Range Image Segmentation for 3-D Object Recognition

Alok Gupta
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/736
For more information, please contact repository@pobox.upenn.edu.
Range Image Segmentation for 3-D Object Recognition

Abstract
Three dimensional scene analysis in an unconstrained and uncontrolled environment is the ultimate goal of computer vision. Explicit depth information about the scene is of tremendous help in segmentation and recognition of objects. Range image interpretation with a view of obtaining low-level features to guide mid-level and high-level segmentation and recognition processes is described. No assumptions about the scene are made and algorithms are applicable to any general single viewpoint range image. Low-level features like step edges and surface characteristics are extracted from the images and segmentation is performed based on individual features as well as combination of features. A high level recognition process based on superquadric fitting is described to demonstrate the usefulness of initial segmentation based on edges. A classification algorithm based on surface curvatures is used to obtain initial segmentation of the scene. Objects segmented using edge information are then classified using surface curvatures. Various applications of surface curvatures in mid and high level recognition processes are discussed. These include surface reconstruction, segmentation into convex patches and detection of smooth edges. Algorithms are run on real range images and results are discussed in detail.

Comments

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/736
RANGE IMAGE SEGMENTATION
FOR 3-D OBJECT RECOGNITION

Alok Gupta

MS-CIS-88-32
GRASP LAB 141

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

May 1988

Acknowledgements: The work reported herein was supported in part by NSF grant DCR-8410771, Airforce grant AFOSR F49620-85-K-0018, Army/DAAG-29-84-K-0061, NSF-CER/DCR82-19196 Ao2, DARPA/ONR N0014-85-K-0807, U.S. Postal Service contract 104230-87-M-0195.
UNIVERSITY OF PENNSYLVANIA
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

RANGE IMAGE SEGMENTATION FOR
3-D OBJECT RECOGNITION

Alok Gupta

Philadelphia, Pennsylvania
May 1988

A thesis presented to the Faculty of Engineering and Applied Science of the University of Pennsylvania in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Science.

Ruzena Bajcsy
(Advisor)

Richard Paul
(Graduate Group Chair)
Abstract

Three dimensional scene analysis in an unconstrained and uncontrolled environment is the ultimate goal of computer vision. Explicit depth information about the scene is of tremendous help in segmentation and recognition of objects. Range image interpretation with a view of obtaining low-level features to guide mid-level and high-level segmentation and recognition processes is described. No assumptions about the scene are made and algorithms are applicable to any general single viewpoint range image. Low-level features like step edges and surface characteristics are extracted from the images and segmentation is performed based on individual features as well as combination of features. A high level recognition process based on superquadric fitting is described to demonstrate the usefulness of initial segmentation based on edges. A classification algorithm based on surface curvatures is used to obtain initial segmentation of the scene. Objects segmented using edge information are then classified using surface curvatures. Various applications of surface curvatures in mid and high level recognition processes are discussed. These include surface reconstruction, segmentation into convex patches and detection of smooth edges. Algorithms are run on real range images and results are discussed in detail.
Acknowledgements

I would like to thank my advisor Dr. Ruzena Bajcsy for the guidance and encouragement. I am grateful to Dr. Kwangyoen Wohn for motivating me to work in the field of range image interpretation. Prof. Wohn also guided me in initial stages of this thesis and furnished programs for least squares fitting. Thanks to Franc Solina for many ideas and suggestions and for superquadrics software. Finally, thanks to Gus Tsikos for help with the range image scanner.

# Contents

## Abstract

Acknowledgements

### 1 Introduction

#### 2 Acquisition and Preprocessing of Range Images

- **2.1 Range Image Acquisition**

- **2.2 Scaling of Range Images**

- **2.3 Smoothing of Range Images**

#### 3 3-D Edges and Segmentation based on edges

- **3.1 Edge detection using $\nabla^2 G(x,y)$**

- **3.2 Segmentation of Range Images using edge information**

- **3.3 Segmentation Results**

- **3.4 Recognition of segmented objects using Superquadrics**

- **3.5 Results of Superquadric Fitting and Classification**

#### 4 Surface Characterization and Segmentation

- **4.1 Differential Geometry of Surfaces**

- **4.2 Computing Surface Characteristics of Range Images**

  - **4.2.1 Estimation of partial derivatives of Depth Maps**

  - **4.2.2 Results of Initial Segmentation**

- **4.3 Post processing of Labeled scenes**

  - **4.3.1 Obtaining Convex patches**

  - **4.3.2 Object Surface Classification**

#### 5 Discussion

A 2nd order Least squares fitting in symmetric neighborhood

B Source Code Listing
Chapter 1

Introduction

Three dimensional scene analysis in an unconstrained and uncontrolled environment is the ultimate goal of computer vision. Most of the effort in this regard has gone in extracting three dimensional information from intensity images and arriving at a meaningful and sufficiently unambiguous interpretation of the scene. However, the problem with monocular vision is the loss of 3-D information thereby making the interpretation process underconstrained. Shape from X methods have been widely studied in last two decades to extract depth of the scene using texture, shading, color, contour and motion. Depth extraction from stereo images is computationally expensive and results in sparse depth maps requiring reconstruction techniques for further interpretation. Range images on the other hand are obtained by realtime depth sensors and provide dense 3-D information of the visible surfaces.

Range images are dense depth maps measuring the distance of the physical surface from a known reference plane. Different types of ranging methods are available to obtain range information according to the application. Magnetic resonance imaging systems give true 3-D images, i.e, all the points in 3-D space are specified. Visible surfaces can be scanned by time of flight laser range finders and amplitude-modulated laser range finders. The most common and cheapest are the triangulation-based scanners. Structured lighting systems scan the scene with a laser stripe to obtain depth information of the visible surface in a
calibrated workspace. Research interest in range image processing has grown tremendously in recent years due to increasing availability of structured lighting range sensors. While these sensors can be employed in closed environment only and suffer from other drawbacks (like shadows, inability to sense highly reflective surfaces and some colors) they are useful for real-time scanning of good quality at low cost.

The range images dealt with in this work are of $z(x, y)$ type, (i.e., monge-patch surfaces) where each pixel gives the $Z$-depth at the coordinate $x$ and $y$. Since range images (or depth maps) contain explicit 3-D information about the scene it is expected that surface description and object recognition should be easier to handle with range images. However if the scene is quite complicated, then the problem cannot be solved that easily by using range images as one might think. Intensity information can be used to complement range information where ambiguity arises in interpretation, but this involves registration and correspondence problems and may even complicate the analysis.

Representation of range images is just like that of reflectance images. A two dimensional array of depth values specifying $(x,y,z)$ coordinates with respect to a known coordinate frame is enough for most applications. This allows many low level intensity image processing techniques to be directly used to process range images by interpreting the pixel value as 'depth' instead of 'reflectance value'. Contrast and brightness however have to be interpreted as surfaces of varying depths.

We have addressed the problem of object and surface segmentation in this report. Segmentation is essentially goal oriented. It can be conveniently divided into two processes: initial segmentation and final segmentation. Initial segmentation process is a result of local computations done in a known neighborhood of every pixel in the image. The final segmentation process does refinement of the initial segmentation using global constraints to arrive at a global interpretation of the scene. We have not assumed any domain knowledge or limited the objects to be of certain type. Our goal is to study boundary based segmentation, surface based segmentation and integration of the two methods. It is possible to segment the scene in flat, convex and concave subparts with detailed description of individual parts using boundary and surface based techniques.
An important aspect of the object recognition problem is the robustness of the recognition approach. It is essential that algorithm be size-invariant, position invariant, orientation invariant and be able to recognize partially occluded objects. As observed by Besl and Jain [9] it is known from the results in differential geometry that Gaussian and mean curvature are visible-invariant features of a surface region in the sense that they do not change under viewpoint transformations that do not affect the visibility of that region. When a surface region is visible, its curvature measurements are invariant to changes in surface parametrization and to translations and rotations. The invariant property is important for 3-D object recognition. Since our final segmentation process will be dependent on the local computations it is necessary that the low-level features be invariant.

While these two approaches are domain independent, any high level recognition approach like model based interpretation makes use of domain specific knowledge. We use a high level volumetric approach using superquadrics described in [23] to illustrate the usefulness of initial segmentation in high level vision. Figure 1 presents the paradigm explored in this work.

It is clear that processing of range images can be divided into three major stages: low level, intermediate level and high level. After range image is acquired from the sensor, it needs to be smoothed before any useful operations can be performed on it. Though it creates localization problems, it reduces the effect of quantization which is important for surface fitting. Low level processing is data-driven with the objective of obtaining useful local features that can be used by higher processing stages. Three dimensional edges constitute important features. We have used the Laplacian of Gaussian operator of [24] to detect step edges. Smooth edges have a different significance in case of range images and are more difficult to detect. This will be discussed in chapter 3 in detail.

Computation of curvature involves computing first and second order derivatives at every pixel in the image. Based on curvature signs, initial segmentation of the scene is performed. This is further improved by region growing done with global constraints. Haralick et al [6] have described a mathematical treatment for describing the topographic primal sketch of the
Figure 1: A paradigm for Range Image Segmentation and 3-D object recognition
underlying gray tone intensity surface of a digital image. They use first and second directional derivatives to classify each picture element as one of peak, pit, ridge, ravine, saddle, flat, and hillside. Michael Brady et al [4, 5, 7] describe a study of classes of curves as a source of constraint on the surface on which they lie, and as a basis for describing it. Their approach gives a curvature primal sketch of the surface. Tracing lines of curvature in real range images is very unreliable due to the low x-y resolution of the scanner and quantization and other sensing errors. Besides it is noise sensitive and computationally expensive. Besl and Jain [25, 9, 13] have done a comprehensive study of invariant surface characteristics and presented an algorithm for variable order surface fitting for image segmentation. They have summarized the field of 3-D object recognition in their excellent survey [3].

A scale-space based algorithm for extraction and representation of physical properties of a surface, using curvature properties of the surface is discussed in Fan, Medioni and Nevatia [14]. Nackman [19] has described the two dimensional critical point configuration graphs for describing the behavior of smooth functions of two variables by extracting peaks (local maxima), pits (local minima) and passes (saddle points) of a surface. Our approach is to not go into too much detail of the surface but to label the surface as flat, convex and concave accurately. Thus local variations are ignored in favour of a more global interpretation. Yang and Kak [33] describe an algorithm to analyze the topmost object in a pile. They compute derivatives by fitting B-splines and use local curvature information to label the object as flat and curved. Their method can only handle one type of surface for the topmost object in the scene and has other problems in assuming that step edges form a closed contour, which is not true in a general range image as described in chapter 3. A new approach for surface classification using characteristic contours is proposed by Sethi and Jayaramamurthy [20]. Characteristic contours are defined as the loci of the points where the surface normals are at a constant inclination to a selected reference vector. However it requires segmented surface and normal vector at every point, which limit its usefulness to surface classification in final stage of recognition process.

There are specific methods available to process images acquired using a light-stripe
rangefinder. Smith and Kanade [34] have done contour classification of light-stripes to produce object centered 3-dimensional descriptions. Another method by Martin Herman [35] extracts detailed, complete descriptions of polyhedral objects from light-stripe rangefinder data.

Segmentation of scene into surface primitives is useful in many applications. Most of the techniques discussed above involve curvature determination. Hebert and Ponce [8] have used surface normals (the Extended Gaussian Images) to classify surfaces into three simple primitive surfaces: planar, cylindrical, and conic regions. Duda, Nitzan and Barrett [36] have presented an algorithm for detecting planar regions using registered range and reflectance data.

Most of the high level recognition approaches include model matching. Kuan and Dravovich [37] have represented the objects as viewpoint-independent volumetric model based on generalized cylinders. They perform feature-to-model matching based on low-level features derived from range imagery. Constructing the 3-D model of an object involves integrating data or descriptions of an object obtained from multiple views and representing this integrated data in a coherent manner. Vemuri and Aggarwal [38] have presented an algorithm for automatic construction of models by determining the orientation of the object in the calibrated workspace and representing the object in cylindrical coordinates. Their method does not require correspondence to be established but requires registered intensity and range data of the scene while building the model. We have used superquadric models to recognize segmented objects. The classification procedure matches superquadric parameters with the parameters of the identifiable models. Since models are well defined by eleven superquadric parameters, there is no need to build models of the objects in advance.
Chapter 2

Acquisition and Preprocessing of Range Images

Range images obtained by different scanners differ in the format of the output. In order to apply low level techniques to the image it is necessary that the image points be quantized in Z-depth format with equal resolution factor in X and Y direction. Once converted into Z-depth format the image is smoothed. This chapter discusses some practical aspects of real range image processing which are important if any useful results are desired.

2.1 Range Image Acquisition

The test images used in this work were acquired by structured lighting triangulation based scanners. Figure 2 shows the ranging geometry of a typical range sensor. The trigonometry of a sensor will not be described here.

Either the laser stripe moves and scans the scene or the workspace moves under a vertical laser stripe. If the viewpoint of the sensing camera is not the same as the laser then shadows (regions with missing data) are obtained. In order to discriminate between shadows and background, (region of known depth on which object is sitting) background is assigned a nonzero depth.
(a) Ranging using structured lighting.

(b) Z-depth format of range images

Figure 2: Ranging geometry of a structured lighting scanner and Z-depth format
Contrary to the popular assumption made by researchers, it may not always be possible to represent the visible surface in Z-depth format, viewing perpendicular to the background. To be able to represent all the scanned points in Z-depth format, it is necessary to digitize the scene watching parallel to camera's line of sight. This may require rotating the scene to align the Z axis along the line of sight of camera thereby rotating the background which is no longer of constant depth. The segmentation procedure should take this into account. Also, this makes the processing viewpoint dependent. To avoid the trouble arising due to this, it is often convenient to fix the viewpoint at the cost of losing some scanned points. This problem is acute with images obtained from white scanner where \( f(x, y) \) is not unique. The solution is to segment the scene from background and then rotate the scene to obtain the Z-depth image.

2.2 Scaling of Range Images

Sampling interval of the scanners depends on the thickness of the laser stripe, value of laser stripe increment and resolution of the camera. More often than not vertical resolution (along Y axis) is different from horizontal resolution (along X axis). Thus the sampled points are not spaced uniformly in X and Y direction. Since we apply neighborhood operators during low level processing of images, it is necessary to rescale the images uniformly in both directions. We have rescaled the Z-depth image by fitting a plane on three neighborhood points. Figure 3 illustrates the difference between unscaled and uniformly scaled images.

2.3 Smoothing of Range Images

Depth resolution of a range image is an important parameter in low level processing. Range scanners usually have depth resolution good enough for most applications. In fact a resolution of 0.01 inch/pixel is too fine and noise sensitive for surface fitting purposes. The problem comes in quantization of z values. If entire scan depth is quantized within 8 bits (most convenient representation), effective depth resolution is drastically reduced thereby
increasing the quantization error. Since surface fitting is very sensitive to quantization error, we have minimized it by following two step procedure:

1. Original depth resolution is preserved by storing the depth value unscaled in 2 bytes. This allows 64k possible quantization levels. Scaling along Z axis is done only when needed.

2. Image is smoothed using a Gaussian operator and smoothed values are stored in floating point buffers so as not to lose any precision.

One way to reduce noise is to perform median filtering of the image. It ensures that isolated noise is reduced and edges are not smoothed.

Our approach is to study the scale-space behavior of range images. We have used Gaussian operator to smooth images. The Gaussian function in two dimensions is given by:

$$G(x, y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$
Since it is separable in X and Y directions, one dimensional Gaussian operator is applied separably on the image. Smoothing is controlled by the size of the operator, which is determined by $\sigma$. Gaussian operator has some nice properties that make it a unique operator for our purposes.

Yuille and Poggio [39] have proved that the Gaussian low-pass filter is the only filter with a nice-scaling behavior for linear derivative operators like the laplacian operator. It also satisfies the following conditions:

1. Filtering is shift-invariant and therefore, a convolution.

$$F \times I(x) = \int F(x - \alpha)I(\alpha)dx$$

2. The filter has no preferred scale. The filter is properly normalized at all the scales.

3. The filter recovers the whole image at sufficiently small scales.

$$\lim_{\sigma \to 0} F(x, \sigma) = \delta(x)$$

where $\delta(x)$ is the Dirac delta function.

4. The position of the center of the filter is independent of scale of the filter. Otherwise zero crossings of a step edge would change their position with change of scale.

5. The filter goes to zero as $|x| \to \infty$ and as $\sigma \to \infty$.

We have studied the behavior of increasing sigma value on edge detection, surface characterization and segmentation. As $\sigma$ value increases, window size of the Gaussian operator increases and details are lost. Figure 4 and figure 5 show the perspective plots of a range image before and after smoothing respectively.

Minor surface perturbations are smoothed easily. But the undesirable effect of uniform application of Gaussian operator is smoothing of all types of edges. Step edges (chapter 3) are smoothed to form roughly convex and concave subparts (chapter 4). This complicates edge detection, specially detection of smooth edges (concave and convex edges) that are
Figure 4: 3-D perspective plot of original image

Figure 5: 3-D perspective plot of smoothed image (σ = 1)
further smoothed. Thin objects tend to merge into the underlying objects, making segmentation difficult. As will be discussed in chapter 3, as we go up the scale objects start merging. We have found sigma value of 1 (window size $5 \times 5$) to be best suited for our experiments. In surface based segmentation technique, smoothing alters the local behavior of the surface, but makes the result more reliable, specially away from the edges. Step edges are shown as adjacent convex and concave regions. Segmentation using these effects of smoothing is discussed in chapter 4 in detail.
allows boundaries to be read off as zero crossings of the LOG operated image. We'll discuss the significance of the LOG operator in view of range images. While step edges pose no particular problem, smooth edges are difficult to detect by local operations. In range image segmentation it is of particular interest that the a pile of objects be segmented into convex subparts. This requires detection of concave edges that will delimit the convex subparts. Mitiche and Aggarwal [28] have presented a probabilistic approach of detecting the convex and concave edges by using domain specific constraints.

Though 3-D edges are quite useful for object recognition, there are some inherent limitations in edge information that make their use limited to aiding the higher level recognition processes along with a host of invariant features. Edge classification depends on the orientation of object in 3-D space and is therefore not an invariant feature. Thus edge information cannot be the only feature used by the recognizer and it has to be used in conjunction with other features. However, as will be seen later, edge information is good enough for early segmentation of range images because the requirement of invariant features does not apply to the initial-segmentation process.

In case intensity information is available range data can be complemented by reflectance data to pick up weak 3-D edges like the step edges created by overlapping thin objects. Wong and Hayrapetian [32] used range information to segment intensity images. Gil, Mitiche and Aggarwal [27] have described experiments in combining intensity and range edges. While intensity images are certainly useful in detecting edges in the scene, they need to be registered in the same way as range images to avoid correspondence problem. This may
possibility of extracting smooth edges using the $\nabla^2 G$ operator.

The Gaussian distribution in one dimension is defined as:

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-x^2}{2\sigma^2}}$$

The first and second derivatives are:

$$G'(x) = \frac{-x}{\sqrt{2\pi}\sigma^3} e^{\frac{-x^2}{2\sigma^2}}$$

$$G''(x) = \frac{-1}{\sqrt{2\pi}\sigma^3} \left(1 - \frac{x^2}{\sigma^2}\right) e^{\frac{-x^2}{2\sigma^2}}$$

The cross-section of a range image and the profiles of first and second order derivative are shown in figure 7.

In two dimensions the LOG operator becomes:

$$G''(x, y) = \frac{-1}{\sqrt{2\pi}\sigma^4} \left(2 - \frac{x^2 + y^2}{\sigma^2}\right) e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$
It is clear that behavior of second order derivatives is unique at every type of change in surface. There are positive spikes at concave ramp edge, negative spikes at convex roof and ramp edges and zero crossings at jump edges. However, there are serious practical limitations in using this response to detect concave and convex edges. The response at these edges is dependent on the convexity or concavity of the edge which is roughly the measure of angle at which the two surfaces meet. If the angle is too small and change in depth is gradual as in most situations, the response would be below or same as that due to local surface changes. See figure 8 for the step, convex and concave responses obtained in a synthetic range image having planar regions. Even in synthetic range image the responses deteriorate as image was smoothed and smooth concave and convex edges virtually disappeared.

Thresholding of zero crossings is necessary in case of range images to avoid local surface perturbations. Responses due to weak concave and convex edges would then be filtered. Zero-crossings generated by weak step edges may also lie below the thresholding value.
range images, Value of zero-crossing has direct relationship with the magnitude of depth discontinuity. Thus selection of a threshold effectively restricts the minimum detectable depth. An object with less than acceptable height would be invisible in the edge image.

As observed in the previous chapter, it is absolutely necessary to smooth an image before attempting any local operation it. LOG operator gives following image:

\[ f(x, y) = (\nabla^2 G) \ast I(x, y) \]

which can be written as:

\[ f(x, y) = \nabla^2 (G \ast I(x, y)) \]

Degree of smoothing depends on the value of sigma, which controls the size of the window. Larger the sigma, greater is smoothing. While this effect is interpreted in intensity images as blurring and hence reduction of details, In range images it is seen in terms of smoothing the surface at the boundaries in addition to reduction of details. This results in all types of boundaries to become smoothed and can have undesirable effects on boundary detection and surface based segmentation. We have observed that with increasing scale value, range images loose vital boundary information presenting difficulties in edge based segmentation. Empirically determined window size of $5$ ( sigma value = $1$ ) is chosen for processing all the images.

The algorithm for edge detection is given below.

1. Read in the range image.
2. Convolve the image with Gaussian operator separably in X and Y direction.
3. Convolve the $G(x,y) \ast I(x,y)$ image with Laplacian operator:

\[
\begin{bmatrix}
0 & -1 & 0 \\
-1 & 4 & -1 \\
0 & -1 & 0 \\
\end{bmatrix}
\]
4. Label the Zero-crossing (with maximum value) at every pixel in the $\nabla^2 G(x, y) \ast I(x, y)$ image. Also mark the direction along which maximum crossing value is found.

5. Threshold the image at a predetermined value to label pixels as belonging to step edges.

Figures 11(b), 12(b) and other figures show the magnitude of zero-crossings detected in range images.

It is observed that threshold selection is important in defining the acceptable depth, which in turn is determined by the amount of detail seen in the filtered image. Also the thresholding value is different at different scales. Threshold value varies inversely with smoothing parameter ($\sigma$). As we go up the scale-space the need for thresholding decreases.

Next we discuss a segmentation technique based on the step edges detected by the LOG operator.

3.2 Segmentation of Range Images using edge information

Segmentation of objects in a range image depends on actual requirements. One should therefore define the problem of segmentation clearly in the relevant context. In order to recognize an object it is necessary to isolate the object, which is not a trivial task. In a practical environment where objects can be of any size and shape, segmentation of individual objects can be a difficult task. Objects in a range image will always be partially occluded making the problem of segmentation and recognition difficult. If the scene consists of a heap of objects and we have to recognize one object then our problem can be simplified by considering the object of immediate importance, the one on the top of the heap. But this method has only specific applications like picking parts out of a bin etc. and is not useful as a general segmentation strategy.

Another approach to a complete segmentation is to segment the picture according to the requirement. Sometimes image segmentation into different surface types may be useful
and at other times convex objects need to be segmented. Whatever the method, a segmentation process working on local information cannot always give requisite results. In fact, segmentation at lower-level of processing can at best give locally valid results which may be conflicting from global point of view. Thus a robust segmentation process has to work with higher stages of processing to yield globally valid results. This introduces the concept of feedback from higher stages so as to work in a closed loop with the goal of object recognition. Then segmentation can be considered as a part of object recognition stage.

As discussed before, 2nd order derivatives give enough information to delineate objects at the step boundaries. We will develop an algorithm for segmenting the topmost object in a heap of arbitrary objects.

First stage of segmentation process involving isolation of objects from background can be easily accomplished by thresholding the image at known background depth. The difficult part is to identify an individual object from the heap of objects. At this point it is essential to define what is meant by an 'individual object'. An object can be a complex combination of various 'primitive objects' like cube, sphere, cylinder etc. Some important questions need answering here before any further progress can be made: What are the end boundaries(edges) of the object and what are the internal boundaries of the object and how to distinguish between them?

Different types of 3-D edges are jump edges, concave roof edges, concave ramp edges, convex roof edges and convex ramp edges. Considering any of these edges as the internal or external boundary of the object is going to put restrictions on the object types that can be segmented. For example, a jump edge (Unless it is an occluding edge against the background) need not be the end boundary of the actual object. Since the segmentation process is essentially a local operation and no other knowledge is used this problem cannot be solved at this segmentation level. There are following solutions to this problem:

1. Make assumptions about the type of the objects. For example, assume the topmost object to be convex. This is a very strong assumption and requires convex internal boundary information. As noted before, this information is difficult to derive using
second order derivatives. We will see in next chapter, how surface characterization techniques can be used to approximate the presence of smooth internal boundaries of an object.

2. a priori knowledge about the scene. But this is against our approach towards a general and robust segmentation algorithm.

3. Feedback from higher stages of object recognition to eliminate need for a priori knowledge and to relax the strong assumptions made at low-level segmentation stage. Since higher levels of object recognition are global processes and may have knowledge about the domain, a closed loop segmentation procedure is bound to perform better than one having no feedback.

Thus to achieve a reliable segmentation of the initial scene, we will assume that the topmost object is delineated by jump boundaries. This may not always be true as two objects can join at convex or concave edges or one object may merge into next one due to negligible thickness of the object at the point of contact. This means that local information cannot give perfect segmentation in all the cases. In such cases we need higher level processing to figure out the right segmentation of the scene. The segmentation based on external boundary information will give only an initial estimate of segmentation. This estimate is reliable to the point that it can distinguish between objects of predetermined depth.

In the context of invariant object recognition it is important to note that step boundaries may vary with orientation of the object. Thus they are used only to segment the object and not to recognize the object. We will discuss the results of recognition and classification of the segmented object using the superquadric technique developed by Franc [23].

The block diagram of segmentation and classification process is shown in figure 9.

A practical problem with using zero-crossing as step boundaries is that they do not form closed contours. The boundaries delineating the object may not be completely enclosing the object, resulting in region growing process to overflow from the object and include the neighboring object as part of the object. This drawback renders the final segmentation result very unreliable. Yang and Kak [33] use a priori knowledge about the width of the
Figure 9: Flow chart of segmentation and recognition process
object and contour tracking to extract the closed contour surrounding the object. Their method does not guarantee success in all the cases. David Heeger [26] has proposed a computational approach to gap filling. It is computationally expensive and not suited for our purpose since we want to avoid explicit contour tracing in the entire image and want the region growing method to take care of it. Peter Allen [29] has used gap filling method based on contour growing proposed by Nevatia and Babu [30]. They perform gap filling on the entire scene using the predecessor-successor graph of all connected contours. See Peter Allen's Ph.D. thesis for details. Contours are then merged based on the requirement of merging N pixel gaps. This approach is again computationally expensive. Peter Allen observes that filling of at most two pixel gaps is acceptable because of the ambiguities resulting with three or more pixel gap-filling requirement. We have implemented two pixel gap filling by constraining the region growing process near the boundaries, thus avoiding the explicit gap filling stage.

One and two pixel gap filling is accomplished by simply requiring that a pixel having a boundary pixel in its 8-neighborhood be not grown recursively. Instead, the pixel and the boundary pixel are marked as grown. Figure 10 illustrates gap filling in one instance.

Thus we are able to avoid the contour tracing explicitly to fill gaps. Three or more pixel gaps cannot be adequately handled by gap-fillers. Some sort of post processing is necessary to further segment the segmented object in this case. One way is to trace all the boundary pixels of the segmented object and use concavity information to segment the object into parts. This approach is being implemented and will not be discussed here.

