Multiscale Methods, Parallel Computation, and Neural Networks for Real-Time Computer Vision

Thesis by

Roberto Battiti

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1990
(submitted November 30, 1989)

Acknowledgments

This work was done while I was a research assistant to Geoffrey Fox in the Caltech Concurrent Computation Program and it benefited in many ways from his advice on parallel computation. I thank Christof Koch for providing the seminal ideas for the work in computer vision and effective assistance.

I am also pleased to acknowledge the useful discussions and suggestions from Wojtek Furmanski, Paul Messina, Demetri Terzopoulos, Roy Williams, and Edoardo Amaldi.

This work was supported in part by DOE grant DE-FG-03-85ER25009, the Program Manager of the Joint Tactical Fusion Office, the National Science Foundation with grant IST-8700064, and by IBM.

Last but not least, thanks to my parents Luciano and Lina and especially to my wife Annamaria for her patience and support during these intense years. Romano, my beautiful newborn son, provided a strong motivation to conclude this work in a short time.

Abstract

This thesis presents new algorithms for low and intermediate level computer vision.

The guiding ideas in the presented approach are those of hierarchical and adaptive processing, concurrent computation, and supervised learning.

Processing of the visual data at different resolutions is used not only to reduce the amount of computation necessary to reach the fixed point, but also to produce a more accurate estimation of the desired parameters. The presented *adaptive multiple scale technique* is applied to the problem of motion field estimation. Different parts of the image are analyzed at a resolution that is chosen in order to minimize the error in the coefficients of the differential equations to be solved. Tests with video-acquired images show that velocity estimation is more accurate over a wide range of motion with respect to the homogeneous scheme. In some cases introduction of *explicit discontinuities* coupled to the continuous variables can be used to avoid propagation of visual information from areas corresponding to objects with different physical and/or kinematic properties.

The human visual system uses concurrent computation in order to process the vast amount of visual data in "real-time." Although with different technological constraints, parallel computation can be used efficiently for computer vision. All the presented algorithms have been implemented on *medium grain distributed memory multicomputers* with a speed-up approximately proportional to the number of processors used. A simple two-dimensional domain decomposition assigns regions of the multiresolution pyramid to the different processors. The inter-processor communication needed during the solution process is proportional to the linear dimension of the assigned domain, so that efficiency is close to 100% if a large region is assigned to each processor.

Finally, learning algorithms are shown to be a viable technique to engineer computer vision systems for different applications starting from multiple-purpose modules. In the last part of the thesis a well known optimization method (*the Broyden-Fletcher-Goldfarb-Shanno memoryless quasi-Newton method*) is applied to simple classification problems and shown to be superior to the "error back-propagation" algorithm for numerical stability, automatic selection of parameters, and convergence properties.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 "Leitmotif" of the Thesis

The main objective of this thesis has been that of studying efficient parallel distributed algorithms for computer vision, in which many interconnected computational units cooperate to reach the desired result. Engineering these computational systems consists of tuning the mutual connections and the local updating rule to the different problems. A guiding principle in this task is that of hierarchical organization (for example, images are analyzed by units at different resolutions). The operations involved in low- and intermediate-level vision can be compared to distillation processes, where information is purified (from noise and irrelevant data) before being used by high level modules.

Biological analogies with our visual systems provided some of the motivation for this work. Nonetheless, emphasis is always on efficiency and possible implementation using available technology. In particular, all algorithms discussed here can be (and have been) implemented on multicomputers with an efficiency close to 100%.

In the following sections I will briefly outline the contribution of this thesis for the different problems considered.

1.2 Properly Coupled Discontinuities for Better Multiscale Vision

Multiscale methods with a new proposal for the coupling of the discontinuity detection elements (*line processes*) on different layers and for their incorporation into the multiscale relaxation process have been applied to two different problems of low level computer vision: piecewise smooth surface reconstruction and estimation of the motion field.

First a fast multiscale scheme for reconstructing a piecewise smooth surface from sparse and noisy data is proposed, in which line element detectors (*line processes*) at different resolutions are coupled in a coherent way to a multiscale "smoothing" algorithm (Gauss-Seidel relaxation) acting on the depth points. The suggested strategy is based on the interaction of the line processes with a neighborhood of depth points and line processes *at various scales.* In this way coarse-scale evidence guides detailed placement of discontinuities at finer resolutions, while fine-scale results improve delineation at coarser resolutions.

