Quasi-Invariant Iconic Object Recognition

James E. Jennens

MSc Information Technology: Knowledge Based Systems
Department of Artificial Intelligence
University of Edinburgh
1994

Abstract

This dissertation documents work towards a model of the iconic image matching system. It is hypothesized that iconic matching is one of two main visual recognition systems employed by human beings. It is the process of matching the image on the retina against a stored model image. To accomplish this various processes are employed: a multi-scale representation to facilitate size constancy; multiple representations (intensity, edge and corner images); a saccading process to direct the point of fixation towards interesting locations and a matching system. The system, so far, has only been tested on a small database of artificially generated test images but it should be possible to extend this in future.

Acknowledgements

I would first of all like to thank my supervisor, Dr Fisher, for his patient and careful guidance during the course of the research reported here. I would also like to express my gratitude to my secondary supervisor, Dibio Borges and to Andrew FitzGibbon for his advice and help.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is hypothesized that human beings employ two visual recognition systems [Farah 84]. One of them uses geometrical descriptions of objects and a matching system based on geometrical reasoning to deduce rotation, translation and possible occlusion of any object it attempts to match.

The second method is thought to be an iconic pattern matching system. This involves the recognition of an object by its image rather than by its features and the spatial relationship between them. This is an area of vision which has not been extensively investigated. There are a number of difficult problems involved with visual recognition of this sort. How do we recognize objects under different contrasts and illumination? How do we cope with rotational and translational differences? How can we recognize objects over a large range of distances from the viewer? All of these affect the size, shape or appearance of the object to some degree and make iconic matching a difficult task.

This project is concerned with the iconic matching aspect of vision. The object is to build a system from a large number of fairly simple modules and to tie these together to provide the input and support functions for a neural net based indexing and matching system. The input to the network would be provided by the supporting functions and include intensity, colour, edge, corner, texture, motion, and surface orientations.

The processes needed to accomplish this should include:

- a foveation process to give images with high detail at the point of interest and progressively lower resolution towards the periphery and a multi-scale representation to facilitate size constancy

- intensity and colour correction routines to deal with changes in illumination intensity and wavelength

- feature detection processes to produce edge/corner/blob images

- an interest operator to determine regions of interest in the visual field which bear closer examination and a saccade process to shift the position of gaze onto these objects

- a figure / ground mechanism to segment the object from its surroundings

- a stable feature frame to act as a short term visual store to build up a representation of the world in de-foveated form and to aid in the recognition process

The work done so far on this project has not included the neural net based matcher because of the time involved in acquiring good training data and the training of the net. The matching has instead been done with a correlation type process. The lightness/colour constancy has not been modelled either restricting the matcher to operating in fixed lighting conditions. Other components have been implemented with varying degrees of success. This project takes the form of a preliminary investigation of the iconic matching process and will hopefully lead to extensive future work in this area.

The system, at present, is capable of recognizing and distinguishing between simple images. It has been tested on artificial images but should hopefully be able to be extended to deal with images of real objects grabbed by a video camera. The images are loaded in, a saccade process indicates useful areas of the image to examine and the images are matched. The result of this is a probability score for each model. These probability scores are used to rescale the data to give a better fit with the model and could be used in some kind of figure / ground process but are not at present. The image recognized is written to the stable feature frame.

Chapter 2

Background

Most theories of visual perception agree that perception involves the computation of three dimensional form derived from the two dimensional retinal image. Iconic vision is an early stage in the visual system whereby objects are first located and then recognized by matching the retinal images against a stored database of icons.

To achieve this, it is theorized that the brain has to carry out a number of different processes. The visual system seems to be organized in a fairly modular way and these processes operate, for the most part, independently. One advantage of modularity is apparent in the evolution of the vision system: different modules could be damaged or altered without destroying the whole system. Physiological and neurological studies have demonstrated that a number of patients, after having suffered brain damage of one sort or another, have selectively lost different aspects of vision [Farah 90] [Heywood *et al.* 87]. These, and other studies, have provided considerable evidence for the initial independence of perception of colour, shape, motion and depth. The separation of these properties not only makes sense on an evolutionary scale but also facilitates the recognition of objects independent of colour, distance or motion enabling us to recognize a person at ten or a hundred metres whether running or standing still. [Zeki 79] has demonstrated that the visual cortex of the monkey seems to be organized into areas concerned primarily with the perception of orientation, colour and motion. All the evidence leads us to believe that the human visual system is also modular.

Foveation

The image received by the brain is in foveal form. The receptors in the eye are much more densely packed in the centre of the retina (the fovea) than they are towards the periphery. Detailed, high resolution information is only available at the position of gaze. Away from this point only low resolution, large scale changes in intensity or colour are perceived.

Boundary Extraction

One purpose of the visual system is to translate images on the retina into a coherent description of the environment. One of the earliest stages in the process will be the extraction of significant data within that image. Significant areas of an image are areas which contain the most information. These include edges, blobs and corners: areas of large intensity or colour discontinuities. These areas will almost always define the distinct regions in an image as different objects rarely reflect light in the same way. These components when combined together at different scales make up what has come to be known as the primal sketch which when combined with other information using stereopsis, shape from shading etc. go to make up the $2\frac{1}{2}d$ sketch. This provides information concerning surface orientation and depth.

Interest Operators

The human eye is attracted by various things in its field of view. Areas of high contrast, moving objects and areas of vertical symmetry seem to be places of interest. The ability of being able to quickly focus attention on moving objects has obvious advantages to any animal at risk of being attacked. Areas of high contrast serve to define boundaries and would therefore also be useful in recognizing objects. The area over which the focus of attention operates is known to vary [Eriksen & Murphy 87]. If the area of attention is small then fine detail can be construed; when the size of the area is larger only coarser resolution information is available.

Colour Constancy

When we view the world we see surfaces and objects in a range of colours. All objects in the world absorb and reflect light of various wavelengths in various combinations. Black objects absorb light of all wavelengths, a white object reflects light of all wavelengths, colours are produced by objects absorbing and reflecting unequal ranges of frequencies. One might imagine that the human brain encodes information about colour on the basis of the wavelengths of the light incident on the retina. However, the perception of colour and lightness is far from straightforward. We can see white objects as such even when poorly illuminated and black ones as black even when under a bright light. An object's colour seems to be the same when illuminated under sunlight as it is indoors where it is illuminated by proportionally more low frequency light. Somehow our brains are able to ignore the particular effects of the ambient lighting and perceive the real colour and lightness of objects we observe. These processes are known as colour constancy and lightness constancy. Zeki [Zeki 80] has demonstrated experimentally that some cells in the V4 region show colour constancy. A good demonstration of this effect was provided by Land [Land 59] in his famous two colour slide experiments. Land and McCann went on to devise a theory to explain this phenomenon [Land & McCann 71] [Land 86] in which reflectance maps are computed within independent channels but comparison of the model with colour change observed by humans has not been encouraging [Brainard & Wandell 86].

Figure / Ground Separation

Figure / ground separation is the term given to the brain's ability to filter out the background information present in an image and to concentrate selectively on certain specific parts of the image. This is certainly related to visual attention. It will depend on which scale an image is being observed whether a high or low resolution figure will be seen. For instance when we look at a tree in a forest we may choose to focus attention on the tree, one of its leaves or the whole coppice in which it stands. An example of this was demonstrated by Navon [Navon 77] when he was advancing the Gestalt approach of recognizing the whole before the parts. His experiments involved having various people look at large letters made up from

smaller ones. Subjects were briefy shown images of the kind shown in Figure 2–1 and asked to report what they had seen.

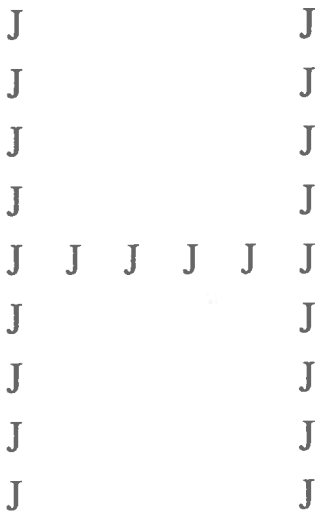

Figure 2–1: Are global or local feature processed first?

Navon's results indicated that the global feature is always recognized first and that some subjects failed to notice that small letters were present at all. However, later studies [Kinchla & Wolf 79] demonstrated that this depends on the size of the whole. Their results indicate that when the large letter exceeds 8 degrees of visual angle the smaller letters are perceived first. Their conclusions indicated that there is an optimal size of feature which will be processed first. Regardless of which feature is processed first, we seem unable to be simultaneously aware of both: we can either attend to the local features (the small J's) or the global one (the large H).

Stable Feature Frame

The image of the world that the human brain interprets appears to be stable. The world, as we perceive it when looking around, does not appear to rush past in a blur as it does when moving a video camera around in the same fashion. It is thought that the brain keeps a mental map of the surrounding area to facilitate this apparent stability. This map is known as the stable feature frame. Awareness of our surroundings is built up gradually. The receptors in the human eye are much

more densely packed towards the centre of the retina and only objects projected onto this area will be seen at maximum resolution. Outside of this area we can still see objects but often not in sufficient detail to recognize them. We have to saccade to the area in question first. As objects and features in the world are recognized they are stored in the stable feature frame and sometimes suggest other areas of the image to examine. For instance, if something in the world has been recognized as an eye then a label indicating this will be present in the stable feature frame. This might lead us to examine the image for the presence of other facial features or to zoom out to recognize a complete face.

There are various theories which attempt to explain aspects of the human visual system. This project is an attempt to tie some of these theories together and to try to simulate the iconic matching aspect of the visual system. It will use an interest operator for gaze determination. A stable feature frame will be employed to store recognized objects. A multi-scale representation of the data will be used to facilitate multi-scale recognition and a foveation routine will be used to sample the data.

Chapter 3

System Description

3.1 Overview

This system attempts to recognize objects by matching data images to stored images in a model base. It can be roughly divided into two sections. The first deals with the collection and pre-processing of input data. A diagram of this is shown in Figure 3–1. The second is concerned with the matching of the data and visual awareness.

The input to the system is a 512^2 image of the world. The system should saccade around this image to identify objects located within it. At each saccade the system obtains a foveal representation in RΘ form (see Section 3.2). This is a circular window centred on the foveation point of radius 128 pixels. This window is then converted into RΘ form. Having the input data in this form makes it easy to hold information on several scales. Each ring is twice the size of the previous one. The outer rings hold low resolution data, the inner rings hold the higher resolutions.

The image of the world can have red, green, blue and intensity components and the foveated image will reflect this. The foveated image is a stack of image planes all registered on the same point in the world. The various planes are red, green, blue, intensity, edge, corner, blob and label. The system extracts the edge, corner and blob images from the RΘ representation of the intensity and colour data. The label image is built as recognition of the object takes place: it records positions of sub-features. Lightness and colour constancy are needed to recognize objects in varied lighting conditions. This could be modelled with a retinex algorithm or normalization. The time based collection processes multiple images sampled over

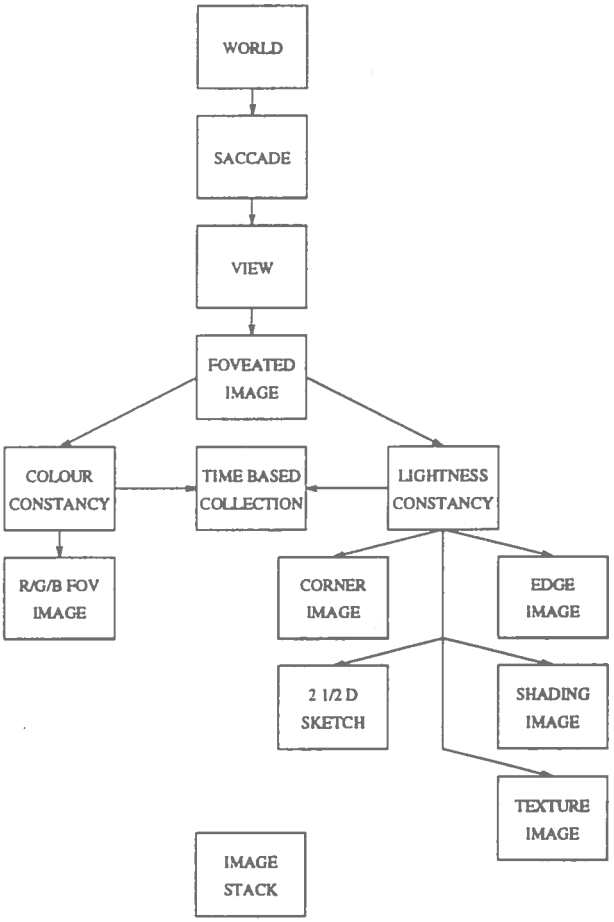

Figure 3–1: Data collection and pre-processing

a period of time. These could be used to compute optical flow and be useful for figure/ground segmentation. The current view image stack is constructed from these various planes and is then passed on to the matching system (Figure 3–2).

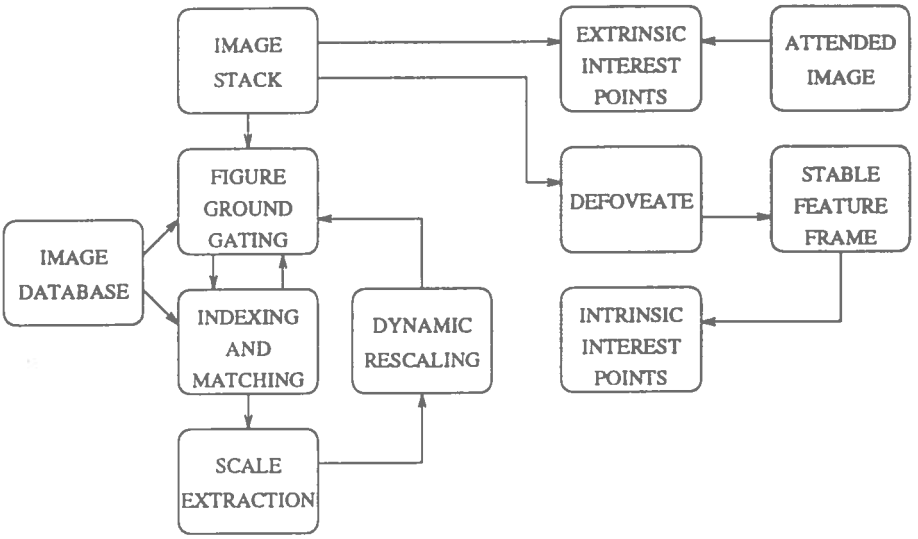

Figure 3–2: The matching system

The image stack is examined for interest points which are extracted and added to the saccade list. Models of objects are stored as image stacks containing the same data planes as the current view image stack. The model is stored at the middle resolution and is registered on a corner of the object. The three scales of data in the current view stack are matched in turn against each model. The model giving the best match score is selected and the data is rescaled according to the match values for the three scales. The rescaled data is then matched again against the model base. The matching process utilizes a weighted mask with pixel weights proportional to the response of the matcher to suppress unwanted data. This loop continues until convergence at which point the recognized object is written to the stable feature frame. Prior to the next saccade the label plane is examined for intrinsic interest points (labels which suggest other areas of interest) which are then added to the saccade list for further examination.

3.2 Representation

The system deals with data in RΘ form. The image taken has 32 evenly spaced sectors and 7 rings with exponentially increasing radius. The radius of each ring increases by a factor of two. This enables a multi-scale representation to be stored in one array. The radius of the outer ring is 128 pixels. This value was chosen as being the largest practical value to move around in the world image 512^2 pixels.