The algorithm for segmentation is given below:

1. Read in the original range image $I(x,y)$ and $\nabla^2G * I(x,y)$ image.
   
   $\text{label\_val} = 0$

2. Segment the objects from background by thresholding at background depth (supplied by the user). In case background is not of uniform depth, a plane can be fitted to represent the background and threshold the objects from the scene.

3. Locate the 3x3 window with maximum height by averaging the pixel values in 3x3
Figure 10: An example of gap filling

(b) Pixels in region. Ungrown Pixels.
window at every pixel in the range image. This gives the seed region for region growing. Clearly the window lies on the topmost object. If there are more than one heap, then also only one seed region is obtained.

4. \( \text{label.val} = \text{label.val} + 1 \)

5. Grow the seed region recursively in all 8 directions. For gap filling procedure to work it is necessary to grow pixels in 8-neighborhood. Let \( p_{i,j} \) be the pixel being grown. A pixel \( \hat{p}_{i,j} \) in 8-connected neighborhood of \( p_{i,j} \) is not grown under one of the following conditions:

   (a) \( \text{depth}(\hat{p}_{i,j}) \leq \text{background\_threshold} \)

   (b) \( \hat{p}_{i,j} \) is already labeled.

   (c) If \( \hat{q}_{i,j} \), any pixel in 8-connected neighborhood of \( \hat{p}_{i,j} \) satisfies:

      \[ \text{laplacian}(\hat{q}_{i,j}) \geq \text{edge\_threshold} \]

      Then \( p_{i,j} \) is 1-pixel distance from an edge pixel and likely to be in a gap. Make:

      \[ \text{label}(\hat{p}_{i,j}) = \text{label}(p_{i,j}) \]

      \[ \text{label}(\hat{q}_{i,j}) = \text{label}(p_{i,j}) \]

6. If number of pixels in the extracted region < acceptable size then region is invalid else it is valid.

7. If region is valid then determine supporting points of the region.

8. The region extracted in the first pass is topmost region. Subsequent regions are grown from top to bottom, left to right. If any more pixels are left to be processed then pick up any unprocessed pixel and go back to step 5 to grow region.

9. Output topmost region, all valid regions and supporting points in separate image files.

### 3.3 Segmentation Results
Figure 11: **Segmentation**: (a) original image from RCA. (b) Edges detected. (c) topmost object. (d) all segmented objects
Figure 12: **Segmentation:** (a) original image from RCA. (b) Edges detected. (c) topmost object. (d) all segmented object.
Figure 13: Segmentation: (a) original image from grasp lab scanner. (b) Edges detected. (c) topmost object. (d) all segmented objects
Figure 14: **Segmentation:** (a) original image from RCA. (b) Edges detected at $\sigma = 1$. (c) segmented parts of one object at $\sigma = 1$. (d) Segmentation at $\sigma = 2$, for the same threshold value.
Figure 15: Segmentation at different scales: (a) image smoothed at $\sigma = 2.0$. (b) objects segmented. (c): image smoothed at $\sigma = 3.0$. (d) objects segmented
The programs are written in C and implemented on a VAX-785 running UNIX. Ikonas graphics and PM format for images are used in all programs. Images are acquired from two sources. Most of the images used as examples are from RCA range image database and remaining are scanned by Grasp lab's range scanner. All the images are digitized in Z-depth format. RCA images have better (12 bits/pixel) resolution than GraspLab images (8 bits/pixel). Hence more detail is seen in the former. It makes difference in detection of thin objects. Due to different Z resolution of two scanners, we have used different threshold values for the two sets of images. All the images are smoothed uniformly with Gaussian of $\sigma = 1$ (window size = $5 \times 5$). Zero-crossings of LOG operator are thresholded to remove response due to minor surface perturbations. The threshold at a given $\sigma$ value also limits the thickness of objects that can be segmented. Threshold values are determined empirically, since histogram of zero-crossings cannot be used in determining threshold automatically as is done in intensity images. However threshold value remains same for all the images acquired from the same source. This is true for all the empirically determined parameters reported in this thesis. Background value is also known to the program and is constant given a scanner. Results of processing the images are in figures 11, 12, 13 and 14. In figure 11 all the objects are segmented correctly. The topmost object is a Cylindrical object. Figure 12 shows merging of objects because of very weak step boundary information. Figure 13 shows results of segmentation on the image obtained from GraspLab scanner. A constant offset of 100 is added to original image depth values and zero-crossings are enhanced for displaying purposes. Figure 14 exhibits different results at two scales for the same edge threshold. The scene has single object, a box with string tied around it, so that the box is divided into 4 partitions. Because of the high depth resolution of the image, edge information due to string is enough to segment the box into three parts at $\sigma = 1$ (figure 14(c)). Increasing the $\sigma$ value to 2, removes the details of the string and whole visible surface is recovered (figure 14(d)).

To study the effect of increasing sigma value on zero-crossing, one of the multiple object images is processed for $\sigma = 1, 2, 3$. Note that objects start to merge as sigma increases, with thin objects undetectable at $\sigma = 2$ (Figures 11, 15)
3.4 Recognition of segmented objects using Superquadrics

The surfaces extracted by the previous algorithm can be classified as one of the eight basic surface types. We will discuss this classification approach in detail in next chapter. In this section we will describe a high level recognition and classification method that classifies the segmented object into four broad categories.

We have used superquadric model recovery method implemented by Franc [23] to recognize the segmented object in a range image. Details of the procedure for superquadric fitting are discussed in Franc's Ph.D. thesis. Superquadrics are a family of parametric shapes that can be used as primitives for shape representation in computer vision [31]. Superquadrics are like lumps of clay that can be deformed and glued together into realistic looking models. However, we will consider only non-deformed superquadric models for classification of the object into one of the categories:

1. **flat**: Object with negligible height compared to length and width;

2. **roll**: A Cylindrical object.

3. **box**: An object with comparable height, length and width.

4. **Irregular**: Any object not falling in any of the above three categories.

Superquadric implicit equation is given by:

\[
\left[ \left( \frac{x}{a_1} \right)^{\varepsilon_1} + \left( \frac{y}{a_2} \right)^{\varepsilon_2} \right]^{\frac{2}{\varepsilon_1}} + \left( \frac{z}{a_3} \right)^{\varepsilon_3} = 1.
\]

Parameters \(a_1, a_2, a_3\) define the superquadric size in \(x, y, z\) direction respectively. \(\varepsilon_1\) is the squareness parameter in the latitude plane and \(\varepsilon_2\) is the squareness parameter in the longitude plane. Based on these parameter values superquadrics can model a large set of standard building blocks, like spheres, cylinders, parallelepipeds and shapes in between. Figure 16 illustrates the various types of shapes obtainable by changing two shape parameters. If both \(\varepsilon_1\) and \(\varepsilon_2\) are 1, the surface defines an ellipsoid. Cylindrical shapes are obtained for \(\varepsilon_1 < 1\) and \(\varepsilon_2 = 1\). Parallelepipeds are obtained for both \(\varepsilon_1\) and \(\varepsilon_2\) are < 1.
Figure 16: Superquadric models as function of shape parameters \((\varepsilon_1, \varepsilon_2)\) for given size parameters \((a_1, a_2, a_3)\)
We have restricted the model recovery procedure to fit the models with \( 0 \leq \varepsilon_1, \varepsilon_2 \leq 1 \). We will not discuss the details of model recovery here.

The Criteria used for classification are three size parameters, two shape parameters and the goodness of fit (GOF) measure. The superquadric procedure returns a GOF measure using the following equation:

\[
GOF = \frac{1}{N} \sum_{i=1}^{N} [a_1 a_2 a_3 (F(x, y, z; a_1, a_2, a_3, \varepsilon_1, \varepsilon_2, \phi, \theta, \psi, p_x, p_y, p_z) - 1)]^2
\]

Where \( F \) is the superquadric inside-outside function described in Franc [23]. \( \phi, \theta, \psi \) define the orientation and \( p_x, p_y, p_z \) define position of superquadric in space.

The object given by the segmentation procedure has only the points visible to the scanner. Much of the volumetric information is lost in the Z-depth format of representation. While this is not a serious problem in case of curved objects like cylinder or segmented surfaces having volumetric information (like a tilted box viewed from above), model fitting becomes ambiguous if the visible surface is flat. If it is known that the original scene had only one object, then the supporting surface can be assumed to be the plane parallel to the known background. The problem complicates in the multiple object scenes, where it becomes impossible to assign correct depth to the segmented object. Given no prior knowledge about the surface type, we need to add points in every case to give volumetric information to the superquadric procedure. Points can be added in two ways:

1. Background is assumed to be the supporting surface of the object. Points are added on the background by backprojecting the visible surface on the background (figure 17(c)). While this is desirable in case of flat surfaces, it is not right for surfaces with volumetric information.

2. Supporting points of the segmented object are used to determine the immediate supporting surface(s) of the object. Points are added vertically (figure 17(e)) to the object. This technique is more flexible since it can handle objects not lying on the background. But it results in more points to be added in addition to assuming that
the object is actually *touching* the neighboring objects, which may not be true in general.

In general it is not possible to extract correct supporting surface information from a single viewpoint. We have used horizontal addition of points in our experiments as it is faster than vertical addition and recovers the desired model.

The algorithm for model fitting, selection and classification is following:

1. Read the segmented object in Z-depth format.

2. **Format conversion and point addition**: Generate a list of points in 3-D space representing the object. Call it *points.orig*. For every point on the visible surface add
a point at the same \((x, y)\) coordinates on the background. Output the list of original and added points in \(points.add\).

3. **Superquadric fitting** : Run Superquadric model fitting procedure on \(points.orig\). Model obtained is \(model.orig\). Run Superquadric model fitting procedure on \(points.add\). Model obtained is \(model.add\). Iterative superquadric fitting is stopped if one of the following conditions is met:

(a) Number of iterations \(\geq 15\).

(b) Goodness of fit of \(i\)th iteration \((i \leq 15)\) is \(\leq Acceptable\ measure\). This measure is empirically determined.

(c) If for the \(j\)th \((j \geq 5)\) iteration:

\[
\sqrt{\frac{1}{5} \sum_{i=j}^{j-4} \left( GOF(i) - \sum_{k=j}^{j-4} GOF(k) \right)^2} \leq Acceptable\_deviation.
\]

Condition (a) assumes that model recovery is complete by 15th iteration. Condition (b) stops the procedure if an acceptable model is obtained early in the process. Condition (c) monitors the rate convergence of fitting procedure. It terminates the fitting procedure if the GOF measures of last five iterations do not vary much. All the values used in the above three conditions are empirically determined.

4. **Model selection** :

If \(GOF(model.add) AND GOF(model.orig) \leq Acceptable\_fit\) THEN GOTO \texttt{volume.criterion}.

ELSE IF \(GOF(model.add) \leq Acceptable\_fit\) THEN \(model = model.add\) GOTO \texttt{classify}.

ELSE IF \(GOF(model.orig) \leq Acceptable\_fit\) THEN \(model = model.orig\) GOTO \texttt{classify}.

37
ELSE

OBJECT = *Irregular*. GOTO Done.

5. **Volume-criterion**: Volume can be approximated as $a_1 \times a_2 \times a_3$.

If($\text{volume.add} < \text{volume.orig}$)

Then $model = model\text{.orig}$

Else $model = model\text{.add}$.

6. **classify**: Classify $model$ using $a_1, a_2, a_3$ and $\varepsilon_1, \varepsilon_2$:

(a) If (($a_3 \ll a_1$) AND ($a_3 \ll a_2$)) AND ($\varepsilon_1 < 0.5$) AND ($\varepsilon_2 < 0.5$))

Then OBJECT = *FLAT*.

(b) Else If (($a_1 \ll a_3$) AND ($a_2 \ll a_3$) AND ($\varepsilon_1 < 0.5$) AND ($\varepsilon_2 < 0.5$))

Then OBJECT = *FLAT*.

(c) Else If (($a_1 > \text{THRESH\_BOX}$) AND ($a_2 > \text{THRESH\_BOX}$) AND ($a_3 > \text{THRESH\_BOX}$) AND ($\varepsilon_1 < 0.5$) AND ($\varepsilon_2 < 0.5$))

Then OBJECT = *BOX*.

(d) Else If (($a_1 > \text{THRESH\_1\_ROLL}$) AND ($a_2 > \text{THRESH\_1\_ROLL}$) AND ($a_3 > \text{THRESH\_2\_ROLL}$) AND ($\varepsilon_1 < 0.5$) AND ($\varepsilon_2 > 0.5$))

Then OBJECT = *ROLL*.

(e) Else OBJECT = *Irregular*

**THRESH\_BOX** is the minimum acceptable dimension of the box.

**THRESH\_1\_ROLL** is the minimum acceptable width and height of the roll.

**THRESH\_2\_ROLL** is the minimum acceptable length of roll.
Figure 18: Parameter values of the recovered models

<table>
<thead>
<tr>
<th>OBJECT</th>
<th>Model</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
<th>e1</th>
<th>e2</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cylinder</td>
<td>Orig</td>
<td>16.30</td>
<td>8.65</td>
<td>75.70</td>
<td>0.24</td>
<td>0.95</td>
<td>56.21</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>16.07</td>
<td>35.29</td>
<td>72.16</td>
<td>0.10</td>
<td>0.14</td>
<td>393.28</td>
</tr>
<tr>
<td>Box</td>
<td>Orig</td>
<td>3.47</td>
<td>33.6</td>
<td>45.29</td>
<td>0.10</td>
<td>0.81</td>
<td>1274.40</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>38.11</td>
<td>51.14</td>
<td>39.47</td>
<td>0.10</td>
<td>0.10</td>
<td>441.16</td>
</tr>
<tr>
<td>Flat</td>
<td>Orig</td>
<td>7.46</td>
<td>46.76</td>
<td>57.72</td>
<td>0.10</td>
<td>0.52</td>
<td>319.32</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>8.86</td>
<td>45.75</td>
<td>57.52</td>
<td>0.10</td>
<td>0.17</td>
<td>245.34</td>
</tr>
<tr>
<td>Others</td>
<td>Orig</td>
<td>4.928</td>
<td>50.41</td>
<td>76.79</td>
<td>0.10</td>
<td>0.43</td>
<td>4083.61</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>9.38</td>
<td>53.00</td>
<td>82.44</td>
<td>0.10</td>
<td>0.10</td>
<td>4178.83</td>
</tr>
</tbody>
</table>

7. **Done**: Output classified model with parameters. Determine Orientation and position of the model in world coordinate system.

### 3.5 Results of Superquadric Fitting and Classification

The superquadric fitting procedure and classifier were run on the objects segmented previously. The superquadric parameters for the four types of recovered objects are shown in figure 18.

Figure 19 shows model recovery on topmost object segmented in figure 14. The model selection process rejected `model.orig` due to large fit-error and accepted `model.add`. Even if `model.orig` had an acceptable error measure, `model.add` would have been selected due
to larger volume. The acceptable error magnitude was empirically determined to be 500. In figure 20 model.orig is selected and classified as roll because of the tremendous error difference between the two acceptable models. model.add is accepted and classified as flat in figure 22 because of volume consideration, although both the models have acceptable error measures. Finally, the film mailer in figure 23 is classified as irregular as the fit-errors of both the models is more than acceptable error measure.

The results are shown for the four classes of objects. Tapered, bent or concave objects cannot be represented by these models and hence will be classified as irregular. Franc Solina’s superquadric method also allows for tapering and bending along with segmentation.
Figure 20: Superquadric fitting and Model selection (a Cylindrical object): (a) original points. (b) fitted model on original points. (c) original and added points. (d) fitted model on original and added points.
Figure 21: **Segmentation:**
- **Upperleft:** original image from grasp lab scanner (*aletter*)
- **Upperright:** Edges detected
- **Lowerleft:** topmost object
- **Lowerright:** all segmented objects
Figure 22: Superquadric fitting and Model selection (a letter) (a) original points. (b) fitted model on original points. (c) original and added points. (d) fitted model on original and added points.
Figure 23: Superquadric fitting and Model selection (a film-mailer): (a) original points. (b) fitted model on original points. (c) original and added points. (d) fitted model on original and added points.
of the complex objects into parts. In the next chapter we will describe a surface classification scheme that uses the output of segmentation routines described in this chapter.
Chapter 4

Surface Characterization and Segmentation

Surface characterization refers to the computational process of partitioning the surfaces into regions with equal characteristics. Since our ultimate goal is object recognition, classification of the surfaces by the characteristics of the surface functions is very useful. Classical differential geometry provides a complete surface description of analytic surfaces so as to obtain a complete set of surface characteristics. Surface characterization can be successfully used in intermediate and high level processing of the object recognition problem.

Important surface characteristics, that are visible-invariant are Gaussian curvature and the mean curvature. They are invariant to changes in surface parametrization and to translations and rotations of object surfaces. Gaussian curvature is an intrinsic property of the surface while mean curvature is an extrinsic property of the curvature.

From differential geometry it is well known that curvature, speed, and torsion uniquely determine the shape of 3-D surfaces. The surface characteristics of our interest are the ones with one-to-one relationship with curve shapes. The mathematics of a general surface representation scheme and calculation of Gaussian and mean curvatures is described in following section.
4.1 Differential Geometry of Surfaces

Parametric form of equation for a regular surface $S$ with respect to a known coordinate system is:

$$S = (x, y, z): x = x_1(u, v), y = x_2(u, v), z = x_3(u, v), (u, v) \in D \subseteq \mathbb{R}^2$$

The surface is a locus of points in Euclidean three-space defined by the end points of the vector $X(u, v)$ with $x_i(u, v)$ the components of the vector. These real functions are assumed to be defined over an open connected domain of a Cartesian $u, v$ plane and to have continuous second partial derivatives there. In our analysis of range images we are assuming that this condition is satisfied.

The second condition for a regular surface is automatically satisfied by Z-depth format images. It requires that the coordinate vectors $X_u = X_1 = \frac{\partial X}{\partial u}, X_v = X_2 = \frac{\partial X}{\partial v}$ are linearly independent:

$$\frac{\partial X}{\partial u} \times \frac{\partial X}{\partial v} = X_1 \times X_2 \neq 0.$$

The surface in range images is given by:

$$X = (x_1, x_2, f(x_1, x_2))$$

and coordinate vectors become:

$$X_1 = \left(1, 0, \frac{\partial f}{\partial x_1}\right),$$

$$X_2 = \left(0, 1, \frac{\partial f}{\partial x_2}\right),$$

These vectors are linearly independent given the first condition.

It can be shown using differential geometry techniques that first and second fundamental forms (which exist only if the surface is analytic) uniquely characterize a general smooth surface. The first fundamental form $I$ of a surface is defined as:
Figure 24: Coordinate frame at the Neighborhood of a point

\[ I(u, v, du, dv) = dX.dX = \begin{bmatrix} du & dv \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} \begin{bmatrix} du \\ dv \end{bmatrix} = du^T [g] du \]

where \([g]\) matrix elements are given by:

\[ g_{11} = E = X_u.X_u \quad g_{22} = G = X_v.X_v \quad g_{12} = g_{21} = F = X_u.X_v \]

The two tangent vectors \(x_u\) and \(x_v\) lie in the tangent plane \(T(u,v)\) of the surface at the point \((u,v)\). \([g]\) matrix is symmetric for an analytic surface.

Figure 24 shows the coordinate frame at the Neighborhood of a point.

The first fundamental form \(I(u, v, du, dv)\) measures the small amount of movement in the parameter space \((du, sv)\). The first fundamental form is invariant to surface parametrization changes and to translations and rotations in the surface. Therefore it depends on the surface itself and not on how it is embedded in the 3-D space. The metric functions \(E, F, G\) determine all the intrinsic properties of the surface. In addition they define the area of a surface:
The second fundamental form of the surface is given by:

$$A = \int \int_R \sqrt{EG - F^2} du \, dv$$

The second fundamental form of the surface is given by:

$$II(u, v, du, dv) = - dx.dn = \left[ \begin{array}{c} du \\ dv \end{array} \right] \left[ \begin{array}{cc} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array} \right] \left[ \begin{array}{c} du \\ dv \end{array} \right] = du^T[b]du$$

Where \([b]\) matrix elements are defined as:

$$b_{11} = L = X_{uu}.n \quad b_{22} = N = X_{vv}.n \quad b_{12} = b_{21} = M = X_{uv}.n$$

$$n(u, v) = \frac{\nabla_x X \nabla_v}{| \nabla_x X \nabla_v |} = \text{unit_normal_vector}$$

Where the double subscript denotes second partial derivatives

$$x_{uu}(u, v) = \frac{\partial^2 x}{\partial u^2} \quad x_{vv}(u, v) = \frac{\partial^2 x}{\partial v^2} \quad x_{uv}(u, v) = x_{vu}(u, v) = \frac{\partial^2 x}{\partial u \partial v}$$

The second fundamental form measures the correlation between the change in the normal vector \(dn\) and the change in the surface position at a point \((u, v)\) as a function of small movement \((du, dv)\) in the parametric space. Besl and Jain [9] have discussed the properties of first and second fundamental forms in detail. We will consider some of the important properties of Gaussian and Mean curvature in the following paragraphs.

It can be shown that the \([g]\) matrix and the \([b]\) matrix elements are the continuous functions with continuous second and first partial derivatives respectively and that they uniquely determine the surface type. From the \([g]\) and \([b]\) matrices calculated above surface shape and intrinsic surface geometry can be uniquely determined.

The Gaussian curvature function \(K\) of a surface can be defined in terms of the two matrices as:

$$K = \det \left( \left[ \begin{array}{cc} g_{11} & g_{12} \\ g_{21} & g_{22} \end{array} \right]^{-1} \right) \det \left( \left[ \begin{array}{cc} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array} \right] \right)$$
The mean curvature of a surface is defined as:

$$H = \frac{1}{2} \text{tr} \left( \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} \right)^{-1} \text{det} \left( \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \right)$$

The two types of curvatures are together referred to as surface curvature functions. They exhibit very important properties that enable them to be used as features for higher level of processing. For detailed discussion on the properties of surface curvature functions see Besi and Jain [9]. Some of the relevant properties are summarized below:

1. Surface types can be determined by the sign of surface curvatures. They are shown in figure 25.
2. Gaussian curvature exhibits isometric invariance properties.
3. Mean curvature is slightly less sensitive to noise than Gaussian curvature.
4. Gaussian curvature function of a convex surface uniquely determines the surface.
5. Mean curvature function of a graph surface taken together with the boundary curve of a graph surface uniquely determines the graph surface from which it was computed.
6. Gaussian and mean curvature are invariant to arbitrary transformations of the \((u, v)\) parameters of a surface as long as the Jacobian of the transformation is always non-zero.

7. Gaussian and mean curvatures are invariant to rotations and translations of a surface. This property enables us to obtain view-independent characteristics.

8. Gaussian curvature is an isometric invariant of a surface. Therefore it is an intrinsic surface quantity. It is independent of the the way the surface is embedded in the 3-D space.

9. Gaussian and mean curvature are local surface properties.

10. Another important property of surface curvatures is that Gaussian curvature indicates the surface shape at individual surface points. When surface is shaped like an ellipsoid in the Neighborhood of \((u, v)\), \(K(u, v) > 0\). It is \(< 0\) for locally saddle-shaped surface and is \(= 0\) if the surface is flat, ridge-shaped or valley-shaped locally. Mean curvature also indicates surface shapes at individual points when considered together with the Gaussian curvature.

The above observations are very important for surface classification and have been widely studied and used in range image processing. In fact surface characteristics constitute an important part in the realization of the ultimate goal of three dimensional object recognition.

4.2 Computing Surface Characteristics of Range Images

Given a range image, our objective is to calculate the Gaussian and mean curvature. To compute surface curvature we need to know the estimates of the first and second partial derivatives of the depth map. Equations to get the partial derivatives can be simplified in the case of the Z-depth format range image. Parameterization takes a very simple form: \(x_u = \begin{bmatrix} u & v & f(u, v) \end{bmatrix}^T\). The \(T\) superscript indicates the transpose. This gives following formulas for the surface partial derivative and the surface normal.
\[
\begin{align*}
\mathbf{x}_u &= \begin{bmatrix} 1 & 0 & f_u \end{bmatrix}^T \\
\mathbf{x}_v &= \begin{bmatrix} 0 & 1 & f_v \end{bmatrix}^T \\
\mathbf{x}_{uu} &= \begin{bmatrix} 0 & 0 & f_{uu} \end{bmatrix}^T
\end{align*}
\]
\[
\begin{align*}
\mathbf{x}_{uv} &= \begin{bmatrix} 0 & 0 & f_{uv} \end{bmatrix}^T \\
\mathbf{x}_{vv} &= \begin{bmatrix} 0 & 0 & f_{vv} \end{bmatrix}^T
\end{align*}
\]
\[
\mathbf{n} = \frac{1}{\sqrt{1 + f_u^2 + f_v^2}} \begin{bmatrix} -f_u & -f_v & 1 \end{bmatrix}^T.
\]

and the six fundamental form coefficients:

\[
g_{11} = 1 + f_u^2 \quad g_{22} = 1 + f_v^2 \quad g_{12} = f_u f_v
\]

\[
b_{11} = \frac{f_{uu}}{\sqrt{1 + f_u^2 + f_v^2}} \quad b_{12} = \frac{f_{uv}}{\sqrt{1 + f_u^2 + f_v^2}} \quad b_{22} = \frac{f_{vv}}{\sqrt{1 + f_u^2 + f_v^2}}
\]

The expression for Gaussian curvature is given by:

\[
K = \frac{f_{uu} f_{vv} - f_{uv}^2}{(1 + f_u^2 + f_v^2)^2}
\]

And the expression for mean curvature is given by:

\[
H = \frac{f_{uu} + f_{uv} + f_{uu} f_v^2 + f_{uv} f_u^2 - 2 f_u f_v f_{uv}}{2 (1 + f_u^2 + f_v^2)^{3/2}}
\]

Thus if we are given a depth map function \(f(u, v)\) that possesses first and second partial derivatives, Gaussian and mean curvature can be computed directly.

4.2.1 Estimation of partial derivatives of Depth Maps

Partial derivatives of the range image can be obtained by fitting a continuous differentiable function that best fits the data. There are various techniques available in mathematics that have been used by computer vision researchers to determine partial derivatives of depth maps.
Using Discrete Orthogonal Polynomials

Besl and Jain [9] used discrete quadratic orthogonal polynomial fitting at each pixel to estimate derivatives. It is possible to control Neighborhood size for making local estimates which is important in case of actual range images.

A quadratic surface is fit at each pixel in the image, using a window convolution operator of size desired by the user.

Each point in the given window is associated with a position \((u, v)\) from the set \(UXU\) where \(N\) is odd:

\[
U = \left[\frac{-(N-1)}{2}, \ldots, -1, 0, 1, \ldots, \frac{(N-1)}{2}\right].
\]

The following discrete orthogonal polynomials provide the quadratic surface fit:

\[
\phi_0(u) = 1 \quad \phi_1(u) = u \quad \phi_2(u) = \left( u^2 - \frac{M(M+1)}{3} \right)
\]

Where \(M = \frac{(n-1)}{2}\). The \(b_i(u)\) functions are normalized orthogonal polynomials:

\[
b_0(u) = \frac{1}{\sqrt{N}} \quad b_1(u) = \frac{3u}{M(M+1)(2M+1)}
\]

\[
b_2(u) = \frac{1}{P(M)} \left( u^2 - \frac{M(M+1)}{3} \right)
\]

Where \(P(M)\) is a fifth-order polynomial in \(M\):

\[
P(M) = \frac{8}{45}M^5 + \frac{4}{9}M^4 + \frac{2}{9}M^3 - \frac{1}{9}M^2 - \frac{1}{15}M.
\]

\(b_i(u)\) vectors are computed according to the window size. First the surface estimate function \(\hat{f}(u, v)\) is calculated:

\[
\hat{f}(u, v) = \sum_{i,j=0}^{2} a_{ij} \phi_i(u) \phi_j(v)
\]

that minimizes the mean square term:

\[
\epsilon = \sum_{(u,v) \in U^2} \left( f(u, v) - \hat{f}(u, v) \right)^2
\]

53
Coefficients are given by:

\[ a_{ij} = \sum_{u,v \in U^2} f(u,v)b_i(u)b_j(v) \]

The first and second partial derivatives can then be directly read from the \( a_{ij} \) coefficients:

\[ f_u = a_{10} \quad f_v = a_{01} \quad f_{uv} = a_{11} \quad f_{uu} = 2a_{20} \quad f_{vv} = 2a_{02} \]

After the first and second partial derivatives are determined, surface characteristics at each pixel are calculated.

