Chapter 5

Numerical Issues

The computer implementations of most components of the segmentation process are straightforward. I describe here three components which are not found in the standard numerical analysis references. The fourth component is an algorithm for finding the 1–dimensionsal skeleton of an n–dimensional binary object.

The first component is the construction of centralized finite differences for general order derivatives. Although the current segmentation code contains only routines for first–order differentiation, later versions may conceivably use higher–order derivatives and higher–order approximations. Most numerical analysis texts mention a few finite differences for low order differentiation. The construction I give is not difficult, but it does allow construction of the appropriate template for convolution for a given order derivative and given order approximation.

The second component is Gaussian blurring using finite differences, but where the discretization is in terms of the scale variable σ (sampled according to a geometric sequence) rather than the usual time variable t (sampled according to an arithmetic sequence). In addition to stating the numerical method for blurring, I give a proof of stability when the base of the sequence is suitably restricted. The question of what base to use for the geometric sampling to guarantee that no information is missed by the sampling has been an open question. Since the front end of the human visual system appears to handle scale in a way that is similar to finite difference methods, the bounds obtained on the base may be of importance in understanding how the human visual system handles scale.

The third component is the solution of eigensystems of the form $A\vec{v} = \lambda B\vec{v}$ where A and B are real symmetric matrices. In the height definition for ridges, A is the negative of a Hessian matrix and B is the identity matrix, so the usual numerical methods apply in finding eigenvalues and eigenvectors. I give a variation on a method using Householder reduction which does some of the initial matrix manipulations analytically rather than numerically. In the principal direction and level definitions for ridges, A represents the second fundamental form and B represents the first fundamental form. The generalized eigensystem is equivalent to $B^{-1}A\vec{v} = \lambda\vec{v}$, a regular eigensystem, but $B^{-1}A$ is usually not symmetric. I give a method which takes advantage of the symmetry of A and B, still uses Householder reduction, and therefore retains the stability inherent in Householder methods.

The fourth component involves finding 1–dimensional skeletons of an n–dimensional object. The skeleton construction preserves the topology (connectivity) of the original object. If the original object has k holes, then the skeleton had k holes. Also, the construction preserves the general geometric shape of the original object by thinning the object layer–by–layer from the outside to inside.

5.1 Finite Differences

This section contains the construction for templates corresponding to central difference formulas for derivatives of functions of a single variable. Derivatives for multivariable functions can be obtained from the one–variable templates by tensor products.

5.1.1 Derivatives of Univariate Functions

Central finite differences are computed using Taylor polynomial approximations to functions. Probably the most familiar finite differences are

$$f'(x) \doteq \frac{f(x+h) - f(x-h)}{2h} \quad \text{and} \quad f''(x) \doteq \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

for a given step size $h > 0$. Each of these approximations is of order $O(h^2)$. In image applications, usually the step size is specified to be $h = 1$ (the pixel spacing), so as a template applied to 1–dimensional images,

$$D_1 = \frac{1}{2}(-1, 0, 1) \quad \text{and} \quad D_2 = (1, -2, 1)$$

for the first and second derivative templates, respectively. Note that the entries of the template, read from left to right, correspond to the coefficients of $f(x-h)$, $f(x)$, and $f(x+h)$. One way to compute these approximations is by expanding the terms $f(x+h)$ and $f(x-h)$ as Taylor polynomials. I use Taylor series in a formal way and ignore the convergence properties.

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \tfrac{1}{2}h^2 f''(x) + \cdots + \tfrac{1}{n!}h^n f^{(n)}(x) + \cdots \\ f(x-h) &= f(x) - hf'(x) + \tfrac{1}{2}h^2 f''(x) - \cdots + \tfrac{(-1)^n}{n!}h^n f^{(n)}(x) + \cdots \end{aligned}$$

Subtracting the two expansions yields

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2}{3!}h^3 f'''(x) + \cdots = 2hf'(x) + O(h^3),$$

so

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2).$$

Similarly, adding the two expansions yields

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + \frac{2}{4!}h^4 f^{(4)}(x) + \cdots = 2f(x) + h^2 f''(x) + O(h^4),$$

so

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2).$$

The same ideas apply to constructing $O(h^2)$ approximations to higher–order derivatives. For example, it can be shown that $O(h^2)$ approximations to the third and fourth derivatives are

$$f^{(3)}(x) \doteq \frac{f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)}{2h^3}$$

and

$$f^{(4)}(x) \doteq \frac{f(x+2h) - 4f(x+h) + 6f(x) - 4f(x-h) + f(x-2h)}{h^4}.$$

Higher–order approximations can also be constructed. For example, the $O(h^4)$ approximations for the first four derivatives are

$$\begin{aligned} f^{(1)}(x) &\doteq \tfrac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h}, \\ f^{(2)}(x) &\doteq \tfrac{-f(x+2h)+16f(x+h)-30f(x)+16f(x-h)-f(x-2h)}{12h^2}, \\ f^{(3)}(x) &\doteq \tfrac{-f(x+3h)+8f(x+2h)-13f(x+h)+13f(x-h)-8f(x-2h)+f(x-3h)}{8h^3}, \text{ and} \\ f^{(4)}(x) &\doteq \tfrac{-f(x+3h)+12f(x+2h)-39f(x+h)+56f(x)-39f(x-h)+12f(x-2h)-f(x-3h)}{6h^4}. \end{aligned}$$

Of course, given the approximation, one can use the expansions to verify its correctness. The problem is to construct the approximations from the expansions where the order of approximation is specified *a priori*. For any integer i, the Taylor expansion

$$f(x+ih) = \sum_{n=0}^{\infty} \frac{(ih)^n}{n!} f^{(n)}(x)$$

is used.

Because I am concerned only with central differences, the order of the approximation will be even. Suppose an approximation to $f'(x)$ with error of order $O(h^{2p})$ for some specified $p \geq 1$ is desired. Observe that the number of template terms in the first–derivative approximations appears to be odd and is in fact $2p+1$. Construct a template $T = (T_{-p}, \ldots, T_0, \ldots, T_p)$ so that

$$\begin{aligned} hf'(x) + O(h^{2p+1}) &= \textstyle\sum_{i=-p}^{p} T_i f(x+ih) \\ &= \textstyle\sum_{i=-p}^{p} T_i \left(\sum_{n=0}^{\infty} \frac{(ih)^n}{n!} f^{(n)}(x) \right) \\ &= \textstyle\sum_{n=0}^{\infty} \left(f^{(n)}(x) \sum_{i=-p}^{p} \frac{i^n}{n!} T_i \right) h^n. \end{aligned}$$

The equalities are true if for $0 \leq n \leq 2p$ the following condition is required:

$$\sum_{i=-p}^{p} \frac{i^n}{n!} T_i = \left\{ \begin{array}{ll} 1, & n = 1 \\ 0, & n \neq 1 \end{array} \right\}.$$

This gives $2p+1$ equations in $2p+1$ unknowns. For $p = 1$, an $O(h^2)$ approximation, the solution is $T = \frac{1}{2}(-1, 0, 1)$ and for $p = 2$, an $O(h^4)$ approximation, the solution is $T = \frac{1}{12}(1, -8, 0, 8, -1)$.

To compute approximations for any derivative $f^{(m)}(x)$ the above construction can be modified. From observation of the second, third, and fourth derivative approximations listed earlier, it appears that a template with $2c+1$ coefficients where $c = p + \lfloor (m-1)/2 \rfloor$ is needed. The function $\lfloor x \rfloor$ is the "floor" function. Construct template $T = (T_{-c}, \ldots, T_0, \ldots, T_c)$ so that

$$\frac{1}{m!} h^m f^{(m)}(x) + O(h^{2p+m}) = \sum_{i=-c}^{c} T_i f(x+ih) = \sum_{n=0}^{\infty} \left(f^{(n)}(x) \sum_{i=-c}^{c} \frac{i^n}{n!} T_i \right) h^n.$$

The equalities are true if for $0 \leq n \leq 2c$ the following condition is required:

$$\sum_{i=-c}^{c} \frac{i^n}{n!} T_i = \left\{ \begin{array}{ll} 1, & n = m \\ 0, & n \neq m \end{array} \right\}.$$

This gives us $2c+1$ equations in $2c+1$ unknowns. For $p=2$, an order $O(h^4)$ approximation, and $m=3$, the value $c=3$ and the solution is $T=\frac{1}{8}(1,-8,13,0,-13,8,-1)$.

In general, the system of equations for T can be written as $AT=B$ where A is a $(2c+1)\times(2c+1)$ matrix, B is a $(2c+1)\times 1$ matrix, each containing only integer coefficients. The system can be solved symbolically for T.

5.1.2 Derivatives of Multivariate Functions

For functions with more variables, the partial derivatives can be approximated by grouping together all of the same variables and applying the univariate approximation for that group. For now I assume that the pixel spacing is uniform in all dimensions, but that is not necessarily so in applications. For nonuniform spacing, the finite differences need to be adjusted appropriately. The first and second partial derivatives of $f(x,y)$ are

$$\begin{aligned}
f_x(x,y) &= \tfrac{f(x+h,y)-f(x-h,y)}{2h}+O(h^2),\\
f_y(x,y) &= \tfrac{f(x,y+h)-f(x,y-h)}{2h}+O(h^2),\\
f_{xx}(x,y) &= \tfrac{f(x+h,y)-2f(x,y)+f(x-h,y)}{h^2}+O(h^2),\\
f_{yy}(x,y) &= \tfrac{f(x,y+h)-2f(x,y)+f(x,y-h)}{h^2}+O(h^2), \text{ and}\\
f_{xy}(x,y) &= \tfrac{f(x+h,y+h)-f(x+h,y-h)-f(x-h,y+h)+f(x-h,y-h)}{4h^2}+O(h^2).
\end{aligned}$$

Let $\delta_{m,p}$ denote the central finite difference operator for derivative m of order $O(h^{2p})$. For example,

$$\delta_{1,1}f(x)=\frac{f(x+h)-f(x-h)}{2h} \text{ and } \delta_{2,1}f(x)=\frac{f(x+h)-2f(x)+f(x-h)}{h^2}.$$

For any p, define $\delta_{0,p}f(x)=f(x)$, the identity operator. Thus, $f'(x)=\delta_{1,1}f(x)+O(h^2)$ and $f''(x)=\delta_{2,1}f(x)+O(h^2)$.