The approach has been tested on "Randomville" images (random collection of quadrilateral structures in the image plane) with promising results. Performance of the algorithm degrades gracefully when the sampling rate of the constraining data is reduced to a small fraction (down to 10%) of the grid points at the finest scale. Comparison with reconstruction time required by the one scale algorithm shows a speed-up of at least two orders of magnitude for 129×129 test images.

The second application has been for estimating the optical flow field (the projection of the velocity field onto the image plane) from a temporal sequence of images. Introduction of *line processes* is useful in order to avoid mixing velocity data from different moving objects during the relaxation phases. Fusion of information about the presence of zero-crossings (obtained after filtering the image with the Laplacian of a Gaussian operator) and about the presence of big differences in nearby velocity values is used to activate the line processes. The complete algorithm is based on an *adaptive multiscale strategy*, where the finest discretization grid is chosen locally using an estimation of the reliability of the obtained optical flow, as will be described in the following section.

The algorithms have been implemented with high efficiency on a MIMD parallel computer with distributed memory. A *coarse grain* domain decomposition is found to be useful for this and other multiscale problems.

1.3 Error Estimation to Improve Optical Flow

Single scale approaches to the determination of the 2-D optical flow field from the time-varying brightness pattern assume that the spatio-temporal discretization necessary to solve the appropriate equation is adequate for representing the patterns and motions in the scene. However, the choice of an appropriate spatial resolution is very difficult because it is subject to conflicting, scene dependent, constraints.

In differential methods, for example, derivative estimation is more accurate for long wavelength and slow motion with respect to the discretization step. On the contrary, short wavelengths and fast motion are required in order to reduce the relative errors caused by noise in the image acquisition and quantization process. Thus, the appropriate discretization step depends heavily on the local image and motion characteristics.

Homogeneous multiscale approaches treating all scales on the same footing, reduce the solution time with respect to single scale schemes, but they cannot avoid the interference that may be induced by conflicting information from different scales.

In this paper we propose a multiscale method for determining the 2-D optical flow, where the discretization scale is chosen locally according to an estimate of the error in the velocity estimation.

Results for 129x129 pixel video acquired images show that this method provides more accurate optical flow estimation than conventional algorithms (for example [56]), while maintaining the typical multiscale speed-up.

1.4 Optimization Techniques to Teach Multilayer Perceptrons

Learning is an essential part of our visual system. It is also clear that the availability of convenient ways to teach or adapt computer vision systems to different applications or different situations would open the road to cheaper and faster system development. It is not unrealistic to think about future off-the-shelf general purpose vision machines, or visual modules that can be assembled according to their use and "programmed" in some automatic way. After all, for example it does not take long for a human to learn a different alphabet set[1].

This work is concerned with the study of fast algorithms for teaching multilayer perceptrons. The considered "retina" is a simple one (a one-dimensional set of units) and the presented "images" have little to do with real images. I selected these test problems because I was concerned about comparing performance of the suggested algorithm with other approaches. It is also important to stress that the most efficient approach is probably not that of presenting images to a *tabula rasa* neural network and hoping that it will solve all your problems. All the available tools have to be used (for distillation of the essential information[2]) before applying learning to determine only a limited set (say of less than one thousand) of critical parameters. The results of the tests show that this is within the reach of widely available computing resources (microprocessor-based workstations with optional accelerator boards).

Standard back-propagation learning for feedforward neural networks is known to have slow convergence properties. Furthermore, no general prescription is given for selecting the appropriate learning rate, so success is dependent on a trial and error process. In this work a well known optimization technique (the *Broyden-Fletcher-Goldfarb-Shanno* memoryless quasi-Newton method) is employed to speed up convergence and to select parameters. The strict locality requirement is relaxed but parallelism of computation is maintained, allowing efficient use of concurrent computation. While requiring only limited changes to the back-propagation algorithm, this method yields a speed-up from one to two orders of magnitude for medium-size networks.

Comparisons are done with back-propagation using optimal parameters and with a version of it employing learning rate adaptation. This last method is in itself interesting, because it converges in a number of iterations close to that of optimized back-propagation, with no need for parameter optimization.

[1]Except if he is learning Chinese...

[2]Essential information is for example contained in zero-crossings, Canny edges, different momenta of the gray level distribution, etc.