**Using Difference Operators**

Brady et al. [4] have used 3 × 3 difference operators to locally compute first and second derivatives of the Gaussian smoothed surface. Neighborhood size cannot be increased in this method. The operators are:

\[
\begin{bmatrix}
-1 & 0 & 1 \\
-1 & 0 & 1 \\
-1 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 \\
0 & 0 & 0 \\
-1 & -1 & -1
\end{bmatrix}
\begin{bmatrix}
-1 & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & -1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & -2 & 1 \\
1 & -2 & 1 \\
1 & -2 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 \\
-2 & -2 & -2 \\
1 & 1 & 1
\end{bmatrix}
\]

**Using B-Spline fitting**

Yang and Kak [33] have derived 3 × 3 operators using B-splines for computing partial derivatives of a range map. These can be combined with Gaussian operator to increase the window size and reduce sensitivity to noise. The operators give partial derivatives at the center pixel of each operator.
Least Squares Polynomial Fitting

We have used a fast least squares fitting method to derive partial derivatives in the symmetric neighborhood of a pixel. This method allows the neighborhood size to be controlled.

A surface fit of order $n$ can be written as:

$$f(x, y) = \sum_{i,j=0}^{i+j\leq n} a_{ij} x^i y^j$$

We have used second ($n=2$) order fitting in the neighborhood of every pixel to compute first and second order derivatives. Clearly, since the pixel at which derivatives are computed is at the origin, we get:

$$x = 0 \text{ and } y = 0$$

$$\frac{\partial f(x, y)}{\partial x} = a_{10} \quad \frac{\partial f(x, y)}{\partial y} = a_{01}$$

$$\frac{\partial^2 f(x, y)}{\partial x^2} = 2a_{20} \quad \frac{\partial^2 f(x, y)}{\partial y^2} = 2a_{02}$$

$$\frac{\partial^2 f(x, y)}{\partial x \partial y} = \frac{\partial^2 f(x, y)}{\partial y \partial x} = a_{11}$$

Thus derivatives are read off directly from the coefficients. We have also used the general least squares fitting procedure for fitting polynomial on surface patches. For the purpose of
computing derivatives it is observed that we always have symmetric Neighborhood around the pixel. This fact simplifies the least squares equations. See appendix B for the simplified least square fitting equations for second order bivariate approximation in a symmetric Neighborhood.

4.2.2 Results of Initial Segmentation

The above mentioned process is applied to actual range images and results are shown in figures 27,28,29, 30,31,32,33.

The smoothing behavior of Gaussian operator was briefly discussed in chapter 2. It is observed that step edges in range images are actually adjacent convex and concave edges. This is further amplified after smoothing the image with any size of Gaussian operator. Brady etal. [4] have restricted the Gaussian application to inside of the region. We have used Gaussian uniformly in the range image with the intention of uniformly smoothing the image for the purpose of obtaining reliable curvature estimates (see figure 26)
Figure 27: curvature estimation (a) original image. 192 × 256 12 bits/pixel image (b) smoothed image. (c) regions. (d) error in fitting
Figure 28: curvature estimation left to right, from top; original image. $150 \times 150$ 8 bits/pixel image; error in fitting; segmented regions; flat regions; convex regions; concave regions.
Figure 29: Initial labeling of scene with different threshold values (a) 0.01 (b) 0.02 (c) 0.03 (d) 0.04.
Figure 30: Labeling of scene: (a) all regions (b) convex (c) concave (d) flat.

Figure 31: Thresholded left: gauss. right: mean. black indicates zero, gray is positive and white is negative value
Figure 32: Histogram of Gaussian Curvature

Figure 33: Histogram of Mean curvature
For surface characterization purpose we use higher sigma value (= 1.5) and for post segmentation processing we work at lower level in scale. Step Edges detected at sigma = 1.0 are used to detect region boundaries in higher level processing stage discussed in next section.

Although the response of Gaussian and Mean curvature is reliable it is necessary to threshold the values around zero. 1% of maximum was used as threshold in all examples. Figures 27 and 28 show the labeled regions and error in fitting second order polynomial at the Neighborhood of each pixel, in images with 12 bits/pixel and 8 bits/pixel respectively. The fit-error is appreciable at boundaries including smooth edges. This means that curvature estimates at the edges are not reliable. At such points curvature magnitude may not be reliable though sign of curvatures is reliable. Further results are shown only for image in figure 27.

Figure 29 shows the effect of threshold values on curvature signs. As threshold values for Gaussian and mean curvature is changed, pixel labeling may change if the curvature magnitude is not appreciable. Image thresholded at 1% of maximum curvature magnitude (see figure 29(c)) has correct labelings. Further results are shown only for this threshold value. Pixels are classified as one of the eight basic types. We can classify the entire image into concave, convex, and flat regions by simply merging all neighboring pixels having similar type of surface, i.e, flat, concave or convex (see figure 30. Thresholded values of Gaussian and mean curvature are shown in figure 31. White patches indicate zero magnitude, gray indicate positive magnitude and black indicates negative curvature magnitude. It is observed that Gaussian curvature is mostly zero except for isolated patches, since the image has no spherical object. Non-zero mean curvature values are obtained at step edges and on a cylindrical object. Histogram of the magnitude of Gaussian and mean curvature (figure 32 and 33) for the entire image show appreciable mean curvature magnitude in the image and no significant Gaussian curvature. It can therefore be inferred that the scene has flat and possibly cylindrical objects.
4.3 Post processing of Labeled scenes

The segmentation done by labeling the individual pixels using sign of Gaussian and mean curvature is local in nature and threshold dependent. In order to interpret these labelings globally, we need to process the labeled image with global constraints. Besl and Jain [25] have proposed a variable order surface fitting algorithm. Surface patches are described as linear, quadric or cubic.

Our approach depends on the actual requirements. We describe two methods, (both are preliminary) to obtain useful segmentation given labeled image. The first method simply groups convex patches to form connected convex subparts of the scene. Second method uses the segmented objects obtained from algorithm described in chapter 3.

4.3.1 Obtaining Convex patches

As noted in third chapter, detection of smooth edges is difficult to extract using only local information. Curvature information at the all types of edges is easy to record. From figure 26 it is clear that edges in smoothed images can be recorded as thin convex and concave regions. In particular, convex edges are of convex cylinder type, with zero Gaussian curvature but appreciable negative mean curvature. Similarly, concave edges are of concave cylinder type, with zero Gaussian curvature but appreciable positive mean curvature. Thus all types of edges give either convex or concave cylindrical response. But the edge response is obtained over wider region due to smoothing and large window size during derivative computation. It is therefore not possible to have exact localization of patches obtained by merging convex regions.

A simple algorithm for obtaining convex patches is given below:

1. Read the labeled image.
2. Label each patch as a region.
3. Initialize the region data structure to record surface type, number of pixels, topmost pixel in the region, neighbours of the region, extremities of the region and the label
Figure 34: Convex patches

Figure 35: Convex patches
assigned to the region.

4. For the next unprocessed topmost region of the type flat or peak (convex sphere) or ridge (convex cylinder) with acceptable number of pixels do:

(a) Extend the original region to include all neighboring regions of type flat or ridge. Other types of regions are considered concave or part of other convex subpart. peak patches are not included because they will be selected as seed region.

(b) Repeat the above step to extend the region, till it is not possible to grow any more.

5. Output the convex subparts. End.

Figures 34 and 35 show convex patches obtained from labeled image obtained in figure 27 and in figure 28(a) respectively. Majority of objects in figure 34 are merged into one convex patch while they are separated in figure 35

4.3.2 Object Surface Classification

Surfaces on the segmented objects can be classified as one of basic surface type using the initial labeling based on sign of curvatures. Yang and Kak [33] have used extended Gaussian images to identify surface type on isolated surfaces. Histogram of labels in an isolated object can give some idea about the surface and guide the surface fitting process.

The classification algorithm is as follows:

1. Read in the segmented objects image and labeled surface image.

2. For each object in the image do:

   (a) Erode the object in labeled image so as to remove points within 5 pixel distance of the object boundary. This reduces the effect of smoothing and window size during curvature estimation which is mainly contributed by pixels near the boundary, and does not reflect the nature of region.
Figure 36: Classified surfaces

(b) Histogram the remaining pixel-label values.

(c) If more than 90% pixels are of one type, either flat or cylinder or sphere then the surface can be classified as such. If there are two or more peaks in the histogram, object has more than one surface type.

(d) In single surface cases fit the best fitting surface on the points. Output the description of surface.

(e) Further processing by region growing by surface fitting is necessary to smoothen the surface patches. Fit surfaces on individual patches and merge them by region growing.

This algorithm is being implemented. Initial classification process will classify the surfaces in figure 11 as 5 plane surfaces, 1 cylindrical and 1 irregular surface (the film mailer). See figure 36.

First and second order polynomials were fitted on flat and non-flat surface patches respectively in image of figure 11. The reconstructed image is shown in figure 37. Besl
and Jain [13, 25] have used initial labeling to obtain seed regions in the final region growing process. They perform variable order surface fitting to approximate the scene as a collection of piece-wise continuous functions.
Chapter 5

Discussion

Though results of running the various algorithms described in previous chapters on images acquired from different scanners are consistent, there is scope for refinement of all the approaches. We will discuss the merits and demerits of each method and suggest improvements.

We need to study the scale-space behavior of range images in detail. This would lead to a better understanding of the scale at which range images should be handled. We have noticed that thresholding of zero-crossings makes the entire segmentation procedure dependent on the threshold value. Though we have obtained consistent results with a fixed empirically determined value for all the images obtained from a particular scanner, threshold selection is not automatic. Secondly, even with right threshold value the region may not be completely bounded by the zero-crossings (in case of overlaps by thin objects or sensor noise) To make the whole process less sensitive to threshold, following post-processing steps (region splitting) are suggested:

1. Read in the segmented object.

2. Trace the contours around the object as it is defined now and also any other boundaries that are now lying inside the object. Except for the bounding contours, other contours may not be closed. They may simply lie within the region and actually are boundary of the real object. In such a case mark the beginning and end of the contour. If the
contour touches the closed contour then mark the point of contact as end of inner contour.

3. In all the contours mark the concavities.

4. Now split the region by connecting two contours (gap filling) or connecting two points of concavity (gap filling or region splitting) or connecting an end point of contour with a concavity, based on predetermined gap filling distance.

5. The output is the segmented object.

The above method should be indifferent to threshold values on higher side as it splits the region consisting of more than one regions. To reduce the sensitivity to low threshold values (which will result too many small regions) some sort of merging is required. Merging is a much difficult task, so it is better to keep the threshold high and have the post-segmentation process perform the splitting, rather than initial segmentation performing splitting due to low threshold value.

Another solution to splitting is to let higher level recognition process make globally valid observations to split the region. The higher level procedure may use a priori information or may make some assumptions or apply global constraints to split the region. Franc Solina’s (see [23]) superquadric procedure can split the regions into identifiable parts by performing model fitting on individual part of the object.

In chapter 4 we noticed that labeling of the scene based on curvature sign is threshold sensitive. While thresholding around zero is necessary to obtain meaningful results, it is not clear how that value can be automatically determined. Curvature determination being local, the labeling is sensitive to noise and surface texture. It is not well understood how to generate a global interpretation of such surfaces.
Bibliography


[12] B.K.P. Horn *Machine Vision*


[37] Darwin Kuan and Robert Drazovich; *Model-based Interpretation of Range Imagery*, Proc. of the National Conference on Artificial Intelligence, Austin, August 6-10, AAAI, pp 210-215.


Appendix A

2nd order Least squares fitting in symmetric neighborhood

The approximating polynomial is written as:

\[ I(x, y) = a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00} \]

The square error term is:

\[ \Sigma^2 = \sum_{i=1}^{N}(Z_i - I_i)^2 \]

\[ \Sigma^2 = \sum_{i=1}^{N}(Z_i^2 - 2Z_iI_i + I_i^2) \]

To minimize the least squares term, let \( \frac{\partial \Sigma^2}{\partial X_i} = 0 \).

which is:

\[ \frac{\partial \Sigma^2}{\partial a_{20}} = -2Z_i x_i^2 + 2x_i^2 I_i = 0 \]

\[ \frac{\partial \Sigma^2}{\partial a_{11}} = -2Z_i x_i y_i + 2x_i y_i I_i = 0 \]

\[ \frac{\partial \Sigma^2}{\partial a_{02}} = -2Z_i y_i^2 + 2y_i^2 I_i = 0 \]
\[
\frac{\partial \Sigma^2}{\partial a_{10}} = -2Z_i x_i + 2x_i I_i = 0
\]
\[
\frac{\partial \Sigma^2}{\partial a_{01}} = -2Z_i y_i + 2y_i I_i = 0
\]
\[
\frac{\partial \Sigma^2}{\partial a_{00}} = -2Z_i + 2I_i = 0
\]

Writing:

\[
S_{jik} = x_i^j y_i^j z_i^k
\]

we get:

\[
\begin{bmatrix}
S_{400} & S_{310} & S_{220} & S_{300} & S_{210} & S_{200} \\
S_{310} & S_{220} & S_{130} & S_{210} & S_{120} & S_{110} \\
S_{220} & S_{130} & S_{040} & S_{120} & S_{030} & S_{020} \\
S_{300} & S_{210} & S_{120} & S_{200} & S_{110} & S_{100} \\
S_{210} & S_{120} & S_{030} & S_{110} & S_{020} & S_{010} \\
S_{200} & S_{110} & S_{020} & S_{100} & S_{010} & S_{000}
\end{bmatrix}
\begin{bmatrix}
a_{20} \\
a_{11} \\
a_{10} \\
a_{11} \\
a_{01} \\
a_{00}
\end{bmatrix}
= \begin{bmatrix}
S_{201} \\
S_{111} \\
S_{021} \\
S_{101} \\
S_{011} \\
S_{001}
\end{bmatrix}
\]

In a symmetric neighborhood:

\[
S_{pq0} = 0 \text{ for odd } p \text{ or } q \text{ and}
\]

\[
S_{pq0} = S_{qp0}
\]

The above system of equations reduces to:

\[
\begin{bmatrix}
S_{400} & 0 & S_{220} & 0 & 0 & S_{200} \\
0 & S_{220} & 0 & 0 & 0 & 0 \\
S_{220} & 0 & S_{040} & 0 & 0 & S_{020} \\
0 & 0 & 0 & S_{200} & 0 & 0 \\
0 & 0 & 0 & 0 & S_{020} & 0 \\
S_{200} & 0 & S_{020} & 0 & 0 & S_{000}
\end{bmatrix}
\begin{bmatrix}
a_{20} \\
a_{11} \\
a_{10} \\
a_{02} \\
a_{01} \\
a_{00}
\end{bmatrix}
= \begin{bmatrix}
S_{201} \\
S_{111} \\
S_{021} \\
S_{101} \\
S_{011} \\
S_{001}
\end{bmatrix}
\]

75
Which can be written as:

\[ a_{10} = \frac{S_{101}}{S_{200}} \quad a_{01} = \frac{S_{011}}{S_{200}} \quad a_{11} = \frac{S_{111}}{S_{220}} \]

and

\[
\begin{bmatrix}
S_{400} & S_{220} & S_{200} \\
S_{220} & S_{040} & S_{020} \\
S_{200} & S_{020} & S_{000}
\end{bmatrix}
\begin{bmatrix}
a_{20} \\
a_{02} \\
a_{00}
\end{bmatrix}
= 
\begin{bmatrix}
S_{201} \\
S_{021} \\
S_{001}
\end{bmatrix}
\]
Appendix B

Source Code Listing

Listings of source programs is included in following pages.

1. space.c : Performs smoothing, median filtering, Gaussian filtering, Laplacian, marking zero-crossings and graphics display of histogram etc. in interactive manner.

2. segment.c : Segments all objects and topmost object in the scene, given original image and zero-crossings of LOG image. Also outputs supporting points of topmost object.

3. rca_calib.c : Generates a list of points for Z-depth format image. Originally written by Franc, modified to read PM files and add points horizontally and vertically.

4. classify.c : Procedure used to select and classify the superquadric models.

5. spline.c : Computes various surface characteristics of the image. Interactively displays results and histograms using quickdraw routines. Outputs labeled image and other characteristic as desired by the user.

6. grad.c : Has code for fast least squares fitting. Polynomial is fitted in the symmetric neighborhood of the pixel.

7. merge.c : Performs processing on the image labeled according to signs of Gaussian and mean curvature. It computes convex parts of the image, fits polynomials on patches using general least squares routines in solver.c.

8. solver.c : Has code for a general least squares fitting procedure.

There are other supporting programs that are vital to the algorithms. Superquadric fitting programs are developed by Franc Solina and are not listed here.
Program for computing scale-space description of the input image and lots of other things in interactive manner. Later will override scale.c.

Modifications for display of histogram done on Feb 18, 1988.

March 24 1988: To read/write in both PM.C and PM.S formats.

Y#include <stdio.h>
Y#include <math.h>
#include <local/pm.h>
Y#include <ik.h>
Ydefine BUFSIZE 256 /* Image buffer size */
Ydefine BUF_NUM 4 /* # of buffers available for manipulation */

float result[BUFSIZE][BUFSIZE]; /* stores original image */
float buffer[BUF_NUM][BUFSIZE]; /* temporarily store a buffer */
float x[BUFSIZE], y[BUFSIZE]; /* to store points */

int threshold; /* Threshold for detecting zero-crossings */
int booll, bool2, bool3;

int sizex, sizey; /* size of the image */

FILE *infile, *out; /* pointers to input image and output image file */
FILE *outs, *output_pm; /* output PM.S file */
char *smooth_cmt; /* smooth comment */
char ikonas_disp[10];
char *out_cmt; /* output comment */
int count;
unsigned char c;
static int nhb_x[8] = {0, 1, 1, 0, 1, -1, -1};
static int nhb_y[8] = {1, 1, 0, -1, 0, -1, 1};
float ndb[8], temp;
int l, j, k, m, n, b;
float nbr[10];
int qsize;
float sigma; /* size and sigma of the gaussian operator */
float sum;
float gauss[60];
float gsum;
int offset;
char input[20];
int offx = 0, offy = 0; /* offset coordinates, initialized */
char outfile[30]; /* name of the output file */
int b1, b2, b3;
unsigned char *pm_point;
int disp_row; /* pointer to short integer to hand PM.S format */
int factor; /* # by which PM.S image pixel to be displayed on IKONAS */

char input_filename[50];
int image_format; /* stores the format of the last image read */

if (argc != 2) {
    printf("usage: scale <input-image-file-pmpic><n>\n");
    exit(0);
}

if (argc != 2) {
    printf("Want to display on ikonas?\n");
    scanf("%s", &ikonas_disp[10]);
}

if(strcmp("\n", &ikonas_disp[10]) == 0) {
    printf("Can't open ikonas. Exiting\n");
    exit(0);
}
get comment line

cmt = pm_cmt(argc,argv);

open input pm file

read_picture(argv[1],0);
read_picture(argv[1],1);

printf("Rows : %d
Columns : %d",sizex,sizey);

processing of the commands starts now:

various available commands are:
"gauss": convolves the image with gaussian filter.
"cross": computes zero and other types of crossings in the image.
"save": saves indicated buffer in a file.
"disp": displays indicated buffer on the ikonas.
"add": adds the two buffers. Result is put in buffer 1.
"sub": subtracts two buffers. Result is put in buffer 1.
"buffer": selects the current buffer.
"offset": offset the picture on ikonas.
"original": indicated buffer gets original picture.
"read": Reads a file in the designated buffer.
"hist": Computes and displays the histogram using quickdraw.
"ikpm": saves the image displayed on ikonas in a file in

five active buffers are maintained to manipulate the original image


while(scanf("%s",input) != EOF) 
{ 
if((strcmp(input,"median") == 0) || (strcmp(input,"m") == 0)) 
{ 
/* do median filtering of the image */

b = readbuffer();
for(i=1;i<(size-1);i++)
for(j=1;j<(size-1);j++)
{ 
    count = 0;
    for(m=1;m<i+1;m++)
    for(n=1;n<j+1;n++)
        
        [ 
            ntr[count] = buffer[b][m][n];
    