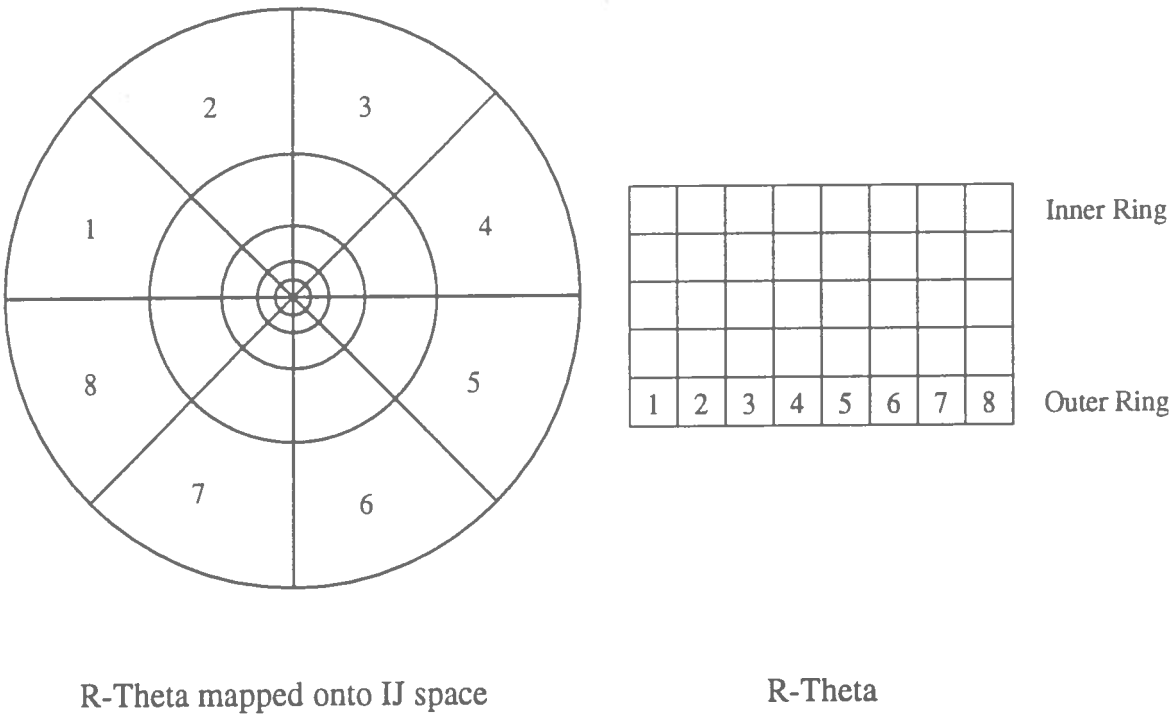

Figure 3–3: RΘ space

Figure 3–3 shows how RΘ is represented. The array shown in the diagram is smaller than that used in the system but the principle is the same. The circular object on the left shows the area covered in IJ space by the RΘ representation on the right. To enable a multi-scale representation to be held we do not use all of the rows in the array. If we wanted to hold three sacles of information in the situation shown in Figure 3–3 we would have the top three rows represent the highest resolution, the middle three rows represent the middle resolution, and the lower three represent the coarsest resolution. However, if me use the outer three rings for the coarsest resolution this will give us a circle with a hole in the

middle when mapped back into IJ space. To get around this the upper rings are averaged to contain the same value based upon their areas when mapped into IJ space. At the middle level of resolution ring 1 will contain a weighted average of 75% ring 1 and 25% ring 0 reflecting their relative contributions to the total area under rings 0 and 1. In a similar fashion at the coarsest resolution ring 2 will contain an intensity value representing 75% ring 2 and 25% of the weighted average of rings 0 and 1 as shown in Figure 3–4. The data in the middle and coarse resolution images (the bottom two) should have their first two and first three rows respectively, identical. Rounding errors in the averaging process have caused the slight differences that can be seen.

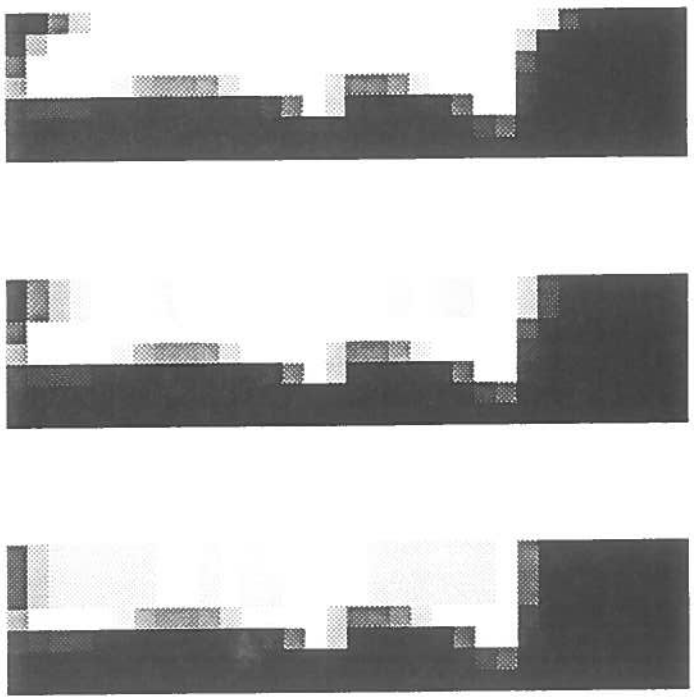

Figure 3–4: The three resolutions of the T figure (see Appendix A) in RΘ form

The image stack is made up of several planes of RΘ data stored in HIPS files, as are the models, and the stable feature frame.

The stable feature frame is an image stack with the same dimensions as the view of the world (512x512). Into it is written objects that the system thinks it has recognized. The data in the stable feature frame represents the systems best estimate of the object in question at a given time. These estimates can be revised

and the data overwritten if a conflict arises. The most probable data (that with the best match score) is stored.

3.3 Control Sequence

The sequence of control in the system is illustrated in Figure 3–5:

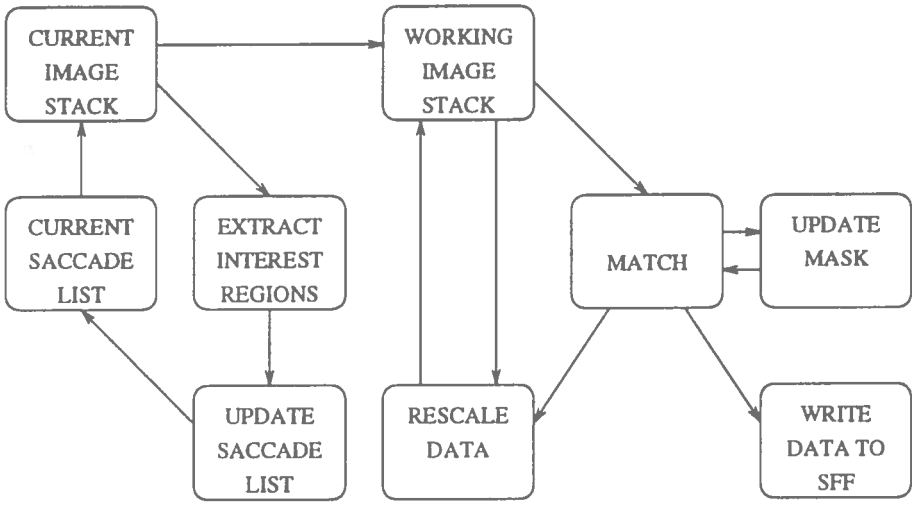

Figure 3–5: The sequence of control in the matching system

The system first saccades to a given location and updates the saccade list to include any areas of interest. The models are registered on the corners of objects so the current view has to be registered at a corner for a correct match to take place. An attempt is made to match anything under the fovea against objects stored in the database. The matcher compares the data at three scales to each of the models and produces three match scores for each model. These match scores are used to rescale the data to give a better fit with the model. The rescaled data is then passed back to the matcher. Upon convergence the recognized object is written to the stable feature frame.

Chapter 4

System Modules

4.1 Coordinate Transformations

Transformations from one representation to the other are performed by the ij2rt and rt2ij routines. These routines make use of pre-loaded mapping tables which indicate the positions and weights of pixels in one coordinate system which make up a pixel in the other. The tables are of the form:

$$P^{x}_{R\Theta}\ P^{y}_{R\Theta}\ P^{x1}_{IJ}\ P^{y1}_{IJ}\ W1\ P^{x2}_{IJ}\ P^{y2}_{IJ}\ W2\ -1000\ -1000\ -1.0$$

This shows which IJ pixels at which weights contribute to the RΘ pixel. The IJ table is similar but the opposite way round. The -1000 -1000 -1.0 on the end signifies end of line.

1 19 3 2 0.172322 2 2 0.483429 2 1 0.189781 1 1 0.154468 -1000 -1000 -1.0

The example above (taken from one of the mapping files) shows that the pixel at ring 1 sector 19 is made from pixels (3,2), (2,2), (2,1) and (1,1),weighted as shown, in IJ space relative to the current point of foveation. An example IJ and its foveated RΘ representation are shown in Figure 4–1.

The rt2ij procedure transforms the RΘ image back into IJ space for use in updating the Stable Feature Frame. The result of applying the rt2ij transform is shown in Figure 4–2.

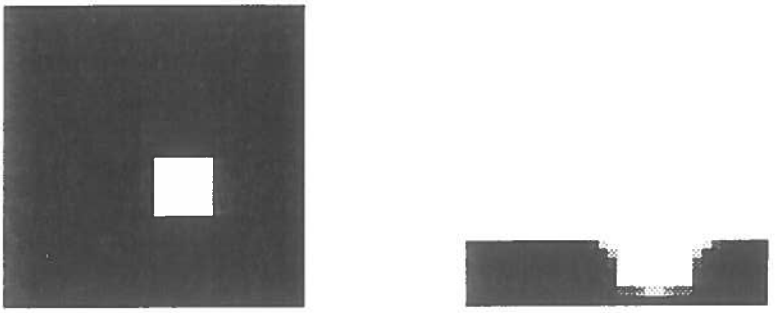

Figure 4–1: Original IJ image before foveation with corresponding RΘ image after foveation

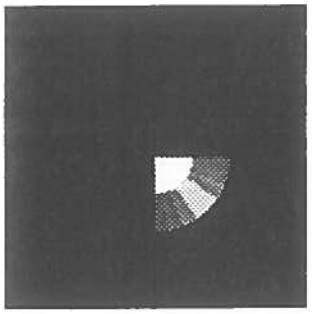

Figure 4–2: Image of square transformed back into IJ space

4.2 Edge Detection

Edge detection is achieved using a Robert's Cross operator (see, for example [Ballard & Brown 82]) of the type shown below:

$$\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}$$

The mask is passed over the intensity image in RΘ form to give a image representing edge magnitude. Because this is an RΘ representation the left side of the array is connected to the right side so the operator is moved all the way across the array and "wraps around" when it gets to the far side. When dealing with colour images this module should process the red, green and blue planes of the current view image stack. This would allow edges to be detected between regions with the same intensity but different colour. An example image is shown in Figure 4–3.

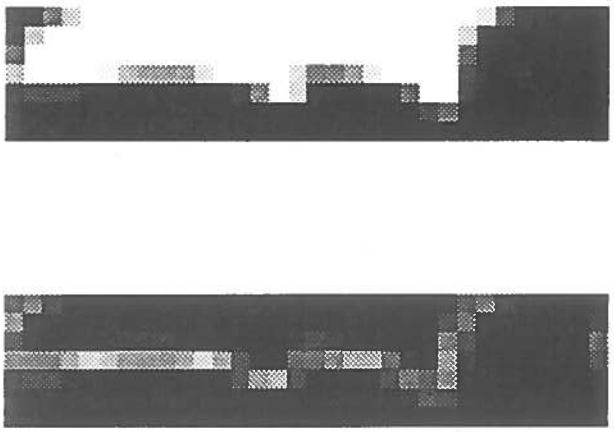

Figure 4–3: RΘ image of the "T" (see Appendix A) before and after edge detection

4.3 Corner Detection

Corners are detected using a Moravec corner detecting template [Moravec 77]. This operates over a three by three region. It's value is the result of squaring the results of subtracting the middle pixel from the two adjacent pixels in each of the four directions and summing them. The minimum value from the four directions is taken.

$$\begin{aligned} C_{xy} = \; & min[(P_{x,y} - P_{x-1,y})^2 + (P_{x,y} - P_{x+1,y})^2 \\ & (P_{x,y} - P_{x-1,y-1})^2 + (P_{x,y} - P_{x+1,y+1})^2 \\ & (P_{x,y} - P_{x-1,y+1})^2 + (P_{x,y} - P_{x+1,y-1})^2 \\ & (P_{x,y} - P_{x,y-1})^2 + (P_{x,y} - P_{x,y+1})^2] \end{aligned}$$

This could not be performed directly on the RΘ representation because the spatial distortions which result from the transformation to RΘ space. Instead, the algorithm cheats by taking a square sub-window from the IJ world view centred on the current foveation point. The Moravec operator is passed over this image to give a corner image. Each corner point is then examined to determine whether it is of a sufficient scale to appear in the RΘ array. This is accomplished by reducing the resolution at each corner by 3 times the ring number in which it would appear in the RΘ representationand repeating the Moravec operation at this point at the new resolution. An IJ corner image is created with a corner being represented as a circle whose radius is proportional to the scale in which it was detected. This is then transformed into RΘ by the standard routine (see Figure 4–4). This figure shows the original "T" (the model is registered at the join between the horizontal and vertical bars on the left side); the result of passing the Moravec operator over this image (with the corners slightly enhanced for display); the IJ image formed after resolution checking and the resultant RΘ image.

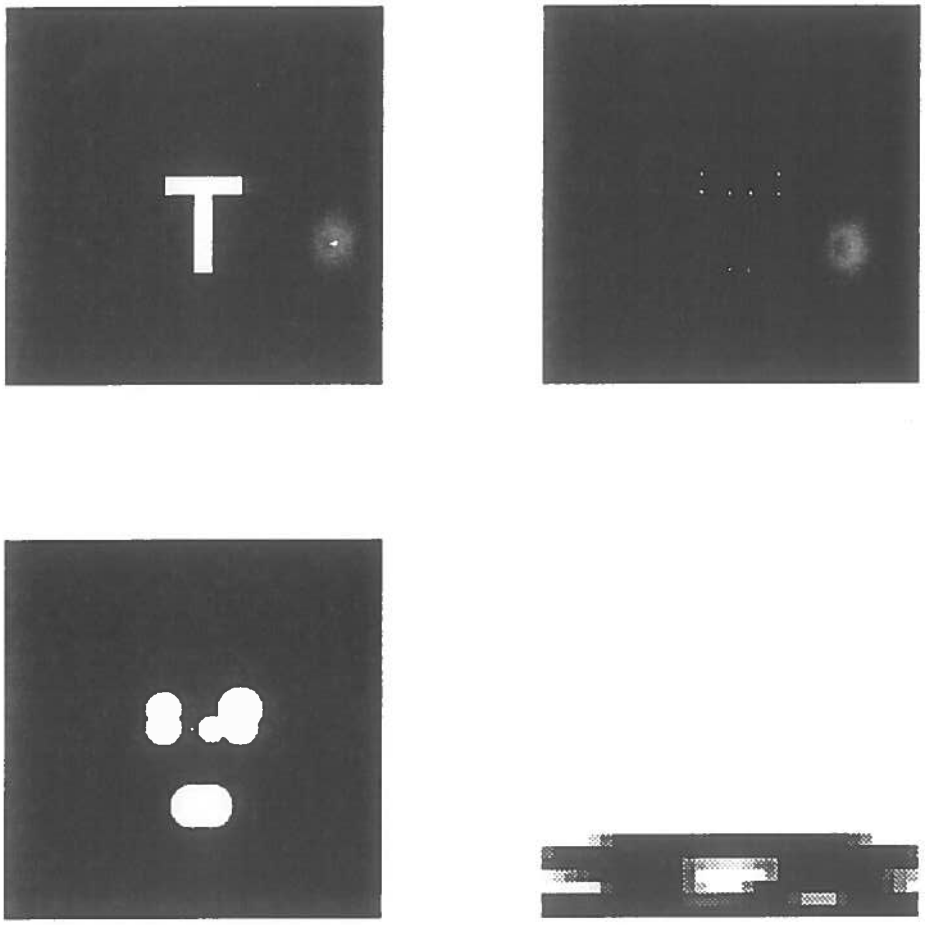

Figure 4–4: Detection of corners

4.4 Extraction of Extrinsic Interest Points

Extrinsic interest points are points in the current field of view which attract the eye's attention. At the moment only the corner image is used to provide interest points. This is accomplished by taking the eight connected local maxima from the RΘ corner image. Most pixels in RΘ space cover many pixels in IJ space and so an accurate determination of the IJ position is difficult to obtain when saccading to the outer rings. At the moment the system records the corner to be at the centre of the RΘ pixel when transformed back into IJ coordinates. These coordinate points are deemed to be of interest and are added to the saccade list. Nearer corners are given a higher priority in the saccade list. Priority is determined by the formula

$$P = 2000 - (R + 1) * 200 + C$$

where P is the priority, R is the ring number that the corner appears in and C is the strength of the corner. This formula was used to ensure nearer corners were processed before distant ones even if the corner overlapped four pixels (and therefore would have a recorded strength of a quarter of the amount it should). Figure 4–5 shows the predicted corner positions for the "T" figure; the result of the local maxima operation; and the result of mapping the local maxima back into IJ space. The two corners at the bottom have been merged into one and an extra one has been detected at the top. The extra corner has been formed from the overlapping regions of two corners producing a local maxima between them. The local maxima procedure detects this and registers it as a corner. This does not affect the matching process because the matcher uses the raw corner data (before the maximua operation).