Part II

Multiscale Low-Level Vision with Line Processes

Chapter 2

3D Surface Reconstruction

2.1 Introduction: Cooperation of Smoothing and Discontinuity Detection

Many processes that are based on "visual" sensors as their main source of information use a preliminary step of segmentation or piecewise smooth surface reconstruction (in the first case the data to be segmented are intensity values, in the second range data or depth values).

The *smoothing* operation filters out irrelevant information in the data (noise derived from the image formation and acquisition process) and spreads information from the sampled points to the nearby regions. During this process the explicit introduction of *discontinuities* (henceforth "line processes") is necessary both to avoid washing away important information under the smoothness requirement and to provide a primitive perceptual organization of the visual input into different elements loosely related to the human notion of parts or objects, to be used by the high-level processing stages. A partial list of references includes [3,8,9,72,74,13,75,76,79].

Neural processing in the brain and practical implementations (see, for example, [11,68,62]) show that the early vision steps can be done *in parallel*. Many computational units (neurons or processors) cooperate to reach the desired solution with a speed-up roughly proportional to their number (at least for regular, local, "trivially parallel" problems [87]).

Recently a multiscale method has been proposed for solving the partial differential equations associated with the smoothing operation [78]. This method can be implemented in a parallel architecture with processors connected in a pyramidal structure [66] or in a two-dimensional grid [2].

Up to now it is not clear how to combine in an effective way the multiscale surface reconstruction process with the discontinuity detection process. This study addresses this problem and suggests a simple scheme that allows cooperation of the two moments without disrupting the regular flow of multigrid computation on the different scales.

Discussion of this central point is preceded by a brief summary of the multigrid method used for solving the PDEs derived from regularization, mainly to establish the terminology and the context.

2.2 Multigrid Method for Regularization

The goal of the surface reconstruction step is to recover the properties of physical surfaces from an array of noisy range data.

In general, the class of admissible solutions is restricted by introducing *a priori* knowledge. In the *regularization method* the desired or plausible properties of the solution are enforced by transforming the reconstruction problem into the *minimization of a functional.*

For example, the energy functional corresponding to an "elastic membrane" ($z(x,y)$) pulled by "springs" connected to the data points ($d(x,y)$) is

$$E(z(x,y)) = \int_{Image} (z(x,y) - d(x,y))^2 + \lambda(z_x^2 + z_y^2)dxdy \tag{2.1}$$

If the functional is quadratic, the minimization problem is straightforward. The energy surface is convex and *gradient descent* will lead the system to the energy minimum.

Some proposals have been made in order to extend the approach for non quadratic functionals [72] [79] . In [72] a "hybrid" approach is used: continuous variables are changed according to the gradient descent scheme [1] (mapped to a resistive network), while the line processors are updated at a slower rate (with a deterministic or stochastic approach). Similarly, a "mixed" annealing strategy has been proposed in [74].

The Bayesian approach [8] to *estimate the most probable image* given a degraded image and a model for the degradation process is likewise reduced to a similar minimization problem.

After applying the calculus of variations, the *stationary* points of the functional [2] are defined by the solutions of the Euler-Lagrange equation.

For the previously introduced functional (eqn. 2.1) one obtains

[1]This can be done because, for a fixed set of line processes, the energy function is quadratic in the continuous variables.

[2]Hopefully local minima.

$$\delta E = \int_{Image} \{2(z(x,y) - d(x,y)) - 2\lambda(z_{xx} + z_{yy})\}\delta z dx dy \tag{2.2}$$

$$\lambda(z_{xx} + z_{yy}) = (z - d); \quad \text{or} \quad \Delta z = (1/\lambda)(z - d) \tag{2.3}$$

In standard methods for solving PDEs, the problem is first discretized on a finite dimensional approximation space. The very large algebraic system obtained is then solved using "relaxation" algorithms , which are local [3] and iterative.

By the local nature of the relaxation process, solution errors on the scale of the solution grid step are corrected in a few iterations; on the contrary, larger-scale errors are corrected very slowly. Intuitively, in order to correct them, information must be spread over a large scale by the "sluggish" neighbor-neighbor influence. If we want a *larger spread of influence* per iteration we need large-scale connections for the processing units, i.e., we need to solve a simplified problem on a coarser scale.

In the words of Brandt [4], we must take advantage of the fact that the algebraic system to be solved does not stand by itself, but is actually an approximation to continuous equations, and therefore can itself be similarly approximated by other (much simpler) algebraic systems on coarser grids. The pyramidal structure of the multigrid solution grids is illustrated in figure 2.1.