    result[i][j] = get_median(ntr);
}
for(k=1;k<(size-1);k++)
for(j=1;j<(size-1);j++)
    buffer[b][i][j] = result[i][j];
} /* end of median filtering */
else if((strcmp(input,"gauss") == 0) || (strcmp(input,"g") == 0) 
{ 
    printf("size of window : ");
    scanf("%d",gsize);
}
/* compute the gaussian array */
gsum = 0;
for(i=-gsize/2;i>gsize/2;i++)
for(j=-gsize/2;j>gsize/2;j++)
{ 
    gauss[j] = (1.0/(sqrt((double)(2.0*3.1415926)*sigma)))*
        gsum += gauss[j];
    printf("gauss[%d] = %f",i,gauss[j]);
}
if (gsum == 0) gsum = 1;
printf("\n gsum = %f \n\n",gsum);
/* separately convolve x-axis */
for(j=0;j<(size);j++)
for(i=gsize/2;i<(size-gsize/2);i++)
{ 
    sum = 0;
    for(k=-gsize/2;k<gsize/2;k++)
    { 
        sum += buffer[b][i+k][j]*gauss[k+gsize/2];
    }
    result[i][j] = sum/gsum;
}
```c
/* seperately convolve y-axis */
for(1=0;1<sizey;1++)
    for(j=gsize/2;j=(sizey-gsize/2);j++)
    {
        sum = 0;
        for(k=-gsize/2;k<=gsize/2;k++)
            sum += result[1][j+k]*gauss[k+gsize/2];
        buffer[b][1][j] = sum/gsum;
    }
else if(strcmp("lap",input) == 0) || (strcmp("1",input) == 0))
    b = readbuffer();
/* apply laplacian operator */
    for(1=1;1<(sizey-1);1++)
        for(j=1;j<(sizey-1);j++)
    {
        result[1][j] = -4*buffer[b][1][j]+buffer[b][1][j-1][j]
                      +buffer[b][1][j-1][j-1]+buffer[b][1][j+1][j]
                      +buffer[b][1][j+1][j+1];
    }
else if(strcmp("crose",input) == 0) || (strcmp("c",input) == 0)
    b = readbuffer();
    printf("step edge magnitude desired ? enter 1 if yes >");
    scanf("d",&bool1);
    printf("concave edge magnitude desired ? enter 1 if yes >");
    scanf("d",&bool2);
    printf("convex edge magnitude desired ? enter 1 if yes >");
    scanf("d",&bool3);
/* trace zeros */
    for(1=0;1<(sizey-1);1++)
        for(j=1;j<(sizey-1);j++)
```
```
else if(strcmp("disp", input) == 0) || (strcmp("d", input) == 0) {
    /* display the indicated buffer in ikonas */
    b = readbuffer();
    printf("offx : %d ; offy : %d \n", offx, offy);
    if(image_format == PM_S)
        printf("factor to divide each pixel by : ");
    scanf("%d", &factor);
    for(i=0; i<sizex; i++)
    {
        if(image_format == PM_S)
            for(j=0; j<sizey; j++)
            {
                temp_buffer[j] = buffer[b][i][j]/factor;
                lwr_n(offx, i, offy, &temp_buffer[0], sizey);
            }
        else
            for(j=0; j<sizey; j++)
            {
                temp_buffer[j] = buffer[b][i][j];
                lwr_n(offx, i, offy, &temp_buffer[0], sizey);
            }
    }
}

} /* end of disp; display on ikonas */
else if(strcmp("ikclose", input) == 0)
    ikclose();
else if(strcmp("ikopen", input) == 0)
    ikopen(NULL);
else if(strcmp("save", input) == 0) || (strcmp("s", input) == 0) {
    /* save the indicated buffer in a file in ikonas format */
    b = readbuffer();
    printf("Enter output filename : ");
    scanf("%s", outfile);
    pm_write(outfile, b);
}

} /* end of save */
else if(strcmp("orig", input) == 0) || (strcmp("o", input) == 0) {
    b = readbuffer();
    /* load the input file in buffer b */
    for(i=0; i<sizex; i++)
        for(j=0; j<sizey; j++)
            buffer[b][i][j] = buffer[0][i][j];
}

/* open output file */
if((out = fopen(outfile, "w")) == NULL)
    { printf("file open error : %s \n", outfile);
      exit(0);
    }
    printf("row : %d col : %d \n", sizex, sizey);
    pm_addcnt(pml, cmnt);

    if(image_format == PM_C)
        { pm_point = (unsigned char *) pml->pm_image;
          for(i=0; i<sizex; i++)
              for(j=0; j<sizey; j++)
              { *(pm_point) = buffer[b][i][j];
                pm_point++;
              }
        }
    else /* format is PM_S */
    { 
        pms_point = (short int *) pml->pm_image;
        for(i=0; i<sizex; i++)
            for(j=0; j<sizey; j++)
            { *(pms_point) = buffer[b][i][j];
                pms_point++;
            }
    }
    pm_write(out, pml);
    fclose(out);
} /* end of save */
else if(strcmp("read", input) == 0) || (strcmp("r", input) == 0) {
    b = readbuffer();
    printf("enter input filename : ");
    scanf("%s", input_filename);
    read_picture(input_filename, b);
}
else if(strcmp("lkm", input) == 0) || (strcmp("lkpm", input) == 0) {
    printf("Nothing happened\n");
}
else if(strcmp("hist", input) == 0) || (strcmp("h", input) == 0) {
    b = readbuffer();
    display_histogram(b);
}
else if(strcmp("row", input) == 0) || (strcmp("rowscan", input) == 0) {
    b = readbuffer();
    printf("row : ");
    scanf("%d", &disp_row);
    while((disp_row < sizey) || (disp_row > 0))
    {
        display_row_histogram(disp_row, b);
        printf("row : ");
        scanf("%d", &disp_row);
    } printf(">");
}

/* compute edge strength and direction */
/* mask is like this : */
-------------
| 5 | 6 | 7 |
-------------
| 4 | x | 0 |
-------------
| 3 | 2 | 1 |
-------------

data_p(intensity, nbd, p_es)
float intensity, nbd[8],
    *p_es;
{
    static double delta[8] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
    static unsigned char tbl[9] = {0, 1, 8, 7, 6, 5, 4, 3, 2};
    float *p_es;
    int i, j, ed, halfgray, maxgray;
    float res, slope;
    res = 0.0;
    ed = -1;
    halfgray = 0;
    maxgray = 255;
    if(bool == 1)
    {
        if(intensity > halfgray)
        {
            for (i=0; i<8; i++)
            {
                if(nbd[i] != halfgray)
                {
                    slope = (float)(intensity - nbd[i])/delta[i];
                    if(slope > res)
                    {
                        res = slope;
                        ed = opposite(i);
                    }
                }
            }
        }
        else if(intensity == halfgray)
        {
            for (i=0; i<8; i++)
            {
                i = opposite(i);
                if((nbd[i] < halfgray) && (nbd[i] > halfgray))
                {
                    slope = (float)(nbd[i] - nbd[i])/delta[i];
                    if(slope > res)
                    {
                        res = slope;
                        ed = i;
                    }
                }
            }
        }
    }
if(bool2 == 1)
    if(intensity > halfgray)
        /* if required check for 0+0
type of crossings because they are generated by
concave surfaces */
        for(i=0; i<8; i++)
            j = opposite(i);
            if((nbd[i] <= halfgray) && (nbd[j] >= halfgray))
                slope = (float)(intensity - nbd[i])/delta[i];
                if(r(es) > res)
                    res = r(es);
                    ed = j;
    }

if(bool3 == 1)
    if(intensity <= halfgray)
        /* if required check for 0- or 0-0
types of crossings because they are generated by
concave surfaces */
        for(i=0; i<8; i++)
            j = opposite(i);
            if((nbd[i] == halfgray) && (nbd[j] <= halfgray))
                slope = (float)(nbd[i] - intensity)/delta[i];
                if(r(es) > res)
                    res = r(es);
                    ed = j;

if(image_format == PM.C)
    if(res == res * 5)
        *p_x = min(res, (float)(maxgray));
        *p_y = tbl[ed+1];

    opposite(1)
    int i;
    { int opp;
        if (i<4)
            opp = i+4;
        else
            opp = i-4;
        return(opp);
    }

    float min(i, j)
    float i, j;
    { if (i<j) return(i);
        else return(j);
    }

    readbuffer()
    { int b;
        printf("buffer ");
        scanf("%d", &b);
        if((b < 0) || (b > 5)) { b=readbuffer(); return(b); } else return(b);
    }

    float abssz(num)
    float num;
    { float i;
        if(num < 0) i = -num; else i = num;
        return(i); }

    /* if required check for 0- or 0-0
    types of crossings because they are generated by
    concave surfaces */
    for(i=0; i<8; i++)
        j = opposite(i);
        if((nbd[i] == halfgray) && (nbd[j] <= halfgray))
            slope = (float)(nbd[i] - intensity)/delta[i];
            if(r(es) > res)
                res = r(es);
                ed = j;
return(1);
}

float get_median(nbr)
float nbr[10];
{
 float nbrl[5];
 sort(nbr,nbrl);
 return(nbrl[4]);
}

sort(nbr,nbr)
float nbr[10],nbrl[5];
{
 int l,m;
 float large;
 int index;
 for(m=0;m<=5;m++)
 {
  large = -100;
  for(l=0;l<m;l++)
   {
    if(large < nbr[l])
    {
     large = nbr[l];
     index = l;
    }
   }
  nbr[m] = large;
  nblr[index] = 0;
 }
}

read_picture(filename,b)
char filename[];
int b;
{
 int 1,j;
 FILE *infs;
 unsigned char *pm_point;
 short int *pms_point;
 int offset;
 /* open input pm file */

 if ((infs = fopen(filename,”r”)) == NULL)
 {
  printf("file open error :\n",filename);
  exit(0);
 }

 /* read inputfile into the pmplc buffer */
 if((pm = pm_read(infs,0)) == NULL)
 {
  printf("error in reading the pmfile \%s",filename);
  exit(0);
 }

 size = (pm->pm_row); /* # of rows */
 sizey = (pm->pm_nocol); /* # of columns */

 image_format = pm->pm_form;

 if(pm->pm_form == PM_C)
 {
  pm_point = (unsigned char *) pm->pm_image;
  for(i=0;i<size;i++)
   for(j=0;j<sizey;j++)
    {
     buffer[i][j] = *pm_point;
     pm_point++;
    }
 }

 else if(pm->pm_form == PM_S)
 {
  pms_point = (short int *) pm->pm_image;
  for(i=0;i<size;i++)
   for(j=0;j<sizey;j++)
    {
     buffer[i][j] = *pms_point;
     pms_point++;
    }
 }

 else fprintf(stderr,"Image in unrecognized format. Buffer not init

 } /* end of read_picture */

display_histogram(buf)
int buff;
{
 int 1,3;
float xmin, xmax, ymin, ymax;
int iopt;
int npts;
int max;
int value;

for (i=0;i<256;i++)
{
    x[i] = i;
    y[i] = 0;
}

qterm(4);
max = -1000;
for (i=0;i<sizeX;i++)
    for (j=0;j<sizeY;j++)
    {
        value = absy((int)(buffer[buf][i][j]));
        y[value] = y[value] + 1;
        if (max < y[value]) max = y[value];
    }

printf(" max : \\
iopt = 0;
npts = 256;

xmin = 0.0;
xmax = 255.0;
ymin = 0.0;
printf("ymax :");
scanf("%f", &ymax);

qkdraw(npts, x, iopt, &xmin, &xmax, &ymin, &ymax);
qdtitel(" R H ");
qxlabl("Ranges");
qylabl("Values");
qdDONE();
}

absy(yvalue)
int value;
{
    if(value < 0) value = -value;
    if(value > 255) return(255);
}

else return(value);
}

display_row_bar_histogram(row, buff)
int row;
int buff;
{
    int 1, j;
    float xmin, xmax, ymin, ymax;
    int iopt;
    int npts;
    int max;
    int value;
    int row_num;
    for (i=0;i<sizeX;i++)
    {
        x[i] = i;
    }
    for (i=0;i<sizeY;i++)
        y[i] = 0;

    qterm(4);
    max = -1000;
    for (j=0;j<sizeY;j++)
    {
        value = absy((int)(buffer[buf][row][j]));
        y[j] = value;
    }
    iopt = 0;
npts = sizeY;
    xmin = 0.0;
xmax = sizeY;
ymin = 0.0;
ymax = 255;
qkdrow(npts, x, iopt, &xmin, &xmax, &ymin, &ymax);
qdtitel(" R H ");
qxlabl("Ranges");
qylabl("Values");
```c
float squarezz(float number)
{
  return (number * number);
}
```
This program performs segmentation on the laplacian edge operator image and marks the segmented object in the range image.

Modified on Jan 28, 1988 to extract supporting surface information for the segmented object.

Feb 1, 1988: Recursive call to grow region modified to incorporate control over the # of pending recursive calls at a given time. This requires maintaining a FIFO queue of open nodes.
March 24, 1988: Modified to accept PM and PM5 format images. Output images in respective format.

Region label 1 is reserved for unaccepted regions. Max regions allowed is determined by MAX_REGION = MAX_REGION -1 to 2.

#include <stdio.h>
#include <math.h>
#include <k.h>
#include <local/pm.h>

#define MIN_VALUE 0  /* =0 if background < shadows (non-zero backgro*/
#define MIN_ACCEPTABLE 300  /* Minimum # of pixels in a valid region*/
#define BUF 256  /* Buffer size in one dimension*/
#define MAX_NUM_CALL 500  /* Maximum # of pending recursive calls*/
#define MAX_REGION 2550  /* Maximum # of regions allowed*/
#define UNACCEPT_REGION 1  /* Label of rejected regions*/
#define PIXEL_STACK_SIZE 10000  /* Size of the pixel stack used for recursive-iterative region growing*/

int range_image_buffer[BUF][BUF];  /* input image buffer*/
int lap_image_buffer[BUF][BUF];  /* laplacian image buffer*/
int third_image_buffer[BUF][BUF];  /* temporary buffer for marking visited points*/
int output_buffer[BUF][BUF];  /* segmented image buffer w/ labeled object*/
int support_image_buffer[2][BUF][BUF];  /* supported surface*/
int average_image[BUF][BUF];  /* average 2X2 stored*/
int vector[65000];
int edge_threshold;  /* threshold for edge strength*/
int rrow,ncol;  /* global variables to store and column values*/

long int count;
int distance;
int seedrow,seedcol;
int back_threshold;
int a_depth;
float squeezer();
char accept_region();
int隊wvalue = 255;
int ikonas_displ[10];
int ppmic[*pm1,*pm2,*pm3];
int label;
int pixels;
int region_count;
int invalid_region_count;

struct region_type {
  int number;
  int valid;
  int size;
  int max;
  int min;
  int region[MAX_REGION];
  int row_max;
  int column_max;
  int max;
}

struct p_stack {
  int row;
  int col;
  int pixel_stack[PIXEL_STACK_SIZE];
}

int rownum;
int column;

int num_call;
int stack_length;
int current_element;
int nextrow,nextcol;
int top_region, label;
int max_pixel_region;
int min_pixel_region;
int gaprow,gapcol;

/* maximum allowable distance the seed region*/
/* seed region coordinates*/
/* threshold for background*/
/* average depth*/
/* Run time display on ikonas*/
/* Pointers to PM-picture st*/
/* label value to identify re*/
/* # of valid regions found*/
/* counts only invalid region*/
/* Label of the region*/
/* =0 if not valid, >1 if val*/
/* # of pixels in the region*/
/* maximum pixel of the region*/
/* minimum pixel of the region*/
/* row number of max depth pixel*/
/* column number of max depth pixel*/
/* depth value of the max depth pixel*/
/* stores the open pixels in circular*/
/* (rownum, colnum) is the next pixel*/
/* popped from the stack*/
/* number of calls pending at a given*/
/* current element if queue is empty*/
/* points to head of the queue*/
/* nextcol and nextcol for the seed re*/
/* label of the topmost region*/
/* maximum pixel of the region*/
/* minimum pixel of the region*/
/* the nbd edge pixel found, returned*/
gapfiller(), if point lies one dist from the edge */

pmpic *read_pmpic_int();

/*****************************************************************************/

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

FILE *range,*laplacian,*outfile,*outfile2,*outfile3;
int i,j,k,l,m,n;
char output[50];    /* generated output file name */
int row,column;    /* # of row and columns in in
text data */
char *cmt;
int done,found;    /* used as booleans */
unsigned char *pm_point,*pm_point2,*pm_point3;
short int *pms_point,*pms_point2,*pms_point3;
unsigned int image_format;    /* stores image format PM_C or

if(argc != 3)
{
    printf("segment : usage : segment <range_file> <lap_edge
exit(1);
}

cmt = pm_cmt(argc,argv);
printf("Want to display on IKONAS ? ");
scanf("%s",<ikonas_disp[0]);

/* open the IKONAS display. value of env. variable is taken */
if(strcmp("y",<ikonas_disp) == 0)
    if(kopen(NULL) != -1)
    {
        printf("Can't open IKONAS. Exiting\n");
        exit(0);
    }

if((range = fopen(argv[1],"r") == NULL)
    printf("segment : Cannot open %s\n",argv[1]);
    exit(0);
}

if((laplacian = fopen(argv[2],"r") == NULL)
    printf("segment : Cannot open %s\n",argv[2]);
    exit(0);
}

if(pml = pm_read(range,0) == NULL)
    printf("Error in reading PM file : %s\n",argv[1]);
    exit(0);

if(pm2 = pm_read(laplacian,0) == NULL)
    printf("Error in reading PM file : %s\n",argv[2]);
    exit(0);

    /* open output files */
    strcpy(output,argv[1]);
    strcat(output,.label";
    if((outfile = fopen(output,"w") == NULL)
    {
        printf("segment : Cannot open output file : %s\n",out
exit(0);
    }

    strcpy(output,argv[1]);
    strcat(output,.top";
    if((outfile = fopen(output,"w") == NULL)
    {
        printf("segment : Cannot open output file : %s\n",out
exit(0);
    }

    strcpy(output,argv[1]);
    strcat(output,.support";
    if((outfile = fopen(output,"w") == NULL)
    {
        printf("segment : Cannot open output file : %s\n",out
exit(0);
    }

    row = pml->pm_row;
column = pml->pm_ncol;
image_format = pml->pm_form;
if((row != pm2->pm_nrow) || (column != pm2->pm_ncol))
{
    printf("Input images are not of same size. Exiting \n");
    exit(0);
}

nrow = row; /* initialize global variables nrow and ncol */
ncol = column;

printf("Edge threshold ":
; scanf("%d",&edge_threshold);

printf("Background threshold ":
; scanf("%d",&background_threshold);

printf("Max. distance from seed region ":
; scanf("%d",&distance);

/* read the input image and laplacian edge operated image */
/* first read the range image */
if(pm1->pm_form == PM_C)
{
    pm_point = (unsigned char *) pm1->pm_image;
    for(i=0;i<row;i++)
        for(j=0;j<column;j++)
        {
            range_image_buffer[i][j] = *(pm_point);
            pm_point++;
        }
}
else if(pm1->pm_form == PM_S)
{
    pms_point = (short int *) pm1->pm_image;
    for(i=0;i<row;i++)
        for(j=0;j<column;j++)
        {
            range_image_buffer[i][j] = *(pms_point);
            pms_point++;
        }
}
/* Now read laplacian operated image */
if(pm2->pm_form == PM_C)
{
    pm_point = (unsigned char *) pm2->pm_image;
    for(i=0;i<row;i++)
        for(j=0;j<column;j++)
        {
            range_image_buffer[i][j] = *(pm_point);
            pm_point++;
        }
}
else if(pm2->pm_form == PM_S)
{
    pms_point = (short int *) pm2->pm_image;
    for(i=0;i<row;i++)
        for(j=0;j<column;j++)
        {
            range_image_buffer[i][j] = *(pms_point);
            pms_point++;
        }
}

/* Initialize output and temporary buffers */
for(i=0;i<row;i++)
    for(j=0;j<column;j++)
    {
        third_image_buffer[i][j] = 0;
        output_buffer[i][j] = 0;
        support_image_buffer[0][i][j] = 0;
        support_image_buffer[1][i][j] = 0;
        bzero((char *) &third_image_buffer[0][0],sizeof(int)*row*column);
        bzero((char *) &output_buffer[0][0],sizeof(int)*row*column);
        bzero((char *) &support_image_buffer[0][0][0],sizeof(int)*row);
        bzero((char *) &support_image_buffer[1][0][0],sizeof(int)*row);
    }

/* Open files for output and initialize buffers */
pm1 = pm_alloc(); /* supporting points image */
pm2 = pm_alloc(); /* Range points */
pm2->pm_nrow = row;
pm2->pm_ncol = column;
pm2->pm_form = image_format;
pm2->pm_image = (char *) malloc(pm_size(pm2));

pm3 = pm_alloc(); /* Top most segmented object */
pm3->pm_nrow = row;
pm3->pm_ncol = column;
pm3->pm_form = image_format;
pm3->pm_image = (char *) malloc(pm_size(pm3));

pm4 = pm_alloc(); /* labeled image */
pm4->pm_nrow = row;

pm1->pm_ncolm = column;
pm1->pm_form = image_format;
pm1->pm_image = (char *) malloc(pm_psizem(pml));

if(image_format == PM_S)
{
    pms_point = (short int *) pm1->pm_image;
    pms_point2 = (short int *) pm2->pm_image;
    pms_point3 = (short int *) pm3->pm_image;
}
else /* image format is PM_C */
{
    pm_point = (unsigned char *) pm1->pm_image;
    pm_point2 = (unsigned char *) pm2->pm_image;
    pm_point3 = (unsigned char *) pm3->pm_image;
}
/* initialize the buffers */
bzero(pm1->pm_image, pm_isz(pm1));
bzero(pm2->pm_image, pm_isz(pm2));
bzero(pm3->pm_image, pm_isz(pm3));

/*** Segmentation of the picture starts ***/

max = -1000;
if(image_format == PM_S)
    label = MAX_REGION; /* starting label+1; for unlabeled*
else
    label = MAX_REGION; /* counts only valid regions */
invalid_region_count = 0; /* counts only invalid regions */
extrow = nextcol = 1; /* initialize the seed region *

************* Loop for segmenting all the objects in the range

    do
        region_count++;
        label--; /*
        if(region_count == 1) find_seed_region();
        else seed_region();
        if(max > 0)
            /* call grow_region for recursive region growing */
        
    while(region_count > 0)

exit 0;

}
while(max > 0));

/* Label all the unwanted regions as UNACCEPT_REGION */
label++; /* label of the last region */

for(i=0; i<row; i++)
  for(j=0; j<column; j++)
    if(third_image_buffer[i][j] == 0)
      output_buffer[i][j] = UNACCEPT_REGION;

region[UNACCEPT_REGION].valid = 0; /* unaccept_region is invalid */

max = 1000;
for(i=0; i<MAX_REGION-1; i++)
  region[i].valid = 1;

if(region[0].max > max)
  { max = region[0].max;
    top_region_label = region[0].number;
    max_pixel_region = region[0].max;
    min_pixel_region = region[0].min;
  }

printf("\nTotal # of Valid regions : %d\n", region_count-1);
printf("Total # of Invalid regions : %d\n", invalid_region_count);
printf("Topmost region : Label = %d Max = %d Min = %d\n", top_region_label, max_pixel_region, min_pixel_region);

/* output the segmented edge image */

for(i=0; i<row; i++)
  {
    output_buffer[i][0] = 0;
    output_buffer[i][column-1] = 0;
  }

for(i=0; i<column; i++)
  {
    output_buffer[0][i] = 0;
    output_buffer[row-1][i] = 0;
  }

if(image_format == PM_C)
  for(i=0; i<row; i++)
    for(j=0; j<column; j++)
      {
        label = output_buffer[i][j];
        if(label < column-1)
          if(output_buffer[i][j] != output_buffer[i][j+1])
            if(region[output_buffer[i][j]].valid == 1)
              { (region[output_buffer[i][j+1]].valid == 1))
                *pm_point = 255;
              if(label == output_buffer[i][j+1])
                if(((region[output_buffer[i][j+1]].valid == 1) ||
                  (region[output_buffer[i][j]].valid == 1))
                  *pm_point = 255;
              if(output_buffer[i][j] == top_region_label)
                *pm_point3 = range_image_buffer[i][j]; /* top */
              if(region[support_image_buffer[i][j]].valid == 1) */
                *pm_point2 = support_image_buffer[0][i][j]; /* sup */
                pm_point++;
                pm_point2++;
                pm_point3++;
          }
          else /* image_format == PM_S */
            for(i=0; i<row; i++)
              for(j=0; j<column; j++)
                {
                  label = output_buffer[i][j];
                  if(label < column-1)
                    if(output_buffer[i][j] != output_buffer[i][j+1])
                      if(((region[output_buffer[i][j+1]].valid == 1) ||
                          (region[output_buffer[i][j]].valid == 1))
                          *pm_point = 255;
                    if(label == output_buffer[i][j+1])
                      if(region[support_image_buffer[i][j]].valid == 1) */
                        *pm_point3 = range_image_buffer[i][j]; /* top */
                      if(region[support_image_buffer[i][j]].valid == 1) */
                        *pm_point2 = support_image_buffer[0][i][j]; /* sup */
                        pm_point++;
                        pm_point2++;
                        pm_point3++;
                }

                pm_write(outfile, pm1);
                pm_write(outfile2, pm2);
                pm_write(outfile3, pm3);
          } /* end of main program */
/***********MAIN***********/

main(argc, argv)
char *argv[];
{
    FILE *range,*laplacian,*outfile,*outfile2,*outfile3;
    int i,j,k,l,m,n;
    char output[50];        /* generated output file name*/
    int row,column;         /* # of row and columns in in
    char *cmt;
    int done,found;          /* used as booleans */
    unsinged char *pm_point,*pm_point2,*pm_point3;
    short int *pms_point,*pms_point2,*pms_point3;
    u_int image_format;      /* stores image format PM.C or
    
    if(argc != 3)
    { printf("segment : usage : segment <range_file> <lap_edge_
    exit(1);
    }
    
    cmt = pm_cmt(argc, argv);
    printf("Want to display on IKONAS ? ");
    scanf("%s",<ikonas_disp[0]);
    /* open the IKONAS display. Value of env. variable is taken *
    if(strcmp("y",ikonas_disp) == 0)
        if(!open(NULL) == -1) 
        { printf("Can't open IKONAS. Exiting\n");
          exit(0);
        }
    if((range = fopen(argv[1],"r")) == NULL)
    { printf("segment : Cannot open %s\n",argv[1]);
      exit(0);
    }

    if((laplacian = fopen(argv[2],"r")) == NULL)
    {
        printf("segment : Cannot open %s\n",argv[2]);
        exit(0);
    }

    if((pm1 = pm_read(range,0)) == NULL)
    {
        printf("Error in reading PM file : %s\n",argv[1]);
        exit(0);
    }
    if((pm2 = pm_read(laplacian,0)) == NULL)
    {
        printf("Error in reading PM file : %s\n",argv[2]);
        exit(0);
    }
    
    /* open output files */
    strcpy(output,argv[1]);
    strcat(output,".label");
    if((outfile = fopen(output,"w")) == NULL)
    { printf("segment : Cannot open output file : %s\n",out
    exit(0);
    }
    
    strcpy(output,argv[1]);
    strcat(output,".top");
    if((outfile = fopen(output,"w")) == NULL)
    { printf("segment : Cannot open output file : %s\n",out
    exit(0);
    }
    
    strcpy(output,argv[1]);
    strcat(output,".support");
    if((outfile = fopen(output,"w")) == NULL)
    { printf("segment : Cannot open output file : %s\n",out
    exit(0);
    }
    
    row = pm1->pm_row;
    column = pm1->pm_ncol;
    image_format = pm1->pm_form;
if((row != pm2->pm_nrow) || (column != pm2->pm_ncol))
{
    printf("Input images are not of same size. Exiting \n");
    exit(0);
}

nrow = row;  /* initialize global variables nrow and ncol */
ncol = column;

printf("Edge threshold : ");
scanf("%d", &edge_threshold);

printf("Background threshold : ");
scanf("%d", &back_threshold);

printf("Max. distance from seed region : ");
scanf("%d", &distance);

/* read the input image and laplacian edge operated image */

/* first read the range image */
if((pm1->pm_form == PM_C))
{
    pm_point = (unsigned char *) pm1->pm_image;
    for(i=0;i<nrow;i++)
        for(j=0;j<column;j++)
        {
            range_image_buffer[i][j] = *(pm_point);
            pm_point++;
        }
}
else if((pm1->pm_form == PM_S))
{
    pms_point = (short int *) pm1->pm_image;
    for(i=0;i<nrow;i++)
        for(j=0;j<column;j++)
        {
            range_image_buffer[i][j] = *(pms_point);
            pms_point++;
        }
}  /* now read laplacian operated image */
if((pm2->pm_form == PM_C))
{
    pm_point = (unsigned char *) pm2->pm_image;
    for(i=0;i<nrow;i++)
        for(j=0;j<column;j++)
        {
            lap_image_buffer[i][j] = *(pm_point);
            pm_point++;
        }
}
else if((pm2->pm_form == PM_S))
{
    pms_point = (short int *) pm2->pm_image;
    for(i=0;i<nrow;i++)
        for(j=0;j<column;j++)
        {
            lap_image_buffer[i][j] = *(pms_point);
            pms_point++;
        }
}
/* Initialize output and temporary buffers */
for(i=0;i<nrow;i++)
    for(j=0;j<column;j++)
    {
        third_image_buffer[i][j] = 0;
        output_buffer[i][j] = 0;
        support_image_buffer[0][i][j] = 0;
        support_image_buffer[1][i][j] = 0;
    }

bzero((char *) 4*third_image_buffer[0][0], sizeof(int)*row*column);
bzero((char *) 4*output_buffer[0][0], sizeof(int)*row*column);
bzero((char *) 4*support_image_buffer[0][0][0], sizeof(int)*row*column);
bzero((char *) 4*support_image_buffer[1][0][0], sizeof(int)*row*column);

/* Open files for output and initialize buffers */

pm2 = pm_alloc();  /* supporting points image */
    pm2->pm_nrow = row;
    pm2->pm_ncol = column;
    pm2->pm_form = image_format;
    pm2->pm_image = (char *) malloc(pm_size(pm2));

pm3 = pm_alloc();  /* top most segmented object */
    pm3->pm_nrow = row;
    pm3->pm_ncol = column;
    pm3->pm_form = image_format;
    pm3->pm_image = (char *) malloc(pm_size(pm3));

pm1 = pm_alloc();  /* labeled image */
    pm1->pm_nrow = row;
    pm1->pm_ncol = column;
    pm1->pm_form = image_format;
    pm1->pm_image = (char *) malloc(pm_size(pm1));
/* Segment the picture starts */

max = -1000;
if (image_format == PM_S)
  label = MAX_REGION; /* starting label+1; for unlabeled */
else
  label = MAX_REGION; /* counts only valid regions */
invalid_region_count = 0; /* counts only invalid regions */
nextrow = nextcol = 1; /* initialize the seed_region star */

************ Loop for segmenting all the objects in the range
  do
    region_count++;
    label++; 

    if(region_count == 1) find_seed_region();
    else
      seed_region();

    if(max > 0)
      { /* call grow_region for recursive region growing */

      }
  }

else printf("\n No more valid regions \n");
while(max > 0));

/* Label all the unwanted regions as UNACCEPT_REGION */
label++; /* label of the last region */
for(i=0;i<row;i++)
for(j=0;j<column;j++)
if(region[i][j] == 0)
    output_buffer[i][j] = UNACCEPT_REGION;
max = -1000;
for(i = (MAX_REGION - 1); i >= label; i--)
if(region[i].valid == 1)
    if(region[i].max > max)
        { max = region[i].max;
        top_region_label = region[i].number;
        max_pixel_region = region[i].max;
        min_pixel_region = region[i].min;
        }
printf("\n\nTotal # of Valid regions : %d\n",region_count-1);
printf("Total # of invalid regions : %d\n",invalid_region_count);
printf("Topmost region : Label = %d Max = %d Min \n",top_r
/* output the segmented edge image */
for(i=0;i<row;i++)
    { output_buffer[i][0] = 0;
    output_buffer[i][column-1] = 0;
    }
for(i=0;i<column;i++)
    { output_buffer[0][i] = 0;
    output_buffer[row-1][i] = 0;
    }
if(image_format == PM_C)
for(i=0;i<row;i++)
    for(j=0;j<column;j++)
        { label = output_buffer[i][j];
        if(j!=column-1)
            if(output_buffer[i][j] != output_buffer[i][j+1])
                if((region[output_buffer[i][j]].valid == 1) ||
                (region[output_buffer[i][j+1]].valid == 1))
                *pm_point = 255;
        if((row==i))
            if(output_buffer[i][j] != output_buffer[i+1][j])
                if((region[output_buffer[i][j]].valid == 1) ||
                (region[output_buffer[i+1][j]].valid == 1))
                *pm_point = 255;
        if(output_buffer[i][j] == top_region_label)
            *pm_point3 = range_image_buffer[i][j]; /*top */
        if(region[support_image_buffer[1][1][j]].valid == 1)
            if(support_image_buffer[1][1][j] == top_region_label)
                *pm_point2 = support_image_buffer[0][1][j]; /*sup*/
                pm_point++;
                pm_point2++;
                pm_point3++;
        else /* image_format == PM_S */
            { label = output_buffer[i][j];
                if(j!=column-1)
                    if(output_buffer[i][j] != output_buffer[i][j+1])
                        if((region[output_buffer[i][j]].valid == 1) ||
                        (region[output_buffer[i][j+1]].valid == 1))
                            *pm_point = 255;
                if((row==i))
                    if(output_buffer[i][j] != output_buffer[i+1][j])
                        if((region[output_buffer[i][j]].valid == 1) ||
                        (region[output_buffer[i+1][j]].valid == 1))
                            *pm_point = 255;
                    if(output_buffer[i][j] == top_region_label)
                        *pm_point3 = range_image_buffer[i][j]; /*top */
                    if(region[support_image_buffer[1][1][j]].valid == 1)
                        if(support_image_buffer[1][1][j] == top_region_label)
                            *pm_point2 = support_image_buffer[0][1][j]; /*sup*/
                            pm_point++;
                            pm_point2++;
                            pm_point3++;
            }
            pm_write(outfile,pml);