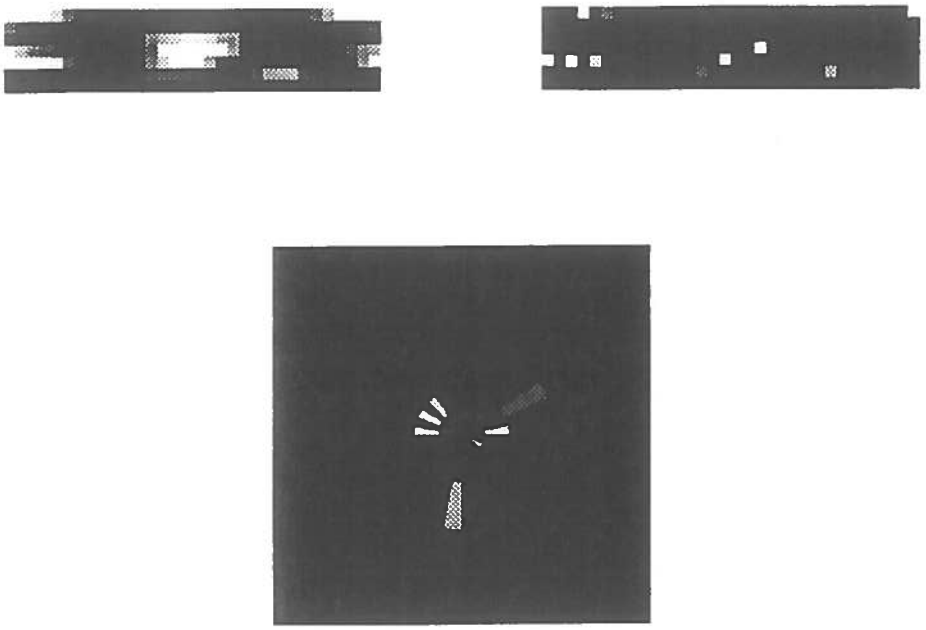

Figure 4–5: Prediciting where corners lie in IJ space

4.5 Updating the Saccade List

The saccade list has to be updated to record interest points in the image which bear closer examination. This is done by examining the corner image and extracting the positions of potential corner locations. The saccade list is an ordered list containing the IJ coordinate to saccade to and the priority of this saccade. The priority is determined by the strength of the corner, as determined by the Moravec operator [Moravec 77] and the distance from the current point of foveation. This priority was originally going to be determined by the formula $\frac{Cornerstrength}{Distance}$. This would be fine but a separate record of strength of corners needs to be kept for this. Instead the formula shown in section 4.4 is used. Currently, corners in the corner image are represented as circular blobs which cover more than one pixel which makes it difficult to extract the strength of the corner. The corners are represented in this way to enable the dynamic rescaling procedure to effectively work with them. Unfortunately this makes it difficult to determine the strength of the corners as they are spread over several pixels. It also merges distant corners causing them to be recorded on the saccade list as one, but this is perfectly acceptable as it should

not be possible to discern accurate corner locations when not examining an area in close proximity to them. Another advantage to having the corners spread over several pixels is the possibility of giving a more accurate estimation of their actual location. Instead of recording the corner location as being in the centre of the RΘ pixel with the largest response, the centre of the corner blob could be determined to give sub-pixel (RΘ) accuracy.

4.6 Matching

Matching is performed by taking the absolute value of subtracting the current data at each scale from each model on a pixel by pixel basis which gives three match values for every model item.

$$M_l = \sum |m_{ij} - d_{ij}|$$

Standard correlation was not used because normalizing the image gave poor results. Normalization was originally used as a substitute for colour constancy but results were poor and this was abandonded. When an image with low contrast is normalized it pulls the data to opposite ends of the spectrum giving a very noisy image, making matching very difficult..

Only the pixels under the model are used during matching and the background is ignored. Previously, a figure/ground system was employed to mask out irrelevant background pixels. Unfortunately this did not work very well. The data was originally matched over figure and background against the model but this gave incorrect results as the background was more important than the figure in the matching process.

The above formula is used for every plane in the data stack (currently intensity, edge and corner). The match score for the whole image stack is the evenly weighted combination of the three scores. The model which gives the best score is assumed to be the object which is currently being observed. The contribution of each plane to the final match score for the image stack is determined by a weight. Each plane of data can be weighted differently to allow for the relative importance of different

features. Rescaling of the data is achieved by taking the relevant rows of data (ie rows 1-5 for the middle resolution).

A problem is encountered when looking at the middle or coarse resolution data. The higher resolution data in the rings above the current scale (ring 0 for the middle scale, 0 and 1 for the coarse scale) cannot simply be discarded. Therefore for input to the matcher the topmost ring at any scale should be a weighted average based on the area of each pixel in IJ space of itself and any rings above it as explained in section 3.2. This works very well for intensity and colour data, reducing the resolution in the correct manner. However, a problem is encountered with the corner data. Corners at the finest scale cover the entire inner ring and at the moment no indication is stored concerning whether the corner will be visible at lower resolution. When moving to a lower resolution: corners tend to disappear. For corner images the top row is copied into the next one to preserve corner information. This is not a very good solution as some corners should disappear from view at lower resolutions simply because they are fine detail. Some record of corner scale should be kept to alleviate this problem or ideally a better corner detector, preferably in RΘ space.

4.7 Rescaling

The dynamic rescaling routine attempts to resize the data to get a better fit with the model. It takes the match scores for the three scales of data and uses them to estimate the correct scale. A linear relationship between scale and match score is assumed and providing that the middle scale gives the best match score the data is rescaled according to the following formula:

$$if(c > a)N = X - \frac{a - c}{2(b - a)}$$

$$if(a > c)N = X - \frac{a - c}{2(b - c)}$$

$N * 100$ represents the percentage change to the current scale. If the middle scale's score is not the highest a new scale and value has to be estimated and

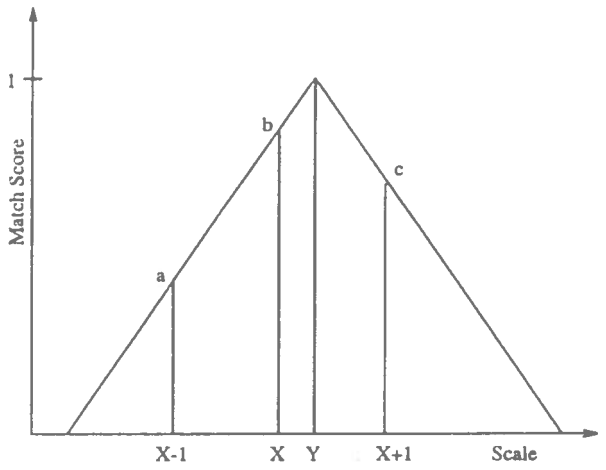

Figure 4–6: Scale Estimation

the system will rescale the data towards the nearest scale using the newly created extra scale as a or c in the formula. A record is kept of the scale of the data to update the stable feature frame.

Problems were encountered with rescaling regarding the corner image. The corners were represented as single pixels in RΘ space after having been detected in IJ space and mapped into RΘ. If the corner lay near a pixel boundary in RΘ this would cause problems with matching and rescaling. The data might be only a few percent bigger than the model, for instance, and the corners in the data and the model would be recorded as being in two separate regions. The corners were changed so that they were plotted as circles in IJ space before transformation into RΘ. This caused a blurring of the corner image which helps rescaling. (see 4–5).

4.8 Stable Feature Frame

All data in Stable Feature Frame is stored in defoveated form - it reflects the geometry of the world. The Stable Feature Frame has been implemented as an image stack. The raw data held in the current view image stack is written to the Stable Feature Frame after it has been recognized. It makes use of the mask plane of the image stack to keep a record of the resolution of data written to it. Higher resolution data overwrites lower resolution data. The mask plane is a 512^2

plane into which is written the ring number (which indicates resolution) of any data written to the rest of the Stable Feature Frame. Before any further data is written, a check is made to ensure that the ring number of the new data is equal to or lower than that of previous data. The label image keeps a record of what object has been recognized and at which scale. New data should only overwrite old data if a more probable match has been found. This hasn't been implemented yet.

Chapter 5

Results

The results presented here were obtained by presenting the system with four data items, one at a time and observing the output of the system. The tests involved positioning the fovea on the registered corner of the data and allowing the system to iterate through the matching sequence until convergence. The tables in sections 1-4 show the iteration number, the match values against each of the four models and the scale of the data (1.0 represents the size of the original data). These experiments are designed to test whether the matching system is capable of identifying the four objects. Section 5 gives an example of the corner finding procedure using the saccade list. Section 6 discusses the results obtained.

Iteration	*Scale*	*Model1*	*Model2*	*Model3*	*Model4*	*Scale*
1	reduced	0.84	0.78	0.80	0.54	1.00
1	normal	0.93	0.85	0.88	0.56	1.00
1	enlarged	0.88	0.85	0.85	0.55	1.00
2	reduced	0.86	0.80	0.81	0.54	0.71
2	normal	0.92	0.87	0.87	0.56	0.71
2	enlarged	0.83	0.82	0.81	0.55	0.71
3	reduced	0.85	0.79	0.81	0.54	0.88
3	normal	0.93	0.86	0.88	0.56	0.88
3	enlarged	0.85	0.84	0.83	0.55	0.88
4	reduced	0.85	0.79	0.81	0.54	0.85
4	normal	0.93	0.87	0.87	0.56	0.85
4	enlarged	0.85	0.84	0.83	0.55	0.85

Table 5–1: Match values returned when observing data figure 1

5.1 Data 1

Figure 5–1 shows images of the data for this test and model number 1 respectively. The model is registered at the upper left side corner. The model is 40 by 40 pixels. The data is 25% bigger than the model.

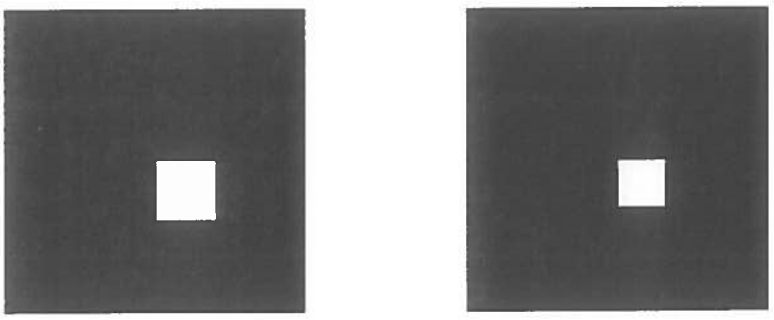

Figure 5–1: Data and Model 1

Iteration	*Scale*	*Model1*	*Model2*	*Model3*	*Model4*	*Scale*
1	reduced	0.81	0.87	0.81	0.53	1.00
1	normal	0.86	0.93	0.84	0.54	1.00
1	enlarged	0.83	0.90	0.82	0.53	1.00
2	reduced	0.82	0.88	0.81	0.53	0.72
2	normal	0.86	0.94	0.84	0.54	0.72
2	enlarged	0.78	0.86	0.79	0.53	0.72
3	reduced	0.82	0.88	0.81	0.53	0.86
3	normal	0.86	0.94	0.84	0.54	0.86
3	enlarged	0.80	0.88	0.81	0.53	0.86

Table 5–2: Match values returned when observing data figure 2

5.2 Data 2

Figure 5–2 shows images of the data for this test and model number 2 respectively. The model is registered at the upper left side corner. The model is 40 by 40 pixels. The data is 25% bigger than the model.

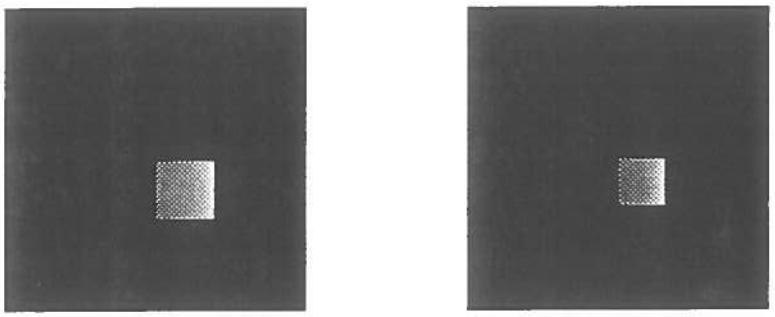

Figure 5–2: Data and model 2

Iteration	*Scale*	*Model1*	*Model2*	*Model3*	*Model4*	*Scale*
1	reduced	0.77	0.76	0.86	0.52	1.00
1	normal	0.82	0.81	0.97	0.54	1.00
1	enlarged	0.72	0.76	0.82	0.51	1.00
2	reduced	0.75	0.76	0.85	0.52	1.18
2	normal	0.82	0.81	0.97	0.54	1.18
2	enlarged	0.73	0.76	0.84	0.52	1.18
3	reduced	0.74	0.76	0.84	0.52	1.22
3	normal	0.81	0.81	0.96	0.54	1.22
3	enlarged	0.73	0.77	0.84	0.52	1.22

Table 5–3: Match values returned when observing data figure 3

5.3 Data 3

Figure 5–3 shows images of the data for this test and model number 3 respectively. The model is registered at the upper left corner. The model is contained in a 40 by 40 pixel box. The data is $12\frac{1}{2}\%$ smaller than the model.

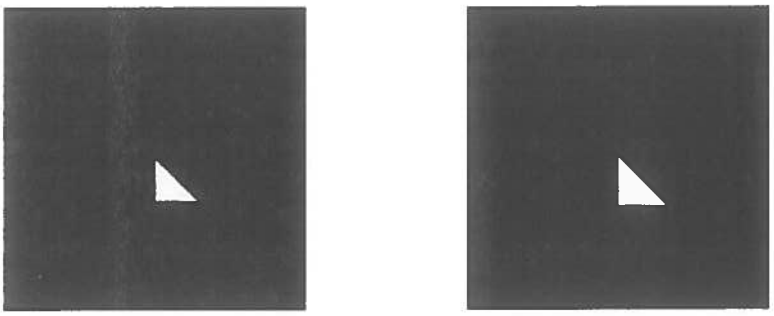

Figure 5–3: Data and Model 3

Iteration	*Scale*	*Model1*	*Model2*	*Model3*	*Model4*	*Scale*
1	reduced	0.70	0.65	0.69	0.71	1.00
1	normal	0.68	0.64	0.73	0.83	1.00
1	enlarged	0.62	0.63	0.68	0.76	1.00
2	reduced	0.72	0.67	0.72	0.75	0.67
2	normal	0.69	0.68	0.75	0.85	0.67
2	enlarged	0.62	0.65	0.67	0.70	0.67
3	reduced	0.71	0.66	0.70	0.73	0.86
3	normal	0.68	0.66	0.74	0.84	0.86
3	enlarged	0.62	0.64	0.67	0.73	0.86
4	reduced	0.71	0.66	0.71	0.73	0.84
4	normal	0.68	0.66	0.74	0.84	0.84
4	enlarged	0.62	0.64	0.67	0.73	0.84

Table 5–4: Match values returned when observing data figure 4

5.4 Data 4

Figure 5–4 shows images of the data for this test and model number 4 respectively. The model is registered at the left side corner between the horizontal and vertical bars of the "T". The model is 50 pixels tall and 40 pixels wide. The data is 33% bigger than the model.

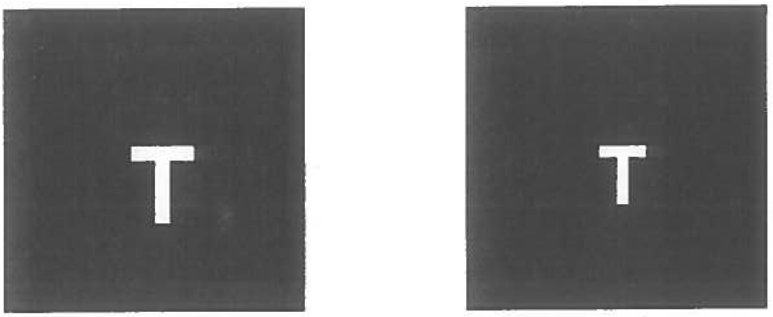

Figure 5–4: Data and Model 4

5.5 Saccading

In this test the system is directed to start examining the scene at (280,280). The corner that the model is registered on is the upper left corner of the square. This is at position (256,256); the system should guide the fixation point towards this location. The saccade list is updated after each saccade to produce an ordered list of points to examine.

Saccade 1 (280,280)

priority 1042 i 265 j 262
priority 1042 i 298 j 265
priority 1041 i 265 j 298
priority 1040 i 295 j 298
priority 1031 i 259 j 291

Position (265,262) is selected as the best corner to go for.

Saccade 2 (265,262)

priority 1263 i 260 j 252
priority 1263 i 254 j 261
priority 1217 i 264 j 251
priority 1042 i 265 j 262
priority 1042 i 298 j 265
priority 1041 i 265 j 298
priority 1040 i 295 j 298
priority 1031 i 259 j 291
priority 1012 i 244 j 251
priority 833 i 252 j 307
priority 829 i 295 j 299
priority 827 i 312 j 258

The system has now moved closer to the corner and is able to give a more accurate estimate (260,252).