This simple idea and its realization in the *multigrid algorithm* not only leads to asymptotically optimal solution times (i.e., convergence in $O(n)$ operations), but also dramatically decreases solution times for a variety of practical problems, as shown in [4].

The multigrid "recipe" is simple. First use *relaxation* to obtain an approximation with smooth error for a fine grid. Then, given the smoothness of the error, calculate corrections to this approximation on a coarser grid, and in order to do this, first relax, then correct *recursively* on still coarser grids. Optionally one can also use the *nested iteration* idea (use of coarser grids to provide a good starting point for finer grids) to speed up the initial part of the computation.

Historically these ideas were developed starting from the sixties by Bakhvalov, Fedorenko and others (see [16] for a review).

[3]The local structure is essential for efficient use of parallel computation.

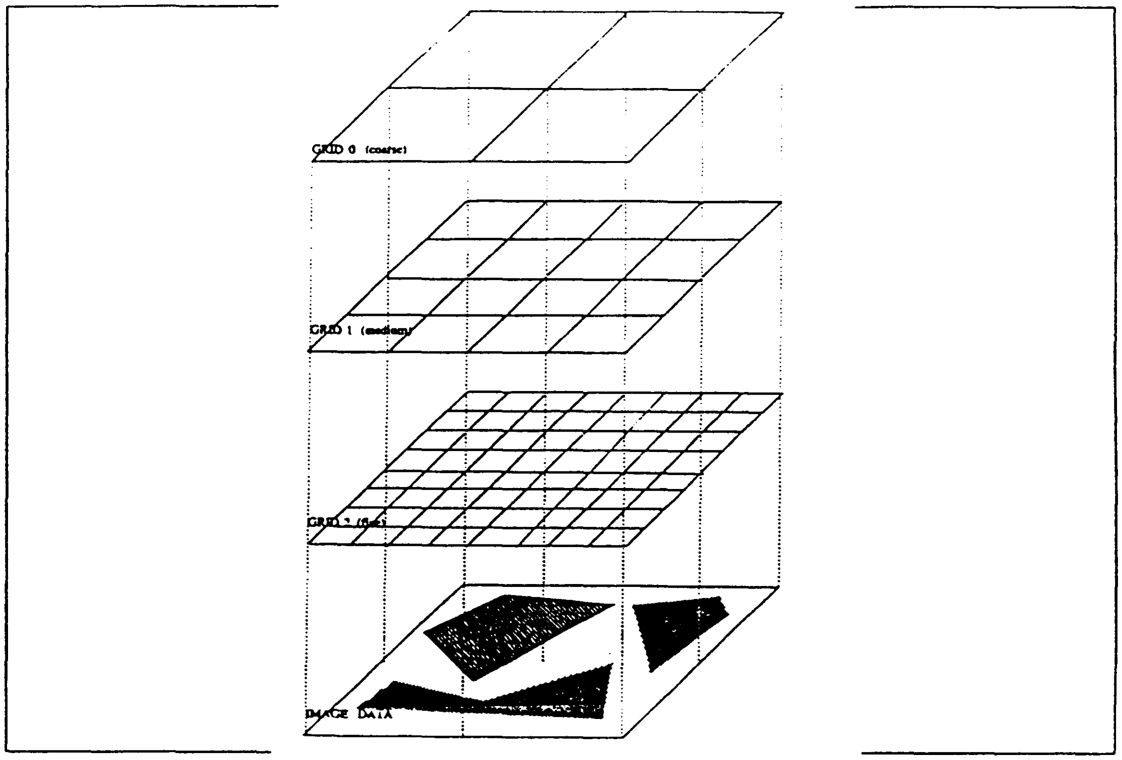


Figure 2.1: Pyramidal structure for multigrid algorithms.

It is shown in [4] that, with a few modifications in the basic algorithms, one can *store the actual solution* (not the error) in each layer. This method is particularly useful for visual reconstruction, where we are interested not only in the finest scale result but also in the multiscale representation developed as a byproduct of the solution process. This is called *full approximation storage algorithm* and it is briefly outlined in what follows.