For functions $f(x,y)$ I use tensor notation to indicate application of finite differences corresponding to different variables. For example,

$$\begin{aligned}
\delta_{0,1}\otimes\delta_{1,1}f(x,y) &= \tfrac{f(x+h,y)-f(x-h,y)}{2h},\\
\delta_{1,1}\otimes\delta_{0,1}f(x,y) &= \tfrac{f(x,y+h)-f(x,y-h)}{2h}, \text{ and}\\
\delta_{1,1}\otimes\delta_{1,1}f(x,y) &= \tfrac{f(x+h,y+h)-f(x+h,y-h)-f(x-h,y+h)+f(x-h,y-h)}{4h^2}.
\end{aligned}$$

Thus, $f_x(x,y)=\delta_{0,1}\otimes\delta_{1,1}f(x,y)+O(h^2)$, $f_y(x,y)=\delta_{1,1}\otimes\delta_{0,1}f(x,y)+O(h^2)$, and $f_{xy}(x,y)=\delta_{1,1}\otimes\delta_{1,1}f(x,y)+O(h^2)$. Note that the derivatives on f are evaluated from left to right, whereas the difference operators are performed right to left. This convention is used to

distinguish between finite differences corresponding to different variables. For example, to obtain an $O(h^4)$ approximation to f_{xxyyy}, compute $f_{xxyyy}(x,y) = \delta_{3,2} \otimes \delta_{2,2} f(x,y) + O(h^4)$. The application of $\delta_{2,2}$ to f approximates the second derivative in x. The application of $\delta_{3,2}$ to f approximates the third derivative in y.

Now I show how to construct the templates to compute the differences as convolutions with templates. Let $\delta_{m,p}$ have corresponding template $T_{m,p}$, and let $\delta_{n,p}$ have corresponding template $T_{n,p}$, as constructed in the previous section. The template corresponding to $\delta_{n,p} \otimes \delta_{m,p}$ is just the tensor product $T_{n,p} \otimes T_{m,p}$.

For example, $\delta_{1,1}$ has template $T_{1,1} = \frac{1}{2}(-1, 0, 1)$, so $\delta_{1,1} \otimes \delta_{1,1}$ has template

$$T_{1,1} \otimes T_{1,1} = \frac{1}{4} \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix},$$

where the upper left corner is the weight for $f(x-h, y-h)$ and the lower right corner is the weight for $f(x+h, y+h)$. This template is used to approximate f_{xy} for an image. As another example, $\delta_{2,1}$ has template $(1, -2, 1)$ and $\delta_{3,1}$ has template $\frac{1}{2}(-1, 2, 0, -2, 1)$, so $\delta_{2,1} \otimes \delta_{3,1}$ has template

$$T_{1,1} \otimes T_{3,1} = \frac{1}{2} \begin{bmatrix} -1 & 2 & 0 & -2 & 1 \\ 2 & -4 & 0 & 4 & -2 \\ -1 & 2 & 0 & -2 & 1 \end{bmatrix}.$$

5.2 Gaussian Blurring

The most popular method used for blurring appears to be convolution of a Gaussian kernel with the image via a fast Fourier transform (FFT). However, the implementations of FFTs usually have two problems. The first problem is that a nonnegative image when blurred by an FFT may have negative values as a result of numerical round–off errors. A region of positive measure for which the initial image is identically zero becomes a region for which there are many sign fluctuations on numbers of small magnitude. The fluctuations create problems in my applications that require Gaussian blurring, such as finding ridges in a blurred image. The second problem is that for large scale blurring, the FFTs produce artifacts near the

four corners of 2–dimensional images which look like bright four–point stars. This causes problems in my applications when scale is large; false ridges are detected in the images.

An alternate approach to Gaussian blurring is to notice that it is equivalent to solving an initial value problem for the partial differential equation $u_t = \nabla^2 u$ over $\mathbb{R}^n$ where the initial data is the image to be blurred. Large t corresponds to large scale blurring. The finite difference method in the time variable is stable only for relatively small time increments. To get a blurred image at large time (scale) requires a tremendous number of iterations, so the method is not cost–effective. However, nonnegative initial data leads to nonnegative solutions, and the method is *asymptotically stable*; that is, as t approaches infinity, the numerical solution approaches the true solution.

I have implemented Gaussian blurring with a similar finite difference scheme, but in terms of the scale variable σ where $t = \sigma^2/2$. The scale samples will form a geometric sequence, unlike the former method which samples time arithmetically. This means blurring to a larger scale can be accomplished by many fewer steps if the scale version is used rather than the time version of the numerical method. Stability is still an issue, except that the constraint will be on the base of the geometric sequence. The partial differential equation in space and scale is $\sigma u_\sigma = \sigma^2 \nabla^2 u$. The following argument is based on one spatial variable. Similar arguments can be given when there are more spatial variables.

5.2.1 An Approximation to Diffusion

Consider the initial value problem $\sigma u_\sigma(x,\sigma) = \sigma^2 u_{xx}(x,\sigma)$ for $\sigma > \sigma_0$ and $x \in \mathbb{R}$, where $u(x,\sigma_0) = I(x)$ is the initial image. Assume that $I(x)$ is zero outside the interval $[-r,r]$. I will approximate the partial derivatives by finite differences as follows. Let $\sigma = b^\tau$ for some base $b > 1$ and let $w(x,\tau) = u(x,\sigma)$. By the Mean Value Theorem (Fulks 1978),

$$w(x,\tau+1) = w(x,\tau) + w_\tau(x,\tau) + \frac{1}{2} w_{\tau\tau}(x,\hat{\tau}(x))$$

for some $\hat{\tau}(x) \in [\tau, \tau+1]$. Transforming back to the old coordinates yields

$$u(x,b\sigma) = u(x,\sigma) + (\ln b)\sigma u_\sigma(x,\sigma) + \frac{(\ln b)^2}{2}[\hat{\sigma}^2 u_{\sigma\sigma}(x,\hat{\sigma}(x)) + \hat{\sigma} u_\sigma(x,\hat{\sigma}(x))]$$

for some $\hat{\sigma}(x) \in [\sigma, b\sigma]$. Therefore,

$$\sigma u_\sigma(x,\sigma) = \frac{u(x,b\sigma) - u(x,\sigma)}{\ln b} + \frac{\ln b}{2}[\hat{\sigma}^2 u_{\sigma\sigma}(x,\hat{\sigma}(x)) + \hat{\sigma} u_\sigma(x,\hat{\sigma}(x))]. \tag{5.1}$$

A similar application of the Mean Value Theorem yields

$$\sigma^2 u_{xx}(x,\sigma) = \frac{u(x+h\sigma,\sigma) - 2u(x,\sigma) + u(x-h\sigma,\sigma)}{h^2} + \frac{h^2}{24}[\sigma^4 u_{xxxx}(\hat{x},\sigma) + \sigma^4 u_{xxxx}(\bar{x},\sigma)] \quad (5.2)$$

for $h > 0$, for some $\hat{x} \in [x, x+h\sigma]$, and for some $\bar{x} \in [x-h\sigma, x]$. An approximation to the diffusion equation is therefore

$$u(x,b\sigma) = u(x,\sigma) + \frac{\ln b}{h^2}[u(x+h\sigma,\sigma) - 2u(x,\sigma) + u(x-h\sigma,\sigma)]. \quad (5.3)$$

At this point discretize only the scale variable, say $\sigma_j = \sigma_0 b^j$ for $j \geq 0$, and let $v_j(x) = u(x,\sigma_j)$. The approximation is now

$$v_{j+1}(x) = v_j(x) + \frac{\ln b}{h^2}\left[v_j(x+h\sigma_j) - 2v_j(x) + v_j(x-h\sigma_j)\right]. \quad (5.4)$$

Now discretize the spatial variable, say $x_i = ih$ for integers i, and let $v_{i,j} = u(x_i,\sigma_j)$. The quantities $x_i \pm h\sigma_j$ are not necessarily spatial grid points. I make another approximation using cubic interpolation, which keeps the error the same order as that for the second derivative approximation. Define $\xi_1 = \lfloor i+\sigma_j \rfloor$, $\xi_0 = \xi_1 - 1$, $\xi_2 = \xi_1 + 1$, and $\xi_3 = \xi_1 + 2$. The points $h\xi_k$ are all valid spatial grid points with $h\xi_1 \leq x_i + h\sigma_j \leq h\xi_2$. The cubic Lagrange polynomials corresponding to the indices ξ_k are $L_k(\xi) = \prod_{m=0,m\neq k}^{3}(\xi - \xi_m)/(\xi_k - \xi_m)$. The tabular values to be interpolated are $v_{\xi_k,j} = u(h\xi_k,\sigma_j)$, so the interpolating polynomial for the indices is

$$p(\xi) = \sum_{k=0}^{3} v_{\xi_k,j} L_k(\xi).$$

Similarly define $\mu_1 = \lfloor i-\sigma_j \rfloor$, $\mu_0 = \mu_1 - 1$, $\mu_2 = \mu_1 + 1$, and $\mu_3 = \mu_1 + 2$, with corresponding Lagrange polynomials $M_k(\mu) = \prod_{m=0,m\neq k}^{3}(\mu - \mu_m)/(\mu_k - \mu_m)$. The tabular values to be interpolated are $v_{\mu_k,j} = u(h\mu_k,\sigma_j)$, so the interpolating polynomial is

$$q(\mu) = \sum_{k=0}^{3} v_{\mu_k,j} M_k(\mu).$$

The approximation to the diffusion equation now becomes

$$\begin{aligned} v_{i,j+1} &= v_{i,j} + \tfrac{\ln b}{h^2}\left(p(i+\sigma_j) - 2v_{i,j} + q(i-\sigma_j)\right) \\ &= v_{i,j} + \tfrac{\ln b}{h^2}\left(\textstyle\sum_{k=0}^{3} v_{\xi_k,j} L_k(i+\sigma_j) - 2v_{i,j} + \sum_{k=0}^{3} v_{\mu_k,j} M_k(i-\sigma_j)\right). \end{aligned} \quad (5.5)$$

Finally, the initial image data is given only for a finite set of indices, say $v_{i,0} = u(x_i,\sigma_0)$, $0 \leq i \leq N-1$. If one of the quantities $x_i \pm h\sigma_0$ is outside the valid range of indices, then

$u(x_i \pm h\sigma_0, \sigma_0)$ is undefined. I remedy this problem by assuming that $v_{i,0} = v_{0,0}$ for $i < 0$ and $v_{i,0} = v_{N-1,0}$ for $i > N-1$. The same restriction is made for other scales: $v_{i,j} = v_{0,j}$ for $i < 0$ and $v_{i,j} = v_{N-1,j}$ for $i > N-1$. Other methods could be used instead for handling the boundary conditions and the occurrence of sample positions being outside the image. For example, one could apodize the boundary values as a way of extending the image outside the sample grid.