Saccade 3 (260,252)

priority 1485 i 255 j 252
priority 1485 i 260 j 257
priority 1485 i 255 j 254
priority 1418 i 255 j 250
priority 1263 i 254 j 261
priority 1263 i 260 j 252
priority 1217 i 264 j 251
priority 1042 i 298 j 265
priority 1042 i 265 j 262
priority 1041 i 265 j 298
priority 1040 i 295 j 298
priority 1031 i 259 j 291
priority 1012 i 244 j 251
priority 835 i 307 j 256
priority 835 i 256 j 299
priority 833 i 252 j 307
priority 829 i 295 j 299
priority 827 i 312 j 258
priority 820 i 290 j 289

The estimate is further refined to (255,252).

Saccade 3 (260,252)

priority 1485 i 255 j 252
priority 1485 i 255 j 254
priority 1485 i 260 j 257
priority 1485 i 255 j 257
priority 1485 i 252 j 256

priority 1452 i 250 j 254
priority 1418 i 255 j 250
priority 1263 i 260 j 252
priority 1263 i 254 j 261
priority 1220 i 260 j 262
priority 1217 i 264 j 251
priority 1042 i 298 j 265
priority 1042 i 265 j 262
priority 1041 i 265 j 298
priority 1040 i 295 j 298
priority 1031 i 259 j 291
priority 1012 i 244 j 251
priority 839 i 302 j 256
priority 835 i 256 j 299
priority 835 i 259 j 299
priority 835 i 307 j 256
priority 833 i 252 j 307
priority 829 i 295 j 299
priority 827 i 312 j 258
priority 820 i 290 j 289
priority 814 i 285 j 289
priority 809 i 242 j 297

The system has again estimated (255,252) due to a slight bug in the algorithm (it doesn't remove saccade positions when saccading to them).

5.6 Discussion

Matching

The matching system works reasonably well when the gaze position is fixated on the position where the model is registered. It is, however, very sensitive to postional change. The match values for incorrect matches (other models) seem

quite high but this is because of the relatively few pixels being matched and the fact that all of the models are located in the same place. This means that regardless of which image we are looking at a good proportion of the pixels will match perfectly. However, the system is able to discriminate between the four models.

Evaluation of Saccade List

Apart from the bug which doesn't remove positions saccaded to, another problem is apparent. The saccade list dramatically expands with each iteration. This is because it is identifying the same corners at different positions. It needs some sort of threshold to prevent it from making too many corner estimates. Perhaps only identifying a point as being interesting if there is no nearby point already flagged as interesting or deleting the old point if the current estimate is thought to be more accurate. The other problem is even after saccading to (255,254), the best estimate after saccade 3, the matching system still does not recognize the image it is looking at. This is a problem with the matching system. A more sophisticated matching system could be used to correct for this.

Chapter 6

Further work and conclusions

The code for this project has only been partially implemented. The system is still quite a long way from being fully operational but the parts of the system that have been implemented show some promise. As can be seen from the results in the previous chapter, the system is capable of recognizing and distinguishing between a few simple images. With more time, it would be interesting to extend the system somewhat. There are numerous things that need to be done.

Program Code

The code has been developed in parallel with the design of the system to some extent which made planning in advance somewhat difficult. This has resulted in a poorly structured implementation. This needs to be re-written.

Fovea

The size of the fovea used was 7 rings of 32 sectors. This gave very coarse resolution: probably insufficient to recognize images of real objects. The log 2 increase in ring diameter is probably too big. It would be interesting to compare performance with different foveas. It is not certain that the decision to hold the internal representation in RΘ form is correct. It produced several problems with corner detection that were not really resolved. It would make locating rotated images very much easier as this could be done by simply moving the sector origin around the circle but it is not biologically plausible.

Corner Detection

The corner detection is not entirely satisfactory. The system cannot adjust the resolution of corner images correctly. Information needs to be kept about the scale of each corner detected to see at which resolution it will be visible in. At the moment corners detected at the highest resolution are assumed to be visible at lower resolution. This will not always be the case. The image should also be examined at multiple scales for corners so that blurred corners would be recognized. Currently, the system detects corners at the highest resolution and then checks these to determine whether they would be visible at lower resolutions. Corners were detected this way to speed up the corner detection process: the whole image is only checked once. This hasn't affected any results obtained so far because only sharp corners are present in the test data but would cause problems with images of real objects.

Input Data

Currently, only binary and grey scale images have been tested. Most of the code exists for dealing with full colour images but they have not been tested at this point. In addition, only very simple, artificial test images have been used. Images of real objects would be more challenging. The blob image was not implemented; it would detect small patches of colour or intensity discontinuity. This would be easy to implement and would prove useful with images of real objects.

Matching and Masking

A neural net based matching system would be preferable to the pseudo-correlation now employed. The fovea is masked against each model when matched at present. It would be better to have a mask develop as recognition of an object proceeds. This could be more readily accomplished with a network implementation. The current implementation gives fairly good match scores to almost any input. This is in large part due to the very small number of pixels being matched but a different classification system could improve upon this. The system is very sensitive to incorrect positioning. Even when only a distance of a couple of pixels away from

the correct location (data and model registered at the same place) the matching system sometimes makes incorrect identifications. A matching system with a degree of positional invariance is needed. The matching process should also inhibit less likely matches with each iteration through the loop.

Interest Points

The saccade list needs work. Problems with corner detection made choosing priority values difficult. The algorithm used to select interest points does not work very well. It needs to be a bit more intelligent about which things are the same corners. It tends to give several (slightly different) co-ordinate positions which all correspond to the same corner. The saccade list does not guide the system to a point where it can make a correct match. It also does not make use of the attended image. The attended image should keep a record of areas that have already been examined. The system should not saccade back to these areas under normal circumstances.

Stable Feature Frame

The stable feature frame has not been fully implemented at present. It should be able to resolve conflicting identifications of objects and remove them from its memory. This has not been implemented. The label image has not been fully implemented either. This would be used to suggest intrinsic interest points. For instance, if an eye has been found, this suggests areas to look for a nose. Future work should use the labels to override the attended image, if necessary) to re-examine areas if it has reason to believe it has better information now.

Testing

In conclusion, this project has not been a complete success. However, it has exposed a number of problems which do not seem insurmountable. This sort of system, although quite difficult to assemble, looks like it could have a reasonable chance of performing acceptably with a few modifications. I have not tested it as thoroughly as I would have liked but from the testing already done it is quite clear

that work remains to be done before further testing would be of very much use. The matching system is too sensitive to positional change and identifies objects incorrectly or as being at a different scale when being very close to having the data registered at the same point as the model. This is disappointing, I had expected better, but a more sophisticated matching system could take care of this.

Bibliography

[Ballard & Brown 82] D.H. Ballard and C.M. Brown. *Computer Vision.* Prentice-Hall, New Jersey, 1982.

[Brainard & Wandell 86] D.H. Brainard and B.A. Wandell. Analysis of the retinex theory of colour vision. *Journal of the Optical Society of America*, 3:1651–1656, 1986.

[Eriksen & Murphy 87] C.W. Eriksen and T.D. Murphy. Movement of attentional focus across the visual field: A critical look at the evidence. *Perception and Psychophysics*, 42:299–305, 1987.

[Farah 84] M. Farah. The neurological basis of mental imagery: A componential analysis. *Cognition*, 18:241–269, 1984.

[Farah 90] M. Farah. *Visual agnosia disorders of object recognition and what they tell us about normal vision.* MIT Press, Cambridge, Mass., 1990.

[Heywood *et al.* 87] C.A. Heywood, B. Wilson, and Cowey A. A case study of cortical colour "blindness" with relatively intact achromatopic discrimination. *Journal of Neurology, Neurosurgery and Psychiatry*, 50:22–29, 1987.

[Kinchla & Wolf 79] R.A. Kinchla and J.M. Wolf. The order of visual processing: "top-down", "bottom-up" or "middle-out". *Perception and Psychophysics*, 25:225–231, 1979.

[Land & McCann 71] E.H. Land and J.J. McCann. Lightness and retinex theory. *Journal of the Optical Society of America*, 61:1–11, 1971.

[Land 59] E.H. Land. Experiments in colour vision. *Scientific American*, 200:84–99, May 1959.

[Land 86] E.H. Land. Recent advance in retinex theory. *Vision Research*, 26:7–21, 1986.

[Moravec 77] H.P. Moravec. Towards automatic visual obstacle avoidance. In *Proceedings of the IJCAI*, page 584, 1977.

[Navon 77] D. Navon. Forest before trees: The precedence of global features in visual perception. *Cognitive Psychology*, 9:353–383, 1977.

[Zeki 79] S.M. Zeki. Uniformity and diversity of structure and function in rhesus monkey prestriate visual cortex. *Journal of Physiology*, 277:273–290, 1979.

[Zeki 80] S.M. Zeki. The representation of colours in the cerebral cortex. *Nature*, 284:412–418, 1980.

