveShape(roundShape);
```

```
#else
   // get the average shape

   shape aveShape(tSet[0]);
   for (i = 1; i < tSet.size(); i++)
           aveShape += tSet[i];
   aveShape /= tSet.size();
#endif

#if 0
   // sanity: all weights are unity

   for (i = 0; i < tSet[0].size(); i++)
           weightSet.push_back(1.0);
#else

   // assign each point a weight according to the variance in its position
   // wrt every other point, accumulated across all the shapes; points that
   // (tend to) stay stationary (have a small variance) get a larger weight.

   // note that this assumes all the sample shapes are the same orientation
   // and scale.  [ It might be worth recalculating the weight after each
   // normalisation iteration ??? ]

   for (int p1 = 0; p1 < tSet[0].size(); p1++) { // for each point

           double sumVar = 0;
           for (int p2 = 0; p2 < tSet[0].size(); p2++) { // for each point

              double sigX = 0.0;
              double sigX2 = 0.0;
              for (int s = 0; s < tSet.size(); s++) { // for each shape
                      vec2 tv = tSet[s][p1] - tSet[s][p2];
                      double del = tv.length2();
                      sigX += sqrt(del);
                      sigX2 += del;
              }
              sumVar += (sigX2 - ((sigX * sigX) / tSet.size())) / tSet.size();
           }
           weightSet.push_back(1.0 / sumVar);
   }

#endif

#if 0
   // for testing purposes rotate average by 30 degrees & scale to 50%

   mat3 trx(vec3(0.433, -0.25, 0),
                vec3(0.25,  0.433, 0),
                vec3(0, 0, 1));

   aveShape *= trx;
#endif

#if 1
   for (int pass = 0; pass < 5; pass++) {
           // align all the shapes to the average one, wrt weights

           shape newAveShape(aveShape);

           for (i = 0; i < tSet.size(); i++) {
              shape aShape(tSet[i]); // work with copies for stability

              (void)align(aShape, aveShape, weightSet);

              if (i == 0)
                      newAveShape = aShape;
              else
                      newAveShape += aShape;
           }
```

```
            // now align the new average shape to the circle (normalisation)

            (void)align(newAveShape, roundShape, weightSet);
            double distance2 = aveShape.distance2(newAveShape);

            cout << "%% distance after pass " << pass << ": " << distance2 << "\n";
            aveShape = newAveShape;

            if (distance2 < 10e-6) // todo: compute this limit
                break;
    }

    // great! we now have a sensible average shape ...
    // align everything once and for all

    for (i = 0; i < tSet.size(); i++)
            align(tSet[i], aveShape, weightSet);

#endif

    // show what we've got
    roundShape.display(cout);
    for (i = 0; i < tSet.size(); i++)
            tSet[i].display(cout);
    showpage(cout);

#else // translate/rotate/scale

    // we have properly aligned shapes
    // get the new mean shape

    shape aveShape(tSet[0]);
    for (i = 1; i < tSet.size(); i++)
            aveShape += tSet[i];
    aveShape /= tSet.size();
#endif

#if 0
    /* to escape the linearness point positions must be related to other
       points in the same shape. (Otherwise points can move only in
       straight lines wrt the average point.)
       */
    // try converting x/y points to incremental rho/theta?  Really need
    // to find centres of rotation and measure rotation wrt them ?  The
    // correlation points need not be in the shape --- how about: invent
    // random points in the shape as rotation points (genetic?) - could
    // either always include them in autocorrelation or pick best ones ...

    // prob with incr. rho/theta is shape may never close ? Needs to be point
    // delta-motion wrt mean shape's corresponding point [[[ ! true ? ]]]
    // or can measure wrt anywhere ?  Depends on matrix ops ? Need to understand.
    // eigenvectors correlate (?) rhos&|thetas whatever they may be.

    // look at original chord analysis (and why its bad) ON2

    // plausible rotation points will stay approximately the same distance from
    // all rotation points ? --- just try rotation about 'next' point in shape ?

#endif

    // subtract the average shape from each shape

    for (i = 0; i < tSet.size(); i++)
            tSet[i] -= aveShape;

    // okay, now we have shapes where each 'point' is the delta to
    // the mean shape.  Perform PCA on the delta information to
    // correlate those deltas that change together.  The result is
    // a set of delta-vectors (that are mutually orthogonal) which
    // represent the 'best fit' correlation. (These are the eigen
    // vectors of the covariance matrix - see eigen.calculate ...)
```

```
    eigenData eigen;
    eigen.calculate(tSet);

    // save all the info for the next step ...

    eigen.export(cerr);
    aveShape.export(cerr);

    return 0;
}

/* end */
```

---

drawlamp.cpp

```
/* draw an anglepoise lamp */

#include <stdio.h>
#include <math.h>

#include "rand.h"

struct point {
    double x;
    double y;
};


point pos;

void emit(point p)
{
#if 1
    printf("%g %g\n", (p.x - pos.x) / 5, (p.y - pos.y) / 5 );
    pos = p;
#else
    printf("%g %g lineto\n", p.x, p.y);
#endif
}


void head(point p, double theta)
{
    point q, r, s;

    q.x = p.x + 20 * sin(theta);
    q.y = p.y + 20 * cos(theta);
    emit(q);

    r.x = q.x + 7 * cos(theta);
    r.y = q.y - 7 * sin(theta);
    emit(r);

    s.x = p.x + 3 * (r.x - p.x);
    s.y = p.y + 3 * (r.y - p.y);
    emit(s);

    r.x = q.x - 7 * cos(theta);
    r.y = q.y + 7 * sin(theta);

    s.x = p.x + 3 * (r.x - p.x);
    s.y = p.y + 3 * (r.y - p.y);

    emit(s);
    emit(r);
}


point arm(point p, double len, double theta)
{
```

```
    point q;

    q.x = p.x + len * sin(theta);
    q.y = p.y + len * cos(theta);

    emit(q);

    return q;
}


point base(point p)
{
    point q;

    q.x = p.x + 20;
    q.y = p.y - 10;
    emit(q);

    q.x -= 40;
    emit(q);

    emit(p);
    return p;
}


int main(int argc, char *argv[])
{
    point t;

    for (int i = 0; i < 100; i++) {
            t.x = -50;
            t.y = -50;

#if 1
            pos.x = 0.0; pos.y = 0.0;
#else
            printf("%g %g moveto\n", t.x, t.y);
#endif

            double fa = nextGaussian() / 2 - 0.3;

            t = base(t);
            t = arm(t, 70, fa);
            t = arm(t, 60, fa += nextGaussian() / 2 + 0.7);
            head(t, fa + nextGaussian() / 2 + 0.6);

#if 1
            printf("0.0 0.0");
#else
            printf("0.1 setlinewidth stroke\n");
#endif
    }

#if 0
    printf("showpage\n");
#endif
    return 0;
}
```