5.2.2 Stability of the Method

In equation (5.4), try a separation of variables by $v_j(x) = \alpha_j f(x)$ where $f(x)$ is a non–constant bounded function and $\alpha_0 = 1$. The equation becomes

$$\frac{\alpha_{j+1}}{\alpha_j} = 1 + \frac{\ln b}{h^2}\left(\frac{f(x+h\sigma_j) - 2f(x) + f(x-h\sigma_j)}{f(x)}\right).$$

Since the left–hand side is independent of x for all j,

$$\frac{\partial}{\partial x}\left(\frac{f(x+h\sigma_j) - 2f(x) + f(x-h\sigma_j)}{f(x)}\right) \equiv 0$$

for all $x \in \mathbb{R}$ and $\sigma \geq \sigma_0$. A solution is $f(x) = \sin(\lambda x)$ for any constant $\lambda \neq 0$. Moreover, for this choice of f,

$$\frac{\alpha_{j+1}}{\alpha_j} = 1 - \frac{4\ln b}{h^2}\sin^2\left(\frac{\lambda h \sigma_j}{2}\right), \quad \alpha_0 = 1,$$

which has the solution

$$\alpha_n = \prod_{j=0}^{n-1}\left[1 - \frac{4\ln b}{h^2}\sin^2\left(\frac{\lambda h\sigma_j}{2}\right)\right]$$

for $n \geq 1$. To guarantee that the numerical method is stable, in fact asymptotically stable, it is necessary that $\alpha_n \to 0$ as $n \to \infty$. This limiting condition is ensured if $0 < \ln(b)/h^2 < 1/2$, in which case all the terms of the product are bounded away from 1 in magnitude.

For d spatial variables with the same grid sizes h, a similar construction will yield the stability condition $0 < \ln(b)/h^2 < 1/(2d)$. If the spatial measurements are in pixel units, $h = 1$, then $b < \exp(1/2) \doteq 1.649$ for dimension 1, $b < \exp(1/4) \doteq 1.284$ for dimension 2, and $b < \exp(1/6) \doteq 1.181$ for dimension 3.

5.2.3 Bounds on Approximation Error

I now construct bounds on the approximation errors made by using equations (5.3) and (5.5). Let $\alpha = \sqrt{\sigma^2 - \sigma_0^2}$ and $w(x, \alpha) = u(x, \sigma)$; then w is a solution to $w_\alpha = \alpha w_{xx}$ for $\alpha > 0$ and $x \in \mathbb{R}$, with initial condition $w(x, 0) = I(x)$, where I represents the input image. The solution to the differential equation is the integral convolution

$$w(x, \alpha) = \int_{\mathbb{R}} \frac{1}{\sqrt{2\pi}\alpha} \exp\left(-\frac{y^2}{2\alpha^2}\right) I(x - y)\, dy. \tag{5.6}$$

By equation (5.1), the error made in using the finite difference approximation for $\sigma u_\sigma(x, \sigma)$ is bounded by a general bound obtained for $|\sigma^2 u_{\sigma\sigma} + \sigma u_\sigma|$. Note that $\sigma^2 u_{\sigma\sigma} + \sigma u_\sigma = \sigma^4 u_{xxxx} + 2\sigma^2 u_{xx}$, so only bounds need be obtained bounds on the second– and fourth–order spatial derivatives of u. By equation (5.2), the error made in using the finite difference for $\sigma^2 u_{xx}$ is bounded by a general bound obtained for $|\sigma^4 u_{xxxx}|$.

From the change of variables follows the identities $\sigma^2 u_{xx} = \sigma^2 w_{xx}$ and $\sigma^4 u_{xxxx} = \sigma^4 w_{xxxx}$. Differentiating equation (5.6) twice, multiplying by σ^2, and setting $z = y/\alpha$ yields

$$\sigma^2 w_{xx} = \int_{x-r}^{x+r} \frac{\sigma^2}{\sqrt{2\pi}\alpha^3}(z^2 - 1) \exp(-z^2/2) I(x - y)\, dy,$$

where I have also used the assumption that $I(x)$ is 0 outside of the interval $[-r, r]$. It can be shown that $|(z^2 - 1)\exp(-z^2/2)| \leq 1$. Thus,

$$|\sigma^2 w_{xx}| \leq \frac{2r \max |I|}{\sqrt{2\pi}\sigma} \left(\frac{\sigma}{\sqrt{\sigma^2 - \sigma_0^2}}\right)^3. \tag{5.7}$$

Similarly, taking more derivatives yields

$$\sigma^4 w_{xxxx} = \int_{x-r}^{x+r} \frac{\sigma^4}{\sqrt{2\pi}\alpha^5}(z^4 - 6z^2 + 3) \exp(-z^2/2) I(x - y)\, dy.$$

It can also be shown that $|(z^4 - 6z^2 + 3)\exp(-z^2/2)| \leq 3$, so

$$|\sigma^4 w_{xxxx}| \leq \frac{6r \max |I|}{\sqrt{2\pi}\sigma} \left(\frac{\sigma}{\sqrt{\sigma^2 - \sigma_0^2}}\right)^5. \tag{5.8}$$

The approximation error in equation (5.1) is therefore bounded by

$$\left|\sigma u_\sigma(x, \sigma) - \frac{u(x, b\sigma) - u(x, \sigma)}{\ln b}\right| \leq C\frac{\ln b}{\sigma}\left[\left(\frac{\sigma}{\sqrt{\sigma^2 - \sigma_0^2}}\right)^3 + 3\left(\frac{\sigma}{\sqrt{\sigma^2 - \sigma_0^2}}\right)^5\right]$$

where $C = r \max|I|/\sqrt{2\pi}$. The approximation error in equation (5.1) is bounded by

$$\left|\sigma^2 u_{xx}(x,\sigma) - \frac{u(x+h\sigma,\sigma) - 2u(x,\sigma) + u(x-h\sigma,\sigma)}{h^2}\right| \leq C\frac{h^2}{2\sigma}\left(\frac{\sigma}{\sqrt{\sigma^2-\sigma_0^2}}\right)^5.$$

The approximation error of the method in equation (5.3) is $O(\ln b)$ in scale discretization and $O(h^2)$ in space discretization. Note that as σ gets large, the derivative approximations become increasingly better. For σ near σ_0, a different argument can be used to get tighter bounds.

The bound on approximation error in equation (5.5) can be found in any standard numerical analysis which discusses Lagrange interpolation of polynomials. The interpolation errors are bounded by

$$|v(x_i + h\sigma_j) - p(i+\sigma_j)| \leq \frac{\max|I^{(4)}|h^4}{24} \text{ and } |v(x_i - h\sigma_j) - q(i-\sigma_j)| \leq \frac{\max|I^{(4)}|h^4}{24}.$$

A bound on the additional error in approximating the finite difference for $\sigma^2 u_{xx}$ by a cubic interpolation is $\max|I^{(4)}|h^2/24$ since the denominator of the finite difference cancels h^2. This additional error is $O(h^2)$, the same order as that of the original finite difference approximation.

5.2.4 Timing

A simple test on a fast workstation (phong, a DEC 5500) for a $256 \times 256 \times 128$ data set showed that for a single step Gaussian blurring via a general purpose FFT program (`/usr/image/bin.MIPS/gauss`) took about 1 hour, which included some swap time, but the finite difference scheme as described took about 5 minutes. For larger scales, the finite difference scheme must be iterated. As long as the number of iterations keeps the total time less than that for the FFT program, the algorithm is effective. Of course, the tradeoff is that the FFT generally has smaller errors than the finite difference method. But the real measure needed for processing large data sets is the *cost*, which balances error and execution time.

5.3 Solving Eigensystems

The height definition and nonmetric definition require solving eigenvalue problems of the form $A\vec{v} = \lambda\vec{v}$, where A is a real, symmetric matrix. Standard packages may be used; in particular, for general dimensions I use routines from Numerical Recipes in C (Press, Flannery, Teukolsky & Vetterling 1988). I use the routine `tred2` for reduction of the matrix to tridiagonal form, followed by `tqli` for computing eigenstuff for a tridiagonal matrix. However, for dimensions 2 through 4, I have written special reduction routines which run about 3 times faster than `tred2`.

The principal direction definition and level definition require solving *generalized eigenvalue* problems of the form $A\vec{v} = \lambda B\vec{v}$ where A and B are both real, symmetric matrices. The matrix B turns out to be positive definite, so the problem could be transformed to a regular eigenvalue problem via a Cholesky decomposition. However, B has a very special form which allows me to make an even simpler transformation using orthogonal matrices.

5.3.1 Eigenstuff for $A\vec{v} = \lambda\vec{v}$

The standard way to solve $A\vec{v} = \lambda\vec{v}$ is to compute an orthogonal matrix Q such that $T = Q^tAQ$ is tridiagonal. The equivalent eigensystem is $T\vec{w} = \lambda\vec{w}$ where $\vec{w} = Q^t\vec{v}$. Such systems are fairly easy to solve numerically since roots of $\det(T - \lambda I) = 0$ can be bounded *a priori* and then located using standard root–finding techniques.