The algebraic system, obtained by discretizing the original problem on the different grids (numbered by k with $0 \leq k \leq L$, $0 =$ coarsest) is

$$\mathrm{A}^{h_k}\mathrm{z}^{h_k} = \mathrm{d}^{h_k} \tag{2.4}$$

The data on the finest grid define d^{h_L}, while for the hierarchy of coarser grids the right hand side $\mathbf{d}^{h_k}$ is obtained using the two *extension* (fine $\longrightarrow$ coarse) and *interpolation* (coarse $\longrightarrow$ fine) operators, respectively $I^{\uparrow}$ and $I^{\downarrow}$ in this way [4]:

$$\mathbf{d}^{h_k} = \mathbf{A}^{h_k}\left(I^{\uparrow}\mathbf{z}^{h_{k+1}}\right) + I^{\uparrow}\left(\mathbf{d}^{h_{k+1}} - \mathbf{A}^{h_{k+1}}\mathbf{z}^{h_{k+1}}\right) \qquad (2.5)$$

Simple injection and bilinear interpolation are used in the present work.

Before computation is begun on a grid *finer* than the current one, the initial values for $\mathbf{z}$ are updated as:

$$\mathbf{z}^{h_k} \longleftarrow \mathbf{z}^{h_k} + I^{\downarrow}\left(\mathbf{z}^{h_{k-1}} - I^{\uparrow}\mathbf{z}^{h_k}\right) \qquad (2.6)$$

while before computation is begun on a grid *coarser* than the current one, the updating is

$$\mathbf{z}^{h_k} \longleftarrow I^{\uparrow}\mathbf{z}^{h_{k+1}} \qquad (2.7)$$

The switching of control between different grids is explained in figure 2.2.

Terzopulos applied the multigrid algorithm for solving PDEs associated with different early vision problems [17,78], like the lightness problem, shape from shading, surface reconstruction, optical flow. We repeated some tests and obtained typical multiscale speed-up factors of at least 100 for 129×129 images.

2.3 Line Processes in Time and Scale-Space

Because "real" images consist of approximately continuous patches separated by discontinuities and because a relevant part of the useful information is contained in these discontinuities, a surface reconstruction algorithm will have to deal with them in a constructive way.

[4]This definition agrees with the idea that coarse-scale corrections are a *top - down* influence. The definition given in "mathematical" texts is usually the opposite, so beware.