Appendix A

The Models

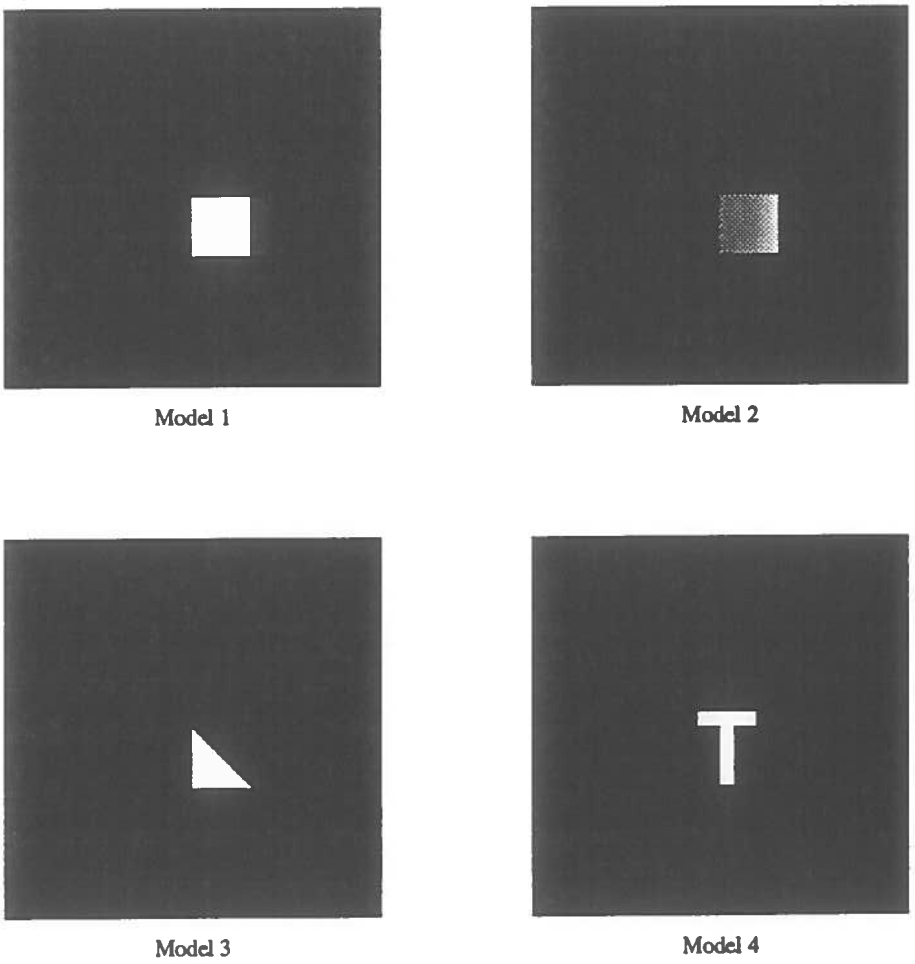

Figure A–1: The four models

Appendix B

Program Code

```
#include <math.h>
#include <stdio.h>
#include <malloc.h>
#include <hipl_format.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <memory.h>

char Progname[]="proj";

#define MAXENTRY        100000
#define MAXSECTORS      32
#define MAXRINGS        8
#define MAXPIXELS       512
#define MAXFOVSIZE      257
#define NUMSCALES       3
#define NUMMATCHCATS    8
#define NUMMODELS       4
#define REDWEIGHT       1.
#define GREENWEIGHT     1.
#define BLUEWEIGHT      1.
#define INTENSITYWEIGHT 1.
#define EDGEWEIGHT      1.
#define CORNERWEIGHT    1.
#define BLOBWEIGHT      1.
#define LABELWEIGHT     1.
#define M_PI 3.14159265358979323846

extern int errno;

struct entryij {
   short i,j;
   double percent;
   struct entryij *next;
```

```
} entIJ[MAXENTRY];

struct entryrt {
   short r,theta;
   double percent;
   struct entryrt *next;
} entRT[MAXENTRY];


struct hips_file {
  struct header* hd;
  unsigned char* image;
};

struct cstruct {
  struct hips_file cimage1;
  struct hips_file cimage2;
};

struct image_stack {
  struct hips_file red;
  struct hips_file green;
  struct hips_file blue;
  struct hips_file intensity;
  struct hips_file edge;
  struct hips_file corner;
  struct hips_file mask;
  struct hips_file label;
  struct hips_file blob;
  float scale;
  float labelinfo[256];
};


struct point {
  int x;
  int y;
};

struct sacp {
    int x;
    int y;
    int pri;
    struct sacp* next;
};

struct sacpstack {
    int length;
    struct sacp* current;
    struct sacp* base;
};
```

```
struct hips_file get_image2(char* hips_file_in);
struct hips_file ij2rt(struct point sp, struct hips_file hfile_in,
                       int NumRings, int NumSectors);
struct hips_file rt2ij(struct point sp, struct hips_file image_rt, int ysize,
                       int xsize, int NumPixels, int NumRings);
struct hips_file rcross(struct hips_file hfile);
struct hips_file constancy(struct hips_file hfile);
struct hips_file create_hips(int,int);
struct hips_file median(struct hips_file hfile);
struct hips_file subtract(struct hips_file, struct hips_file);
struct hips_file moravec(struct hips_file hfile);
struct hips_file create_cv_mask(int x,int y);
struct hips_file corner3(struct hips_file hfile,char* Ofile);
struct hips_file rescale1(struct hips_file hf1,float scale_val);
struct hips_file clip(struct hips_file hf,struct point p,int size);
struct hips_file reduce(struct hips_file hfile,int scale);
struct hips_file expand(struct hips_file hfile,int scale);
struct hips_file cornerij(struct hips_file hf,struct hips_file mask,
                          int NumRings, int NumSectors);
struct hips_file localmax(struct hips_file hfile);
struct hips_file corner(struct hips_file hf,struct point saccade_point,
                        int NumPixels, int NumRings,int NumSectors);
struct cstruct cornerij2(struct hips_file hfile,int NumRings,
                          int NumSectors);
struct hips_file putcircle(struct hips_file c1,int i1,int j1,int scale,
                          int dataout);

struct image_stack rescaleN(struct image_stack current,float* score,
                           struct image_stack orig);
struct image_stack init_is();
struct image_stack create_sff_stack(int x, int y);
struct image_stack isrt2ij(struct image_stack rt,struct point centre,
                          int NumPixels, int NumRings);
struct hips_file updatemask(float match_score[NUMMODELS][NUMSCALES],
                            struct hips_file match_mask,
                            struct image_stack models[NUMMODELS]);
struct image_stack rescale(struct image_stack current,float* scores);

struct sacp* newsacp();
void load_models(struct image_stack models[NUMMODELS],int NumPixels,
                 int NumRings, int NumSectors);
void mkfilenames(char* OfileRT, char* OfileIJ, int NumRings, int NumSectors,
                 int NumPixels, char* fstub);
void unalloc(struct hips_file hf);
void unallocim(struct image_stack im);
void load_map_file(char* table_file_rt, char* table_file_ij, int* NumRings,
                   int* NumSectors, int* NumPixels);
void write_file(struct hips_file hfile, char* file_name);
void write_to_sff(struct hips_file sff, struct hips_file h,struct point p);
void error_msg(char *RoutineName, char *Message);
void run_xv(char* filename);
void dump_ims(struct image_stack ims,char* fstub);
```

```
void update_sff(struct image_stack sff, struct image_stack view,
                struct point p, int NumPixels);
void itoa(int n, char s[]);
void reverse(char s[]);
void printsaclist(struct sacpstack stk);
struct sacpstack sortstack(struct sacpstack stk);
struct sacpstack initsacpstack();
struct sacpstack pop(struct sacpstack stk,struct sacp* entry);

int max3(float a,float b, float c);
int match(struct image_stack models[NUMMODELS],struct image_stack cv,
          float score[NUMMODELS][NUMSCALES],struct hips_file match_mask);
double match1(struct hips_file Data, struct hips_file Model,int scale,
              struct hips_file mask,struct hips_file int_mask, char*);
double match2(struct hips_file Data, struct hips_file Model,int scale,
              struct hips_file match_mask,struct hips_file int_mask,
              char* ftype);
struct sacpstack updatesaclist(struct sacpstack stk,struct image_stack ims,
                               struct point sp);


struct entryij *head_rt[MAXRINGS][MAXSECTORS];
struct entryrt *head_ij[MAXFOVSIZE][MAXFOVSIZE];

void main()
{
        int SFFx=512,SFFy=512;
        char name[80],hin[80],hout[80];
        int i,j;
        int NumRows,NumCols,NumRings,NumSectors,NumPixels;
        struct hips_file hfile_in,hfile_ij,hfile_rc,hfile_norm,cv_mask,
                         hfile_ij_rc,hfile_mf,hfile_corner,hfile_corner2,
                         hfile_ijc,hfile_ijc2,hfile_mrvc,hfile_ijmrvc,
                         hfcorner3,hfijc3,hftest,hfcorner4,hfijc4,hfclip,
                         hfreduce,hfexpand,hfcij1,hfcij,match_mask;
        struct image_stack stable_ff;
        struct image_stack input_image,original_view;
        struct image_stack current_view,current_view_new;
        struct image_stack current_view_new_ij,current_view_ij;
        struct point saccade_point;
        struct point centre,sff_point;
        struct sacpstack saccadelist;

        int best;
        int update;
        int com;
        char fname[80];
        short loop=1;
        char OfileRT[80],OfileIJ[80];
        int v1,v2,v3;
        struct image_stack models[NUMMODELS];
        float match_score[NUMMODELS][NUMSCALES];
```

```
    printf("Loading mapping files..\n");

    current_view=init_is();

    stable_ff=create_sff_stack(SFFx,SFFy);
    input_image.intensity=get_image2("in1");
    saccadelist=initsacpstack();

    saccade_point.x=256;
    saccade_point.y=256;
    mkfilenames(OfileRT,OfileIJ,7,32,128,"tab");
    load_map_file(OfileRT,OfileIJ,&NumRings,&NumSectors,&NumPixels);

    match_mask=create_hips(NumSectors,NumRings);
    for(i=0;i<NumSectors*NumRings;i++)
        {
            match_mask.image[i]=255;
        }

    centre.x=NumPixels;
    centre.y=NumPixels;

    printf("Loading models..\n");
    load_models(models,NumPixels,NumRings,NumSectors);

    sff_point.x=saccade_point.x-centre.x;
    sff_point.y=saccade_point.y-centre.y;
    original_view=init_is();

while(1)
  {
    while(loop)
    {
      printf("\n1-saccade 2-xv 3-stop  5-new >");
      scanf("%d",&com);
      switch(com)
        {
        case 1:/* Saccade */
          scanf("%d",&saccade_point.x);
          scanf("%d",&saccade_point.y);
          sff_point.x=saccade_point.x-centre.x;
          sff_point.y=saccade_point.y-centre.y;
          loop=0;
          break;
        case 2:/* run xv */
          scanf("%s",fname);
          run_xv(fname);
          loop=1;
```

```
      break;
    case 3:/*quit*/
      exit(0);
    case 5:/* load new world view*/
      scanf("%s",fname);
      unalloc(input_image.intensity);
      input_image.intensity=get_image2(fname);
      unallocim(stable_ff);
      stable_ff=create_sff_stack(SFFx,SFFy);
      saccadelist=initsacpstack();
      run_xv(fname);
      loop=1;
      break;

    case 6:/* allow matcher to perform another iteration*/
      loop=0;
      break;
    case 7:/* load new table files */
      scanf("%d %d %d",&NumRings,&NumSectors,&NumPixels);
      printf("Loading mapping files..\n");
      mkfilenames(OfileRT,OfileIJ,NumRings,NumSectors,NumPixels,
                  "tab");
      load_map_file(OfileRT,OfileIJ,&NumRings,
                    &NumSectors,&NumPixels);
      centre.x=NumPixels;
      centre.y=NumPixels;
      load_models(models,NumPixels,NumRings,NumSectors);


      loop=1;
      break;

    default:
      loop=1;
    }
}
loop=1;

if(com!=6)
  { /* get original view of world*/
    unallocim(original_view);
    original_view=init_is();
    original_view.intensity=ij2rt(saccade_point,
                                  input_image.intensity,
                                  NumRings,NumSectors);

    original_view.edge=rcross(original_view.intensity);

    original_view.corner=corner(input_image.intensity,saccade_point,
                                NumPixels,NumRings,NumSectors);

    current_view=init_is();
```

```
    current_view.intensity=ij2rt(saccade_point,
                                 input_image.intensity,
                                 NumRings,NumSectors);

    current_view.edge=rcross(current_view.intensity);

    current_view.corner=corner(input_image.intensity,saccade_point,
                               NumPixels,NumRings,NumSectors);

  }
else
  {
    current_view=current_view_new;
  }

dump_ims(current_view,"viewrt");

current_view_ij=init_is();
current_view_ij=isrt2ij(current_view,centre,
                        NumPixels,NumRings);
dump_ims(current_view_ij,"viewij");

saccadelist=updatesaclist(saccadelist,current_view,saccade_point);
saccadelist=sortstack(saccadelist);
printsaclist(saccadelist);

best=match(models,current_view,match_score,match_mask);
printf("Best match=%d\n",best);

for(j=0;j<NUMSCALES;j++)
  {
    printf("%d ",j);
    if (j==0) printf("reduced ");
    if (j==1) printf("normal ");
    if (j==2) printf("enlarged ");
    for(i=0;i<NUMMODELS;i++)
      printf("%1.2f ",match_score[i][j]);
    printf("%1.2f",current_view.scale);
    printf("\n");
  }

/*match_mask=updatemask(match_score,match_mask,models);*/
/*write_file(match_mask,"mmrt");*/

current_view_new=init_is();
current_view_new=rescaleN(current_view,match_score[best],
                          original_view);

update=0; /* has matcher converged ? */
if (fabs(1-(current_view_new.scale/current_view.scale))<.02)
  update=1;
```

```
        printf("cvnew scale %f       old scale %f\n",
               current_view_new.scale,current_view.scale);

        if (update)
          { /* if converged update SFF */
            printf("Scale -------> %f\n",current_view_new.scale);
            original_view.scale=current_view_new.scale;
            update_sff(stable_ff,original_view,sff_point,NumPixels);
            dump_ims(stable_ff,"sff");
            unallocim(current_view_new);
            unallocim(original_view);
            loop=1;
          }
        unallocim(current_view);
        unallocim(current_view_ij);
      }
}



struct hips_file updatemask(float match_score[NUMMODELS][NUMSCALES],
                            struct hips_file match_mask,
                            struct image_stack models[NUMMODELS])
{   /* update f/g mask based on match scores - not used */
    struct hips_file newmask;
    double fit,weight,total=0.;
    int max,i,j,k;
    float threshold=0.;
    int cols=match_mask.hd->cols;
    int rows=match_mask.hd->rows;
    int np=rows*cols;
    char* rtn_name="updatemask";

    unalloc(match_mask);
    newmask=create_hips(cols,rows);

    for (i=0;i<NUMMODELS;i++)
      {
        max=max3(match_score[i][0],match_score[i][1],match_score[i][2]);
        fit=match_score[i][max];
        if (fit>threshold) total+=fit;
      }


    for (k=0;k<NUMMODELS;k++)
      {
        max=max3(match_score[k][0],match_score[k][1] ,match_score[k][2]);
        fit=match_score[k][max];
        weight=fit/total;
        if (fit>threshold)
          for(j=0;j<rows;j++)
```

```
            for(i=0;i<cols;i++)
              {
                if(models[k].intensity.image[i+j*cols])
                      newmask.image[i+j*cols]+=weight*255;
              }
       }
    return(newmask);
}


struct sacp* newsacp(int x, int y, int pri)
{  /* alloc mem for saccade list element */
    struct sacp* sp;
    char* rtn_name="newsacp";

    if (!(sp=malloc(sizeof(struct sacp))))
        error_msg(rtn_name,"Not again? Ran out of memory!!");
    sp->x=x;
    sp->y=y;
    sp->pri=pri;
    sp->next=(struct sacp*)NULL;

    return sp;
}

struct sacpstack initsacpstack()
{ /* initalize saccade list */
  struct sacpstack stk;

  stk.length=0;
  stk.current=(struct sacp*)NULL;
  stk.base=(struct sacp*)NULL;

  return stk;
}

void printsaclist(struct sacpstack stk)
{ /* display saccade list */
  struct sacp* ptr;

  ptr=stk.base;

  while (ptr != (struct sacp*)NULL)
    {
      printf("priority %d    i %d     j %d\n",ptr->pri,ptr->x,ptr->y);
      ptr=ptr->next;
    }
}

struct sacpstack updatesaclist(struct sacpstack stk,struct image_stack ims,
                               struct point sp)
{   /* add pos'ns of corners to saccade list */
```

```
    int i,j,x,y,pri;
    float theta,r;
    int threshold=50;
    int cols=ims.corner.hd->cols;
    int rows=ims.corner.hd->rows;
    struct hips_file lmax,lmij;
    struct point p;

    p.x=128;p.y=128;
    lmax=localmax(ims.corner);
    /*write_file(lmax,"LMAX");*/
    lmij=rt2ij(p,lmax,257,257,128,rows);
    /*write_file(lmij,"LMIJ");*/

    for(j=0;j<rows;j++)
        for(i=0;i<cols;i++)
            {
                if(lmax.image[i+j*cols]>threshold)
                    {   /* calc. centre or RT pixel */
                        if (i)
                          theta=2*M_PI*i/cols+M_PI/cols;
                        else
                          theta=M_PI/cols;
                        r=pow(2,j+1)-0.5*pow(2,j);
                        x=-r*cos(theta);
                        y=-r*sin(theta);
                        if (sqrt(x*x+y*y) >3)
                          {
                            x+=sp.x;
                            y+=sp.y;
                            pri=(2000-(j+1)*200)+
                              ims.corner.image[i+j*cols]/(j+1);
                            if (stk.length)
                              {
                                stk.current->next=newsacp(x,y,pri);
                                stk.current=stk.current->next;
                              }
                            else
                              {
                                stk.base=newsacp(x,y,pri);
                                stk.current=stk.base;
                              }
                            stk.length++;
                          }
                    }
            }
    unalloc(lmax);
    return stk;
}


struct sacpstack sortstack(struct sacpstack stk)
```

```
{  /* sort saccade list into priority order */

  static int sacpcompare(struct sacp* i,struct sacp* j)
  {
    return(i->pri - j->pri);
  }

  struct sacp *sacparray,entry;
  char* rtn_name="sortstack";
  int i,exists=0,length=0;

  if (!(sacparray=malloc(sizeof(struct sacp)*stk.length)))
        error_msg(rtn_name,"Not again? Ran out of memory!!");


  while (stk.base != (struct sacp*)NULL)
    {
      stk=pop(stk,&entry);
      exists=0;
      for(i=0;i<length;i++)
        if ((entry.x!=sacparray[i].x ||
             entry.y!=sacparray[i].y) && !exists)
          exists=0; else exists=1;
      if (!exists) sacparray[length++]=entry;
    }

  qsort(sacparray,length,sizeof(struct sacp),sacpcompare);

  for (i=length-1;i>=0;i--)
    {
      if (stk.length)
        {
          stk.current->next=
            newsacp(sacparray[i].x,sacparray[i].y,sacparray[i].pri);
          stk.current=stk.current->next;
        }
      else
        {
          stk.base=newsacp(sacparray[i].x,sacparray[i].y,sacparray[i].pri);
          stk.current=stk.base;
        }
      stk.length++;

    }

  free(sacparray);
  return(stk);
}

struct sacpstack pop(struct sacpstack stk,struct sacp* sptr)
{  /* remove element from list */
  *sptr=*stk.base;
```

```
  free(stk.base);
  stk.base=sptr->next;
  stk.length--;

  return stk;
}

struct hips_file corner(struct hips_file hf,struct point saccade_point,
                        int NumPixels,int NumRings,int NumSectors)
{  /* get corner image */
    struct hips_file hfclip;
    struct point centre;
    struct cstruct corners;

    centre.x=NumPixels;
    centre.y=NumPixels;

    hfclip=clip(hf,saccade_point,NumPixels);
    corners=cornerij2(hfclip,NumRings,NumSectors);