            pm_write(outfile2,pml);
            pm_write(outfile3,pml);
} /* end of main program */
double pm1->pm_ncol = column;
pm1->pm_format = image_format;
pm1->pm_image = (char *) malloc(pm_psize(pm1));

if(image_format == PM_S)
    {
        pm_point = (short int *) pm1->pm_image;
        pm_point2 = (short int *) pm2->pm_image;
        pm_point3 = (short int *) pm3->pm_image;
    }
else /* image format is PM_C */
    {
        pm_point = (unsigned char *) pm1->pm_image;
        pm_point2 = (unsigned char *) pm2->pm_image;
        pm_point3 = (unsigned char *) pm3->pm_image;
    }

/* initialize the buffers */
bzero(pm1->pm_image,pminsize(pm1));
bzero(pm2->pm_image,pminsize(pm2));
bzero(pm3->pm_image,pminsize(pm3));


max = -1000;
if(image_format == PM_S)
    {
        label = MAX_REGION; /* starting label+1; for unlabelled
        else
        label = MAX_REGION;
        region_count = 0; /* counts only valid regions */
        invalid_region_count = 0; /* counts only invalid regions */
        nextrow = nextcol = 1; /* initialize the seed region star

        ************ Loop for segmenting all the objects in the range
        do
        region_count++;
        label++;
        if(region_count == 1) find_seed_region();
        else
            seed_region();
        if(max > 0)
        { /* call grow_region for recursive region growing */
        }
    }
else printf("\n No more valid regions \n");
seedrow = row_max; /* seed region coordinates in globa
seedcol = column_max;
third_image_buffer[seedrow][seedcol] = 1;
pixels=0;
num_call = 1;
stack_length = 0;
current_element = 0;
max_pixel_region = -1000;
min_pixel_region = 1000;
grow_region(row_max,column_max);
while(stack_length != current_element)
    {
        get_pixel(); /* returns the pixel */
        num_call = 1;
        /*printf("AAAAA");*/
        if(output_buffer[rownum][colnum] != label)
            /*printf(" BBBBB ");*/
            /*grow_region(rownum,colnum);
    }
smooth_region(label);
if(accept_region(label) == 'n')
    {
        region_count--;
        invalid_region_count++;
        region[region].number = label;
        region[region].valid = 0;
        region[region_count].size = pixels;
    }
else
    {
        printf("label = %d ",label);
        printf("Top:(%d,%d) = %d \n",row_max,column_max,ma
        printf("pixels : %d ACCEPT ",pixels);
        printf("Max : %d Min td \n",max_pixel_region,min
        region[label].number = label;
        region[label].valid = 1;
        region[label].size = pixels;
        region[label].max = max_pixel_region;
        region[label].min = min_pixel_region;
        determine_support(label);
    }
else printf("\n No more valid regions \n");
while (max > 0)
{
    /* Label all the unwanted regions as UNACCEPT_REGION */
    label++; /* label of the last region */
    for (i=0; i<row; i++)
        for (j=0; j<column; j++)
            if ((region[output_buffer[i][j]].valid == 0)
                output_buffer[i][j] = UNACCEPT_REGION;
            region[UNACCEPT_REGION].valid = 0; /* unacceptable region is inv

        max = -1000;
        for (i = (MAX_REGION - 1); i >= label; i--)
            if (region[i].valid == 1)
            if (region[i].max > max)
            {
                max = region[i].max;
                top_region_label = region[i].number;
                max_pixel_region = region[i].max;
                min_pixel_region = region[i].min;
            }

            printf("\n\nTotal # of Valid regions : %d\n", region_count - 1);
            printf("Total # of Invalid regions : %d\n", invalid_region_count);
            printf("Topmost region : Label = %d Max = %d Min %d \n", top_r

    } /* output the segmented edge image */
    for (i=0; i<row; i++)
    {
        output_buffer[i][0] = 0;
        output_buffer[i][column-1] = 0;
    }

    for (j=0; j<column; j++)
    {
        output_buffer[0][j] = 0;
        output_buffer[row-1][j] = 0;
    }

    if (image_format == PM_C)
    for (i=0; i<row; i++)
        for (j=0; j<column; j++)
        {
            label = output_buffer[i][j];
            if (j<column-1)
                if (output_buffer[i][j] != output_buffer[i][j+1])
                if ((region[output_buffer[i][j]].valid == 1) ||
                    (region[output_buffer[i][j+1]].valid == 1))
                *pm_point = 255;
            if (output_buffer[i][j] != output_buffer[i+1][j])
                if ((region[output_buffer[i+1][j]].valid == 1) ||
                    (region[output_buffer[i][j]].valid == 1))
                *pm_point = 255;

    if (output_buffer[i][j] == top_region_label)
        *pm_point3 = range_image_buffer[i][j]; /* top */
    if (region[support_image_buffer[i][j]].valid == 1)
        if (support_image_buffer[i][j].valid == 1)
            *pm_point2 = support_image_buffer[0][j][j]; /* sup */
            pm_point++;
            pm_point2++;
            pm_point3++;

            else /* image_format == PM_S */
            for (i=0; i<row; i++)
                for (j=0; j<column; j++)
                {
                    label = output_buffer[i][j];
                    if (j<column-1)
                        if (output_buffer[i][j] != output_buffer[i][j+1])
                            if ((region[output_buffer[i][j+1]].valid == 1) ||
                                (region[output_buffer[i][j]].valid == 1))
                            *pm_point = 255;
                        if (output_buffer[i][j] != output_buffer[i+1][j])
                            if ((region[output_buffer[i+1][j]].valid == 1) ||
                                (region[output_buffer[i][j]].valid == 1))
                            *pm_point = 255;

                    if (output_buffer[i][j] == top_region_label)
                        *pm_point3 = range_image_buffer[i][j]; /* top */
                    if (region[support_image_buffer[i][j]].valid == 1)
                        if (region[output_buffer[i][j]].valid == 1)
                            *pm_point3 = support_image_buffer[0][j][j]; /* sup */
                            pm_point3++;
                            pm_point2++;
                            pm_point3++;
                }

            pm_write(outfile1, pm1);
            pm_write(outfile2, pm2);
            pm_write(outfile3, pm3);

        } /* end of main program */
/************ GROW_REGION ************
/
/* Following is the modified recursive call for the segmentation. 
* now a pixel is examined before routine is called rather than calling 
* the routine and then checking the pixel type. this was done to opti 
* the stack-space which overflows in the case of large objects. 
* even after this modification stack overflows if the object is 
* therefore recursive region growing is not possible if the object is 
*/

grow_region(row,col)
int row;
int col;
{
    int i,j;       /* loop control variables */
    int dist;
    int cross_grad;

    if(strcmp("y",ikonas_disp) == 0)
        lwr[col,row,&value];
        output_buffer[row][col] = label;

    if(max_pixel_region < range_image_buffer[row][col])
        max_pixel_region = range_image_buffer[row][col];
    if(min_pixel_region > range_image_buffer[row][col])
        min_pixel_region = range_image_buffer[row][col];

    pixels++;

    /* if((row < (nrow-1)) && (col < (ncol-1))) */
    cross_grad = sqrt((double)
        (squarezz(range_image_buffer[row][col] -
        range_image_buffer[row+1][col+1] -
        range_image_buffer[row][col+1] -
        range_image_buffer[row+1][col]))
    
    else /*
        cross_grad = 0;
        if(cross_grad !=0) cross_grad = 0;
        if(cross_grad < edge_threshold) 
        
    
    if((i == row) || (j == col))
    
    if((i > 0) && (i < (nrow-1)) && (j > 0) && (j < (ncol-1)) &&

    dist = (int)(sqrt(double)((seedrow - i)*(seedrow - i) +
                (seedcol - j)*(seedcol - j)));
    if (dist <= distance)
    
    if(lap_image_buffer[1][j] < edge_threshold)
    
    if(gap_filler(i,j) == 0)
    
    else /* point is actually one dista 
    from an edge pixel*/
    
    
    /* first mark the pixels visited 
    output_buffer[1][j] = label;
    pixels++;
    third_image_buffer[1][j] = 1;
    if(strcmp("y",ikonas_disp) == 0)
        lwr[j,i,&value];
    
    else /* distance exceeded */
    
    
    */
    
}
output_buffer[i][j] = label;
            pixels++;
            third_image_buffer[i][j] = 1;
            if(strcmp("y", ikonas_disp) == 0)
             lwr(j, i, value);
        } /* acceptable pixel */
    } /* cross_grad acceptable */
    num_call++;
} /* end of grow_region */

/**************************** STORE_PIXEL & GET PIXEL ******************************/

store_pixel(rrow,ccol)
int rrow;
int ccol;
{
        num_call--;
        / * printf(" PPPP "); */
        pixel_stack[stack_length].row = rrow;
        pixel_stack[stack_length].col = ccol;
        stack_length++;
        if(stack_length == PIXEL_STACK_SIZE) stack_length = 0;
        if(stack_length == current_element)
            printf("\n in segment : Stack Collision While region growing \n");

    get_pixel()
    {
        rownum = pixel_stack[current_element].row;
        colnum = pixel_stack[current_element].col;
        current_element++;
        if(current_element == PIXEL_STACK_SIZE) current_element = 0;
    } /*

    float squarez(n)
    int n;
    { float f;
      f = (float)(n);
      return(f*f);
    } /*

    min(x,y)
    int x,y;
    { /*

/**** gap_filler checks if the pixel can be considered as the part of the edge. This is done by checking if the 8-connected neighbour has any edge pixel. If yes, it is considered a neighbour of edge and coordinates of edge pixel is returned in gaprow,gapcol. The pixel is not further grown. This procedure fills in 2-pixel gap without undergoing the pain of gap-filling using contour tracing which is computationally expensive. See Peter Allen's thesis for performance of gap-filler due to Naratia & Babu. He observes that filling of at most 2-pixel gaps is acceptable in most cases.****/ gap_filler(rrow,ccol) int rrow,ccol;
{
    int i,j;
    for(i=rrow-1;i<=(rrow+1);i++)
        for(j=ccol-1;j<=(ccol+1);j++)
            if(lap_image_buffer[i][j] > edge_threshold)
            { /* edge pixel in 8-connected hdbd is found */
                gaprow = i;
                gapcol = j;
                return(i);
            }
   /* no edge pixel in 8-connected neighbourhood is found */
   return(0);
}

smooth_region(label_value)
int label_value;
{
    int i,j;
    /* for(i=0;i<nrow;i++)
        for(j=0;j<ncol;j++)
            if(output_buffer[i][j] == label_value)
            { */
            /*
        */
        min(x,y)
    int x,y;
    { /*
/*************** GROW_REGION ***************
/* following is the modified recursive call for the segmentation, now a pixel
  is examined before routine is called rather than calling
  the routine and then checking the pixel type. this was done to
  optimize the stack-space which overflows in the case of large objects. 
  even after this modification stack overflows if the object is
  therefore recursive region growing is not possible if the object is */

grow_region(row, col)
int row;
int col;
{
  int 1,j;  /* loop control variables */
  int dist;
  int cross_grad;

  if(strcmp("y",ikonasDisp) == 0)
    lwr[row][col][value];
  output_buffer[row][col] = label;

  if(max_pixel_region < range_image_buffer[row][col])
    max_pixel_region = range_image_buffer[row][col];
  if(min_pixel_region > range_image_buffer[row][col])
    min_pixel_region = range_image_buffer[row][col];

  pixels++;

  /* if((row < (nrow -1)) && (col < (ncol -1)))
   cross_grad = sqrt((double)
     (squarezz(range_image_buffer[row][col] -
       range_image_buffer[row+1][col+1] -
       range_image_buffer[row+1][col] -
       range_image_buffer[row][col+1];
   */

  else /*
   cross_grad = 0;
  if(cross_grad != 0) cross_grad *= 0;
  if(cross_grad < edge_threshold)
    {
      for(1=row-1; 1<row+2; 1++)
        for(1=col-1; 1<col+2; 1++)
        if((1 != row) || (1 != col))
          if((1 > 0) && (1 < (nrow-1)) && (j > 0) && (j < (ncol-1)))
            lwr[1][1][value];
    }
  
  if(lap_image_buffer[1][1] < edge_thres)
    output_buffer[1][1] = label;
  if(gap_filler(1) == 0)
    third_image_buffer[1][1] = back_thr;
  if(gap_filler(1) == 0)
    third_image_buffer[1][1] = back_thr;
  if(num_call++;
  if(num_call >= MAX_NUM_CALL)
    store_pixel(1,1);
    else
      grow_region(1,1);
  }/* point is actually one dista
  from an edge pixel*/

  /* first mark the pixels visted
  output_buffer[1][1] = label;
  pixels++;
  third_image_buffer[1][1] = 1;
  if(strcmp("y",ikonasDisp) == 0)
    lwr[1][1][value];
  /* now mark the edge pixel visit
  output_buffer[1][1] = pixels++;
  third_image_buffer[1][1] = pixels++;
  if(strcmp("y",ikonasDisp) == 0)
    lwr[1][1][value];
  }/* distance exceeded */
}
output_buffer[i][j] = label;
pixels++;
third_image_buffer[i][j] = 1;
if(strcmp("y",ikonas_disp) == 0)
    lwr(j, i, &value);
}
/* acceptable pixel */
} /* cross grad acceptable */
num_call--; /* end of grow_region */

/*********************** STORE_PIXEL & GET PIXEL **********************

store_pixel(rrow, ccol)
int rrow;
int ccol;
{
    num_call--; /*
    printf("PPPP *"); */
    pixel_stack[stack_length].row = rrow;
    pixel_stack[stack_length].col = ccol;
    stack_length++;
    if(stack_length == PIXEL_STACK_SIZE) stack_length = 0;
    if(stack_length == current_element)
        printf("\nsegment : Stack Collision While region growing \n
");
get_pixel()
{
    rownum = pixel_stack[current_element].row;
    colnum = pixel_stack[current_element].col;
    current_element++;
    if(current_element == PIXEL_STACK_SIZE) current_element = 0;
}
float square2z(n)
int n;
{ float f;
  f = (float)(n);
  return(f*f); }

/**** gap filler checks if the pixel can be considered as the part the edge. This is done by checking if the 8-connected neigbou has any edge pixel. If yes, it is considered a neighbour of edg and coordinates of edge pixel is returned in gaprow, gapcol. The pixel is not further grown. This procedure fills in 2-pixel gap without undergoing the pain of gap-filling using contour track which is computationally expensive. See peter allen's thesis fo performance of gap-filler (due to Narati & Babu). He observes that filling of at most 2-pixel gaps is acceptable in most case

*****/

gap_filler(rrow, ccol)
int rrow, ccol;
{
    int i, j;
    for(i=rrow-1;i<=(rrow+1);i++)
        for(j=ccol-1;j<=(ccol+1);j++)
            if(lap_image_buffer[i][j] >= edge_threshold)
                /* edge pixel in 8-connected hbd is found */
                gaprow = i;
                gapcol = j;
                return(1);
            }
    /* no edge pixel in 8-connected neighbourhood is found */
    return(0);
}

smooth_region(label_value)
int label_value;
{
    int i, j;
    /* for(i=0;i<nrow;i++)
        for(j=0;i<ncol;i++)
            { if(output_buffer[i][j] == label_value)
                }
    */
    min(x, y)
int x, y;
{
if (x<y) return(x);
    else    return(y);
}

char accept_region(lab)
int lab;
{  
    int i,j,k;
    if(pixels < MIN_ACCEPTABLE) return('n');
    else return('y');
}

determine_support(lab)
int lab;
{  
    int i,j,k;
    /* determine boundary points for all rows first */
    for(i=0;i<row;i++)
        for(j=0;j<ncol;j++)
            {  
                if(output_buffer[l][j] != lab && output_buffer[l][j+1] ==
                   det_med_support(lab,1,j+1))
                    else if(output_buffer[l][j] == lab && output_buffer[l][j+1] ==
                       det_med_support(lab,1,j+1))
            
    /* determine boundary points for all columns */
    for(j=0;j<nrow;j++)
        for(i=0;i<nrow;i++)
            {  
                if(output_buffer[i][j] != lab && output_buffer[i+1][j] ==
                   det_med_support(lab,i+1,j))
                    else if(output_buffer[i][j] == lab && output_buffer[i+1][j] ==
                       det_med_support(lab,i+1,j))
            }

    det_med_support(lab, row, col)
int lab;

int row;
int col;
{  
    int i,j,k;
    int count = 0;
    int num;
    int wsize = 3;
    /* printf("det_med_support for lab = %d row %d col %d called ",lab,row,col);
    for(i=0;i<256;i++) vector[i] = 0;
    for(i=(row-wsize);i<=(row+wsize);i++)
        for(j=(col-wsize);j<=(col+wsize);j++)
            if((i>=0) && (i<nrow) && (j>=0) && (j<ncol))
                if(output_buffer[i][j] != lab)
                    {  
                        count++;
                        vector[range_image_buffer[i][j]]++;
                    }
    num = count/2;  This is for median */
    num = 1;     /* This picks up the smallest depth */
    /* printf("num = %d count = %d",num,count);*/
    if(num == 0)
        {  
            support_image_buffer[0][row][col] = range_image_buffer[row][col];
            support_image_buffer[i][row][col] = lab;
        }
    else
        {  
            for(i=0;i<row;i++)
                {  
                    num=num-vector[i];
                    if(vector[i] > 0) j=1;
                    support_image_buffer[0][row][col] = j;
                    support_image_buffer[l][row][col] = lab;
                }
        }
}

finished()
{  
    int i,j,k;
    int finish = 1;
for(i=1;i<row;i++)
    for(j=1;j<col;j++)
        if((third_image_buffer[i][j] == 0) &&
           (range_image_buffer[i][j] > back_threshold)) finish = 0;

return(finish);

/******************************************************************************
****** FIND_SEED_REGION *****************************/

find_seed_region()
{
    int i,j,k,m,n;
    int ok_point;
    /* Boolean */
    int average_depth; /* average depth of the 3x3 window */
    int account;

    max = -1000;
    for(i=2;i<row-1;i++)
        for(j=2;j<col-1;j++)
            if((range_image_buffer[i][j] > back_threshold) &&
               (third_image_buffer[i][j] == 0))
                {
                    account = 0;
                    average_depth = 0;
                    ok_point = 1;
                    if(region_count == 1) /* do it only the first time */
                        {
                            for(m=1;m<i+2;m++)
                                for(n=1;n<j+2;n++)
                                    if((lap_image_buffer[m][n] < edge_threshold) &&
                                       (range_image_buffer[m][n] > back_threshold) &&
                                       (third_image_buffer[m][n] == 0))
                                        {
                                            account++;
                                            average_depth += range_image_buffer[m][n];
                                            
                                        } /* if(account > 0) average_image[1][j] = average_depth/account */
                            else ok_point = 0;
                        }
                    if(ok_point == 1)
                        {
                            if((lap_image_buffer[i][j] < edge_threshold) &&
                               (range_image_buffer[i][j] > back_threshold) &&
                               (third_image_buffer[i][j] == 0))
                                {
                                    (average_image[1][j] > max))
                                        {
                                            max = average_image[1][j];
                                            row_max = i;
                                            column_max = j;
                                        }
                                    
                                } /* end of find_seed_region */

/******************************************************************************
****** SEED_REGION *****************************/

seed_region()
{
    int i,j,done,end;
    int m,n;
    int account;
    int cross_grad;

    done = 0;
    end = 0;

    while((done == 0) && (end == 0))
        {
            if((third_image_buffer[nextrow][nextcol] == 0) &&
               (lap_image_buffer[nextrow][nextcol] < edge_threshold) &&
               (range_image_buffer[nextrow][nextcol] > back_threshold))
                account = 0;
                for(m=nextrow-1;m<nextrow+2;m++)
                    for(n=nextcol-1;n<nextcol+2;n++)
                        /* cross_grad = sqrt((double)
                           (squares(range_image_buffer[m][n] -
                               range_image_buffer[m+1][n+1]) +
                           squarezz(range_image_buffer[m][n]-
                               range_image_buffer[m+1][n])); */
                    cross_grad = 0;
            if((lap_image_buffer[m][n] < edge_threshold) &&
               (range_image_buffer[m][n] > back_threshold) &&
               (third_image_buffer[m][n] == 0))
                account++;
            
        }

if(account > 0)
segment.c  Tue Apr 10 18:31:58 1988  19

    { done = 1;
      max = range_image_buffer[nextrow][nextcol];
    }
    row_max = nextrow;
    column_max = nextcol;
    nextcol++;
    if(nextcol > (ncol-2))
      { nextrow++;
        nextcol = 0;
        if(nextrow > (nrow - 2))
          { end = 1;
            max = -1000;
          }
      }
}
} /* end of seed_region */

/**************************** FILL THE GAPS ***************************
fill_gaps()
{
  int i, j, k, l;
}
}/*fill_gaps */

/**************************** CONTOUR TRACING **************************
//** Following values are range-image scanner dependent. These correspond to the SRI test database **/

#define VERTICAL 1.531
#define HORIZONTAL 1.531
#define HEIGHT 0.245
#define BACK_DEPTH 5.0

/* Following are for RCA range image scanner, measured in mm/pixel */

#define VERTICAL 1.8796
#define HORIZONTAL 1.8796
#define HEIGHT 0.0254

#define VERTICAL 1.5240 /* X, varying rows 0.060 */
#define HORIZONTAL 1.8796 /* Y, varying cols 0.74 */
#define HEIGHT 0.0254 /* Z, depth 0.001 */

/* Range image scanner dependent */
#define BACK_DEPTH 320 /* background depth */
#define TRESH 0 /* original value = 5 */
#define BORDER 10

/* Big array declarations are moved out of the main program body to avoid run time memory fault. */
*** Important to have the image read in as unsigned char so that pix values are from 0 to 255 and not from -127 to +127 ***

int picture[PICDIM_X][PICDIM_Y];
int sup_points[PICDIM_X][PICDIM_Y];

double point[POINTS][3], support[POINTS][3], T[4][4], newpoint[3], ve
*/

*** additional declarations for adding hidden points */

double vect[3], addpoint[POINTS][3], vectpoint[3]; /* horizontal double sup_vect[3], side_add_points[POINTS][3], sup_point[3]; */

pmpic *pm1; /* Input file is in PM-format */
int sizey; /* # of rows in input picture */
int sizex; /* # of columns in input picture */

main (argc, argv)
int argc;
char *argv[];
{
    int i, j, k, k2, k3, m, n, grid;
double x1, y1, z1, x2, y2, z2, x3, y3, z3, a, b, c;
double sqrt(), sq(), distance;
char g[10], h[10], sup[10], orig[10];
int sup_count = 0;
int cause;
FILE *orig_file,*add_file;
if ((argc < 3) || (argc > 4))
    printf("usage: rca_calib grid_density inpic [sup_prc] \n");
else
    grid = atoi(argv[1]);
    read_picture(argv[2], picture);
    if (argc == 4) read_picture(argv[3], sup_points);
make_matrix(T);
/* compute the supporting plane from three points */
x1 = 15.0;
y1 = 5.0;
z1 = BACK_DEPTH;
/* z1 = (double)picture((int)x1[(int)y1];*/
  fprintf(stderr, "z1 = \f", z1);
x2 = 230.0;
y2 = 5.0;
z2 = BACK_DEPTH;
/* z2 = (double)picture((int)x2[(int)y2];*/
  fprintf(stderr, "z2 = \f", z2);
x3 = 30.0;
y3 = 165.0;
z3 = BACK_DEPTH;
/* z3 = (double)picture((int)x3[(int)y3];*/
  fprintf(stderr, "z3 = \f", z3);
c = ((x3 - x1)*(x3*y2 - x2*y3) - (x3 - x2)*(x3*y1 - x1*y3))/(x3*x2 - x2*x3)*(x3*y1 - x1*y3) - (x3*z1 - x1*z3)*(x3*y2 - x2);
b = (-c * (x3*z1 - x1*z3) - (x3 - x1))/(x3*y1 - x1*y3);

a = (-1 - c*x1 - b*y1)/x1;
fprintf(stderr,"a = \f b = \f c = \f\n",a,b,c);
fprintf(stderr,"Add hidden points horizontally (y/n)?");
scanf("%s", h);
if(argc == 4) /* supporting points file present */
    fprintf(stderr,"Add hidden points vertically (y/n)?");
    scanf("%s", sup);
} else
    fprintf(sup,"%s","no")

k = 0;
k2 = 0;
k3 = 0;
for (); j < size - BORDER; j = j + grid
    { for (i = BORDER; i < size - BORDER; i = i + grid)
        { vector[0] = j * HORIZONTAL;
            vector[1] = (239 - i) * VERTICAL;
            matrix_mult(vector, T, newpoint);
            if(h[0] == 'y')
                { vector[0] = vector[0];
                    vector[1] = vector[1];
                    vector[2] = ((-1.0-a*b-j)/c)*HEIGHT;
                    matrix_mult(vector, T, vectpoint);
                }
            distance = (a + 1 + b + j + c * (double)(picture[1][j]) + 1)
                      sqrt(sq(a) + sq(b) + sq(c));
            fprintf(stderr,"%d j = %d d = \f d distance = \f \n",i,j,distance,
            if(distance < -TRESH)
                { point[k][0] = newpoint[0];
                    point[k][1] = newpoint[1];
                    point[k][2] = newpoint[2];
                    if(h[0] == 'y')
                        { addpoint[k][0] = vectpoint[0];
                            addpoint[k][1] = vectpoint[1];
                            addpoint[k][2] = vectpoint[2];
                        }
                    k = k + 1;
else if (distance < TRESH)
    { 
    support[k2][0] = newpoint[0];
    support[k2][1] = newpoint[1];
    support[k2][2] = newpoint[2];
    k2 = k2 + 1;
    }
else () // points is in shadow */
}

if (sup[0] == 'y')
    for (j = BORDER; j < sizey - BORDER; j++)
    { 
    for (i = BORDER; i < sizex - BORDER; i++)
    { 
    if(sup_points[i][j] > 0)
    { 
    sup_count++;
    cause = 0;
    for(m=i-grid/2;m=<i+grid/2;m++)
    for(n=j-grid/2;n=<j+grid/2;n++)
    if((sup_points[m][n] == 255)) cause++;
    if(cause == 0)
    { 
    for(m=sup_points[1][j]; m < picture[1][j]; m= 
    { 
    sup_vect[0] = j * HORIZONTAL;
    sup_vect[1] = (239 - i) * VERTICAL;
    sup_vect[2] = m*HEIGHT;
    matrix_mult(sup_vect, T, sup_point);
    side_add_points[k3][0] = sup_point[0];
    side_add_points[k3][1] = sup_point[1];
    side_add_points[k3][2] = sup_point[2];
    k3++;
    }
    sup_points[i][j] = 255;
    }
    }
    }

fprintf(stderr,"Remove supporting surface (y/n)? ");
scanf("%s", g);

fprintf(stderr,"Add original points ?(y/n) ");
scanf("%s", orig);
result[2] = matrix[2][0] * vector[0] +
           matrix[2][1] * vector[1] +

/***************************************************************
* function for making a T matrix
***************************************************************/
make_matrix(TR)
    double TR[4][4];
    { double rx, ry, rz, x, y, z;
      x = 80.0;
      y = -180.0;
      z = 0;
      rx = 90.0;
      ry = 45.0;
      rz = -15.0;
      rx = rx * PI/180;
      ry = ry * PI/180;
      rz = rz * PI/180;
      /* homogenous transformation : Euler angles */
      TR[0][0] = cos(rx)*cos(ry)*cos(rz) - sin(rx)*sin(rz);
      TR[0][1] = -cos(rx)*cos(ry)*sin(rz) + sin(rx)*cos(rz);
      TR[0][2] = cos(rx)*sin(ry);
      TR[0][3] = x;
      TR[1][0] = sin(rx)*cos(ry)*cos(rz) + cos(rx)*sin(rz);
      TR[1][1] = -sin(rx)*cos(ry)*sin(rz) + cos(rx)*cos(rz);
      TR[1][2] = sin(ry)*sin(rz);
      TR[1][3] = y;
      TR[2][0] = -sin(ry)*cos(rz);
      TR[2][1] = sin(ry)*sin(rz);
      TR[2][2] = cos(ry);
      TR[2][3] = z;
      TR[3][0] = 0;
      TR[3][1] = 0;
      TR[3][2] = 0;
      TR[3][3] = 1;
    }

double sq(x)
    double x;
    { return(x*x); }

read_picture(filename,buffer)
    char filename[50];
    int buffer[PICDIM_X][PICDIM_Y];
    { int i,j;
      FILE *ifs;
      unsigned char *pm_point;
      short int *pm_point;
      /* open input pm file */
      if ((ifs = fopen(filename,"r")) == NULL)
        { printf("file open error :\n",filename);
          exit(0);
        }
      /* read input file into the pm file buffer */
      if (pm = pm_read(ifs,0)) == NULL)
        { printf("error in reading the pm file \n",filename);
          exit(0);
        }
      sizey = (pm->pm_row); /* # of rows */
      sizex = (pm->pm_ncol); /* # of columns */
      fprintf(stderr,"rows : %d ; columns : %d\n",sizex,sizey);
      if(pm->pm_form == PM_C)
        { pm_point = (unsigned char *) pm->pm_image;
          for(i=0;i<sizex;i++)
            for(j=0;j<sizey;j++)
              { buffer[i][j] = *pm_point;
               pm_point++;
             }
        } else if(pm->pm_form == PM_S)
          { ..
pms_point = (short int *) pm1->pm_image;
for(i=0;i<sizex;i++)
    for(j=0;j<sizey;j++)
    {
        buffer[i][j] = *(pms_point);
        pms_point++;
    }
else
    {
        printf("Image file in unrecognized format. Exiting.\n");
        exit(0);
    }
/* end of read picture */
classify.c Wed Mar 9 16:00:45 1988

/*****************************/

Program to classify the superquadric model into one of the four
categories:

- flat,
- box,
- roll,
- IPP

Format of the input file is as output from the rec9.out program.
Run as:
% classify fit*.originalpoints fit*.addedpoints

0.00 <= e1,e2 <= 1.00

/***************************************************************************/

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

#define TIM 3 /* to implement << or >> */
#define FLAT 1
#define BOX 2
#define NULL 3
#define IPP 4

double a1[3], a2[3], a3[3]; /* to store a1, a2 and a3 */
double e1[3], e2[3]; /* to store e1 and e2 */
double measured_goodness1, goodness; /* goodness of fit */
double measured_goodness2;

char dummy[100], orig[50], add[50];

int type1, type2; /* type determined by looking at the a1
and a3 values and e1 and e2 values of
fitted model */

double dum;

double l1_box = 10.0, l1_roll = 10.0, l2_roll = 15.0;

FILE *ins, *addfile, *outfile;

main(argc, argv)
int argc;
char *argv[];
{
  int i, j, k, l;
  int good1, good2;

strcpy(orig, argv[1]);
strcpy(add, argv[2]);

if(!ins = fopen(argv[1], "r")) exit(0);
{
  printf("classify : File open error : %s\n", argv[1]);
  exit(0);
}

if(!addfile = fopen(argv[2], "r")) exit(0);
{
  printf("classify : File open error : %s\n", argv[2]);
  exit(0);
}

if(!outfile = fopen(argv[3], "w")) exit(0);
{
  printf("classify : File open error : %s\n", argv[3]);
  exit(0);
}

fscanf(ins, "%s", dummy);

fscanf(ins, "%f %f %f", &a1[1], &a2[1], &a3[1]);

fscanf(ins, "%f %f %f", &a1[2], &a2[2], &a3[2]);

fscanf(ins, "%f %f %f", &e1[1], &e2[1]);

fscanf(ins, "%f %f %f %f", &dum, &dum, &dum, &dum);

fscanf(ins, "%s", dummy);

fscanf(ins, "%f,%f,%f", &measured_goodness1);

printf("%f,%f,%f", a1[1], a2[1], a3[1]);

printf("%f,%f,%f", e1[1], e2[1]);

printf("%f,%f", measured_goodness1);

fscanf(addfile, "%s", dummy);

fscanf(addfile, "%f %f %f", &a1[2], &a2[2], &a3[2]);

fscanf(addfile, "%f %f %f", &a1[1], &a2[1], &a3[2]);

fscanf(addfile, "%f %f", &e1[2], &e2[2]);

fscanf(addfile, "%f %f", &e1[2], &e2[2]);
```c
int num;
{
    int i, j, k;
    if((TIM*a3[num] < a1[num]) && (TIM*a3[num] < a2[num]) &&
        (a2[num] < 0.5) && (a1[num] < 0.5))
        return(FLAT);
    else if((a1[num]*TIM < a3[num]) || (a2[num]*TIM < a3[num]) &&
        (el[num] < 0.5) && (e2[num] < 0.5))
        return(FLAT);
    else if((a1[num] > k_box) && (a2[num] > k_box) && (a3[num] > k_box) &&
        (el[num] < 0.5) && (e2[num] < 0.5))
        return(BOX);
    else if((a1[num] > k1_roll) && (a2[num] > k1_roll) && (a3[num] > k2) &&
        (el[num] < 0.5) && (e2[num] > 0.5))
        return(ROLL);
    else
        return(IFF);
}