The reduction $T = Q^tAQ$ is obtained where Q is a product of a sequence of $n - 2$ Householder transformations, where n is the size of the matrix. Since A is symmetric, the matrix T is symmetric, so only the main diagonal and the adjacent subdiagonal need to be computed. A description of the algorithm is given in (Press et al. 1988). The routine which implements the reduction is

```
void NRC_tred2 (int n, float **mat, float *diag, float *subd)

// input:   n = size of matrix
//          mat = nxn real, symmetric A
// output:  mat = orthogonal Q
//          diag = diagonal entries of tridiagonal T, diag[0..n-1]
//          subd = subdiagonal entries of T, subd[0..n-2]

{
```

```
    int i, j, k, ell;

    for (i = n-1, ell = n-2; i >= 1; i--, ell--) {
        float h = 0, scale = 0;

        if ( ell > 0 ) {
            for (k = 0; k <= ell; k++)
                scale += fabs(mat[i][k]);
            if ( scale == 0 )
                subd[i] = mat[i][ell];
            else {
                for (k = 0; k <= ell; k++) {
                    mat[i][k] /= scale;
                    h += mat[i][k]*mat[i][k];
                }
                float f = mat[i][ell];
                float g = ( f > 0 ? -sqrt(h) : sqrt(h) );
                subd[i] = scale*g;
                h -= f*g;
                mat[i][ell] = f-g;
                f = 0;
                for (j = 0; j <= ell; j++) {
                    mat[j][i] = mat[i][j]/h;
                    g = 0;
                    for (k = 0; k <= j; k++)
                        g += mat[j][k]*mat[i][k];
                    for (k = j+1; k <= ell; k++)
                        g += mat[k][j]*mat[i][k];
                    subd[j] = g/h;
                    f += subd[j]*mat[i][j];
                }
                float hh = f/(h+h);
                for (j = 0; j <= ell; j++) {
                    f = mat[i][j];
                    subd[j] = g = subd[j] - hh*f;
                    for (k = 0; k <= j; k++)
                        mat[j][k] -= f*subd[k]+g*mat[i][k];
                }
            }
        }
        else
            subd[i] = mat[i][ell];

    diag[i] = h;
}

diag[0] = subd[0] = 0;
for (i = 0, ell = -1; i <= n-1; i++, ell++) {
    if ( diag[i] ) {
        for (j = 0; j <= ell; j++) {
            float sum = 0;
```

```
                for (k = 0; k <= ell; k++)
                    sum += mat[i][k]*mat[k][j];
                for (k = 0; k <= ell; k++)
                    mat[k][j] -= sum*mat[k][i];
            }
        }
        diag[i] = mat[i][i];
        mat[i][i] = 1;
        for (int j = 0; j <= ell; j++)
            mat[j][i] = mat[i][j] = 0;
    }

    // re-ordering if NR_tqli is used subsequently
    for (i = 1, ell = 0; i < n; i++, ell++)
        subd[ell] = subd[i];
    subd[n-1] = 0;
}
```

I modified the code so that array indexing starts at 0 rather than at 1. I also have the first $n-1$ entries of `subd` storing the subdiagonal values rather than the last $n-1$ (`subd[1..n-1]`) as in NRC. In the book, routine `tred2` returns the subdiagonal entries in the last $n-1$ positions. The code `tqli` for finding eigenstuff of tridiagonal matrices rotates the input subdiagonal entries so that the first $n-1$ entries are valid, and the last entry is arbitrary. I placed the rotation in `tred2` because I have my own reduction code which can be called in place of `tred2`, so there would be no reason to waste time rotating in `tqli`.

The computation of eigenstuff for tridiagonal T is accomplished by factoring $T = QL$ where Q is orthogonal and L is lower triangular. The diagonal entries of L are the eigenvalues of T and the columns of Q are the corresponding eigenvectors. The routine below is found in (Press et al. 1988) and uses implicit shifting to accelerate the convergence and avoid loss of precision of the eigenvalues.

```
void NRC_tqli (int n, float *diag, float *subd, float **mat)

// input:   n = size of matrix
//          diag = diagonal entries of tridiagonal T, diag[0..n-1]
//          subd = subdiagonal entries of T, subd[0..n-2]
//          mat = the output matrix from NRC_tred2 if the eigenstuff
//                of real, symmetric A is desired (run NRC_tred2 on A
//                first), otherwise just the identity matrix if
//                eigenstuff desired for T only
// output:  diag = eigenvalues of T (and of A if NRC_tred2 used first)
//          mat = eigenvectors of T (or of A if NRC_tred2 used first),
//                column k of mat is an eigenvector for the eigenvalue
```

```
//                diag[k]

{
    const int eigen_maxiter = 30;

    for (int ell = 0; ell < n; ell++) {
        for (int iter = 0; iter < eigen_maxiter; iter++) {
            for (int m = ell; m <= n-2; m++) {
                float dd = fabs(diag[m])+fabs(diag[m+1]);
                if ( (float)(fabs(subd[m])+dd) == dd )
                    break;
            }
            if ( m == ell )
                break;

            float g = (diag[ell+1]-diag[ell])/(2*subd[ell]);
            float r = sqrt(g*g+1);
            if ( g < 0 )
                g = diag[m]-diag[ell]+subd[ell]/(g-r);
            else
                g = diag[m]-diag[ell]+subd[ell]/(g+r);
            float s = 1, c = 1, p = 0;
            for (int i = m-1; i >= ell; i--) {
                float f = s*subd[i], b = c*subd[i];
                if ( fabs(f) >= fabs(g) ) {
                    c = g/f;
                    r = sqrt(c*c+1);
                    subd[i+1] = f*r;
                    c *= (s = 1/r);
                }
                else {
                    s = f/g;
                    r = sqrt(s*s+1);
                    subd[i+1] = g*r;
                    s *= (c = 1/r);
                }
                g = diag[i+1]-p;
                r = (diag[i]-g)*s+2*b*c;
                p = s*r;
                diag[i+1] = g+p;
                g = c*r-b;

                for (int k = 0; k < n; k++) {
                    f = mat[k][i+1];
                    mat[k][i+1] = s*mat[k][i]+c*f;
                    mat[k][i] = c*mat[k][i]-s*f;
                }
            }
            diag[ell] -= p;
            subd[ell] = g;
            subd[m] = 0;
```

```
            }
            if ( iter == eigen_maxiter ) {
                printf("NR_tqli - exceeded maximum iterations");
                exit(1);
            }
        }
    }
```

As before, all array indexing has been modified to start at 0 rather than at 1.

For my applications I need the eigenvalues to be sorted. The following routine is a selection sort. The order of the parameters is different than in NRC.

```
void NRC_eigsrt (int n, float *eigval, float**eigvec)

// input:   eigval = eigenvalues in arbitrary order
//          eigvec = column k of matrix is eigenvector for eigval[k]
// output:  eigval = eigenvalues in descending order, maximum in
//                   eigval[0], minimum in eigval[n-1]
//          eigvec = column k of matrix is eigenvector for eigval[k]

{
    // sort eigenvalues so that eigval[i] >= eigval[j] for i < j
    for (int i = 0, k; i <= n-2; i++) {
        float max = eigval[k=i];
        for (int j = i+1; j < n; j++)
            if ( eigval[j] > max )
                max = eigval[k=j];
        if ( k != i ) {
            // swap eigenvalues
            eigval[k] = eigval[i];
            eigval[i] = max;

            // swap eigenvectors
            for (j = 0; j < n; j++) {
                float tmp = eigvec[j][i];
                eigvec[j][i] = eigvec[j][k];
                eigvec[j][k] = tmp;
            }
        }
    }
}
```

Array indexing also starts at 0 rather than at 1.

5.3.2 Symbolic Tridiagonalization

The majority of my image analysis is on images of dimensions 2, 3, and 4. I worked out the Householder reductions for tridiagonalization symbolically and then implemented them

N	time for NRC_tred2	time for tridiag_N
2	9.23	5.99
3	43.52	15.33
4	99.12	27.98

Table 5.1: Timing for numeric versus symbolic tridiagonalization

to see if they were faster. The routines are `tridiag_N` for $N = 2, 3, 4$. I did profiling by iterating $100,000$ times a block of code which assigned a matrix its values, tridiagonalized (using `NRC_tred2` or `tridiag_N`), and computed eigenstuff with `NRC_tqli`. I did the timing on an Intel 80486 at 33 MHz running under Microsoft Windows 3.1. Table 5.1 gives the timing information where the times are total seconds of execution time as measured by using the `clock()` routine in `time.h`. The reduction routines were timed on matrices of the form $M = [m_{ij}]$ where $m_{ij} = i + j + 1$.

Reduction of 2×2 matrices

Since a 2×2 matrix is already tridiagonal, there is nothing to do theoretically. However, the implementation must do the assignment of numbers to the actual parameters needed for further processing. The method which does this is

```
void tridiag_2 (float **mat, float *diag, float *subd)

// input:   mat = 2x2 real, symmetric A
// output:  mat = identity matrix I
//          diag = diagonal entries of A, diag[0] = a00, diag[1] = a11
//          subd = subdiagonal entry of A, subd[0] = a01

{
    // matrix is already tridiagonal

    diag[0] = mat[0][0];
    diag[1] = mat[1][1];
    subd[0] = mat[0][1];
    subd[1] = 0;
    mat[0][0] = 1;  mat[0][1] = 0;
    mat[1][0] = 0;  mat[1][1] = 1;
}
```

Reduction of 3×3 matrices

There are many ways to reduce A to tridiagonal form. I used a Householder transformation which does a rotation and a reflection in the x_1x_2–plane, where $\vec{x} = (x_0, x_1, x_2)$. Let the matrix entries be labeled

$$A = \begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix}.$$

If $c = 0$, the matrix is already tridiagonal, so the orthogonal transformation is just the identity matrix $Q = I$. If $c \neq 0$, a Householder transformation Q and corresponding tridiagonal matrix $T = Q^t A Q$ are

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & u & v \\ 0 & v & -u \end{bmatrix} \quad \text{and} \quad T = \begin{bmatrix} a & L & 0 \\ L & d + vq & e - uq \\ 0 & e - uq & f - vq \end{bmatrix}$$

where $L = \sqrt{b^2 + c^2}$, $u = b/L$, $v = c/L$, and $q = 2ue + v(f - d)$. The method for the reduction is

```
void tridiag_3 (float **mat, float *diag, float *subd)

// input:   mat = 3x3 real, symmetric A
// output:  mat = orthogonal matrix Q
//          diag = diagonal entries of T, diag[0,1,2]
//          subd = subdiagonal entry of T, subd[0,1]

{
    float a = mat[0][0], b = mat[0][1], c = mat[0][2],
                         d = mat[1][1], e = mat[1][2],
                                        f = mat[2][2];

    diag[0] = a;
    subd[2] = 0;
    if ( c != 0 ) {
        float ell = sqrt(b*b+c*c);
        b /= ell;
        c /= ell;
        float q = 2*b*e+c*(f-d);
        diag[1] = d+c*q;
        diag[2] = f-c*q;
        subd[0] = ell;
        subd[1] = e-b*q;
        mat[0][0] = 1; mat[0][1] = 0; mat[0][2] = 0;
```

```
            mat[1][0] = 0; mat[1][1] = b; mat[1][2] = c;
            mat[2][0] = 0; mat[2][1] = c; mat[2][2] = -b;
        }
        else {
            diag[1] = d;
            diag[2] = f;
            subd[0] = b;
            subd[1] = e;
            mat[0][0] = 1; mat[0][1] = 0; mat[0][2] = 0;
            mat[1][0] = 0; mat[1][1] = 1; mat[1][2] = 0;
            mat[2][0] = 0; mat[2][1] = 0; mat[2][2] = 1;
        }
    }
```