    unalloc(hfclip);
    unalloc (corners.cimage2);

    return corners.cimage1;
}


void mkfilenames(char* OfileRT, char* OfileIJ, int NumRings, int NumSectors,
                 int NumPixels, char* fstub)
{ /* create filename for table files give num rings, sectors, pixels */
        char tmp[5];

        strcpy(OfileRT,fstub);
        strcpy(OfileIJ,fstub);
        strcat(OfileRT,"RT");
        strcat(OfileIJ,"IJ");
        itoa(NumRings,tmp);
        strcat(OfileRT,tmp);
        strcat(OfileIJ,tmp);
        itoa(NumSectors,tmp);
        strcat(OfileRT,tmp);
        strcat(OfileIJ,tmp);
        itoa(NumPixels,tmp);
        strcat(OfileRT,tmp);
        strcat(OfileIJ,tmp);
}


void unalloc(struct hips_file hf)
{ /* free mem for hips files */
  free(hf.hd);
  free(hf.image);
```

```
}

void unallocim(struct image_stack im)
{  /* free mem for image stack */
    if (im.red.image) unalloc(im.red);
    if (im.green.image) unalloc(im.green);
    if (im.blue.image) unalloc(im.blue);
    if (im.intensity.image) unalloc(im.intensity);
    if (im.edge.image) unalloc(im.edge);
    if (im.corner.image) unalloc(im.corner);
    if (im.label.image) unalloc(im.label);
    if (im.blob.image) unalloc(im.blob);
    if (im.mask.image) unalloc(im.mask);
}

void itoa(int n, char s[])
  {
    int i,sign;
    if ((sign =n) <0)
      n=-n;
    i=0;
    do {
      s[i++] = n % 10 +'0';
    } while ((n/=10)>0);
    if (sign <0)
      s[i++] ='-';
    s[i]='\0';
    reverse(s);
  }

void reverse(char s[])
{
  int c,i,j;

  for (i=0,j=strlen(s)-1;i<j;i++,j--) {
    c=s[i];
    s[i]=s[j];
    s[j]=c;
  }
}


void dump_ims(struct image_stack ims,char* fstub)
{ /* write image stack to disk */
  char fname[40];

  strcpy(fname,fstub);
  strcat(fname,".r");
  if (ims.red.image!=NULL) write_file(ims.red,fname);
  else printf("%s - red image null.\n",fname);
  strcpy(fname,fstub);
  strcat(fname,".g");
```

```
  if (ims.green.image!=NULL) write_file(ims.green,fname);
  else printf("%s - green image null.\n",fname);
  strcpy(fname,fstub);
  strcat(fname,".b");
  if (ims.blue.image!=NULL) write_file(ims.blue,fname);
  else printf("%s - blue image null.\n",fname);
  strcpy(fname,fstub);
  strcat(fname,".i");
  if (ims.intensity.image!=NULL) write_file(ims.intensity,fname);
  else printf("%s - intensity image null.\n",fname);
  strcpy(fname,fstub);
  strcat(fname,".e");
  if (ims.edge.image!=NULL) write_file(ims.edge,fname);
  else printf("%s - edge image null.\n",fname);
  strcpy(fname,fstub);
  strcat(fname,".co");
  if (ims.corner.image!=NULL) write_file(ims.corner,fname);
  else printf("%s - corner image null.\n",fname);
  strcpy(fname,fstub);
  strcat(fname,".bl");
  if (ims.blob.image!=NULL) write_file(ims.blob,fname);
  else printf("%s - blob image null.\n",fname);
  strcpy(fname,fstub);
  strcat(fname,".l");
  if (ims.label.image!=NULL) write_file(ims.label,fname);
  else printf("%s - label image null.\n",fname);
  strcpy(fname,fstub);
  strcat(fname,".m");
  if (ims.mask.image!=NULL) write_file(ims.mask,fname);
  else printf("%s - mask image null.\n",fname);
}

struct image_stack create_sff_stack(int x, int y)
{ /* create sff */
  char* rtn_name="create_sff_stack";
  struct image_stack ims;
  int i;
  ims=init_is();

  ims.red=create_hips(x,y);
  ims.green=create_hips(x,y);
  ims.blue=create_hips(x,y);
  ims.intensity=create_hips(x,y);
  ims.edge=create_hips(x,y);
  ims.corner=create_hips(x,y);
  ims.mask=create_hips(x,y);
  ims.label=create_hips(x,y);
  ims.blob=create_hips(x,y);
  for(i=0;i<x*y;i++) /* mask image id's resolution of data written to sff */
    ims.mask.image[i]=255;
  return ims;
}
```

```
void load_models(struct image_stack models[NUMMODELS],int NumPixels,
                 int NumRings, int NumSectors)
{ /* load IJ images of models and save RT images */
  void deresmodel(struct hips_file model,int corner)
    {
      int i;

      if (corner)
         for(i=0;i<model.hd->cols;i++)
           {
             model.image[i+model.hd->cols]=model.image[i];
             model.image[i]=0;
           }
      else
        for(i=0;i<model.hd->cols;i++)
          {
            model.image[i+model.hd->cols]*=.75;
            model.image[i+model.hd->cols]+=.25*model.image[i];
            model.image[i]=0;
          }
    }

    /* Need to change for colour (edges&corners etc)*/
    char fstub[100],fname[100];
    char tmp[10];
    int i,j;
    struct point centre;
    struct hips_file hftemp;

    centre.x=NumPixels;
    centre.y=NumPixels;

    for(i=0;i<NUMMODELS;i++)
            {
                models[i]=init_is();
                strcpy(fstub,"model");
                models[i]=init_is();
                itoa(i+1,tmp);
                strcat(fstub,tmp);
                strcpy(fname,fstub);
                strcat(fname,".i");
                hftemp=get_image2(fname);
                models[i].intensity=ij2rt(centre,hftemp,
                                          NumRings,NumSectors);
                models[i].edge=rcross(models[i].intensity);
                models[i].corner=corner(hftemp,centre,NumPixels,NumRings,
                                        NumSectors);
                deresmodel(models[i].intensity,0);
                deresmodel(models[i].edge,0);
                deresmodel(models[i].corner,1);
                strcpy(fstub,"modelrt");
```

```
                itoa(i+1,tmp);
                strcat(fstub,tmp);
                strcpy(fname,fstub);
                strcat(fname,".i");
                write_file(models[i].intensity,fname);
                strcpy(fname,fstub);
                strcat(fname,".e");
                write_file(models[i].edge,fname);
                strcpy(fname,fstub);
                strcat(fname,".co");
                write_file(models[i].corner,fname);
                unalloc(hftemp);
            }
}


struct hips_file create_hips(int x, int y)
{ /* alloc mem for hips file header and image array x*y */
  struct hips_file hf;
  char* rtn_name="create_hips";

  if (!(hf.hd=malloc(sizeof(struct header))))
        error_msg(rtn_name,"Not again? Ran out of memory!!");

  /* output r-theta array */
  if ((hf.image=(unsigned char *)calloc(y*x,1))
      <=(unsigned char *)0)
    error_msg(rtn_name,"SYSTEM OVERLOAD! Ran out of memory!!");

  init_header(hf.hd,"","",1,"",y,x,8,0,0,"","");
  return hf;
}


struct hips_file create_cv_mask(int NumRings, int NumSectors)
{ /* Make current view mask for updating SFF */
  char* rtn_name="create_cv_mask";
  struct hips_file cv_mask;
  int i,j;

  cv_mask=create_hips(NumSectors,NumRings);

  for(j=0;j<NumRings;j++)
    {
      for(i=0;i<NumSectors;i++)
        {
          cv_mask.image[i+j*NumSectors]=j+1;
        }
    }
  return cv_mask; /* an RT image with the rows numbered 1-NumRings */
```

```
}


void update_sff(struct image_stack sff, struct image_stack viewrt,
                struct point p,int NumPixels)
{ /* write data to sff if higher resl'n than existing data */
  /* should only update if more probable than existing data */
  /* but doesn't */
  int i,i2,j,j2,cols=sff.mask.hd->cols,rows=sff.mask.hd->rows,ok=1;
  struct image_stack view;
  int freeslot;
  struct point centre;

  centre.x=NumPixels;
  centre.y=NumPixels;

  freeslot=1;
  while (sff.labelinfo[freeslot]>-99)
    freeslot++;

  viewrt.mask=create_cv_mask(viewrt.intensity.hd->rows,
                             viewrt.intensity.hd->cols);

  view=isrt2ij(viewrt,centre,NumPixels,viewrt.intensity.hd->rows);


  if (ok)
    {
      sff.labelinfo[freeslot]=view.scale;
      for(j=p.y,j2=0;j<p.y+view.mask.hd->rows && j+p.y < rows;
          j++,j2++)
        for(i=p.x,i2=0;i<p.x+view.mask.hd->cols && i+p.x < cols;
            i++,i2++)
          {
            if (view.intensity.image[j2*view.mask.hd->cols+i2]>0 &&
                view.mask.image[j2*view.mask.hd->cols+i2] <
                sff.mask.image[j*cols+i])
              {
                if (view.red.image!=NULL)
                  sff.red.image[j*cols+i]=
                    view.red.image[j2*view.red.hd->cols+i2];
                if (view.green.image!=NULL)
                  sff.green.image[j*cols+i]=
                    view.green.image[j2*view.green.hd->cols+i2];
                if (view.blue.image!=NULL)
               sff.blue.image[j*cols+i]=
                  view.blue.image[j2*view.blue.hd->cols+i2];
                  if (view.intensity.image!=NULL)
                    sff.intensity.image[j*cols+i]=
                      view.intensity.image[j2*view.intensity.hd->cols+i2];
```

```
                if (view.edge.image!=NULL)
                  sff.edge.image[j*cols+i]=
                    view.edge.image[j2*view.edge.hd->cols+i2];
                if (view.corner.image!=NULL)
                  sff.corner.image[j*cols+i]=
                view.corner.image[j2*view.corner.hd->cols+i2];
                sff.mask.image[j*cols+i] =
                  view.mask.image[j2*view.mask.hd->cols+i2];
                sff.label.image[j*cols+i]=freeslot;
              }
            }
      }
    else
      printf("SFF not updated - more probable object already present!\n");
    unalloc(viewrt.mask);
    unallocim(view);
}


struct image_stack isrt2ij(struct image_stack rt,struct point centre, int NumPi
{ /* return IJ image stack when given RT im. stack */
    struct image_stack ij;
    ij=init_is();

    if(rt.red.image!=NULL)
      ij.red=rt2ij(centre,rt.red,2*NumPixels+1,2*NumPixels+1, NumPixels,
                   NumRings);
    if(rt.green.image!=NULL)
      ij.green=rt2ij(centre,rt.green,2*NumPixels+1,2*NumPixels+1, NumPixels,
                     NumRings);
    if(rt.blue.image!=NULL)
      ij.blue=rt2ij(centre,rt.blue,2*NumPixels+1,2*NumPixels+1, NumPixels,
                    NumRings);
    if(rt.intensity.image!=NULL)
      ij.intensity=rt2ij(centre,rt.intensity,2*NumPixels+1,2*NumPixels+1,
                         NumPixels,NumRings);
    if(rt.edge.image!=NULL)
      ij.edge=rt2ij(centre,rt.edge,2*NumPixels+1,2*NumPixels+1,
                    NumPixels,NumRings);
    if(rt.corner.image!=NULL)
      ij.corner=rt2ij(centre,rt.corner,2*NumPixels+1,2*NumPixels+1,
                      NumPixels,NumRings);
    if(rt.blob.image!=NULL)
      ij.blob=rt2ij(centre,rt.blob,2*NumPixels+1,2*NumPixels+1,
                    NumPixels,NumRings);
    if(rt.mask.image!=NULL)
      ij.mask=rt2ij(centre,rt.mask,2*NumPixels+1,2*NumPixels+1,
                    NumPixels,NumRings);
    if(rt.label.image!=NULL)
      ij.label=rt2ij(centre,rt.label,2*NumPixels+1,2*NumPixels+1,
                    NumPixels,NumRings);
```

```
  return ij;
}

struct image_stack init_is()
{ /* initalize image stack */
  struct image_stack nullis;
  int i;

  nullis.red.image=NULL;
  nullis.green.image=NULL;
  nullis.blue.image=NULL;
  nullis.intensity.image=NULL;
  nullis.edge.image=NULL;
  nullis.corner.image=NULL;
  nullis.blob.image=NULL;
  nullis.label.image=NULL;
  nullis.mask.image=NULL;
  for(i=0;i<256;i++)
    nullis.labelinfo[i]=-99;
  nullis.scale=1;

  return nullis;
}

void run_xv(char* filename)
{
  char xv_path[100]="xv ";
  char* bg="&";
  strcat(xv_path,filename);
  strcat(xv_path,bg);
  system(xv_path);
}

void load_map_file(char* table_file_rt, char* table_file_ij, int* NumRings,
                   int* NumSectors, int* NumPixels)
{ /* load table files */
        double xr,xs,xp,percent;
        int r,theta,i,j,numentry=0;
        FILE *fp;


        /* load mapping file */
        fp = fopen(table_file_rt,"rb");
        if (fp == NULL)
         {printf("WARNING! open failed on mapping file: %s\n",table_file_rt);
                    exit(0);}
        fscanf(fp,"%lf %lf %lf",&xr, &xs, &xp);
        *NumRings = (int) xr;
        *NumSectors = (int) xs;
        *NumPixels = (int) xp;
        for (r=0; r<*NumRings; r++)
            for (theta=0; theta<*NumSectors; theta++)
```

```
        head_rt[r][theta] = (struct entryij *) -1;
fscanf(fp,"%d %d",&r,&theta);

while (r != -1000)
{
        fscanf(fp,"%d %d %lf",&i,&j,&percent);
        while (percent >= 0)
        {
                entIJ[numentry].i = i;
                entIJ[numentry].j = j;
                entIJ[numentry].percent= percent;
                entIJ[numentry].next = head_rt[r][theta];
                head_rt[r][theta] = &entIJ[numentry];
                numentry++;
                if (numentry >= MAXENTRY)
                {
                        printf("Out of entry space\n");
                        exit(0);
                }
                fscanf(fp,"%d %d %lf",&i,&j,&percent);
        }
        fscanf(fp,"%d %d",&r,&theta);
}
fclose(fp);
printf("RT mapping file loaded\n");

/* load IJ mapping file */

numentry=0;

fp = fopen(table_file_ij,"rb");
if (fp == NULL)
 {printf("WARNING! open failed on mapping file: %s\n",table_file_ij);
            exit(0);}
fscanf(fp,"%lf %lf %lf",&xr, &xs, &xp);

for (j=0; j<(2 * *NumPixels+1); j++)
    for (i=0; i<(2 * *NumPixels+1); i+=1)
       head_ij[j][i] = (struct entryrt *) -1;
fscanf(fp,"%d %d",&i,&j);

while (i != -1000)
{
        fscanf(fp,"%d %d %lf",&r,&theta,&percent);
        while (percent >= 0)
        {
            entRT[numentry].r = r;
            entRT[numentry].theta = theta;
            entRT[numentry].percent= percent;
            entRT[numentry].next=head_ij[*NumPixels+j][*NumPixels+i];
            head_ij[*NumPixels+j][*NumPixels+i] = &entRT[numentry];
            numentry++;
```

```
                    if (numentry >= MAXENTRY)
                    {
                       printf("Out of entry space\n");
                       exit(0);
                    }
                    fscanf(fp,"%d %d %lf",&r,&theta,&percent);
                }
                fscanf(fp,"%d %d",&i,&j);
        }
        fclose(fp);
        fprintf(stderr,"IJ mapping file loaded\n");

}



struct hips_file get_image2(char* hips_file_in)
{ /* load hips image into hips_file data struct and return it */
        int fhi;
        long picsize;
        struct hips_file hfile;
        char* rtn_name="get_image2";

        if (!(hfile.hd=malloc(sizeof(struct header))))
            error_msg(rtn_name,"Not again? Ran out of memory!!");

        if ((fhi=open(hips_file_in,O_RDONLY,0))<0)
            {
                fprintf(stderr,"get_image2: couldn't open file %s\n",
                        hips_file_in);
                exit(1);
            }

        fread_header(fhi,hfile.hd);
        if (hfile.hd->pixel_format != PFBYTE)
        {
                fprintf(stderr,"get_image2: frame must be in byte format\n");
                exit(1);
        }
        if (hfile.hd->num_frame != 1)
        {
                printf("Number of images must be 1\n");
                exit(0);
        }
        picsize=hfile.hd->cols * hfile.hd->rows;

        if ((hfile.image=(unsigned char *)calloc(picsize,1))
            <=(unsigned char *)0)
        {
                fprintf(stderr,"Can't allocate input array\n");
                exit(1);
        }
```

```
        pread(fhi,hfile.image,picsize);

        /* close input file */
        close(fhi);
        return hfile;
}

void write_file(struct hips_file hfile, char* hips_file_out)
{ /* write hips_file to disk */
        long picsize = hfile.hd->cols * hfile.hd->rows;
        int fho;
        char command[100];

        strcpy(command,"chmod ugo+rw-x ");
        strcat(command,hips_file_out);

        fho = open(hips_file_out,(O_WRONLY | O_CREAT));
        if (fho<0)
        {
            printf("\nERROR %d\n",errno);
            exit(1);
        }
        fwrite_header(fho,hfile.hd);
        write(fho,hfile.image,picsize);
        close(fho);
        system(command);
        printf("Wrote %s\n",hips_file_out);


}

struct hips_file ij2rt(struct point sp, struct hips_file hfile,
                       int NumRings, int NumSectors)
{ /* return RT hips file when given IJ */
        int di,dj,r,theta;
        struct entryij *ptr;
        double sum;
        struct hips_file hfile_rt;
        char* rtn_name="ij2rt";


        if (!(hfile_rt.hd=malloc(sizeof(struct header))))
            error_msg(rtn_name,"Not again? Ran out of memory!!");

        /* output r-theta array */
        if ((hfile_rt.image=(unsigned char *)calloc(NumRings*NumSectors,1))
            <=(unsigned char *)0)
        {
                fprintf(stderr,"Can't allocate output array\n");
                exit(1);
        }
```

```
        for (r=0; r<NumRings; r++)
          {

        for (theta=0; theta<NumSectors; theta++)
        {
                ptr = head_rt[r][theta];
                sum = 0.0;
                while (ptr != (struct entryij *) -1)
                {
                        dj = ptr->j + sp.y;
                        if (dj < 0 || dj >= hfile.hd->rows) goto xxx1;
                        di = ptr->i + sp.x;
                        if (di < 0 || di >= hfile.hd->cols) goto xxx1;
                        sum += ptr->percent * *(hfile.image +