classification(num, type)
int num;
int type;
{
    char str[50];
    char string[10];
    switch(type)
    {
    case FLAT : sprintf(string, "%s", "Flat");break;
    case BOX : sprintf(string, "%s", "Box");break;
    case ROLL : sprintf(string, "%s", "Roll");break;
    case IFF : sprintf(string, "%s", "IFF");break;
    }
    if(num == 1)
    {
        printf("Points not added in the fit\n");
        printf("Object classified as %s in", string);
        fprintf(outfile, "%s", string);
        fprintf(stderr, "CP", orig, " fit final ");
        system(str);
    }
}
```
else if(num == 2)
{
    printf("Points added to obtain the fit.\n");
    printf("Object classified as %s \n",string);
    fprintf(outfile, "%s", string);
    sprintf(str, "%s %s %s %.0f%" fit.final");
    system(str);
}

else
{
    printf("Object Classified as %s \n", string);
    fprintf(outfile, "%s", string);
    sprintf(str, "%s %s %s %.0f%" orig, fit.final);
    system(str);
}

volume_criterion()
{
    double vol1, vol2;
    vol1 = a1[1]*a2[1]*a3[1];
    vol2 = a1[2]*a2[2]*a3[2];
    printf("Following volume criterion \n");
    if(vol1 <= vol2)
    classification(2,type2);
    else
    classification(1,type1);
}

good_enough(goodn)
double goodn;
{
    if(goodn < goodness)
    return(1);
    else
    return(0);
}
double sq(x)

double x;
{
    return(x*x);
}

read_picture(filename,buffer)
char filename[50];
int buffer[PICDIM_X][PICDIM_Y];
{
    int i,j;
    FILE *fins;
    unsigned char *pm_point;
    short int *pm_point;
    /* open input pm file */
    if ((fins = fopen(filename,"r")) == NULL) {
        printf("file open error: \n",filename);
        exit(0);
    }
    /* read input file into the pm pic buffer */
    if(pml = pm_read(fins,0)) == NULL) {
        printf("error in reading the pm file \n",filename);
        exit(0);
    }
    sizev = (pml->pm_row);
    /* # of rows */
    sizey = (pml->pm_col);
    /* # of columns */
    fprintf(stderr,"rows : \%d ; columns : \%d\n",sizev,sizey);
    if(pml->pm_form == PM_C) {
        pm_point = (unsigned char *) pml->pm_image;
        for(i=0;i<sizev;i++)
            for(j=0;j<sizey;j++)
                buffer[i][j] = pm_point;
        pm_point++;
    }
    else if(pml->pm_form == PM_S) {
        // code for PM_S
    }
}
pms_point = (short int *) pm1->pm_image;
for(i=0;i<sizex;i++)
    for(j=0;j<sizey;j++)
    {
        buffer[i][j] = *(pms_point);
        pms_point++;
    }
else
{
    printf("Image file in unrecognized format. Exiting.\n");
    exit(0);
}

/*/ end of read picture */
Program to classify the superquadric model into one of the four broad categories:
flat,
box,
roll,
IPP.

Format of the input file is as output from the rec9.out program.
Run as:
% classify fit_.originalpoints fit_.addedpoints

0.00 <= e1,e2 <= 1.00

#include <stdio.h>
#include <math.h>
#define TIM 3      /* to implement << or >> */
#define FLAT 1
#define BOX 2
#define NULL 3
#define IPP 4

double a1[3],a2[3],a3[3];    /* to store a1, a2 and a3 */
double e1[3],e2[3];           /* to store e1 and e2 */
double measured_goodness1,goodness;  /* goodness of fit */
double measured_goodness2;

char dummy[100],orig[50],add[50];

int type1,type2;      /* type determined by looking at the a1 and a3 values and e1 and e2 values of fitted model */

double dum;

double l1_box = 10.0 ,
l1_roll=10.0 ,
l2_roll=15.0 ;

FILE *ifs,*addfile,*outfile;

main(argc,argv)
int argc;
char *argv[];
{
int i,j,k,l;
int good1,good2;

strcpy(orig,argv[1]);
strcpy(add,argv[2]);

if((ifs = fopen(argv[1],"r")) == NULL)
{
printf("classify : File open error : %s\n",argv[1]);
exit(0);
}

if((addfile = fopen(argv[2],"r")) == NULL)
{
printf("classify : File open error : %s\n",argv[2]);
exit(0);
}

if((outfile = fopen(argv[3],"w")) == NULL)
{
printf("classify : File open error : %s\n",argv[3]);
exit(0);
}

fscanf(ifs,"%s",dummy);

fscanf(ifs,"%f %f %f",&a1[1],&a2[1],&a3[1]);

fscanf(ifs,"%f %f %f",&a1[1],&a2[1],&a3[1]);

fscanf(ifs,"%f %f",&e1[1],&e2[1]);

fscanf(ifs,"%f %f",&e1[1],&e2[1]);

fscanf(ifs,"%f %f",&measured_goodness1);

printf("%f,%f,%f\n",a1[1],a2[1],a3[1]);

printf("%f,%f,%f\n",e1[1],e2[1]);

printf("%f\n",measured_goodness1);

fscanf(addfile,"%s",dummy);

fscanf(addfile,"%f %f %f",&a1[2],&a2[2],&a3[2]);

fscanf(addfile,"%f %f %f",&a1[2],&a2[2],&a3[2]);

fscanf(addfile,"%f %f",&e1[2],&e2[2]);

fscanf(addfile,"%f %f",&e1[2],&e2[2]);
fscanf(addfile,"%d %d %d", &dum,&dum,&dum);
fscanf(addfile,"%s", dummy);
fscanf(addfile,"%f", &measured_goodness2);
printf("%f,%f,%f\n", a1[2], a2[2], a3[2]);
printf("%f,%f,%f\n", e1[2], e2[2]);
printf("%f\n", measured_goodness2);
/* Read in the Threshold values */
printf("K for Box : ");
scanf("%f", &k_box);
printf("K1 and K2 for Roll : ");
scanf("%f %f", &k1_roll, &k2_roll);
printf("Goodness of fit measure : ");
scanf("%f", &goodness);
/* FIRST classify the object according to the a1,a2,a3,e1,e2 values*/
type1 = classified(1);
type2 = classified(2);
good1 = good_enough(measured_goodness2);
good2 = good_enough(measured_goodness2);
if((good1 == 1) && (good2 == 0))
  /* first fit is better than second fit */
  classification(1, type1);
else if((good1 == 0) && (good2 == 1))
  /* second fit is better than the first fit */
  classification(2, type2);
else if((good1 == 1) && (good2 == 1))
  /* both the fits are acceptable */
  volume_criterion();
else
  /* Both the fits are unacceptable */
  classification(3, IPP);
}
classified(num)
else if((num == 2))
{  
  printf("Points added to obtain the fit.\n");  
  printf("Object classified as %s \n",string);  
  fprintf(outfile,"%s",string);  
  sprintf(str,"%s %s %s","cp ",add," fit.final ");  
  system(str);  
}

else
{
  printf("Object Classified as %s \n",string);  
  fprintf(outfile,"%s",string);  
  sprintf(str,"%s %s %s","cp ",orig," fit.final ");  
  system(str);
}

volume_criterion()
{
  double vol1,vol2;
  vol2 = a[2]a[3];

  printf("Following volume criterion \n");
  if(vol1 <= vol2)
    classification(2,type2);
  else
    classification(1,type1);
}

good_enough(goodn)
double goodn;
{
  if(goodn < goodness)
    return(1);
  else
    return(0);  
}
Program to compute the first and second order derivatives of the image and finally the Gaussian and Mean curvature at all the image points. The program outputs the sign map of Gaussian and Mean curvature.

Mar 17, 1988 Interactive processing. Can handle PM_S and PM_C images. Outputs only PM_C images.


******************************************************************************

#include <stdio.h>
#include <math.h>
#include <local.pl.h>
#include <lib.h>
#include "~/usr/users/loj/advanced/spline_include.h"

#define SCALE 1.978  /* should be same as xscale and yscale as mm/px for Gus' scanner */
#define ZSCALE 1.500  /* zscale in the digitized image */
#define REGION_SIZE 1000
#define NSYS 25
#define NDATA 1000

struct region_type {
    int order;  /* order of the surface fitted */
    int surf_type;  /* classification of the region surface */
    float fit_error;  /* surface fit error */
    int label;  /* identifying label of the region */
    int size;  /* # of pixels in the region */
    float normal[3];  /* unit normal to the surface */
    float a[REGION_SIZE];
} regions[REGION_SIZE];

struct _image {
    float xu, xv, xu, xv, xuv, xuv;
    float fit_error;
    float Gauss, mean;
    int label;
    int th_mean;
    int th_gauss;
    float q;
    float sqrtq;

    float cosphl;
    int image, pm[BUF_SIZE][BUF_SIZE];  /* for parameters of pixels */
    float ri[BUF_SIZE][BUF_SIZE];  /* range image */
    int buffer[BUF_SIZE][BUF_SIZE];
    int line[BUF_SIZE];
    float linew[BUF_SIZE];
    int offx, offy;

    struct maxval{ /* stores the maximum and minimum of value */
        float fit_error;
        float Gauss;
        float mean;
        float q;
        float sqrtq;
        float cosphl;
        float depth;
        int label;
        int th_gaus;
        int th_mean;
        jmaxv, minv;
    } maxv;

    float x[1024];
    float y[1024];
    float value;
    float xmn, xmx, ymn, ymx;
    int xpt;
    int ypt;

    /* following parameters are returned by the least square fitting */
    double a[30], a21, a22, a03, a20, a11, a02, a10, a01, a00;
    double xux, xuy, xuv, xuv;
    double fit_error;  /* the surface fit error */
    int x_order;  /* the order of the surface fitted in the nbd */
    int y_order;  /* the order of the surface fitted in the nbd */

    /* following are global to this file */
    int csz, rzsz;
    int offset1;
    int approx;
    int s_order;  /* fitted surface order */
    double lsgwr[1];
    FILE *dumpfile;
    int todump;
char name_ext[16][20] = {
    "", /* name-extensions to be appe
    ".fit-error", /* to the input file-name to
    ".quad-var", /* output-file-name
    ".zero-mean",
    ".zero-ccf",
    ".coaf",
    ".sign-mean",
    ".sign-ccf",
    ".mag-pcd",
    ".pda",
    ".mgc",
    ".mmc",
    ".n-critical",
    ".critical",
    ".region"};

int mask_u[5][5] = {
    {-1,-6,-10,-6,-1},
    {-2,-20,-52,-20,-2},
    {0,0,0,0,0},
    {2,20,52,20,2},
    {1,6,10,6,1}};

int mask_v[5][5] = {
    {-1,-2,0,2,1},
    {-6,-20,0,20,6},
    {-10,-52,0,52,10},
    {-6,-20,0,20,6},
    {-1,-2,0,2,1}};

int mask_uu[5][5] = {
    {1,6,10,6,1},
    {0,8,32,8,0},
    {-2,-28,-84,-28,-2},
    {0,8,32,8,0},
    {1,6,10,6,1}};

int mask_vv[5][5] = {
    {1,0,-2,0,1},
    {6,8,-28,8,6},
    {10,32,-84,32,10},
    {6,8,-28,8,6},
    {1,0,-2,0,1}};

int mask_uv[5][5] = {
    {1,2,0,-2,-1},
    {2,12,0,-12,-2},
    {0,0,0,0,0},
    {-2,-12,0,12,-2},
    {-1,-2,0,2,1}};

int op_u[3][3] = { {-1,-4,-1},
    {0,0,0},
    {1,4,1}};

int op_v[3][3] = {
    {-1,0,1},
    {-4,0,4},
    {-1,0,1}};

int op_uv[3][3] = {
    {1,4,1},
    {-2,-8,-2},
    {1,4,1}};

int op_vw[3][3] = {
    {1,-2,1},
    {4,-8,4},
    {1,-2,1}};

int op_uv[3][3] = {
    {1,0,-1},
    {0,0,0},
    {-1,0,1}};

float weight_u = 288.0;
float weight_v = 288.0;
float weight_uw = 144.0;
float weight_vw = 144.0;
float weight_uv = 96.0;

float convolv();
float convolv1();
float absz();

float div_u = 12.0;
float div_v = 12.0;
float div_uw = 6.0;
float div_vw = 6.0;
float div_uv = 4.0;

float max_mean = -1000.0;
float max_gauss = -1000.0;
float min_mean = 1000.0;
float min_gauss = 1000.0;

main(argc, argv)
int argc;
char *argv[];
{
    int i,j,k,l,m,n; /* Loop control variables */
    ppmic *pml,*pm2;
    FILE *infile,*outfile;
    int row,col;
    char *cmt;
    unsigned char *pm_point uchar;
    short int *pm_point short;
    int option;
    float gauss,mean;
    char temp[40];
    int offset;
    ppmic *ppmic[16];
    unsigned char *uchar point[16];
    int temp value;
    int mean val,gauss val;
    int bool[17];
    FILE *fp[16];
    float th mean,th gauss;
    float t1,t2,t3;
    float q;
    char c;
    char smooth;
    int back threshold;
    int to smooth;
    int to _print files; /* =1, if all outputs desired in a file */
    char string[30];
    float sf;
    ppmic *pm;  
    FILE *fp;
    double scale,zscale; /* scale = yscale = xscale; zscale is for d 
    u_int image format;  
    int subs;

    if(argc > 3) || (argc < 2)
    {
        printf("Usage : spline input filename output_diagonistics_file\ 
        exit(0);
    }
    if(infile=fopen(argv[1],"r")) == NULL)
    {
        printf("Can’ t open %s\n",argv[1]);
        exit(0);
    }

todump = 0;
if(argc == 3)
{
    if((dumpfile=fopen(argv[2],"w")) == NULL)
    {
        printf("Can’ t open %s\n",argv[2]);
        exit(0);
    }
    todump = 1;
}

/* if(ikopen(NULL) == -1)
    printf("Can’ t open /KONUS \n");*/

init structure(); /* initializes the structures */
cmt = (char *pm_cmt(argc,argv));  /* get the command line */

if((pml = pm_read(infile,0)) == NULL)
    {
        printf("Error in reading PM-Format file : %s\n",argv[1]);
        exit(0);
    }

row = pml->pm_row;
col = pml->pm_ncol;
rsise = row;
size = col;
offx = offy = 0;

/* copy the input image into the image */

if(pml->pm_form == PM_C) /* one byte per pixel picture */
{
    pm_point uchar = (unsigned char *) pml->pm_image;
    image format = PM_C;
    zscale = zscale;  /* for Gus’ scanner */
    scale = scale;
    printf("Initializing buffer \n");
    for(i=0;i<row;i++)
        for(j=0;j<col;j++)
        {
            ri[i][j] = (float) *pm_point uchar; *(float)zscale/scal
            if(maxv.depth < ri[i][j]) maxv.depth = ri[j][j];
        }
if(minv.depth > r[i][j]) minv.depth = r[i][j];
    pm_point uchar++;  
}
}

else if(pml->pml_form == PM_S) /* short integer picture */
{
    pm_point_short = (short int *) pml->pml_image;
    image_format = PM_S;
    zscale = 1.00;
    scale = 76.20;
    printf("initializing buffer \n");
    for(i=0;i<row;++i)
        for(j=0;j<col;++j)
        {
            r[i][j] = ((float) *pm_point_short) * (float) (zscale / scale)
            if(maxv.depth < r[i][j]) maxv.depth = r[i][j];
            if(minv.depth > r[i][j]) minv.depth = r[i][j];
            pm_point_short++;  
        }
}
else fprintf(stderr,"unrecognized PM format in image");
exit(0);
}

/* background threshold is the background value in image as it is r 
and not in uniformly scaled (in X direction) image */

printf("Background Threshold = ");
scanf("%d", &back_threshold);
back_threshold = (back_threshold * zscale) / scale;
read_specs();  /* read specifications */

printf("Gaussian-smooth the picture ?(1 if yes) ");
scanf("%d", &to_smooth);

if(to_smooth == 1) gaussian(row, col);

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* 
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(to_smooth == 1) gaussian(i, j);  

if(to_smooth == 1) gaussian(row, col);
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(to_smooth == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

/* compute Gaussians  
for(i = 0; i < row; ++i)
    for(j = 0; j < col; ++j)
        if(p[i][j] == 1) gaussian(i, j);  

*/

/* Now compute the derivatives and Gaussian and Mean curvature at e 
image point
*/
/*
   printf("%s",col,offset1); row,col,offset1));
   printf("size : %d  csize : %d approx : %d\n",size,cszie,approx)
   compute_marray();
   for(i=offset1;i<row=offset1;i++)
   { for(j=offset1;j<col=offset1;j++)
     if(ri[i][j] > back_threshold)
     { if((approx == 1) || (approx == 2) || (approx == 3))
         { poly2[i,j];
           xu = 1j;
           xv = iy;
           xuu = 1ix;
           xvv = 1iy;
           xuv = iyx;
         }
     else
     if(approx == 5) /* B-spline fitting using Kak-Hwang mask */
     { xu = convol[i,j,mask_u,weight_u,5];
       xv = convol[i,j,mask_v,weight_v,5];
       xuu = convol[i,j,mask_uv,weight_uv,5];
       xv = convol[i,j,mask_u,weight_u,5];
       xuv = convol[i,j,mask_v,weight_v,5];
       xuv = convol[i,j,mask_v,weight_v,5];
     }
     else /* without smoothing B-spline fitting using Kak-Hwan */
     { xu = convol[i,j,op_u,div_u,3];
       xv = convol[i,j,op_v,div_v,3];
       xuu = convol[i,j,op_uv,div_uv,3];
       xvv = convol[i,j,op_u,div_u,3];
       xuv = convol[i,j,op_v,div_v,3];
     }
   }
   (parm[i][j].xu) = xu;
   parm[i][j].xv = xv;
   parm[i][j].xuu = xuu;
   parm[i][j].xvv = xvv;
   parm[i][j].xuv = xuv;
   gauss = (xuu*xv-xuv*xyv)/
   ((float) (pow((double) 1+xu*xv+xv*xyv), (double) (2)));
   mean = (xuu*xv+xxv*xv+xyv+xu-2.0*xu*xv*xyv)/
*/

{float}(pow((double) 1+xu*xv+xv*xyv), (double) (1.5))
if(gauss > maxv.gauss) maxv.gauss = gauss;
if(maxv.mean > maxv.mean) maxv.mean = mean;
if(gauss < minv.gauss) minv.gauss = gauss;
if(mean < minv.mean) minv.mean = mean;
if(fit_error > 255) fit_error = 255;
if(fit_error > maxv.fit_error) maxv.fit_error = fit_error;
if(fit_error < minv.fit_error) minv.fit_error = fit_error;
parm[i][j].fit_error = fit_error;
parm[i][j].gauss = gauss;
parm[i][j].mean = mean;
}
} /* End of loop to compute derivatives and curvature values */

/* open ikonas display */
if(ikopen(NULL) == 1)
    printf("Can't open IKONAS \n");

printf("ENTER > ");
scanf("%s",string);
while((strcmp("exit",string) != 0) && (strcmp("quit",string) != 0))
  { /* interactive loop to do things starts here */
    if(strcmp("comp",string) == 0)
      { /* compute general things */
      for(i=offset1;i<row=offset1;i++)
      for(j=col=offset1;j++)
      if(ri[i][j] > back_threshold)
        { *** sqrt q ***
          parm[i][j].squrtg = (float)sqrt((double) l+parm[i][j].
          parm[i][j].xu+parm[i][j].xv+parm[i][j].xuv;
          if(parm[i][j].squrtg > maxv.squrtg)
            maxv.squrtg = parm[i][j].squrtg;
          if(parm[i][j].squrtg < minv.squrtg)
            minv.squrtg = parm[i][j].squrtg;
        }
      } /* End of loop to compute derivatives and curvature values */
    } /* compute general things */
  } /* interactive loop to do things starts here */
/* open ikonas display */
if(ikopen(NULL) == 1)
    printf("Can't open IKONAS \n");

printf("ENTER > ");
scanf("%s",string);
while((strcmp("exit",string) != 0) && (strcmp("quit",string) != 0))
  { /* interactive loop to do things starts here */
    if(strcmp("comp",string) == 0)
      { /* compute general things */
      for(i=offset1;i<row=offset1;i++)
      for(j=col=offset1;j++)
      if(ri[i][j] > back_threshold)
        { *** sqrt q ***
          parm[i][j].squrtg = (float)sqrt((double) l+parm[i][j].
          parm[i][j].xu+parm[i][j].xv+parm[i][j].xuv;
          if(parm[i][j].squrtg > maxv.squrtg)
            maxv.squrtg = parm[i][j].squrtg;
          if(parm[i][j].squrtg < minv.squrtg)
            minv.squrtg = parm[i][j].squrtg;
        }
      } /* End of loop to compute derivatives and curvature values */
    } /* compute general things */
  } /* interactive loop to do things starts here */
/* open ikonas display */
if(ikopen(NULL) == 1)
    printf("Can't open IKONAS \n");

printf("ENTER > ");
scanf("%s",string);
while((strcmp("exit",string) != 0) && (strcmp("quit",string) != 0))
  { /* interactive loop to do things starts here */
    if(strcmp("comp",string) == 0)
      { /* compute general things */
      for(i=offset1;i<row=offset1;i++)
      for(j=col=offset1;j++)
      if(ri[i][j] > back_threshold)
        { *** sqrt q ***
          parm[i][j].squrtg = (float)sqrt((double) l+parm[i][j].
          parm[i][j].xu+parm[i][j].xv+parm[i][j].xuv;
          if(parm[i][j].squrtg > maxv.squrtg)
            maxv.squrtg = parm[i][j].squrtg;
          if(parm[i][j].squrtg < minv.squrtg)
            minv.squrtg = parm[i][j].squrtg;
        }
      } /* End of loop to compute derivatives and curvature values */
    } /* compute general things */
  } /* interactive loop to do things starts here */
2*parm[1][j].xuv*parm[1][j].xu
parm[1][j].xuv*parm[1][j].xuvv)
if(parm[1][j].q > maxv.q)
  maxv.q = parm[1][j].q;
if(parm[1][j].q < minv.q)
  minv.q = parm[1][j].q;

/*** zeros of cos phi ***/
parm[1][j].cosphi = (float)/(parm[1][j].xu*parm[1][j].xu*parm[1][j].xu +
  (float)(sqrt((double)(1+parm[1][j].xu*parm[1][j].xu +
  parm[1][j].xu*parm[1][j].xu*parm[1][j].xu +
  parm[1][j].xu*parm[1][j].xu*parm[1][j].xu))/));
if(parm[1][j].cosphi > maxv.cosphi)
  maxv.cosphi = parm[1][j].cosphi;
if(parm[1][j].cosphi < minv.cosphi)
  minv.cosphi = parm[1][j].cosphi;

else if(strcmp("ikclose",string) == 0)
  ikclose();
else if(strcmp("ikopen",string) == 0)
  ikopen(NULL);
else if(strcmp("thresh",string) == 0)
{
  printf("threshold for Gaussian curvature > ");
  scanf("%f",&th_gauss);
  printf("threshold for Mean curvature > ");
  scanf("%f",&th_mean);
  for(i=offset1;i<row-offset1;i++)
    for(j=offset1;j<col-offset1;j++)
      if(v[i][j] > back_threshold)
        {
          if(fabs((double)parm[1][j].gauu) <= th_gauss)
            parm[1][j].th_gauss = 0; /* Gaussian curv = 0 */
          else if(fabs((double)parm[1][j].gauu) > th_gauss)
            parm[1][j].th_gauss = 1; /* Gaussian curv is -ve */
          else if(fabs((double)parm[1][j].gauu) > th_gauss)
            parm[1][j].th_gauss = -1; /* Gaussian curv is +ve */
          if(fabs((double)parm[1][j].th_gauss) > maxv.th_gauss)
            maxv.th_gauss = parm[1][j].th_gauss;
          if(fabs((double)parm[1][j].th_gauss) > minv.th_gauss)
            minv.th_gauss = parm[1][j].th_gauss;
          if(fabs((double)parm[1][j].mean) <= th_mean)
            parm[1][j].th_mean = 0; /* mean curv = 0 */
          else if(fabs((double)parm[1][j].mean) < 0)
            parm[1][j].th_mean = 2; /* mean curv is -ve */
          else parm[1][j].th_mean = 1; /* mean curv is +ve */
        }
/* end of label */

else if(strcmp(string,"temp") == 0)
{
/* put the indicated parameter in the buffer */
for(i=0;i<row;i++)
    for(j=0; j<col; j++)
        buffer[i][j] = parm[i][j].gauss;
}

else if(strcmp(string,"disp") == 0)
{
    scanf("%d",&option);
    printf("scale factor >");
    scanf("%f",&sf);
    for(i=0;i<row;i++)
    {
        for(j=0; j<col; j++)
            switch(option)
            {
            case 1: line[i] = sf*parm[i][j].fit_error;break;
            case 2: line[i] = sf*parm[i][j].gauss; break;
            case 3: line[i] = sf*parm[i][j].mean; break;
            case 4: line[i] = sf*parm[i][j].label; break;
            case 5: line[i] = sf*parm[i][j].th_gauss; break;
            case 6: line[i] = sf*parm[i][j].th_mean; break;
            case 7: line[i] = sf*parm[i][j].q; break;
            case 8: line[i] = sf*parm[i][j].sqrtg; break;
            case 9: line[i] = sf*parm[i][j].cosphi; break;
            case 10: line[i] = sf*parm[i][j]; break;
            }
    lw_n(offx,i+offy,line[0],col);
}

else if(strcmp(string,"offset") == 0)
{
    scanf("%f %f",&offx,&offy);
}

else if(strcmp(string,"hist") == 0)
/* display histogram using qkdaw */

scanf("%d",&option);

printf("# xmin : ");
scanf("%f",&xmin,&xmax);

printf("# of points desired >");
scanf("%d",&npts);

for(i=0;i<npts;i++)
    { x[i] = xmin + ((float)(i) * (xmax-xmin))/(float)(npts);
    y[i] = 0;
    }
for(i=0; i<row; i++)
    for(j=0; j<col; j++)
    {
    switch(option)
    {
    case 1: value = parm[i][j].fit_error;break;
    case 2: value = parm[i][j].gauss; break;
    case 3: value = parm[i][j].mean; break;
    case 4: value = parm[i][j].label; break;
    case 5: value = parm[i][j].th_gauss; break;
    case 6: value = parm[i][j].th_mean; break;
    case 7: value = parm[i][j].q; break;
    case 8: value = parm[i][j].sqrtg; break;
    case 9: value = parm[i][j].cosphi; break;
    case 10: value = ri[i][j]; break;
    }
    sub((int)((value - xmin)*(npts-xmin))/(xmax-xmin));
    if(subs < 1000) & (subs > 0) y[subs] = y[subs] +
    ymin = 1000;
    ymax = -1000;
for(i=0;i<npts;i++)
{ if(ymin > y[i]) ymin = y[i];
  if(ymax < y[i]) ymax = y[i];
}
printf("ymin = \$f, ymax = \$f\n", ymin, ymax);
scanf("\$f \$f", &ymin, &ymax);
lopt = 0;
qterm(4);
qkdraw(npts,x,y,lopt,&xmin,&xmax,&ymin,&ymax);
qdtitl("Histogram");
qklbl("<= values -->");
qylab("freq");
qdone();
}
else if(strcmp(string,"row") == 0)
{ /* row histogram display using qkdraw */
  scanf("%d",&option);
  printf(" row =\n");
  scanf("%d",&i);
  while((i < row) & & (i >= 0))
  {
    ymin = 1000; ymax = -1000;
    for(j=0;j<col;j++)
    {
      switch(option)
      {
      case 1: linet[j] = parm[i][j].fit_error; break;
      case 2: linet[j] = parm[i][j].gauss; break;
      case 3: linet[j] = parm[i][j].mean; break;
      case 4: linet[j] = parm[i][j].label; break;
      case 5: linet[j] = parm[i][j].th_gauss; break;
      case 6: linet[j] = parm[i][j].th_median; break;
      case 7: linet[j] = parm[i][j].q; break;
      case 8: linet[j] = parm[i][j].sqrtq; break;
      case 9: linet[j] = parm[i][j].cosphi; break;
      case 10: linet[j] = r[i][j]; break;
      }
      if(linet[j] < ymin) ymin = linet[j];
      if(linet[j] > ymax) ymax = linet[j];
    }
    xmin = 0.0;
    xmax = col;
    printf("\n");
    printf("x = \$f, y = \$f\n", x[i], y[i]);
    scanf("%d",&i);
"
case 4: *(pm_point uchar) = (unsigned char) sf* 
case 5: *(pm_point uchar) = (unsigned char) sf* 
case 6: *(pm_point uchar) = (unsigned char) sf* 
case 7: *(pm_point uchar) = (unsigned char) sf* 
case 8: *(pm_point uchar) = (unsigned char) sf* 
case 9: *(pm_point uchar) = (unsigned char) sf* 
case 10:* (pm_point uchar) = (unsigned char) sf* 

pm_point uchar++; 
}

pm_write(fp1, pm);
fclose(fp1);

/* interactive processing loop ends */

/* put the outputs in respective files */
/* print("\nputting results in output files \n");
for(l=1; l<16; l++) 
{
  if(l != 100) & (bool[l] == 1))
    if(pm_write(fp[l], pmp[l]) == NULL)
      printf("can't write output files \n");
}

*/

/* End of main program */

float conv0(indi, indj, mask, base, size)
int indi, indj;
int mask[3][3];
float base;
int size;
{
  int i, j, k, l;
  float sum;

    for(k=0,1=indi-1; i<=indi+1,1++; k++)
      for(l=0, j=indj-1; j<=indj+1, j++; j++)
        sum += (float) mask[k][1] * (float) ri[l][j];

    final = sum/base;
    return(final);
}

float conv0_1(indi, indj, mask, base, size)
int indi, indj;
int mask[3][3];
float base;
int size;
{
  int i, j, k, l;
  float sum;
  float final;
  
  sum = 0.0;
  for(k=0, 1=indi-1; i<=indi+1,1++; k++)
    for(l=0, j=indj-1; j<=indj+1, j++; j++)
      sum += (float) mask[k][1] * (float) ri[l][j];

    final = sum/base;
    return(final);
}

float abszz(num)
float num;
{
  float b;
  if(num > 0) b = num; else b = -num;
  return(b);
}

init_structure()
{
  int i, j;
  /* initialize the parm structure first */
bzero(&parm[0][0], sizeof(image)*BUFSIZE*BUFSIZE);

/* initialize maxv and minv structures */

maxv.fit_error = -1000.00;
maxv.gauss = -1000.00;
maxv.mean = -1000.00;
maxv.q = -1000.00;
maxv.sqrtg = -1000.00;
maxv.cosphi = -1000.00;
maxv.label = -1000;
maxv.th_gauss = -1000;
maxv.th_mean = -1000;
maxv.depth = -1000.00;

minv.fit_error = 1000.00;
minv.gauss = 1000.00;
minv.mean = 1000.00;
minv.q = 1000.00;
minv.sqrtg = 1000.00;
minv.cosphi = 1000.00;
minv.label = 1000;
minv.th_gauss = 1000;
minv.th_mean = 1000;
minv.depth = 1000.00;

} /* end of init */
#define RTD (180.0/3.14159)  
#define MAXGRAY 255  
#define WINDOW 5  
#define IFLOAT 4  
#define NARGS 2  
#define MAXORDER 3  
#define NSYS 10  
/* maximum # of coefficients possible */  
#include <stdio.h>  
#include <math.h>  
#include <strings.h>  
#include <local.pm.h>  
#include "/usr/users/alok/advanced/spline_include.h"  

char *malloc(), *strcpy();  
int getline(), body();  
double sqrt(), atan2();  
double isqrrt();  

double a[200][NSYS], a2[200][NSYS], a3[300][NSYS];  
/* matrices to store A of Ax-b; for 1st, 2nd and 3rd order fitt1  
external double lx, ly, lxx, lyy, lxy, lxxx, lyyy, lxyy,  
external double a30, a21, a12, a03, a20, a11, a02, a10, a01, a  
external double fit_error;  
external int s_order;  
external int c_size, r_size;  
external int offset1;  
external int approx;  
external float ri[BUFSIZE][BUFSIZE];  
external FILE *dumpfile;  
external int n_dump;  

read_specs()  
{  
  printf("area, *(newarea[4]) ;  
  FILE *fdin, *fdout[3];  
  char buffer[BUFSIZE],  
  *cmt, *cmd,  
  *ptr, window;  
  int i, j,  
  ncol, now,  
  count, nfiles, nf,  
  grad, mingrad, maxgrad;
compute_amatrix()
{
    int i, j, k;
    k = 0;
    for(i = -offset1; i <= offset1; i++)
        for(j = -offset1; j <= offset1; j++)
            a3[k] = a2[k] = a1[k] = 1;
    k++;
}

/* end of compute_amatrix */

/* --------------------------------------------poly2--------------------------------------------*/

poly2(row, col)
int row, col ;
{
    int x, y ;
    int r, c ;
    int x1, y1 ;
    int k ;
    double z ;
    double s000, s200, s400, s220, s600, s420 ;
    double s001, s011, s021, s031, s101, s111, s121, s201, s211, s301
    double coeff1[NSYS], b_mat[100], coeff2[NSYS], coeff3[NSYS];
    double *p_verb;
    double x2, y4, y2 ;
    double det1, det2 ;
    double invl[10], inv2[10] ;
    double determ();
    double error1, error2, error3;

    k = 0;
    s000 = s200 = s400 = s220 = s420 = s600 = s420 = s600 = 0 ;
    s001 = s011 = s021 = s031 = s101 = s111 = s121 = s201 = s211 = s301

    for( y = -offset1 ; y <= offset1 ; y++ )
        for( x = -offset1 ; x <= offset1 ; x++ )
        {
            c = col + x ;
            r = row + y ;
            if( c < 0 ) c = 0 ;
            else if( c >= csize ) c = csize - 1 ;
            if( r < 0 ) r = 0 ;
            else if( r >= rsize ) r = rsize - 1 ;
            b_mat[k] = r = (double) r1[r][c] ;
            k++ ;

            s000 = s000 + 1 ;
            x2 = x * x ;
            y4 = y * y ;
            s200 = s200 + x2 ;
            s400 = s400 + x4 ;
            s000 = s000 + 1 ;
            x2 = x * x ;
            x4 = x2 * x2 ;
            y2 = y * y ;
            s200 = s200 + x2 ;
            s400 = s400 + x4 ;
        }

poly1(row, col)
int row, col ;
{
    double z, z1, z2 ;
    z = (double) r1[row][col] ;
    z1 = (double) r1[body(rsize, row)][body(csize, col+1)] ;
    z2 = (double) r1[body(rsize, row+1)][body(csize, col)] ;
    lx = z1 - z ;
    ly = z2 - z ;
}
s220 = s220 + x2 * y2;
s600 = s600 + x4 * x2;
s420 = s420 + x4 * y2;

s001 = s001 + z;
s011 = s011 + y * z;
s021 = s021 + y2 * z;
s031 = s031 + y2 * y * z;
s101 = s101 + x * z;
s111 = s111 + x * y * z;
s121 = s121 + x * y2 * z;
s201 = s201 + x2 * z;
s211 = s211 + x2 * y * z;
s301 = s301 + x2 * x * z;
}

if (approx==1) {
    coeff1[0] = a10 = s101 / s200;
    coeff1[1] = a01 = s011 / s200;
    coeff1[2] = a00 = s001 / s000;
    a30 = a21 = a12 = a03 = 0.0;
    a20 = a11 = a02 = 0.0;
    fit_error = error1 = lsqerr1(a1,coeff1,b_mat,k,10,6p_averb); /* c
}
else if (approx==2) {
    coeff2[3] = a10 = s101 / s200;
    coeff2[4] = a01 = s011 / s200;
    coeff2[5] = a00 = s001 / s000;
    det2 = determin(s400,s220,s200,s400,s200,s000);
    inverse(s400,s220,s200,s400,s200,s000,inv2,inv2);
    a30 = a21 = a12 = a03 = 0.0;
    coeff2[2] = a11 = s111 / s220;
    fit_error = error2 = lsqerr1(a2,coeff2,b_mat,k,6,6p_averb);
}
else if (approx==3) {
    det1 = determin(s600,s420,s400,s420,s220,s200);
    det2 = determin(s600,s220,s200,s400,s200,s000);
    inverse(s600,s420,s400,s420,s220,s200,inv1,inv1);
    inverse(s400,s220,s200,s400,s200,s000,inv2,inv2);

    coeff3[6] = a11 = s111 / s220;
    fit_error = error3 = lsqerr1(a3,coeff3,b_mat,k,10,6p_averb);
}

/* the program computes the least square fitting error by calling the
   lsqerr1 routine in the source file solver.c. the calling parama
   a: the n*m matrix,
   x: m*1 matrix having the values of coeficients.
   b: observed values at the n points.
   p_averb : value returned = average value at n points.
   all the variables are doble except for n and m which a
   integers. 
   called as: double lsqerr1(a,x,b,n,m,p_averb)
   returns lsq error (double). */

if((error1 <= error2) & & (error1 <= error3))
{
    s_order = 1;
    fit_error = error1;
    a01 = coeff1[1];
    a10 = coeff1[0];
    a00 = coeff1[2];
    a30 = a21 = a12 = a03 = 0.0;
    a20 = a11 = a02 = 0.0;
}
else if(error2 <= error3)
{
    s_order = 2;
    fit_error = error2;
    a00 = coeff2[5];
    a01 = coeff2[4];
    a10 = coeff2[3];
    a11 = coeff2[2];
    a20 = coeff2[1];
    a21 = coeff2[0];
}
else
{
    s_order = 3;
    fit_error = error3;
}
*/

/*
if(todump == 1) printf(dumpfile,"x = \%d y = \%d error = \%f \n",row,c
/* if(todump == 1) printf(dumpfile,"x = \%d y = \%d error1 = \%f error2

ix = a10;
ly = a01;
ixx = 2.0 * a20;
lyy = 2.0 * a02;
ixy = a11;
lyx = a11;
ixxx = 6.0 * a30;
lyyy = 6.0 * a03;
ixyy = 2.0 * a12;
lyxy = 2.0 * a21;

/* printf("ix %f iy %f ixx %f iyy %f ixy %f\n",ix,ly,ixx,lyy,ixy);
return(0);
} /* Main */


*/

---

compute determinant of symmetric 3 x 3 matrix. ---

double detemn( b11, b12, b13, b22, b23, b33 )

double b11, b12, b13, b22, b23, b33;

{ double templ, temp2;
  templ = b11 * b22 + b33 + b12 * b23 + b13 * b22 + b12 * b23 + b13;
  temp2 = b13 * b22 + b12 * b23 + b33 + b12 * b23 + b13;
  return( templ - temp2 );
}

---

get inverse of symmetric matrix. ---

inverse( b11, b12, b13, b22, b23, b33, det, inv )

double b11, b12, b13, b22, b23, b33;

double det;

double inv[];

{ inv[0] = 0.0;
  inv[1] = (b22 * b33 - b23 * b23) / det;
  inv[2] = -(b12 * b33 - b13 * b23) / det;
  inv[3] = (b12 * b23 - b22 * b13) / det;
  inv[5] = (b11 * b33 - b13 * b13) / det;
  inv[6] = -(b11 * b23 - b12 * b13) / det;
  inv[7] = inv[3];
  inv[8] = inv[6];
  inv[9] = (b11 * b22 - b12 * b12) / det;
  return(0);
}

int body(size,x);

int size, x;

{ if (x<0) return(0);
  else if (x==size)
    return(size-1);
  else
    return(x);
}

getline(s)

char s[];

{ int c, 1;
  l = 0;
  while( (c=getchar()) != '\n' && c != '\0' )
    s[++l] = c;
  s[l] = '\0';
  return(l);
}

/**************************** lsgerr1 ****************************
double lsqerr1(A[], x[], b[], n, p_verb)
    double A[][NSYS], x[], b[],
    *p_verb;
    int n;
{
    int i, j;
    double sum, errsum, bsum;

    errsum = bsum = 0.0;
    for (i = 0; i < m; i++) {
        sum = 0.0;
        for (j = 0; j < n; j++)
            sum = sum + A[i][j]*x[j];
        sum = sum - b[i];
        errsum = errsum + fabs(sum);
        bsum = bsum + fabs(b[i]);
    }
    *p_verb = bsum / (double)n;
    return(errsum);
}
/* :lsqerr */
The program calls routines in solver.c to fit the surface on specific data. The program calls routines in solver.c to fit the surface on specific data. The program calls routines in solver.c to fit the surface on specific data. The program calls routines in solver.c to fit the surface on specific data.
merge.c