Reduction of 4×4 matrices

The reduction of a 4×4 matrix is more complicated. The orthogonal transformation is a motion in the $x_1x_2x_3$–plane where the general 4– dimensional coordinates are $\vec{x} = (x_0, x_1, x_2, x_3)$. Let the matrix entries be labeled

$$A = \begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix} = \left[\begin{array}{c|c} a & \vec{v}^t \\ \hline \vec{v} & S \end{array}\right]$$

where the right–hand side is in block matrix form where $\vec{v}$ is a 3×1 column vector and S is a symmetric 3×3 submatrix of A. I seek an orthogonal transformation in block form $Q = \text{diag}(1, P)$ where 1 is a scalar, P is a 3×3 orthogonal matrix, and $T = Q^tAQ$ is tridiagonal. The tridiagonal matrix T is then given by

$$T = Q^tAQ = \left[\begin{array}{c|c} a & \vec{v}^tP \\ \hline P^t\vec{v} & P^tSP \end{array}\right].$$

Let the columns of P be labeled $\vec{p}_k$, $k = 1, 2, 3$. For T to be tridiagonal, $\vec{v}^tP$ must be parallel to $(1, 0, 0)$. This implies that $\vec{v}$ is orthogonal to both $\vec{p}_2$ and $\vec{p}_3$. If $\vec{v} = (0, 0, 0)$, choose $\vec{p}_1 = (1, 0, 0)$; otherwise, set $\vec{p}_1 = \vec{v}/|\vec{v}|$.

Also for T to be tridiagonal, the upper right–hand corner entry of P^tSP must be 0; that is, $\vec{p}_1^{\,t}S\vec{p}_3 = 0$. This implies that the vector $S\vec{p}_1$ is in the plane of $\vec{p}_1$ and $\vec{p}_2$. If $S\vec{p}_1$ is parallel to $\vec{p}_1$ (*i.e.* $\vec{p}_1$ is an eigenvector for S), then choose any $\vec{p}_2$ and $\vec{p}_3$ such that P is

orthogonal. If $\vec{p}_1 = (\alpha, \beta, \gamma)$, choose

$$P = \begin{bmatrix} \alpha & -\beta & -\gamma \\ \beta & 1 + \frac{(\alpha-1)\beta^2}{\sqrt{\beta^2+\gamma^2}} & \frac{(\alpha-1)\beta\gamma}{\sqrt{\beta^2+\gamma^2}} \\ \gamma & \frac{(\alpha-1)\beta\gamma}{\sqrt{\beta^2+\gamma^2}} & 1 + \frac{(\alpha-1)\gamma^2}{\sqrt{\beta^2+\gamma^2}} \end{bmatrix}$$

if $\sqrt{\beta^2 + \gamma^2} \neq 0$. If this square root is zero, simply choose $\vec{p}_2 = (0, 1, 0)$ and $\vec{p}_3 = (0, 0, 1)$. If $S\vec{p}_1$ is not parallel to $\vec{p}_1$, choose

$$\vec{p}_3 = \frac{\vec{p}_1 \times S\vec{p}_3}{|\vec{p}_1 \times S\vec{p}_3|} \quad \text{and} \quad \vec{p}_2 = \vec{p}_3 \times \vec{p}_1.$$

After constructing P, the code also includes construction of the entries for $P^t S P$ which is itself a tridiagonal matrix, so only 5 values need be computed. The method for the reduction is

```
void tridiag_4 (float **mat, float *diag, float *subd)

// input:   mat = 4x4 real, symmetric A
// output:  mat = orthogonal matrix Q
//          diag = diagonal entries of T, diag[0,1,2,3]
//          subd = subdiagonal entry of T, subd[0,1,2]

{
    // save matrix M
    float a = mat[0][0], b = mat[0][1], c = mat[0][2], d = mat[0][3],
                         e = mat[1][1], f = mat[1][2], g = mat[1][3],
                                        h = mat[2][2], i = mat[2][3],
                                                       j = mat[3][3];

    diag[0] = a;
    subd[3] = 0;

    mat[0][0] = 1; mat[0][1] = 0; mat[0][2] = 0; mat[0][3] = 0;
    mat[1][0] = 0;
    mat[2][0] = 0;
    mat[3][0] = 0;

    if ( c != 0 || d != 0 ) {
        float q11, q12, q13;
        float q21, q22, q23;
        float q31, q32, q33;

        // build column Q1
        float len = sqrt(b*b+c*c+d*d);
        q11 = b/len;
```

```
q21 = c/len;
q31 = d/len;

subd[0] = len;

// compute S*Q1
float v0 = e*q11+f*q21+g*q31;
float v1 = f*q11+h*q21+i*q31;
float v2 = g*q11+i*q21+j*q31;

diag[1] = q11*v0+q21*v1+q31*v2;

// build column Q3 = Q1x(S*Q1)
q13 = q21*v2-q31*v1;
q23 = q31*v0-q11*v2;
q33 = q11*v1-q21*v0;
len = sqrt(q13*q13+q23*q23+q33*q33);
if ( len > 0 ) {
    q13 /= len;
    q23 /= len;
    q33 /= len;

    // build column Q2 = Q3xQ1
    q12 = q23*q31-q33*q21;
    q22 = q33*q11-q13*q31;
    q32 = q13*q21-q23*q11;

    v0 = q12*e+q22*f+q32*g;
    v1 = q12*f+q22*h+q32*i;
    v2 = q12*g+q22*i+q32*j;
    subd[1] = q11*v0+q21*v1+q31*v2;
    diag[2] = q12*v0+q22*v1+q32*v2;
    subd[2] = q13*v0+q23*v1+q33*v2;

    v0 = q13*e+q23*f+q33*g;
    v1 = q13*f+q23*h+q33*i;
    v2 = q13*g+q23*i+q33*j;
    diag[3] = q13*v0+q23*v1+q33*v2;
}
else {  // S*Q1 parallel to Q1, choose any valid Q2 and Q3
    subd[1] = 0;

    len = q21*q21+q31*q31;
    if ( len > 0 ) {
        float tmp = q11-1;
        q12 = -q21;
        q22 = 1+tmp*q21*q21/len;
        q32 = tmp*q21*q31/len;

        q13 = -q31;
        q23 = q32;
```

```
            q33 = 1+tmp*q31*q31/len;

            v0 = q12*e+q22*f+q32*g;
            v1 = q12*f+q22*h+q32*i;
            v2 = q12*g+q22*i+q32*j;
            diag[2] = q12*v0+q22*v1+q32*v2;
            subd[2] = q13*v0+q23*v1+q33*v2;

            v0 = q13*e+q23*f+q33*g;
            v1 = q13*f+q23*h+q33*i;
            v2 = q13*g+q23*i+q33*j;
            diag[3] = q13*v0+q23*v1+q33*v2;
        }
        else {  // Q1 = (+-1,0,0)
            q12 = 0; q22 = 1; q32 = 0;
            q13 = 0; q23 = 0; q33 = 1;

            diag[2] = h;
            diag[3] = j;
            subd[2] = i;
        }
    }

    mat[1][1] = q11; mat[1][2] = q12; mat[1][3] = q13;
    mat[2][1] = q21; mat[2][2] = q22; mat[2][3] = q23;
    mat[3][1] = q31; mat[3][2] = q32; mat[3][3] = q33;
}
else {
    diag[1] = e;
    subd[0] = b;
    mat[1][1] = 1;
    mat[2][1] = 0;
    mat[3][1] = 0;

    if ( g != 0 ) {
        float ell = sqrt(f*f+g*g);
        f /= ell;
        g /= ell;
        float Q = 2*f*i+g*(j-h);

        diag[2] = h+g*Q;
        diag[3] = j-g*Q;
        subd[1] = ell;
        subd[2] = i-f*Q;
        mat[1][2] = 0;  mat[1][3] = 0;
        mat[2][2] = f;  mat[2][3] = g;
        mat[3][2] = g;  mat[3][3] = -f;
    }
    else {
        diag[2] = h;
        diag[3] = j;
```

```
                subd[1] = f;
                subd[2] = i;
                mat[1][2] = 0;  mat[1][3] = 0;
                mat[2][2] = 1;  mat[2][3] = 0;
                mat[3][2] = 0;  mat[3][3] = 1;
            }
        }
    }
```

5.3.3 Eigenstuff for $A\vec{v} = \lambda B\vec{v}$

If the matrix B is invertible, a simple reduction to regular eigenvalue problems is $B^{-1}A\vec{v} = \lambda\vec{v}$. However, the matrix $B^{-1}A$ may not be symmetric, so more general numerical schemes are needed for computing eigenstuff, for example, reduction to Hessenberg form followed by inverse iteration.

The case of interest is when both A and B are symmetric. In the principal direction ridge construction, B corresponds to the first fundamental form and A corresponds to the second fundamental form, both of which are symmetric. The same type of generalized eigenvalue problem arises in statistical pattern recognition where the two matrices represent *scatter within clusters* and *scatter between clusters*. Generally the cases to consider are

- B is positive definite (all eigenvalues are positive),
- B is nonnegative definite (all eigenvalues are nonnegative),
- B is indefinite (some eigenvalues positive, some negative).

The matrix B could also be negative definite or nonpositive definite, but in that case you can multiply the eigensystem by -1 to obtain the cases above. The same cases hold for matrix A if you consider instead the system $B\vec{v} = (1/\lambda)A\vec{v}$. Thus, if either A or B is positive definite, use the associated method on the corresponding eigensystem.