                                                dj*hfile.hd->cols + di);
xxx1:
                        ptr = ptr->next;
                }
                if (sum < 0) sum = 0;
                if (sum > 255) sum = 255;
                *(hfile_rt.image + r*NumSectors + theta) =
                  (unsigned char) sum;
        }
          }
        init_header(hfile_rt.hd,hfile.hd->orig_name,hfile.hd->seq_name,
                    hfile.hd->num_frame,hfile.hd->orig_date,NumRings,
                    NumSectors,hfile.hd->bits_per_pixel,
                    hfile.hd->bit_packing,hfile.hd->pixel_format,
                    hfile.hd->seq_history,hfile.hd->seq_desc);
        return hfile_rt;
}


struct hips_file rt2ij(struct point sp, struct hips_file hfile,
                       int ysize, int xsize, int NumPixels, int NumRings)
{ /* return IJ hips_file when given RT */

        int r,theta,i,j,di,dj;
        struct entryrt *ptr;
        double sum,percent;
        struct hips_file hfile_ij;
        char* rtn_name="rt2ij";
        /*int outer_ring = inner_ring+NumRings-NUMSCALES;*/

        if (!(hfile_ij.hd=malloc(sizeof(struct header))))
            error_msg(rtn_name,"Not again? Ran out of memory!!");

        if (ysize < 1 || ysize > MAXPIXELS)
        {
          printf("Number of rows must be 1-%d\n",MAXPIXELS);
          exit(0);
        }
```

```
if (xsize < 1 || xsize > MAXPIXELS)
{
  printf("Number of columns must be 1-%d\n",MAXPIXELS);
  exit(0);
}


/* foveate this pixel */
if (sp.x < 0 || sp.x >= xsize || sp.y < 0 || sp.y >= ysize)
  {
    printf("Foveated output must be in the range (0,0)-(%d,%d)\n",
           ysize-1,xsize-1);
    exit(0);
  }

/* output ij array */
if ((hfile_ij.image=(unsigned char *)calloc(ysize*xsize,1))
    <=(unsigned char *)0)
{
        fprintf(stderr,"Can't allocate output array\n");
        exit(1);
}

for (j = -NumPixels; j<=NumPixels; j++)
for (i = -NumPixels; i<=NumPixels; i++)
{
  ptr = head_ij[NumPixels+j][NumPixels+i];
  sum = 0.0;
  while (ptr != (struct entryrt *) -1)
  {
      /*    if (ptr->r >= inner_ring && ptr->r <= outer_ring)*/

        sum += ptr->percent * *(hfile.image+ (ptr->r)*hfile.hd->cols
                              +ptr->theta);

        ptr = ptr->next;
  }
  dj = j+sp.y;
  di = i+sp.x;
  if (di < 0 || dj < 0 || di >= xsize || dj >= ysize) continue;
  if (sum < 0) sum = 0;
  if (sum > 255) sum = 255;
  *(hfile_ij.image + dj*xsize + di) = (unsigned char) sum;
}
init_header(hfile_ij.hd,hfile.hd->orig_name,hfile.hd->seq_name,
            hfile.hd->num_frame,hfile.hd->orig_date,ysize,
            xsize,hfile.hd->bits_per_pixel,
            hfile.hd->bit_packing,hfile.hd->pixel_format,
            hfile.hd->seq_history,hfile.hd->seq_desc);
return hfile_ij;
}
```

```
void error_msg(char *RoutineName, char *Message)
/* print error message and exit program */
{
  fprintf(stderr, "ERROR in %s : \n\t- %s\n", RoutineName, Message);
  exit(-1);
}


struct hips_file rcross(struct hips_file hfile)
{ /* perform rcross operation */
       struct hips_file hfile_out;
       unsigned dataout;
       int i,j;
       char* rtn_name="rcross";

       if (!(hfile_out.hd=malloc(sizeof(struct header))))
            error_msg(rtn_name,"Not again? Ran out of memory!!");

       if ((hfile_out.image=
            (unsigned char *)calloc(hfile.hd->rows*hfile.hd->cols,1))
            <=(unsigned char *)0)
                error_msg(rtn_name,"Can't allocate output array");

       /* apply cross operator */
       for (j=0; j<hfile.hd->rows; j++)
          for (i=0; i<hfile.hd->cols; i++)
          {
            if (i!=hfile.hd->cols-1 && j!=hfile.hd->rows-1)
              {
                dataout=
                   (abs((unsigned)hfile.image[i+j*hfile.hd->cols]-
                        (unsigned)hfile.image[i+1+(j+1)*hfile.hd->cols])+
                    abs((unsigned)hfile.image[i+1+j*hfile.hd->cols]-
                        (unsigned)hfile.image[i+(j+1)*hfile.hd->cols]))/2;
                        if (dataout > 255) dataout = 255;
              }
            else if (i==hfile.hd->cols-1 && j==hfile.hd->rows-1)
              {
                dataout=
                   (abs((unsigned)hfile.image[i+j*hfile.hd->cols]-
                        (unsigned)hfile.image[i-(hfile.hd->cols-1)+
                                              j*hfile.hd->cols])+
                    abs((unsigned)hfile.image[i-(hfile.hd->cols-1)
                                              +j*hfile.hd->cols]-
                        (unsigned)hfile.image[i+j*hfile.hd->cols]))/2;
                        if (dataout > 255) dataout = 255;
              }
            else if (i==hfile.hd->cols-1)
              {
                dataout=
                   (abs((unsigned)hfile.image[i+j*hfile.hd->cols]-
```

```
                        (unsigned)hfile.image[i-(hfile.hd->cols-1)
                                              +(j+1)*hfile.hd->cols])+
                    abs((unsigned)hfile.image[i-(hfile.hd->cols-1)
                                              +j*hfile.hd->cols]-
                        (unsigned)hfile.image[i+(j+1)*hfile.hd->cols]))/2;
                        if (dataout > 255) dataout = 255;
                  }
               else if (j==hfile.hd->rows-1)
                  {
                    dataout=
                       (abs((unsigned)hfile.image[i+j*hfile.hd->cols]-
                            (unsigned)hfile.image[i+1+j*hfile.hd->cols])+
                        abs((unsigned)hfile.image[i+1+j*hfile.hd->cols]-
                            (unsigned)hfile.image[i+j*hfile.hd->cols]))/2;
                            if (dataout > 255) dataout = 255;
                  }

               /* put output pixel */
               hfile_out.image[i+j*hfile.hd->cols]=dataout;
             }
        init_header(hfile_out.hd,hfile.hd->orig_name,hfile.hd->seq_name,
                    hfile.hd->num_frame,hfile.hd->orig_date,hfile.hd->rows,
                    hfile.hd->cols,hfile.hd->bits_per_pixel,
                    hfile.hd->bit_packing,hfile.hd->pixel_format,
                    hfile.hd->seq_history,hfile.hd->seq_desc);
        return hfile_out;
}


struct hips_file moravec(struct hips_file hfile)
{  /* perform moravec operation and return moravec'd hips_file */

        struct hips_file hfile_out;
        int i,j,horiz,vert,diag1,diag2,r1,r2;
        char* rtn_name="moravec";
        int cols=hfile.hd->cols;
        int rows=hfile.hd->rows;
        int numpels=rows*cols;
        float* dataout;
        float range,low=999999.0,high=-99999.;


        if (!(hfile_out.hd=malloc(sizeof(struct header))))
            error_msg(rtn_name,"Not again? Ran out of memory!!");

        if ((hfile_out.image=
             (unsigned char *)calloc(rows*cols,1))
             <=(unsigned char *)0)
                 error_msg(rtn_name,"Can't allocate output array");

        if ((dataout=
             (float *)calloc(rows*cols,sizeof(float)))
```

```
      <=(float *)0)
          error_msg(rtn_name,"Can't allocate dataout array");

/* apply moravec operator */
for (j=1; j<rows-1; j++)
   for (i=1; i<cols-1; i++) /*cols +1 for horizontal wrap */
   {
     vert=diag1=diag2=horiz=0;

     vert=((hfile.image[i%cols+(j-1)*cols] -
               hfile.image[i%cols+j*cols]) *
              (hfile.image[i%cols+(j-1)*cols] -
               hfile.image[i%cols+j*cols]))+
             (hfile.image[i%cols+j*cols] -
               hfile.image[i%cols+(j+1)*cols]) *
              (hfile.image[i%cols+j*cols] -
               hfile.image[i%cols+(j+1)*cols]);
     diag1=(hfile.image[(i-1)%cols+(j-1)*cols] -
               hfile.image[i%cols+j*cols]) *
              (hfile.image[(i-1)%cols+(j-1)*cols] -
               hfile.image[i%cols+j*cols])+
              ((hfile.image[i%cols+j*cols] -
               hfile.image[(i+1)%cols+(j+1)*cols])*
              (hfile.image[i%cols+j*cols] -
               hfile.image[(i+1)%cols+(j+1)*cols]));
     diag2=(hfile.image[(i+1)%cols+(j-1)*cols] -
               hfile.image[i%cols+j*cols]) *
              (hfile.image[(i+1)%cols+(j-1)*cols] -
               hfile.image[i%cols+j*cols])+
              (hfile.image[i%cols+j*cols] -
               hfile.image[(i-1)%cols+(j+1)*cols]) *
              (hfile.image[i%cols+j*cols] -
               hfile.image[(i-1)%cols+(j+1)*cols]);
     horiz=(hfile.image[i%cols+j*cols] -
               hfile.image[(i+1)%cols+j*cols]) *
              (hfile.image[i%cols+j*cols] -
               hfile.image[(i+1)%cols+j*cols])+
              (hfile.image[(i-1)%cols+j*cols] -
               hfile.image[i%cols+j*cols]) *
              (hfile.image[(i-1)%cols+j*cols] -
               hfile.image[i%cols+j*cols]);

     if (vert<horiz) r1= vert; else r1=horiz;
     if (diag1<diag2) r2 = diag1; else r2=diag2;
     if (r1<r2)
       dataout[i%cols+j*cols]=r1;
     else
       dataout[i%cols+j*cols]=r2;


     if (dataout[i%cols+j*cols]>low)
       error_msg(rtn_name,"Dataout out of range.");
```

```
        }
      for (j=0; j<numpels; j++)
        {
          if (dataout[j]>high) high = dataout[j];
          if (dataout[j]<low) low = dataout[j];
        }
      range=high-low;
      for (j=0; j<numpels; j++)
        {
          hfile_out.image[j]=(unsigned)((dataout[j]-low)*255/range);
        }

      init_header(hfile_out.hd,hfile.hd->orig_name,hfile.hd->seq_name,
                  hfile.hd->num_frame,hfile.hd->orig_date,hfile.hd->rows,
                  hfile.hd->cols,hfile.hd->bits_per_pixel,
                  hfile.hd->bit_packing,hfile.hd->pixel_format,
                  hfile.hd->seq_history,hfile.hd->seq_desc);
      free(dataout);
      return hfile_out;
}



struct hips_file localmax(struct hips_file hfile)
{  /* return 8-connected local max of hips file */

      struct hips_file hfile_out;
      int i,j;
      char* rtn_name="localmax";
      int cols=hfile.hd->cols;
      int rows=hfile.hd->rows;
      int numpels=rows*cols;
      int a,b,c,d,e,f,g,h,k;
      if (!(hfile_out.hd=malloc(sizeof(struct header))))
          error_msg(rtn_name,"Not again? Ran out of memory!!");

      if ((hfile_out.image=
           (unsigned char *)calloc(rows*cols,1))
           <=(unsigned char *)0)
               error_msg(rtn_name,"Can't allocate output array");

      for (j=0; j<rows; j++)
        for (i=0; i<cols; i++)
          hfile_out.image[i+j*cols]=hfile.image[i+j*cols];

      for (j=0; j<rows; j++)
         for (i=1; i<cols+1; i++) /*cols +1 for horizontal wrap */
         {
           a=b=c=d=e=f=g=h=k=0;
           if (j>0 && j<rows-1)
             {
               a=hfile_out.image[(i-1)%cols+(j-1)*cols];
```

```
                b=hfile_out.image[i%cols+(j-1)*cols];
                c=hfile_out.image[(i+1)%cols+(j-1)*cols];
                d=hfile_out.image[(i-1)%cols+j*cols];
                e=hfile_out.image[i%cols+j*cols];
                f=hfile_out.image[(i+1)%cols+j*cols];
                g=hfile_out.image[(i-1)%cols+(j+1)*cols];
                h=hfile_out.image[i%cols+(j+1)*cols];
                k=hfile_out.image[(i+1)%cols+(j+1)*cols];
              }
            else
            if (j==0)
              {
                d=hfile_out.image[(i-1)%cols+j*cols];
                e=hfile_out.image[i%cols+j*cols];
                f=hfile_out.image[(i+1)%cols+j*cols];
                g=hfile_out.image[(i-1)%cols+(j+1)*cols];
                h=hfile_out.image[i%cols+(j+1)*cols];
                k=hfile_out.image[(i+1)%cols+(j+1)*cols];
              }
            else
            if (j==rows-1)
              {
                a=hfile_out.image[(i-1)%cols+(j-1)*cols];
                b=hfile_out.image[i%cols+(j-1)*cols];
                c=hfile_out.image[(i+1)%cols+(j-1)*cols];
                d=hfile_out.image[(i-1)%cols+j*cols];
                e=hfile_out.image[i%cols+j*cols];
                f=hfile_out.image[(i+1)%cols+j*cols];
              }

            if (e>a && e>b && e>c && e>d && e>f && e>g && e>h && e>k)
                hfile_out.image[i%cols+j*cols]=e;
            else
                hfile_out.image[i%cols+j*cols]=0;
          }



        init_header(hfile_out.hd,hfile.hd->orig_name,hfile.hd->seq_name,
                    hfile.hd->num_frame,hfile.hd->orig_date,hfile.hd->rows,
                    hfile.hd->cols,hfile.hd->bits_per_pixel,
                    hfile.hd->bit_packing,hfile.hd->pixel_format,
                    hfile.hd->seq_history,hfile.hd->seq_desc);

        return hfile_out;
}


double match1(struct hips_file Data, struct hips_file Model,int scale,
              struct hips_file match_mask,struct hips_file int_mask,
              char* ftype)
```

```
{ /* match Data against Mask by totalling the abs of subtracting */
  /* one image from the other (only pixels under model are matched)*/
  /* returns sum of 1-abs((m(ij)-d(ij))/255) */
  /* divided by number of pixels matched */

  /* match 5 rows of data against middle 5 rows of model */
       struct hips_file hfile_out;
       int numpels=0;
       int rows=Data.hd->rows, cols=Data.hd->cols;
       char* rtn_name="match1";
       int i,j,k;
       double dataout;
       double result=0.;
       int threshold=0;
       unsigned char row[2][32];
       struct hips_file adjusted,IJ;
       char tmp[6],fname[30]="adjusted";
       struct point p;

       /* adjust image resolution */
       adjusted=create_hips(cols,rows);
       for(j=0;j<rows;j++)
         for(i=0;i<cols;i++)
           adjusted.image[i+j*cols]=Data.image[i+j*cols];
       if (scale)
         {
           for (i=0;i<cols;i++)
            {
              adjusted.image[i+cols]*=.75;
              adjusted.image[i+cols]+=.25*adjusted.image[i];
              adjusted.image[i]=adjusted.image[i+cols];
            }
         }
       if (scale==2)
         {
           for (i=cols;i<2*cols;i++)
             {
               adjusted.image[i+cols]*=.75;
               adjusted.image[i+cols]+=.25*adjusted.image[i];
               adjusted.image[i]=adjusted.image[i+cols];
               adjusted.image[i-cols]=adjusted.image[i+cols];
             }
         }
       strcat(fname,ftype);
       itoa(scale,tmp);
       strcat(fname,tmp);
       write_file(adjusted,fname);
       strcat(fname,".ij");
       p.x=128;p.y=128;
       IJ=rt2ij(p,adjusted,256,256,128,7);
       write_file(IJ,fname);
       unalloc(IJ);
```

```
        if (Data.hd->rows != Model.hd->rows || Data.hd->cols!=Model.hd->cols)
            error_msg(rtn_name,"Match on unequal files");
        /* scale 0 matches model against 1st 5 rows of data */
        for (j=scale,k=1;j<rows-(NUMSCALES-scale-1);j++,k++){
        for (i=0;i<cols;i++)
          /* only match if pixel is under intensity mask */
          if (int_mask.image[i+k*cols] > 0)
          {
            dataout=abs(adjusted.image[i+j*cols]-Model.image[i+k*cols]);
            if (dataout>0) dataout/=255.;
            result+=(1-dataout)*match_mask.image[i+k*cols]/255.;
            /*printf("%0.2f ",1-dataout);*/
            numpels++;
          } /*else printf("     ");*/
       /* printf("\n");*/
      }
        printf("%d pixels matched\n",numpels);
        if (result) result/=numpels;
        printf("%f\n",result);
        unalloc(adjusted);
        return (result);
}

double match2(struct hips_file Data, struct hips_file Model,int scale,
              struct hips_file match_mask,struct hips_file int_mask,
              char* ftype)
{ /* match Data against Mask by totalling the abs of subtracting */
  /* one image from the other (only pixels under model are matched)*/
  /* returns sum of 1-abs((m(ij)-d(ij))/255) */
  /* divided by number of pixels matched */
  /* match function for corners (copies high res corners rather */
  /* than averaging them to maintain corner strength in resolution */
  /* changing procedure - not good */
  /* match 5 rows of data against middle 5 rows of model */
        struct hips_file hfile_out;
        int numpels=0;
        int rows=Data.hd->rows, cols=Data.hd->cols;
        char* rtn_name="match2";
        int i,j,k;
        double dataout;
        double result=0.;
        int threshold=0;
        unsigned char row[2][32];
        struct hips_file adjusted,IJ;
        char tmp[6],fname[30]="adjusted";
        struct point p;

        adjusted=create_hips(cols,rows);
        for(j=0;j<rows;j++)
          for(i=0;i<cols;i++)
            adjusted.image[i+j*cols]=Data.image[i+j*cols];
        if (scale)
```

```
        {
          for (i=0;i<cols;i++)
           {
             adjusted.image[i+cols]=adjusted.image[i];
             adjusted.image[i]=0;
           }
        }
      if (scale==2)
        {
          for (i=cols;i<2*cols;i++)
            {
              adjusted.image[i+cols]= adjusted.image[i];
              adjusted.image[i]=0;
            }
        }
      strcat(fname,ftype);
      itoa(scale,tmp);
      strcat(fname,tmp);
      write_file(adjusted,fname);
      strcat(fname,".ij");