```c
int row;
int col;
} pixel_stack[PIXEL_STACK_SIZE];

int rownum;  /* (rownum, colnum) is the next pixel */
int colnum;  /* popped from the stack */

int num_call;  /* number of calls pending at a given stack_length;  /* points to the tail of the queue */
int current_element;  /* points to head of the queue */

main(argc, argv)
int argc;
char *argv[];
{
    int i,j,k,l,m,n;  /* local variables */
    ppmic *pm1,*pm2,*pm3;  /* ppmic pointers to range and labeled in
    double scale, zscale;  /* scale of the input image */
    int image_format;  /* scores input image format */
    unsigned char *uchar_point;
    float *float_point;  /* original image is scaled and smoothed short int *short_point;
    FILE *imagefile, *labelfile;
    int want;
    double x,y,averb;
    int valid = 0, invalid = 0;
    int imageformat;
    int region_num;
    int pixel_val;
    float fit_error, min_error;
    int found;
    int lab;
    int surface_fit;  /* -1 : if surface fit is to be done

    if(argc != 4)
    {  fprintf(stderr, "Usage: merge {scaled & smoothed input PM_F ima
        exit(0);
    }

    printf("Want to display region growing on IKONAS. y if yes > ");
    scanf("%s", &ikonas_disp[0]);

printf("Edge_threshold : ");
    scanf("%d", &edge_threshold);

    if((imagefile = fopen(argv[1], "r")) == NULL)
    {  fprintf(stderr,"Cannot open input range image file : %s \n", argv
        exit(0);
    }

    if((labelfile = fopen(argv[2], "r")) == NULL)
    {  fprintf(stderr,"Cannot open image label file : %s \n", argv[2]);
        exit(0);
    }

    if((lapfile = fopen(argv[3], "r")) == NULL)
    {  fprintf(stderr,"Cannot open laplacian operated file : %s \n", argv
        exit(0);
    }

    outputfile = fopen("log", "w");
    if((pm1 = pm_read(imagefile, 0)) == NULL)
    {  fprintf(stderr,"Error in reading PM-format range image file : %
        exit(0);
    }

    if((pm2 = pm_read(labelfile, 0)) == NULL)
    {  fprintf(stderr,"Error in reading PM-format image file : %
        exit(0);
    }

    if((pm3 = pm_read(lapfile, 0)) == NULL)
    {  fprintf(stderr,"Error in reading PM-format Laplacian image file : %
        exit(0);
    }

    row = pm1->pm_row;
    col = pm1->pm_ncol;
    if((row != pm2->pm_row) || (col != pm2->pm_ncol) || (row != pm3->p
        || (col != pm3->pm_ncol))
    {  fprintf(stderr,"Rows and/or columns not same in range and label
```
merge.c

```c
exit(0);
}

if(pm2->pm_form != PM_C)
{
    fprintf(stderr,"Label image is not in PM_C format \n")
    exit(0);
}

if(pm1->pm_form == PM_F){
    float_point = (float *) pm1->pm_image;
    imageformat = PM_F;
    scale = 74.20; /* for RCA images */
    zscale = 1.00;
    for(i=0; i<row; i++)
        for(j=0; j<col; j++)
            (buffer[i][j] = *(float_point);float_point++);
}
else
    if(pm1->pm_form == PM_S){
        imageformat = PM_S;
        short_point = (short int *) pm1->pm_image;
        scale = 74.20; /* for RCA images */
        zscale = 1.00;
        for(i=0; i<row; i++)
            for(j=0; j<col; j++)
                (buffer[i][j] = *(short_point));
    }
else
    if(pm1->pm_form == PM_C){
        imageformat = PM_C;
        uchar_point = (unsigned char *) pm1->pm_image;
        zscale = 1.5; /* for GUS images */
        scale = 1.978;
        for(i=0; i<row; i++)
            for(j=0; j<col; j++)
                (buffer[i][j] = *(uchar_point));
    }
else
    printf("unrecognized format in the input image \n");
    exit(0);
}

uchar_point = (unsigned char *) pm2->pm_image;
for(i=0; i<row; i++)
    for(j=0; j<col; j++)
        (label[i][j] = 0 - *(uchar_point++));
short_point = (short int *) pm3->pm_image;

/* initialize the structure */
for(i=0; i<row; i++)
    for(j=0; j<col; j++)
        lap[i][j] = *(short_point++);

if(pm1->pm_form == PM_F)
{
    printf("Want to smooth? (1/0) > ");
    scanf("%d", &want);
    if(want == 1) gaussian(buffer,row,col);
}

bzero((char *)&rec_image[0][0], sizeof(short int)*BUFSIZE*BUFSIZE);
bzero((char *)&regions[0], sizeof(reg)*REGION_SIZE);