A positive definite matrix B can be factored into $B = M^tM$ for some square matrix M, called the *Cholesky decomposition*. Usually the factorization is done via the LDU decomposition, $B = LDU$ where L is lower triangular, D is diagonal, and U is upper triangular. Since B is positive definite, $L = U^t$ and D has all positive diagonal terms. Thus, $M = \sqrt{D}U$ where $\sqrt{D}$ is the diagonal matrix whose diagonal terms are the square roots of the diagonal

terms of D. The eigensystem can be rewritten as

$$C\vec{w} = \left(M^{-t}AM^{-1}\right)(M\vec{v}) = \lambda(M\vec{v}) = \lambda\vec{w}.$$

The matrix C is symmetric, so the eigenstuff construction described earlier can be used to compute eigenvalues λ and eigenvectors $\vec{w}$. The generalized eigenvectors to the original system are obtained by computing $\vec{v} = M^{-1}\vec{w}$.

In the principal direction definition for ridges, the principal curvatures λ and principal directions $\vec{v}$ are determined by the generalized eigensystem $A\vec{v} = \lambda B\vec{v}$, where $A = \mathrm{H}(f)/\sqrt{1+|\nabla f|^2}$ in Euclidean space or $A = \frac{1}{2}\mathrm{H}(|\vec{x}|^2 + f^2)/\sqrt{1+|\nabla f|^2}$ in scale space, and where $B = I + \nabla f \nabla f^t$ when constructing ridges in Euclidean space or scale space (assuming no spatial distortion). The special form for B allows a different decomposition for B. Let $\vec{d} = \nabla f(x)$. Let Q be an orthogonal matrix such that $Q^t\vec{d} = |\vec{d}|\vec{u}$, where $\vec{u}$ is the vector whose components are all 0 except for the last one which is 1; then $Q^tBQ = \mathrm{diag}(1,\ldots,1,1+|\vec{d}|^2)$. Define $D = \mathrm{diag}(1,\ldots,1,(1+|\vec{d}|^2)^{-1/2})$. The eigensystem now becomes

$$C\vec{w} = \left(D^tQ^tAQD\right)\left(D^{-1}Q^t\vec{v}\right) = \lambda\left(D^{-1}Q^t\vec{v}\right) = \lambda\vec{w}.$$

Again, C is symmetric, so the regular eigenstuff construction can be used. The matrix Q may be chosen as follows. Let $\vec{d} = (d_1,\ldots,d_n)$ and define $\vec{\alpha} = (d_1,\ldots,d_{n-1})$. If $\vec{\alpha} = \vec{0}$, just choose Q to be the identity matrix. Otherwise, define $\vec{\beta} = \vec{\alpha}/|\vec{\alpha}|$ and let

$$Q = \left[\begin{array}{c|c} I + (d_n - 1)\vec{\beta}\vec{\beta}^t & \vec{\alpha} \\ \hline -\vec{\alpha} & d_n \end{array}\right],$$

where I is the $(n-1)\times(n-1)$ identity matrix.

5.4 Skeletons of Objects

Ridges were defined as points which were zeros to certain systems of equations. My segmentation algorithm requires 1–dimensional ridge structures. In the discrete algorithm, candidate ridges were identified as those points for which the defining equations showed sign changes rather than those points for which the equations yielded zeros. The candidate ridges form a binary set which is generally more than 1 pixel thick, but appears to consist of curvilinear

components. I needed a thinning algorithm which would take the candidates and reduce the components to the thickness of 1 pixel. The thinning algorithm makes use of distance transforms and connected component labeling. I discuss the implementations of these algorithms first. A description of the thinning algorithm is given with an example of how it works on a 2–dimensional binary object.

5.4.1 Distance Transforms

In order to thin the candidate ridges in a way that preserves the general geometric shape, I wanted to thin from the outside to inside of the sets. This requires computing a distance transform of the original binary set so that pixels are labeled with their distances from the boundary of the set. I have implemented a distance transform for the L_1 metric where the distance between two points is the maximum of the absolute differences of the individual coordinates. The algorithm computes the transform in–place. The routine takes as input an n–dimensional binary image whose pixels are labeled as 0 (outside an object) or 1 (inside an object). The input image may have multiple binary objects. Each of the output image pixels has an integer value corresponding to its distance to the nearest object boundary pixel of the object which contains it. The distances are measured in pixel units.

The L_1 distance transform is slow for large binary objects. If D is the maximum distance from a pixel to its component's boundary, and if the image has N pixels, the routine is $O(DN)$. If the binary objects are small and sparse, the routine runs much faster. I believe that I have already optimized the transform as much as possible. The algorithm is essentially a simulation of a recursive routine. On the first pass, any pixel with value 1 and whose $2n$ neighbors one unit distance away all have value 1 is assigned the value 2. On completion of the pass, the boundary pixels of objects have value 1, and the interior pixels of objects have value 2. The same labeling scheme is applied to the subobjects which have all pixel values 2. The boundary pixels of the subobjects have value 2 and the interior pixels have value 3. The process is repeated until the pixels of maximum distance are reached. The C pseudocode for the L_1 distance transform is given below.

```
Input:   binary image, im
Output:  L1 distance transform, im  (in-place algorithm)
```

```
for (change_made = TRUE, value = 1; change_made; value++) {
    change_made = FALSE;
    for (each pixel x with im(x) == value) {
        if (all y with L1_dist(x,y) == 1 satisfy im(y) >= value) {
            im(x) = value+1;
            change_made = TRUE;
        }
    }
}
```

I have also implemented a distance transform for the L_2 metric which is the usual Euclidean distance metric. This transform approximates the true Euclidean distance transform. It is based on integer arithmetic, and distances from a pixel to its adjacent neighbors (as a fully connected graph) are approximated by rational numbers. The real distances are $1, \sqrt{2}, \ldots, \sqrt{n}$. For example, at pixel $x \in \mathbb{R}^3$, the distance to neighbor $x + (1,0,0)$ is 1 unit, the distance to neighbor $x+(1,1,0)$ is $\sqrt{2}$ units, and the distance to neighbor $x+(1,1,1)$ is $\sqrt{3}$ units. The rational number approximations are based on a user–specified denominator, call it D, whose default value is 10. The numerator N_d for the approximation to $\sqrt{d}$, $1 \leq d \leq n$, is determined at run–time to be the positive integer which minimizes $|N_d - D\sqrt{d}|$. In the previous example with $D = 10$, we have $N_1 = 10$, $N_2 = 14$ and $N_3 = 17$. We additionally define $N_0 = 0$ as an aid in implmenting the algorithm.

The algorithm takes as input an n–dimensional binary image whose pixels are labeled as 0 (outside an object) or 1 (inside an object). The input may have multiple binary objects. The algorithm computes the transform using the original image and a temporary buffer, but the transform values are stored in the input image buffer. The routine is essentially an in–place algorithm. Each of the output image pixels has an integer value proportional to its distance to the nearest object boundary pixel of the object which contains it. The distances are *not* in pixel units.

The input is an n–dimensional binary image. Pixels with initial value 0 retain that value, but pixels with initial value 1 are initialized to `INFINITY`, an integer which is larger than any quantity obtained from the arithmetic operations performed in the algorithm. The algorithm makes multiple passes and updates the distances from the boundary of objects. The algorithm terminates when no changes are made to the current distance values. Let $\rho_k(x)$ be the current distance value at pixel x after k iterations. Let $S(x)$ be the set of

immediate neighbors of x. For each $y \in S(x)$, let $d(y)$ denote the L_1 distance from x to y. Note that $d(x) = 0$ and $1 \leq d(y) \leq n$ for $y \neq x$. The update of distance on each pass is defined by

$$\rho_{k+1}(x) = \min_{y \in S(x)} \left(\rho_k(y) + N_{d(y)} \right).$$

The C pseudocode for the L_2 distance transform is given below.

```
Input:   binary image, im
         N[0..n], numerators for rational approximations of distance to
                  neighbors of pixels
Output:  L2 distance transform, im  (in-place algorithm)

for (each pixel x with im(x) == 1 )
    im(x) = INFINITY;
image temp;  // same dimensions as im, initialized to zero
change_made = TRUE;
while ( change_made ) {
    change_made = FALSE;
    for (each pixel x with im(x) > 0) {
        temp(x) = im(x);
        for (each pixel y adjacent to x)
            if ( im(y)+N[L1_dist(x,y)] < temp(x) )
                temp(x) = im(y)+N[L1_dist(x,y)];
        if ( temp(x) != im(x) )
            change_made = TRUE;
    }
    im = temp;
}
```

5.4.2 Component Labeling

In order to label ridge segments obtained from a 1–dimensional skeleton with unique positive integer values, I need a routine to label connected components in an n–dimensional binary image. The components are treated as fully connected; that is, connections are checked between a pixel and all its $3^n - 1$ neighbors. I have implemented the component labeling using a scan approach. Treating the image as 1–dimensional, all components are labeled. As a prerequisite, the image boundary values must all be zero. To get back to the n–dimensional setting, labels are merged based on how "scan hyperplanes" and labeled pixels are adjacent to each other. All the previously scanned immediate neighbors of a labeled pixel are analyzed and cycles of related labels are created using associative memory. The idea is best illustrated by some examples.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	0	0	1	0
2	0	0	1	0	1	0
3	0	0	1	1	0	0
4	0	0	0	1	0	0
5	0	0	0	0	0	0

binary image

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	0	0	2	0
2	0	0	3	0	4	0
3	0	0	5	5	0	0
4	0	0	0	6	0	0
5	0	0	0	0	0	0

1D labeling

Figure 5.1: Illustration of component labeling

Consider the 6×6 binary image shown in Figure 5.1. The second image is the one obtained after labeling components if the first image was treated as a 1–dimensional image (in row major order). The associative array is initially `a[] = (0,1,2,3,4,5,6)`. The zero component never changes since the background is not associated with any other regions. After the 1–dimensional labeling, the image is scanned a second time. Each step below indicates reaching a positive–labeled pixel and shows the actions taken.