      p.x=128;p.y=128;
      IJ=rt2ij(p,adjusted,256,256,128,7);
      write_file(IJ,fname);
      unalloc(IJ);
      if (Data.hd->rows != Model.hd->rows || Data.hd->cols!=Model.hd->cols)
          error_msg(rtn_name,"Match on unequal files");
      /* scale 0 matches model against 1st 5 rows of data */
      for (j=scale,k=1;j<rows-(NUMSCALES-scale-1);j++,k++){
      for (i=0;i<cols;i++)
        /* only match if pixel is under intensity mask */
        if (int_mask.image[i+k*cols] > 0)
        {
          dataout=abs(adjusted.image[i+j*cols]-Model.image[i+k*cols]);
          if (dataout>0) dataout/=255.;
          result+=(1-dataout)*match_mask.image[i+k*cols]/255.;
          /*printf("%0.2f ",1-dataout);*/
          numpels++;
        } /*else printf("     ");*/
      /*printf("\n");*/
    }
      printf("%d pixels matched\n",numpels);
      if (result) result/=numpels;
      printf("%f\n",result);
      unalloc(adjusted);
      return (result);
}


int match(struct image_stack models[NUMMODELS],
          struct image_stack cv, float score[NUMMODELS][NUMSCALES],
          struct hips_file match_mask)
{  /* match image stack against model stacks */
```

```
int i,j,max;
float wfac,maxscore=0.;
char ftype[10];
char tmp[5];


for(i=0;i<NUMMODELS;i++)
  for(j=0;j<NUMSCALES;j++)
    {
        itoa(i,tmp);
        score[i][j]=0;
        wfac=0;
        if (cv.red.image)
            {
                score[i][j]+=
                  REDWEIGHT*match1(cv.red,models[i].red,j,
                                   match_mask,models[i].intensity,"");
                wfac+=REDWEIGHT;
            }
        if (cv.green.image)
            {
                score[i][j]+=
                    GREENWEIGHT*match1(cv.green,models[i].green,j,
                                       match_mask,models[i].intensity,"");
                wfac+=GREENWEIGHT;
            }
        if (cv.blue.image)
            {
                score[i][j]+=
                    BLUEWEIGHT*match1(cv.blue,models[i].blue,j,
                                      match_mask,models[i].intensity,"");
                wfac+=BLUEWEIGHT;
            }
        if (cv.intensity.image)
            {
                strcpy(ftype,tmp);
                strcat(ftype,".i.");
                printf("intensity \n");
                score[i][j]+=INTENSITYWEIGHT*
                    match1(cv.intensity,models[i].intensity,j,
                           match_mask,models[i].intensity,ftype);
                wfac+=INTENSITYWEIGHT;
            }
        if (cv.edge.image)
            {
                printf("edge \n");
                score[i][j]+=
                    EDGEWEIGHT*match1(cv.edge,models[i].edge,j,
                                      match_mask,models[i].intensity,
                                      ".e");
                wfac+=EDGEWEIGHT;
            }
```

```
            if (cv.corner.image)
                {
                    printf("corner \n");
                    score[i][j]+=
                        CORNERWEIGHT*match2(cv.corner,models[i].corner,j,
                                            match_mask,models[i].intensity,
                                            ".co");
                    wfac+=CORNERWEIGHT;
                }
            if (cv.blob.image)
                {
                    score[i][j]+=
                        BLOBWEIGHT*match1(cv.blob,models[i].blob,j,
                                          match_mask,models[i].intensity,"");
                    wfac+=BLOBWEIGHT;
                }
            if (cv.label.image)
                {
                    score[i][j]+=
                        LABELWEIGHT*match1(cv.label,models[i].label,j,
                                           match_mask,models[i].intensity,"");
                    wfac+=LABELWEIGHT;
                }
            score[i][j]/=wfac;
            if (score[i][j]>maxscore)
                {
                    max = i;
                    maxscore=score[i][j];
                }
            printf("model %d scale %d score %f ",i+1,j,score[i][j]);
            if (j==0) printf("(object is distant)\n");
            if (j==1) printf("(normal)\n");
            if (j==2) printf("(object is nearby)\n");
        }
    return max;
}

struct hips_file rescale1(struct hips_file hf1,float scale_val)
{ /* rescale hips_file * 1+scale_val */
  struct hips_file hfile_out;
  int rows=hf1.hd->rows;
  int cols=hf1.hd->cols;
  char* rtn_name="rescale";
  int i,j,dataout;

/* +scale_val -> zoom in scale_val   -scale_val -> zoom out */
/* scale_val =+1.0 -> zoom in to double size of image in x and y */

  if (!(hfile_out.hd=malloc(sizeof(struct header))))
    error_msg(rtn_name,"Not again? Ran out of memory!!");

  if ((hfile_out.image=
```

```
        (unsigned char *)calloc(hf1.hd->rows*hf1.hd->cols,1))
       <=(unsigned char *)0)
     error_msg(rtn_name,"Can't allocate output array");

   if (scale_val > 0)
     {
       for (j=rows-1; j>0; j--)
         for (i=0; i<cols; i++)
           hfile_out.image[i+j*cols]=(1-scale_val)*hf1.image[i+j*cols]+
                                     scale_val*hf1.image[i+(j-1)*cols];

       for (i=0;i<cols;i++)
         hfile_out.image[i]=(1-scale_val)*hf1.image[i];
     }
   else
     {
       scale_val*=-1;
       for (j=0; j<rows; j++)
         for (i=0; i<cols; i++)
             {
                 if (j==rows-1)
                     hfile_out.image[i+j*cols]=
                         (1-scale_val)*hf1.image[i+j*cols];
                 else
                     hfile_out.image[i+j*cols]=
                         (1-scale_val)*hf1.image[i+j*cols]+
                         scale_val*hf1.image[i+(j+1)*cols];
             }
     }

   init_header(hfile_out.hd,hf1.hd->orig_name,hf1.hd->seq_name,
               hf1.hd->num_frame,hf1.hd->orig_date,hf1.hd->rows,
               hf1.hd->cols,hf1.hd->bits_per_pixel,
               hf1.hd->bit_packing,hf1.hd->pixel_format,
               hf1.hd->seq_history,hf1.hd->seq_desc);
   return hfile_out;
}

struct hips_file rescale2(struct hips_file hf1,float scale_val)
{ /* rescale hips_file * 1+scale_val */
   /* keeps top row when zooming out - doesn't get rid of corners */
   /* (horrible kludge)*/
   struct hips_file hfile_out;
   int rows=hf1.hd->rows;
   int cols=hf1.hd->cols;
   char* rtn_name="rescale";
   int i,j,dataout;

/* +scale_val -> zoom in scale_val   -scale_val -> zoom out */
/* scale_val =+1.0 -> zoom in to double size of image in x and y */

   if (!(hfile_out.hd=malloc(sizeof(struct header))))
```

```
    error_msg(rtn_name,"Not again? Ran out of memory!!");

  if ((hfile_out.image=
       (unsigned char *)calloc(hf1.hd->rows*hf1.hd->cols,1))
      <=(unsigned char *)0)
    error_msg(rtn_name,"Can't allocate output array");

  if (scale_val > 0)
    {
      for (j=rows-1; j>0; j--)
        for (i=0; i<cols; i++)
          hfile_out.image[i+j*cols]=(1-scale_val)*hf1.image[i+j*cols]+
                                    scale_val*hf1.image[i+(j-1)*cols];

      for (i=0;i<cols;i++)
        hfile_out.image[i]=(1-scale_val)*hf1.image[i];
    }
  else
    {
      scale_val*=-1;
      for (j=1; j<rows; j++)
        for (i=0; i<cols; i++)
            {
                if (j==rows-1)
                    hfile_out.image[i+j*cols]=
                        (1-scale_val)*hf1.image[i+j*cols];
                else
                    hfile_out.image[i+j*cols]=
                        (1-scale_val)*hf1.image[i+j*cols]+
                        scale_val*hf1.image[i+(j+1)*cols];
            }
      for (i=0;i<cols;i++)
        hfile_out.image[i]=hf1.image[i]+scale_val*hf1.image[i+cols];
    }

  init_header(hfile_out.hd,hf1.hd->orig_name,hf1.hd->seq_name,
              hf1.hd->num_frame,hf1.hd->orig_date,hf1.hd->rows,
              hf1.hd->cols,hf1.hd->bits_per_pixel,
              hf1.hd->bit_packing,hf1.hd->pixel_format,
              hf1.hd->seq_history,hf1.hd->seq_desc);
  return hfile_out;
}

int max3(float a,float b, float c)
{ /*returns 0,1 or 2 if a,b or c respectively is largest */
    int max;

    if (a>b)
        {
            if(a>c)
                max=0;
            else
```

```
                max=2;
        }
    else if (b>c)
        max=1;
    else
        max=2;

    return max;
}



struct image_stack rescaleN(struct image_stack current,float* score,
                            struct image_stack orig)
/* rescale original view to most probable model/scale */
{
    float jump=0.1;
    int max,second;
    struct image_stack im;
    float rescale_val;
    max=max3(score[0],score[1],score[2]);

    printf("%f   %f   %f   %d\n",score[0],score[1],score[2],max);
    im=init_is();

    if (max==1)
        {
            if (score[0]>score[2])
                {
                    second=0;
                    rescale_val=(score[0]-score[2])/(2*(score[1]-score[0]));
                }
            else
                {
                    second=2;
                    rescale_val=(score[0]-score[2])/(2*(score[1]-score[2]));
                }
        }
    if (max==2)
        rescale_val=-jump;
    if (max==0)
        rescale_val=jump;

    rescale_val=(1+rescale_val)*current.scale-1;
    printf("Rescale val %f\n",rescale_val);

    if (orig.red.image)
      im.red=rescale1(orig.red,rescale_val);
    if (orig.green.image)
      im.green=rescale1(orig.green,rescale_val);
    if (orig.blue.image)
      im.blue=rescale1(orig.blue,rescale_val);
```

```
    if (orig.intensity.image)
      im.intensity=rescale1(orig.intensity,rescale_val);
    if (orig.edge.image)
      im.edge=rescale1(orig.edge,rescale_val);
    if (orig.corner.image)
      im.corner=rescale1(orig.corner,rescale_val);
    if (orig.blob.image)
      im.blob=rescale1(orig.blob,rescale_val);
    if (orig.label.image)
      im.label=rescale1(orig.label,rescale_val);

    im.scale=(1+rescale_val);
    return im;
}



struct hips_file clip(struct hips_file hf,struct point p,int size)
{  /* remove square region from hips centred on point p */
  struct hips_file hfile_out;
  int rows=size*2;
  int cols=size*2;
  char* rtn_name="clip";
  int i,j,i2,j2;
  int left,top;

  if (!(hfile_out.hd=malloc(sizeof(struct header))))
    error_msg(rtn_name,"Not again? Ran out of memory!!");

  if ((hfile_out.image=
       (unsigned char *)calloc(rows*cols,1))
      <=(unsigned char *)0)
    error_msg(rtn_name,"Can't allocate output array");

  top=p.y-(size);
  left=p.x-(size);
  for(j=top,j2=0;j<top+cols;j++,j2++)
      for(i=left,i2=0;i<left+rows;i++,i2++)
          hfile_out.image[i2+j2*cols]=hf.image[i+j*hf.hd->rows];

  init_header(hfile_out.hd,hf.hd->orig_name,hf.hd->seq_name,
              hf.hd->num_frame,hf.hd->orig_date,rows,
              cols,hf.hd->bits_per_pixel,
              hf.hd->bit_packing,hf.hd->pixel_format,
              hf.hd->seq_history,hf.hd->seq_desc);
  return hfile_out;
}


struct hips_file reduce(struct hips_file hfile,int scale)
{  /* reduce resolution of hips file by factor of scale */
   /* average values under region (scale*scale) */
```

```
    struct hips_file hfile_out;
    int dataout;
    int i,j,k1,k2;
    char* rtn_name="reduce";
    int cols,rows;

    if (scale>1)
      {
        rows=hfile.hd->rows/scale;
        cols=hfile.hd->cols/scale;
        if (!(hfile_out.hd=malloc(sizeof(struct header))))
          error_msg(rtn_name,"Not again? Ran out of memory!!");

        if ((hfile_out.image=
             (unsigned char *)calloc(rows*cols,1))
            <=(unsigned char *)0)
          error_msg(rtn_name,"Can't allocate output array");

        for (j=0; j+scale-1<hfile.hd->rows; j+=scale)
          for (i=0; i+scale-1<hfile.hd->cols; i+=scale)
            {
              dataout=0;
              for(k1=0;k1<scale;k1++)
                for(k2=0;k2<scale;k2++)
                  dataout+=hfile.image[i+k1+(j+k2)*hfile.hd->cols];
              hfile_out.image[i/scale+j/scale*cols]=dataout/(scale*scale);
            }
      }
    else
      {
        rows=hfile.hd->rows;
        cols=hfile.hd->cols;
        if (!(hfile_out.hd=malloc(sizeof(struct header))))
          error_msg(rtn_name,"Not again? Ran out of memory!!");

        if ((hfile_out.image=
             (unsigned char *)calloc(rows*cols,1))
            <=(unsigned char *)0)
          error_msg(rtn_name,"Can't allocate output array");
        memcpy(hfile_out.image,hfile.image,rows*cols);
      }

    init_header(hfile_out.hd,hfile.hd->orig_name,hfile.hd->seq_name,
                hfile.hd->num_frame,hfile.hd->orig_date,rows,
                cols,hfile.hd->bits_per_pixel,
                hfile.hd->bit_packing,hfile.hd->pixel_format,
                hfile.hd->seq_history,hfile.hd->seq_desc);
    return hfile_out;
}
```

```
struct cstruct cornerij2(struct hips_file hfile,int NumRings,
                         int NumSectors)
{ /* perform corner detection on IJ image */
  struct cstruct corners;
  struct hips_file hfile_rt,hf_mor,hfmor2,hfreduce,hfclip,c1,c2;
  int i,j,i1,j1,m[9],a,r,t,horiz,scale,dataout,k1,k2,r1,r2,max;
  float percent,total,maxnum;
  int rows=hfile.hd->rows;
  int cols=hfile.hd->cols;
  float *mrvcdat,low=9999999,high=0,range;
  char* rtn_name="cornerij";
  struct entryrt* ptr;
  struct point p,centre;

  centre.x=cols/2;centre.y=rows/2;
  hf_mor=moravec(hfile); /* detect corners at highest resl'n */
  write_file(hf_mor,"m");
  if ((mrvcdat=
       (float *)calloc(NumRings*NumSectors,sizeof(float)))
      <=(float *)0)
    error_msg(rtn_name,"Can't allocate dataout array");

  hfile_rt=create_hips(NumSectors,NumRings);
  c1=create_hips(cols,rows);

  for (j1=3*(NumRings-1); j1<rows-6*(NumRings-1); j1++)
    for (i1=3*(NumRings-1); i1<cols-6*(NumRings-1); i1++)
      {
        if (hf_mor.image[i1+j1*hf_mor.hd->cols]>50 &&
            head_ij[j1][i1]!=(struct entryrt *) -1)
          { /* corner detected at highest resl'n so determine scale */
            if (head_ij[j1][i1]->r <2)
              scale=1; /* scale=1 if point in ring 0 or 1 */
            else
              scale=head_ij[j1][i1]->r*3;  /* scale = 3*ring number */
            p.x=i1;p.y=j1;
            hfclip=clip(hfile,p,2*scale); /*clip local region around corner*/

            if (scale>1)
              {
                hfreduce=reduce(hfclip,scale);/* reduce resl'n */
                unalloc(hfclip);
              }
            else
              hfreduce=hfclip;

            hfmor2=moravec(hfreduce); /* perform moravec */

            unalloc(hfreduce);

            /* 0  1  2  3
```

```
                4  5  6  7
                8  9 10 11
               12 13 14 15 */

            if (hfmor2.image[5]>hfmor2.image[6])
              r1=hfmor2.image[5];
            else
              r1=hfmor2.image[6];
            if (hfmor2.image[9]>hfmor2.image[10])
              r2=hfmor2.image[9];
            else
              r2=hfmor2.image[10];
            if (r2>r1)
              dataout=r2;
            else
              dataout=r1; /* dataout = highest response Of 5,6,9,10 */
                          /* which are possible corner locations */
                          /* original corner detected at centre */
                          /* between 5,6,9 and 10 */
            unalloc(hfmor2);


            if (dataout>0) /* write circle to IJ corner image */
              {
                c1=putcircle(c1,i1,j1,scale,dataout);
                ptr=head_ij[j1][i1];
                do
                  { /* not used */
                    r=ptr->r;
                    t=ptr->theta;
                    percent=ptr->percent;
                    if (percent*dataout>mrvcdat[t+r*NumSectors])
                      mrvcdat[t+r*NumSectors]=percent*dataout;
                    ptr=ptr->next;
                  }
                while (ptr!=(struct entryrt *) -1);
              }
          }
      }
/* fix inner ring */
max=0;total=0;maxnum=0;
for (j=0;j<NumSectors;j++) /* not used */
  {
    if (mrvcdat[j]>maxnum)
      {
        max=j;
        maxnum=mrvcdat[j];
      }
    total+=mrvcdat[j];
    mrvcdat[j]=0.;
  }
mrvcdat[max]=total;
```

```
  c2=ij2rt(centre,c1,NumRings,NumSectors); /* convert IJ corner image */
  write_file(c2,"newc");                   /* to RT */
  write_file(c1,"newcij");
  for (j=0; j<NumRings*NumSectors; j++)
    {/* not used */
      if (mrvcdat[j]>high) high = mrvcdat[j];
      if (mrvcdat[j]<low) low = mrvcdat[j];
    }
  range=high-low;
  for (j=0; j<NumRings*NumSectors; j++)
    {
      if (mrvcdat[j]>0)
        hfile_rt.image[j]=(unsigned)((mrvcdat[j]-low)*255/range);
    }
  unalloc(c1);
  unalloc(hf_mor);
  free(mrvcdat);
  corners.cimage1=c2;
  corners.cimage2=hfile_rt; /* not used - old corner image */
  return corners;
}

struct hips_file putcircle(struct hips_file hf,int x,int y,int r,int dataout)
{ /* draw circle */
  int i,j;
  for (j=y-r;j<=y+r;j++)
    for(i=x-sqrt(r*r-(j-y)*(j-y));i<=x+sqrt(r*r-(j-y)*(j-y));i++)
      hf.image[i+j*hf.hd->cols]=dataout;
  return hf;
}
```