/* fit a second order polynomial on every patch */
printf(stderr,"Starting surface fitting on individual regions \n")
seedr = 0;
seedc = 0;
region_label = 0;
surface_fit = 1; /* no surface fitting to be done */
kvalue = 100;
to_displace = 0;
while(next_seed() == 0) /* while there are seed regions */
{
    surf_type = label[seedrow][seedcol]; /* label of the region */
    region_label += surf_type;
    num_points = 0;
cmin = 1000;
cmax = 1000;
cmin = -1000;
cmax = -1000;
label[seedrow][seedcol] = region_label;
grow_seed(seedrow,seedcol,surf_type,region_label);
}
if(surface_fit == 1)
if(num_points < MIN_REGION_SIZE) /* region is invalid if # of */
    regions[region_label].valid = 0;
    invalid++;}
else

```
if((0 - surf_type) == FLAT)
{
    lsquare(avect,bvect,result,num_points,3);
    regions[region_label].fit_error = lsqerr(avect,result,bve)
} else
{
    lsquare(avect,bvect,result,num_points,6);
    regions[region_label].fit_error = lsqerr(avect,result,bve)
}

regions[region_label].coeffs[0] = result[0];/*a00,a01,a02
regions[region_label].coeffs[1] = result[1];
regions[region_label].coeffs[2] = result[2];

if((0 - surf_type) != FLAT)
{
    regions[region_label].coeffs[3] = result[3];
    regions[region_label].coeffs[4] = result[4];
    regions[region_label].coeffs[5] = result[5];
    regions[region_label].order = 2;
} else
{
    regions[region_label].order = 1;
}
valid++;
regions[region_label].valid = 1;

}

regions[region_label].label = region_label;
regions[region_label].size = num_points;

regions[region_label].surf_type = -surf_type;
regions[region_label].center[0] = seedrow;
regions[region_label].center[1] = seedcol;
regions[region_label].num_neigh = 0;
regions[region_label].cmin = cmin;
regions[region_label].cmax = cmax;
regions[region_label].rmin = rmin;
regions[region_label].rmax = rmax;
regions[region_label].done = 0;
regions[region_label].dmax = dmax;
con_label[region_label] = 0; /* not assigned to any convex re

fprint(outputfile," region : %d points : %d surf_type %d (row,col)=
if(num_points >= MIN_REGION_SIZE)fprintf(stderr," region : %d

fprint(outputfile," a20 : %f all : %f a02 : %f a01 :

*/

if((0 - surf_type) == FLAT) /* while loop to fit surfaces */
{
    printf("# of regions found : \d invalid : \d valid \d\n",region_label)
    fclose(outputfile);
    tr = region_label; /* total # of regions */
}
else
{
    printf(stderr,"Marking neighbours \d\n");
    /* initialize neighbours */
    for(i=0;i<row;i++)
    for(j=0;j<col;j++)
        if(label[i][j] > 0)
            if(label[i][j] != label[i+1][j])
                make_neighbour(label[i][j],label[i+1][j]);
            make_neighbour(label[i][j],label[i][j+1]);
            if(label[i][j] != label[i][j+1])
                make_neighbour(label[i][j],label[i][j+1]);
                make_neighbour(label[i+1][j],label[i+1][j+1]);
}

/
/*--------------------------------- SURFACE FITTING / REGION GROWING */
if(surfac_fit == 1)
{/* TO BE DONE ONLY IF SURFACE FITTING BASED REGION GROWING DESIRED */
    merge invalid regions with neighbouring valid region with least
to_merge = 0;
    if(to_merge == 1)
    for(i=0;i<tr;i++)
        if(regions[1],valid == 0) /* for all regions do */
        if(regions[1].valid == 0) /* if the region is invalid then do */
            /* returns the list of neighbours */
            min_error = 1000.00;
            round = 0;
            for(j=0;j<regions[1].num_neigh;)++
                /* find error in fitting the polynomial of neigh[j] on */
                if(regions[regions[1].neighour[j]].valid == 1)
                    found = 1;
                    fit_error = error_in_fit(regions[1].label,regions[1]
                    if(min_error > fit_error)
                        if(regions[regions[1].neighour[j]].valid == 1)
merge.c Wed Apr 20 21:59:01 1988 10

) /* END OF THE REGION TO BE DONE ONLY IF SURFACE FITTING BASED REGIO
GROWING IS DESIRED */

/#-------- OUTPUT IMAGES -------- */

/* output the reconstructed and error image */

for(i=1;i<=(row-1);i++)
    for(j=1;j<=(col-1);j++)
        if(label[i][j] > 0)
            {
                /* for all pixels do */
                x = i - regions[label[i][j]].center[0];
                y = j - regions[label[i][j]].center[1];
                rec_image[i][j] = x*x*regions[label[i][j]].coeffs[5] +
                    y*y*regions[label[i][j]].coeffs[4] +
                    x*y*regions[label[i][j]].coeffs[3] +
                    x*regions[label[i][j]].coeffs[2] +
                    y*regions[label[i][j]].coeffs[1] +
                    i*regions[label[i][j]].coeffs[0];
            }

/* initialize pm buffers */

printf("Saving images. Stored as PM_S images \n");

rec_file = fopen("recimage","w");
loc_file = fopen("errorimage","w");
lab_file = fopen("labimage","w");

pm = pm_alloc();
pm->pm_row = row;
pm->pm_ncol = col;
pm->pm_form = PM_R;
pm->pm_image = (char *) malloc(pm_isize(pm));
 hovered(pm->pm_image,pm_isize(pm));

short_point = (short int *) pm1->pm_image;
for(i=0;i<row;i++)
    for(j=0;j<col;j++)
        {
            short_point = (short int *) fabs((double)(rec_image[i][j])*scale
                short_point++;
        }

pm_write(rec_file,pm1);
fclose(rec_file);
bzero(pml->pm_image, pm_isize(pml));
short_point = (short int *) pml->pm_image;
for(i=0;i<row;i++)
    for(j=0;j<col;j++)
        { *short_point = (short int )fabs((double)rec_image[i][j] - short_point++);
    }

pm_write(loc_file,pml);
fclose(loc_file);

/* output new label image */
pml->pm_form = PM_C;
pml->pm_image = (char *)malloc(pm_isize(pml));
bzero(pml->pm_image, pm_isize(pml));
uchar_point = (unsigned char *)pml->pm_image;
for(i=0;i<row;i++)
    for(j=0;j<col;j++)
        { *uchar_point = regions[label[i][j]].surf_type;
            uchar_point++;
        }

pm_write(lab_file,pml);
fclose(lab_file);

if(imageformat != PM_F)
    {
        pml = pm_alloc();
        pml->pm_nrow = row;
        pml->pm_ncol = col;
        pml->pm_form = PM_F;
        pml->pm_image = (char *)malloc(pm_isize(pml));
bzero(pml->pm_image, pm_isize(pml));
        float_point = (float *) pml->pm_image;
        for(i=0;i<row;i++)
            for(j=0;j<col;j++)
                { *float_point = buffer[i][j];
                    float_point++;
                }

        outfile = fopen("image", "w");
        pm_write(outfile,pml);
        fclose(outfile);
    }

/****** CONVEX PATCHES GROWING******/

/* mark all the regions as undone */
for(i=1;i<tr;i++)
    regions[i].done = 0;

/* label convex subparts in the image */
printf(stderr,"Label convex subparts in the image \n");
convex_label = 0;
pixel_val = 100;
while((lab = get_next_region()) != -1)
    { pixel_val = pixel_val + 2;
        convex_label++;
        color(convex_label) = pixel_val;
        regions[lab].done = 1;
        congrowlab = lab;
        extend_region(lab);
    }

/* output the convex/concave image */
con_file = fopen("confile", "w");

printf(stderr, "# of convex patches found: \d", convex_label);

pml = pm_alloc();
pml->pm_nrow = row;
pml->pm_ncol = col;
pml->pm_form = PM_C;
pml->pm_image = (char *)malloc(pm_isize(pml));
bzero(pml->pm_image, pm_isize(pml));
uchar_point = (unsigned char *)pml->pm_image;
con_label[0] = 0;
for(i=0;i<row;i++)
    for(j=0;j<col;j++)
        { if((con_label[label[i][j]] != 0)
            *uchar_point = color[con_label[label[i][j]]];
            if((con_label[label[i][j]] != con_label[label[i][j]+1]) ||
                (con_label[label[i][j]] != con_label[label[i][j]+1]))
                *uchar_point = 255;
            uchar_point++;
        }

pm_write(con_file,pml);
fclose(con_file);
void next_seed()
{
    int i, j, k, l;
    while ((buffer[seedr][seedc] == 0) || (label[seedr][seedc] >= 0)) {
        seedc++;
        if (seedc == (col-1)) {
            seedr++;
            seedc = 0;
        }
        if (seedr == (row-1)) return(-1);
    }
    seedrow = seedr;
    seecol = seedc;
    return(0);
}

void grow_seed(srow, scol, stype, slabel)
{
    int trow;
    int tscol;
    int tstype;
    int tslabel;
    int i, j;
    double x, y;
    if (to_disp == 1) lwr(scol, srow, &k);
    if ((rmin > srow) rmin = srow;
    if ((cmax < srow) cmax = srow;
    if ((rmin > scol) rmin = scol;
    if ((cmax < scol) cmax = scol;
    if ((dmmax < buffer[srow][scol]) dmax = buffer[srow][scol];
    num_points++;
    x = srow - seedrow;
    y = scol - seedcol;
    avect[num_points][5] = x*x;
    avect[num_points][8] = y*y;
    avect[num_points][9] = x*y;
    avect[num_points][10] = y*x;
    avect[num_points][4] = x*y;
    avect[num_points][3] = y*y;
    avect[num_points][2] = x;
    avect[num_points][1] = y;
    avect[num_points][0] = 1;
    bvecr(num_points) = buffer[srow][scol];
    for (i = srow-1; i <= scol+1; i++)
    for (j = scol-1; j <= scol+1; j++)
        if ((i > 0) && (i < (row - 1)) && (j > 0) && (j < (col - 1)))
            if (label[i][j] == stype)
                label[i][j] = slabel;
                grow_seed(i, j, stype, slabel);
}   }    }   */

void make_neighbour(label1, label2)
{
    int i, j, k, l;
    int done;
    done = 0;
    k = 0;
    if (regions[label1].num_neigh == 0)
    {
        regions[label1].num_neigh = 1;
        regions[label1].neighbour[0] = label2;
    }
    else
        while (done == 0)
        {
            if (k == regions[label1].num_neigh)
                /* neighbour not yet marked in the structure */
                regions[label1].num_neigh = regions[label1].num_neigh + 1;
                regions[label1].neighbour[k] = label2;
                done = 1;
            }
            else
            { }
merge.c  Wed Apr 20 21:59:01 1988  15

if(regions[label1].neighbour[k] == label2)
        /* nothing to be done */
        done = 1;
    }
    else k++;
}
/* end of make neighbourhood */

/****** ERROR IN FIT **************

float error_in_fit(label1,label2)
int label1;
int label2;
{
int i,j,k,l;
double error;
double averb;
float x,y;
k = 0;
for(i=regions[label1].rmin;i<=regions[label1].rmax;i++)
    for(j=regions[label1].cmin;j<=regions[label1].cmax;j++)
        if(label1[j] == label1)
        {
            x = i - regions[label2].center[0];
            y = j - regions[label2].center[1];
            averb[k][0] = x*x;
            averb[k][1] = y*y;
            averb[k][2] = x*y;
            averb[k][3] = x;
            averb[k][4] = y;
            averb[k][5] = 1;
            bveck[k] = buffer[i][j];
            k++;
        }
        result[0] = regions[label2].coeffs[0];
        result[1] = regions[label2].coeffs[1];
        result[2] = regions[label2].coeffs[2];
        result[3] = regions[label2].coeffs[3];
        result[4] = regions[label2].coeffs[4];
        result[5] = regions[label2].coeffs[5];
        error = lse-lse[averb,bveck,k,5,5,averb];
        return(float)error;
} /* error in fit */

merge.c  Wed Apr 20 21:59:01 1988  16

/****** MERGE REGION ***********/

merge_region(label1,label2)
int label1;
int label2;
{
    int i,j,k,l;
    k = 0; /* counting # of points */
    for(i=regions[label1].rmin;i<=regions[label1].rmax;i++)
        for(j=regions[label1].cmin;j<=regions[label1].cmax;j++)
            if(label1[j] == label1)
            {
                label1[j] = label2;
                k++;
            }
    regions[label2].size = regions[label2].size + k;
} /* end of merging regions. that was easy */

/****** ERODE ***********/
grows the parent by removing pixels from the child, only if RMS error

criterion is met and the child is of right type.

erode(parent,child)
int parent;
int child;
{
    int i,j,k,l;
    int lab;
    curr++;
    switch(regions[parent].surf_type)
    {
        case 1: /* parent region is flat */
        {
            /* for all types of neighbours grow the region */
            attempt[regions[parent].center[0]][regions[parent].center[1]]
            grow_region(regions[parent].center[0],regions[parent].center[1]);
        }
        break;
        case 91: /* peak; sphere; convex */
        case 159: /* ridge, cylinder, convex */
        case 31: /* minimal, */
        case 127: /* saddle ridge */
        {
            switch(regions[child].surf_type)
            {
            }
        }
    }
}
merge.c  Wed Apr 20 21:59:01 1988  17

{
    case 1: /* flat */
    case 63: /* pit, sphere, concave */
    case 233: /* valley, cylinder, concave */
    case 191: /* saddle valley, concave */
    attempt[regions[parent].center[0]][regions[parent].center[1]][
    grow_region(regions[parent].center[0], regions[parent].center[1])
    break;
}

    case 63: /* pit, sphere, concave */
    case 233: /* valley, cylinder, concave */
    case 191: /* saddle valley, concave */
    {  
        switch(regions[child].surf_type)
        {  
            case 1: /* flat */
            case 91: /* peak; sphere; convex */
            case 159: /* ridge, cylinder, convex */
            case 31: /* minimal, */
            case 127: /* saddle ridge */
            attempt[regions[parent].center[0]][regions[parent].center[1]][
            grow_region(regions[parent].center[0], regions[parent].center[1])
            break;
        }
    }

    }/* end of switch */
    ; /* end of erode */

/******************************************GROW_REGION************************/

 grows parent at the expense of the child.

/*****************************/

    grow_region(nrow,ncol,parent)
    int nrow;
    int ncol;
    int parent;
    {    
        int i,j;
        float x,y;
        float value;

        if(to_disp == 1) lwr(ncol,nrow,$1$kvalue);
        if(label[nrow][ncol] != parent)

merge.c  Wed Apr 20 21:59:01 1988  18

{  
    /* pixel to work on */
    x = nrow - regions[parent].center[0];
    y = ncol - regions[parent].center[1];
    value = x**regions[parent].coeffs[5] +
    y**regions[parent].coeffs[4] +
    x*y**regions[parent].coeffs[3] +
    x*regions[parent].coeffs[2] +
    y*regions[parent].coeffs[1] +
    1*regions[parent].coeffs[0];

    if(fabs((double)(value - buffer[nrow][ncol])) <=
    fabs((double)(regions[parent].fit_error)) + THRESH_ERROR)
    {  /* acceptable pixel */
        label[nrow][ncol] = parent;
        regions[parent].size = regions[parent].size + 1;
        regions[label[nrow][ncol]].size = regions[label[nrow][ncol]]
        for(i=(nrow-1);i<=(nrow+1);i++)
        for(j=(ncol-1);j<=(ncol+1);j++)
            if((i >= 0) && (i < nrow) && (j >= 0) && (j < ncol))
            if((i == nrow) || (j == ncol))
            {  
                if((lap[i][j] < edge_threshold) &&
                (attempt[i][j] != curr))
                {  
                    attempt[i][j] = curr;
                    /* check for the # of pending calls */
                    num_call++;
                    if(num_call >= MAX_NUM_CALL)
                        store_pixel(i,j);
                    else
                        grow_region(i,j,parent);
                }
            }
        }
    }
    else
    {  
        for(i=(nrow-1);i<=(nrow+1);i++)
        for(j=(ncol-1);j<=(ncol+1);j++)
            if((i >= 0) && (i < nrow) && (j >= 0) && (j < ncol))
            if((i == nrow) || (j == ncol))
            {  
                if((lap[i][j] < edge_threshold) &&
                (attempt[i][j] != curr))
                {  
                    attempt[i][j] = curr;
                    num_call++;
                    if(num_call >= MAX_NUM_CALL)
store_pixel(i, j);
    else
        grow_region(i, j, parent);
    }
}
num_call--;
} /* region_grow */

/*************************** STORE_PIXEL & GET_PIXEL ***************************/

store_pixel(rrow, ccol)
int rrow;
int ccol;
{
    num_call--;
    pixel_stack[stack_length].row = rrow;
    pixel_stack[stack_length].col = ccol;
    stack_length++;
    if(stack_length == PIXEL_STACK_SIZE) stack_length = 0;
    if(stack_length == current_element)
        printf("\nsegment : Stack Collision While region growing \n"
    }
    
get_pixel()
{
    rownum = pixel_stack[current_element].row;
    colnum = pixel_stack[current_element].col;
    current_element++;
    if(current_element == PIXEL_STACK_SIZE) current_element = 0;
}

/*************************** PICKUPSEED ***********************************/
returns the seed label.
*/

pick_up_seed()
{
    int i, j, k, l;
    /* first look for flat, spherical, cylindrical regions */

for(i=1; i<=tr; i++)
    {
        if(((regions[i].surf_type == FLAT) || (regions[i].surf_type == RIDGE))&&
            (regions[i].done == 0) && (regions[i].fit_error <= ACCEPT_ERROR) &&
            (regions[i].size > MIN_REGION_SIZE))
            {regions[i].done = 1;
                return(i);
            }
}

/* Now look for any type of acceptable region */
for(i=1; i<=tr; i++)
    {
        if((regions[i].done == 0) && (regions[i].size > MIN_REGION_SIZE)
            && (regions[i].fit_error <= ACCEPT_ERROR))
            { regions[i].done = 1;
                return(i);
            }
}

/* No suitable seed region is available */
return(-1);
} /* pickupseed */

/*************************** GET NEXT REGION *************************/
returns the label of next region to be grown as convex region */

get_next_region()
{
    int i, j, k, l;
    int max;
    int tlabel;
    max = -1000;
    for(i=1; i<=tr; i++)
        {
            if((regions[i].done == 0) && ((regions[i].surf_type == FLAT) ||
                (regions[i].surf_type == PEAK)) &&
                (regions[i].size >= MIN_REGION_SIZE) &&
                (regions[i].dmax > max))
                { max = regions[i].dmax;
                    tlabel = i;
                }
        }
    return(tlabel);
}
merge.c  Wed Apr 20 21:59:01 1988  21

    tlabel = 1;
    
  }

  if(max != -1000) return(tlabel);
  else return(-1);
}

/***********************************************************************/
extend_region
/***************************************************************************/
used to extend a convex region recursively at the region level.
*/
extend_region(int lab)
int lab;
{
  int i, j;
  int n;
  con_label[lab] = convex_label;
  for(i=0;i<regions[lab].num_neigh;i++)
  {
    n = regions[lab].neighbour[i];
    if((regions[n].done == 0) &&
      // (regions[congrowlab].dmax > regions[n].dmax) &&
      ((regions[n].surf_type == FLAT) ||
      (regions[n].surf_type == PEAK)
      (regions[n].surf_type == RIDGE) ||
      (regions[n].surf_type == SADRID))
    {
      if(regions[n].size >= MIN_REGION_SIZE)
      {
        regions[n].done = 1;
        extend_region(n);
      }
      else
      {
        regions[n].done = 1;
        con_label[n] = convex_label;
      }
    }
  }
}
/*
   Used for calculating the least square fit-error in the grad.c program.

   Caution. Warning. Danger:
   1) Include the following in the main program:
       
       #define NSYS <n>
       #define NDATA <m>

   2) If you want to compute least-squares error, you MUST have the following in the calling program:
       
       double lsqerr();
       
       It returns E = |Ax-b| * 100

Parameters passed to the least square program are:
   A[1][], x[], b[], m, n
where
   A x = b.

To check the singularity, small constant 'epsilon' is used.
Current value is set to 0.00001.
This is O.K. for most application, but depending on your application, you may want to change the value.
*/

#include <stdio.h>
#include <math.h>
#include "usr/users/alok/advanced/spline_include.h"

#define NSYS 20
#define NDATA 10000

int prnt = 1; /* -1 for printing else no printing */
double epsilon = 0.00001;
```c
#include <math.h>

double lsqerr(A, x, b, m, n, p_averb)
{
    int m, n;
    double sum, errsum, bsum;
    errsum = bsum = 0.0;
    for (i=0; i<m; i++)
    {
        sum = 0.0;
        for (j=0; j<n; j++)
            sum = sum + A[i][j]*x[j];
        errsum = errsum + fabs(sum);
        bsum = bsum + fabs(b[i]);
    }
    *p_averb = bsum / (double)m;
    return(errsum/m);
}

double lsqrel(A, x, b, m, n)
{
    double A[][NSYS], x[], b[];
    for (i=0; i<n; i++)
        x[i] = 0.0;
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            A[i][j] = 1.0;
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            b[i] = (i+j+1); // Example data
    solver(A, x, b, m, n);
    printf("The solution is:");
    for (i=0; i<n; i++)
        printf("%d", x[i]);
    printf("\nThe sum of residuals is: %f\n", errsum);
    printf("\nThe relative error is: %f\n", errsum/m);
    return(0);
}
```

for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
        AS[i][j] = A[i][j];
    AS[i][n] = b[i];
    x[i] = 0.0;
}

if ((k=gauss(AS,p,n))!-0) {
    if (print==1) printf("Singular Matrix. Order = \d.\n", k);
    else
        k = n;
    backsue(AS,p,x,n,k);
    return(k);
} /* Solver */

/*------------------------- Gauss -------------------------*/
/*/ Gaussian elimination with full pivoting. */
int gauss (A, p, n)
  double A[|][NSYS] ;
  int p[], n ;
{
    int i, j, k, row, col ;
    int dtemp ;
    double pivot, ratio, dtemp ;
    double epsilon ;
    epsilon = 0.0000001 ;
    /* Initialize permutation vector. */
    for (i=0; i<n; i++)
        p[i] = i ;

    for (k=0; k<n-1; k++) {
      /* Find the next pivot element. */
      pivot = A[k][k] ;
      row = col = k ;
      for (i=k; i<n; i++)
        for (j=k; j<n; j++)
          if(fabs(pivot)<fabs(A[i][j])) {
            pivot = A[i][j] ;
            row = i ;
            col = j ;
          }
      if(fabs(pivot)<epsilon) return(k) ;

      /* Exchange row */
      if (k != row) {
        for (i=0; i<n; i++)
          dtemp = A[k][i] ;
        A[row][i] = dtemp ;
      }
      /* Exchange column */
      if (K != col) {
        for (i=0; i<n; i++)
          dtemp = p[col] ;
        p[col] = p[k] ;
        p[k] = dtemp ;
      }
    }

    /* Elimination. */
    for (i=k+1; i<n; i++)
      ratio = A[i][k] / pivot ;
    A[i][k] = 0.0 ;
    for (j=k+1; j<n; j++)
}

if (fabs(A[n-1][n-1])<epsilon) return(n-1) ;
return(0) ;
/* Gauss */
/* Back substitution. */

backsub (A[ ], p[ ], soln[ ], N, n)

double A[ ][NSYS], soln[ ];

int p[ ], N, n ;

{ int i, j, k ;
  double sum, sol[NSYS] ;

  for (k=n-1; k>=0; k--) {
    sum = 0.0 ;
    for (j=k+1; j<n; j++)
      sum = sum + A[k][j] * sol[j] ;
    sol[k] = (A[k][N] - sum ) / A[k][k] ;
  }

  for (k=0; k<n; k++) {
    i = p[k] ;
    soln[i] = sol[k] ;
  }

  return ;
} /* Backsub */

/* Multiplication Product */

product (B[ ], c[ ], m, n)

double B[ ][NSYS], c[ ][NSYS];

int m, n ;

{ int i, j, k ;
  double sum ;
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) AS[i][j] = A[i][j];
    AS[i][n] = b[i];
    x[i] = 0.0;
}

if ((k=gauss(AS,p,n))!=0) {
    if (print==1) printf("Singular Matrix. Order = %d, \n", k);
    else
        k = n;
    backsolv(AS,p,x,n,k);
    return(k);
} /* Solver */

/*============================================================================
/* Gaussian elimination with full pivoting. */
/*============================================================================*/

int gauss (A, p, n)

double A[][][NSYS];
int p[], n;

{ int i, j, k, row, col;
  int dtemp;
  double pivot, ratio, dtemp;
  double epsilon;
  epsilon = 0.0000001;

  /* Initialize permutation vector. */
  for (i=0; i<n; i++) { p[i] = i; }

  for (k=0; k<n-1; k++) {
    /* Find the next pivot element. */
    pivot = A[k][k];
    row = col = k;
    for (i=k; i<n; i++)
      for (j=k; j<n; j++)
  }

  if (fabs(pivot)<fabs(A[i][j])) {
    pivot = A[i][j];
    row = i;
    col = j;
  }

  if(fabs(pivot)<epsilon) return(k);

  /* Exchange row */
  if (k != row) {
    for (i=0; i<n; i++) {
      dtemp = A[k][i];
      A[k][i] = A[row][i];
      A[row][i] = dtemp;
    }
  }

  /* Exchange column */
  if (k != col) {
    dtemp = p[col];
    p[col] = p[k];
    p[k] = dtemp;
    for (i=0; i<n; i++) {
      dtemp = A[i][k];
      A[i][k] = A[i][col];
      A[i][col] = dtemp;
    }
  }

  /* Elimination. */
  for (i=k+1; i<n; i++) {
    ratio = A[i][k] / pivot;
    A[i][k] = 0.0;
    for (j=k+1; j<n; j++)
  }

  if(fabs(A[n-1][n-1])<epsilon) return(n-1);
  return(0);
} /* Gauss */
# Backsubstitution

```c
#include <stdio.h>

double A[]; B[]; C[]; N; double p[N];

void backsub(double A[], double B[], double C[], double p[], double sol[], int N, int n) {
    int i, j, k;
    double sum, sol[N];

    for (k=n-1; k>=0; k--) {
        sum = 0.0;
        for (j=k+1; j<=n; j++)
            sum = sum + A[k][j] * sol[j];
        sol[k] = (A[k][n] - sum) / A[k][k];
    }

    for (k=0; k<n; k++) {
        i = p[k];
        sol[i] = sol[k];
    }

    return;
} /* Backsub */
```

# Mult

```c
void mult(double A[][], double B[], double C[], int m, int n) {
    int i, j;
    double sum;

    for (i=0; i<m; i++) {
        sum = 0.0;
        for (j=0; j<n; j++)
            sum = sum + A[i][j] * B[j];
        C[i] = sum;
    }

    return;
} /* Mult */
```

# Product

```c
void product(double A[], double B[], double C[], int m, int n) {
    int i, j, k;
    double sum;

    for (i=0; i<m; i++) {
        sum = 0.0;
        for (j=0; j<n; j++)
            sum = sum + A[i][j] * B[j];
        C[i] = sum;
    }

    return;
} /* Product */
```
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) as[i][j] = a[i][j];
    x[i] = 0.0;
}

if ((k = gauss(as, p, n)) != 0) {
    if (print == -1) printf("Singular Matrix. Order = %d, \n", k);
    else
        k = n;
    backsolve(as, p, x, n, k);
    return(k);
} /* Solver */

/************************************************** Gauss **************************************************
/* Gaussian elimination with full pivoting. */
int gauss (A, p, n)
double A[][NSYS];
int p[], n;
/
int i, j, k, row, col;
int itemp;
double pivot, ratio, dtemp;
double epsilon;
epsilon = 0.0000001;
/
/* Initialize permutation vector. */
for (i = 0; i < n; i++) { p[i] = i; }
for (k = 0; k < n; k++) {
    /* Find the next pivot element. */
    pivot = A[k][k];
    row = col = k;
    for (i = k; i < n; i++)
        for (j = k; j < n; j++)
            if (fabs(p[i]) < fabs(A[i][j]))
                pivot = A[i][j];
        row = i;
        col = j;
    }
    if(fabs(p[row]) < epsilon) return(n);
/
/* Exchange row */
if (k = row) {
    for (i = 0; i < n; i++)
            dtemp = A[k][i];
        A[k][i] = A[row][i];
    A[row][k] = dtemp;
}
/
/* Exchange column */
if (k = col) {
    itemp = p[col];
    p[col] = p[k];
    p[k] = itemp;
    for (i = 0; i < n; i++)
            dtemp = A[i][k];
        A[i][k] = A[i][col];
    A[i][col] = dtemp;
}
/
/* Elimination. */
for (i = k + 1; i < n; i++) {
    ratio = A[i][k] / pivot;
    A[i][k] = 0.0;
    for (j = k + 1; j < n; j++)
}
/
if (fabs(A[n-1][n-1]) < epsilon) return(n-1);
return(0);
} /* gauss */
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        AS[i][j] = A[i][j];

x[i] = 0.0;

if (k == gauss(AS, p, n))
    if (print == 1) printf("Singular Matrix. Order = %d\n", k);
else
    k = n;
backsub(AS, p, x, n, k);
return(k);
}  /* Solver */

/*-------------------------- Gauss --------------------------*/
/* Gaussian elimination with full pivoting. */

int gauss(A, p, n)

double A[NSYS];
int p[ ], n;

int i, j, k, row, col;
int itemp;
double pivot, ratio, dtemp;
double epsilon;

/* Initialize permutation vector. */
for (i=0; i<n; i++)
    p[i] = i;

for (k=0; k<n-1; k++)

    /* Find the next pivot element. */
    pivot = A[k][k];
    row = col = k;
    for (i=k; i<n; i++)
        for (j=k; j<n; j++)

        if (fabs(pivot) < fabs(A[i][j]) 
            pivot = A[i][j];
            row = i;
            col = j;
        }
    if (fabs(pivot) < epsilon) return(k);

    /* Exchange row */
    if (k != row)
        for (i=0; i<n; i++)
            dtmp = A[k][i];
            A[k][i] = A[row][i];
            A[row][i] = dtmp;

    /* Exchange column */
    if (k != col) 
        itemp = p[col];
        p[col] = p[k];
        p[k] = itemp;

        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                dtmp = A[i][k];
                A[i][k] = A[i][col];
                A[i][col] = dtmp;

    /* Elimination. */
    for (i=k+1; i<n; i++)
        ratio = A[i][k] / pivot;
        A[i][k] = 0.0;
        for (j=k+1; j<n; j++)

    if (fabs(A[n-1][n-1]) < epsilon) return(n-1);
return(0);
}  /* gauss */
```c
// Back substitution.

backsub (A, p, soln, N, n) 

double A[][SYS], soln[] ;
int p[], N, n ;
{
    int i, j, k;
    double sum, sol[NSYS] ;

    for (k=N-1; k>=0; k--) {
        sum = 0.0 ;
        for (j=k+1; j<N; j++)
            sum = sum + A[k][j] * sol[j] ;
        sol[k] = (A[k][N] - sum) / A[k][k] ;
    }

    for (k=0; k<N; k++) {
        soln[p[k]] = sol[k] ;
    }
    return ;
} /* Backsub */

// Product

product (B, C, m, n) 

double B[][SYS], C[][SYS] ;
int m, n ;
{
    int i, j, k ;
    double sum ;
    for (i=0; i<N; i++)
        for (j=0; j<m; j++)
            sum = sum + B[i][j] * C[j][i] ;
    return ;
} /* Product */
```