1. $(1,1)$ (label 1) has no previously scanned neighbors. No action taken on `a[]`.
2. $(4,1)$ (label 2) has no previously scanned neighbors. No action taken on `a[]`.
3. $(2,2)$ (label 3) has previously scanned neighbor $(1,1)$ (label 1). Swap `a[1]` and `a[3]`. The array is now `a[] = (0,3,2,1,4,5,6)`. We have a single cycle $\langle 1,3 \rangle$.
4. $(4,2)$ (label 4) has previously scanned neighbor $(4,1)$ (label 2). Swap `a[2]` and `a[4]`. The array is now `a[] = (0,3,4,1,2,5,6)`. We have two cycles $\langle 1,3 \rangle$ and $\langle 2,4 \rangle$.
5. $(2,3)$ (label 5) has previously scanned neighbor $(2,2)$ (label 3). Swap `a[3]` and `a[5]`. The array is now `a[] = (0,3,4,5,2,1,6)`. We have two cycles $\langle 1,3,5 \rangle$ and $\langle 2,4 \rangle$.
6. $(3,3)$ (label 5) has three previously scanned neighbors.
   (a) $(2,2)$ (label 3): The labels 3 and 5 are already part of a cycle in `a[]`, so no action is taken.

(b) $(2,3)$ (label 5): The labels are the same so no action is taken.

(c) $(4,2)$ (label 4): Swap `a[4]` and `a[5]`. The array is now `a[] = (0,3,4,5,1,2,6)`. We now have a single cycle $\langle 1,3,5,2,4 \rangle$.

7. $(3,4)$ (label 6) has previously scanned neighbor $(3,3)$ (label 5). Swap `a[5]` and `a[6]`. The array is now `a[] = (0,3,4,5,2,6,1)`. We have a single cycle $\langle 1,3,5,6,2,4 \rangle$.

Note that it is important *not* to swap array locations if the labels are already part of the *same* cycle. Doing so will not yield the correct component labeling.

After scanning, the associative memory contains disjoint cycles of labels. A pass is made through the memory and values are replaced by their minimum cycle value. This approach to identifying cycles produces a nonconsecutive numbering. Alternatively, number the cycles and replace the array values by the cycle number to get consecutive numbering. Once this has been done, a last pass is made through the image and pixels are relabeled pixels via `im[i] = a[im[i]]`. In the above example we would have a single labeled connected component. The C pseudocode for connected component labeling is given below.

```
Input:  binary image, im  (image boundaries labeled with 0)
Output:  image with labeled components, im  (in-place algorithm)

// scan image as 1D array and label components
component = 0;
for (i = 0; i < im.pixel_count; i++)
    if ( im[i] ) {  // found an object
        component++;
        while ( im[i] == 1 )  // label the object with a component number
            im[i++] = component;
    }

if ( image is 1 dimensional ) return;  // no merging to do

// merge equivalent components, use associate memory
int assoc[component+1];  // memory size = number of components+background
for (i = 0; i < component+1; i++)  // initially components are distinct
    assoc[i] = i;

for (each pixel x with im(x) > 0 ) {
    for (each already scanned pixel y with L1_dist(x,y) == 1) {
        if ( im(y) > 0 && im(x) != im(y) ) {
            // check if component is already in cycle
            search = im(y);
            do {
```

```
                search = assoc[search];
            } while (search != im(x) && search != im(y));

            // add to cycle if necessary
            if (search == im(y)) {
                temp = assoc[im(x)];
                assoc[im(x)] = assoc[im(y)];
                assoc[im(y)] = temp;
            }
        }
    }
}

// replace each cycle by a single label
compact_count = 0;
for (i = 1; i <= component; i++)
    if ( i <= assoc[i] ) {
        compact_count++;
        current = i;
        while ( assoc[current] != i ) {
            next = assoc[current];
            assoc[current] = compact_count;
            current = next;
        }
        assoc[current] = compact_count;
    }
// compact_count = number of connected components

// relabel image
for (i = 0; i < im.pixel_count; i++)
    if ( im[i] )
        im[i] = assoc[im[i]];
```

5.4.3 Thinning Algorithm

The skeletonizing process should preserve the topology of the original set; that is, the number of holes in the original set should be the same as the number in the thinned set. Also, the process should produce a skeleton that is as "central"in the original set as possible. Segmenting the skeleton into curvilines or simple closed curves might also be desired. Classification of a skeleton point can be made based on how many neighbors it has. Isolated points have no neighbors. End points typically have 1 neighbor unless they occur in a small clump, such as a 5 point "plus", in which case end points can have more than 1 neighbor. Interior line points typically have 2 neighbors, and branch points typically have 3 or more neighbors. I segment the ridge set by removing the branch points and labeling the remaining connected

components.

Skeletonizing

The skeletonizing is designed to produce 1–dimensional structures. For a 3–dimensional binary object, such as an ellipsoid, the output of the routines will be a line segment in the direction of the longest axis. The output will not be a 2–dimensional manifold such as the plate contained in the plane of the two largest axes. These structures fall under the category of medial manifolds and are not discussed here.

The binary object is thinned from the outside to inside, one "layer" at a time. On each pass, each interior pixel (in the L_1 distance sense) is marked as 2, so we have a ternary image with exterior pixels marked as 0, object boundary pixels marked as 1, and interior pixels marked as 2. Each object boundary pixel which is adjacent (in the fully connnected sense) to an interior pixel is removed if it does not disconnect the binary object and if it does not create a new hole. Once pixels are removed, the original interior pixels (marked as 2) are reset to 1 to allow the next pass of the trimming.

The test for disconnection involves all the immediate neighbors (fully connected, test all object pixels in the 3^n neighborhood). At each 1–pixel in the binary image a copy is made of the adjacent pixels. The center pixel is removed. The remaining graph is searched for connected components assuming $(3^n - 1)$–connectedness and using a depth first search. The center pixel is labeled as a branch point if and only if there is more than one connected component.

The test for creating a hole by removal of a pixel involves only the neighbors which are one unit distance away ($2n$–connectedness). An inverted copy of the binary image in the immediate neighborhood is made. The center pixel is set to 1. The graph is searched for connected components using a depth first search assuming $2n$–connectedness. The center pixel is labeled as a cork point (if removed, a new hole has been added) if and only if there is more than one connected component.

After this initial trimming step, there may remain some *positive measure* pixels. A 1–pixel is said to be positive measure if it has at least one adjacent 1–pixel for every coordinate direction. For example, in 2 dimensions, the pixel $(0, 0)$, with value 1, is positive measure

when $(0,1)$ or $(0,-1)$ is a 1 and when $(1,0)$ or $(-1,0)$ is a 1. These pixels are not interior pixels of the object, but are "almost" interior in that they can be identified by a structuring element which is "almost" the entire plus symbol template. The removal of positive measure pixels is similar to the original thinning step and is done iteratively until no change is made. In effect, the first trimming step is an erosion using a full–plus structuring element, whereas the second trimming step is an erosion using partial–plus structuring elements at $2n$ different orientations.

Even with the above two trimming steps, some nonessential pixels remain due to the grid geometry. Two additional trimming steps are used. The first removes pixels which have more than 2 neighbors, but are neither branch points nor cork points. The second removes pixels which have more than 1 neighbor, but are not branch points.

Classification of Pixels

After skeletonizing, I classify pixels by doing a neighborhood count. Pixels are labeled as *end points* if they have 0 or 1 neighbor, as *interior points* if they have exactly 2 neighbors, and as *branch points* if they have 3 or more neighbors. This classification is not always accurate, but other than an exhaustive consideration of all possible pixel configurations in a 3^n cube, I see no way to derive an algorithm that gives you always what you want. The pathological problem is illustrated in the 2–dimensional case with a 5–point plus, say pixels at $(0,0)$, $(\pm 1,0)$, and $(0,\pm 1)$. Clearly the center point should be labeled as a branch point, but probably the other four points should be labeled as end points since they appear to be end points of *very* short line segments (too small to detect based on the scale of the grid). My classification scheme labels all 5 pixels as branch points. Perhaps artificially increasing the resolution (by subdivision), applying the same algorithm, and labeling the original pixel with the most frequent type of its subpixels, would work better.

Segmenting the Skeleton

A copy is made of the skeleton minus its branch points. This copy is passed to the connected component labeler so that each curviline or closed curve in the skeleton is uniquely identified (with positive integer labels). An attempt is then made to reinsert the branch points. Each

such point has its neighborhood searched for an already labeled pixel. If one is found, then the branch point inherits its label. If one is not found (the pathological case, such as the 5–point plus), then a new label is created and assigned to the pixel. The skeleton is therefore segmented into small pieces.

Example

In all the figures the periods indicate zero pixel values. The original binary image is shown in the left in Figure 5.2. The right image shows the interior pixels (using 4–connectedness) marked with values 2.

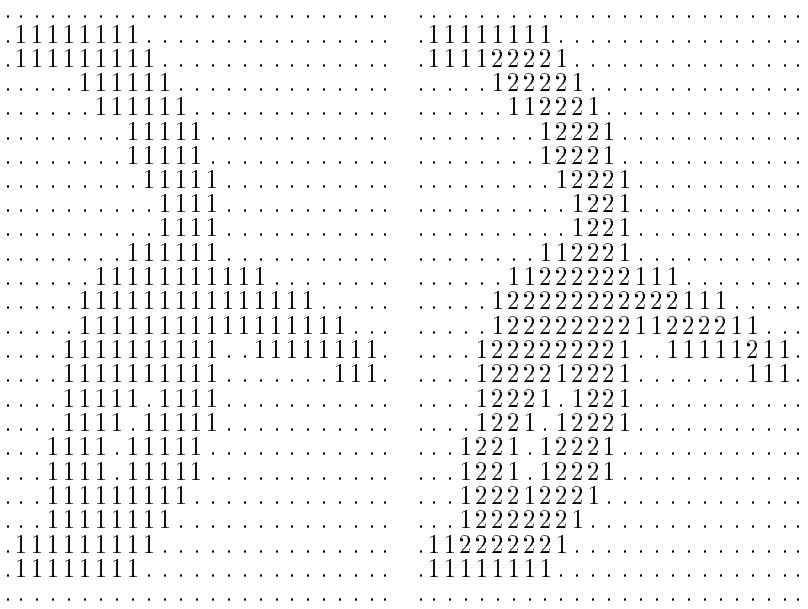

Figure 5.2: Original image, marked image after pass 1

The left image in Figure 5.3 shows the result after trimming those outermost 1–pixels which neither disconnected the set nor created any new holes. The right image was obtained by resetting all the pixels in the left image to 1 and then marking all interior pixels with 2.

The left image in Figure 5.4 shows the result after the next pass of trimming those outermost 1–pixels which neither disconnected the set nor created any new holes. The right image was obtained by resetting all the pixels in the left image to 1 and then marking all interior pixels with 2.

The left image in Figure 5.5 shows the result after the next pass of trimming those outermost 1–pixels which neither disconnected the set nor created any new holes. The right image was obtained by resetting all the pixels in the left image to 1 and then marking all interior pixels with 2.

The left image in Figure 5.6 shows the result after the next pass of trimming those outermost 1–pixels which neither disconnected the set nor created any new holes. The right image was obtained by resetting all the pixels in the left image to 1 and then marking all interior pixels with 2.

The second trimming removes nonessential positive measure pixels without disconnecting the skeleton or creating new holes. The left image of Figure 5.7 shows the result of the first trimming. The right image shows the result after removing positive measure pixels.

The third trimming, removing nonbranch–noncork pixels with 3 or more neighbors, and the fourth trimming, removing nonbranch–noncork pixels with 2 or more neighbors, do nothing in this example. The final skeleton is therefore the last image shown. However, in some simple 3–dimensional examples, both trimmings are needed. I tried the algorithm on a binary object which was the dilated union of two objects. The first was a rectangular solid which was $2 \times 2 \times 16$ in dimension and the second was a thick diagonal line (i, i, i), $(i, i, i+1)$, $(i, i+1, i)$, and $(i, i+1, i+1)$ for 16 different values of i. I arranged for the two objects to intersect, then dilated them. The resulting skeletonization looked good, but the region of intersection still looked a bit thick.

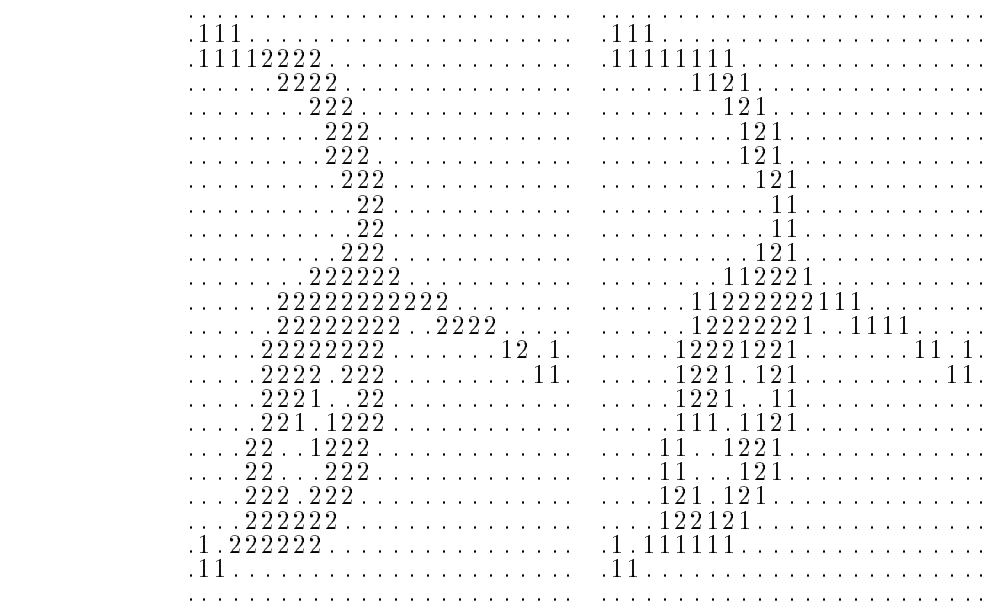

Figure 5.3: Trimmed image (not reset) after pass 3, marked image after pass 2

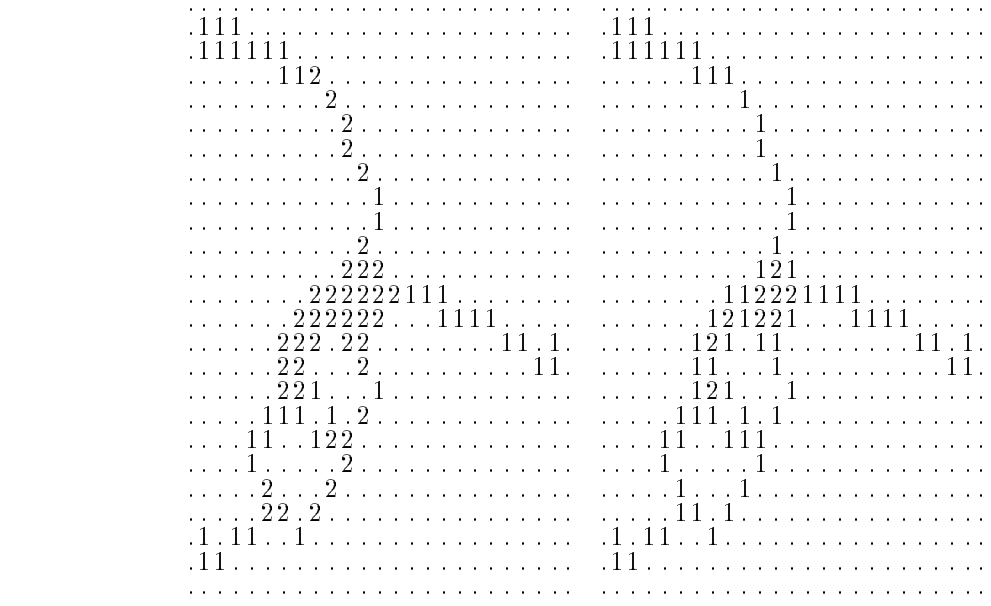

Figure 5.4: Trimmed image (not reset) after pass 2, marked image after pass 3

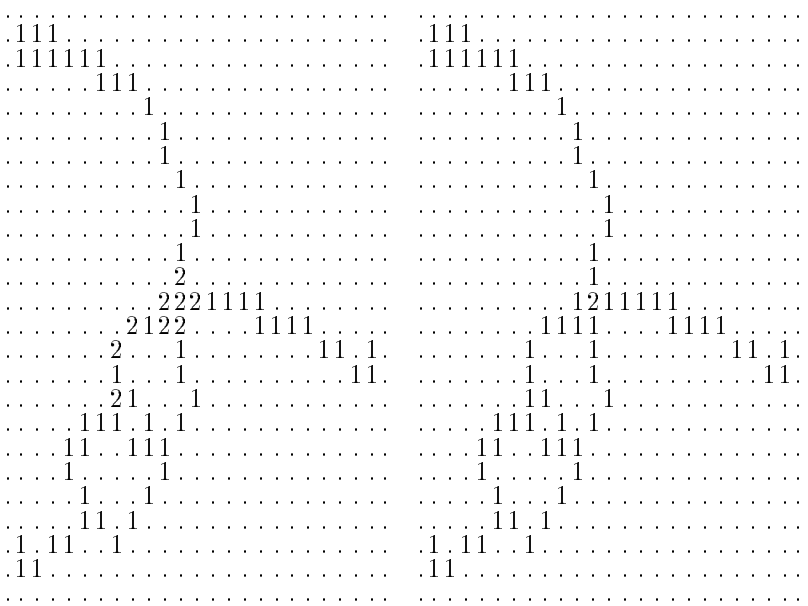

Figure 5.5: Trimmed image (not reset) after pass 3, marked image after pass 4

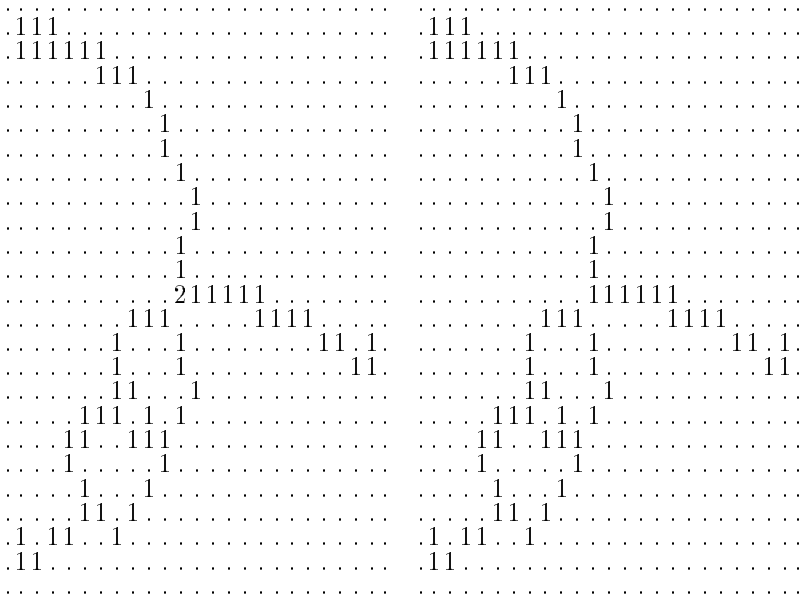

Figure 5.6: Trimmed image (not reset) after pass 4, result of first trimming

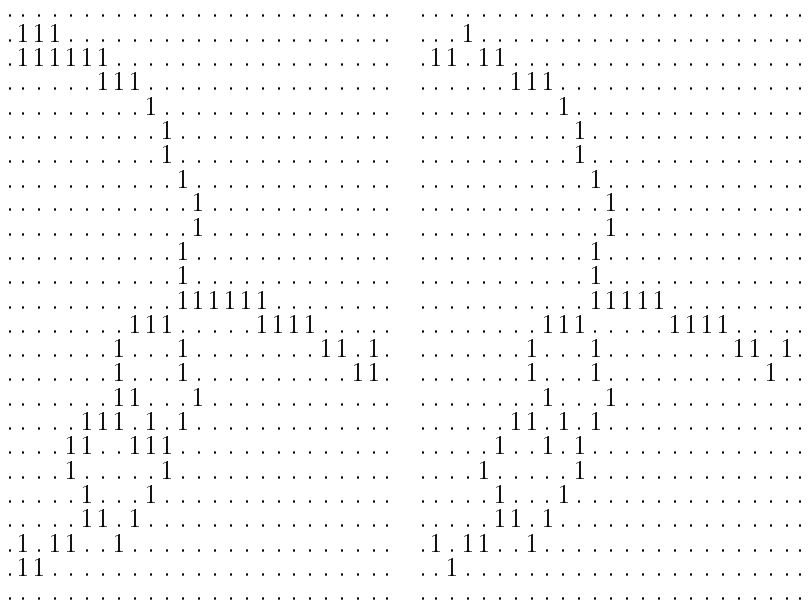

Figure 5.7: Result of first trimming, positive measure pixels removed