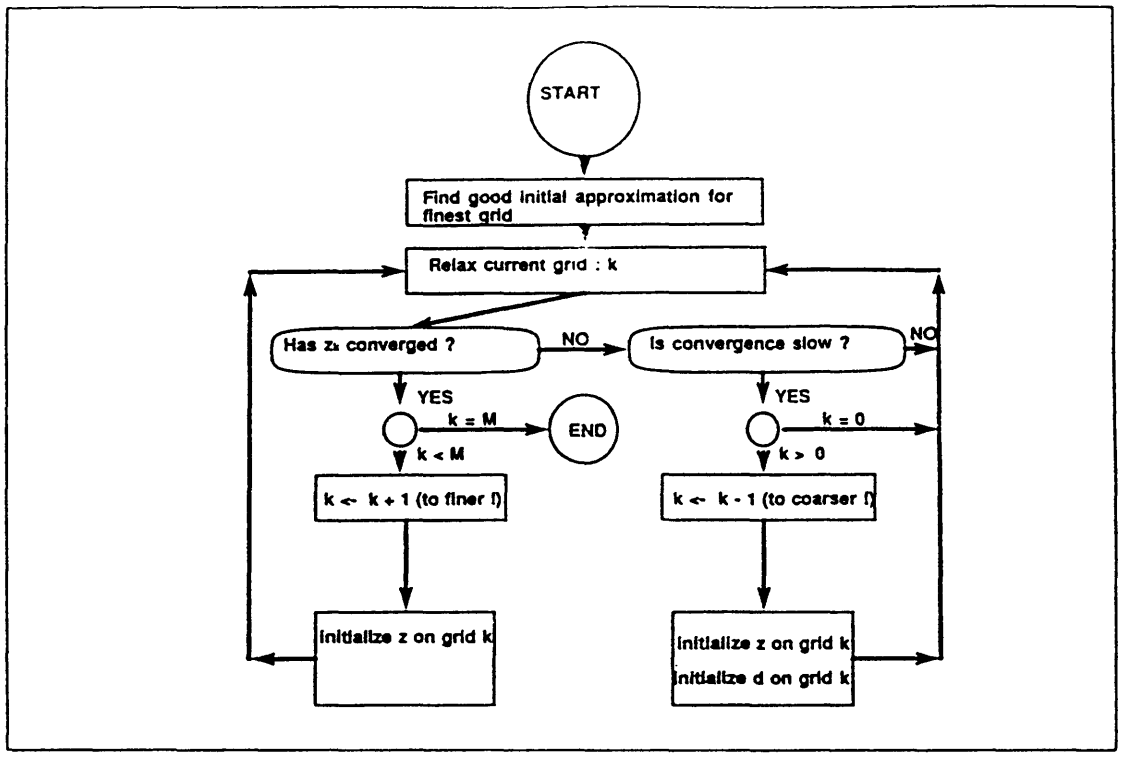


Figure 2.2: Flow of control in sequential multigrid (adapted from Brandt).

Even if there are some results in the literature [79,74,8], up to now it has not been clear how to combine the surface reconstruction and the discontinuity detection processes in an optimal way (speed is naturally one of the considered parameters).

One has to distinguish clearly between very different approaches, distinct by the *degree of cooperativity* of the two processes, considering both time and scale.

In some cases the discontinuity detection step is assigned to a separate *preliminary* process. Assuming this, in a *regularization approach* the smoothness constraint will not be enforced *globally*, but *locally*, depending on the presence or absence of line processes. For example, eqn. 2.1 will be transformed into

$$E(z(x,y)) = \int_{Image} (z(x,y) - d(x,y))^2 + \lambda(x,y)(z_x^2 + z_y^2)dxdy \qquad (2.8)$$

with $\lambda(x,y)=0$ in the presence of a line process, so that a break in the surface will not influence the value of the energy function. The danger in this case is that unessential discontinuities are introduced.

In other schemes, discontinuities are detected *after* the smoothing step (that could hide some of them), for example, by taking derivatives (error in derivatives will be smaller after regularization) and thresholding them appropriately.

Finally, other proposals consider cooperation of the two processes *in time* but do not consider the problem of organizing the cooperation *in scale.*

In [72] for example a new term is added to the energy function to favor a *good discontinuity structure.* If we introduce a function $G(\lambda)$ measuring the local "goodness" of the discontinuities, eqn. 2.8 becomes

$$E(z(x,y)) = \int_{Image} (z(x,y) - d(x,y))^2 + \lambda(x,y)(z_x^2 + z_y^2)dxdy + \int_{Image} G(\lambda(x,y))dxdy \tag{2.9}$$

In the hardware implementation suggested by the authors of [72], an analog network minimizes the "smoothness and data agreement energy" while, in a cyclic way, a digital network updates the line processes minimizing the "discontinuity energy."

The line processor network is updated at a much slower speed than the analog one. This is due to the fact that LPs are delimiting large-scale structures and must wait for the relaxation process to spread information over large distances before committing themselves to a yes or no decision.

Summarizing, in the first two approaches one process cannot make use of information exchange with the "dual" one, while in the last one the computation tends to be very slow for large images, because many cycles are required for convergence of the two coupled networks.

Our suggested approach to the problem will be illustrated in the following sections. It is based on the introduction of *line processes* at different scales, "connected" to neighboring *depth points* (containing the z values of the surface, henceforth called DPs) at the same scale and to neighboring *line processes* (henceforth LPs) on the finer and coarser scale. A heuristic function (called *Cost* function) is then responsible for embedding into the computational system the requirement of proper discontinuity structure. Finally, the multiscale algorithm is adapted for dealing with discontinuities in a simple but effective way.

2.3.1 Mutual Connections of Line Processes

During the course of the reconstruction, a given *line process* updates its value in a manner depending on the values of some other LPs. This is by definition the *neighborhood* and we feel free to refer to its members as the processes *connected* to the original one. It is useful to define three different subsets of this neighborhood: the set of connected LPs *at the same scale*, called SSN , its subset SSN* lacking the two *parallel line processes* (defined as the LPs at both sides of the given one and with the same orientation, see figure 2.3) and the set DSN containing the connected LPs *at the coarser and finer scales.*

Considering first the SSN, inside a given layer a LP is connected to other LPs with a "snowflake" pattern, as in figure 2.3. The influence of the parallel discontinuities is essential in the multi-scale scheme, to avoid duplication of lines caused by the "excitatory" coarse $\rightarrow$ fine influence.

The choice of the *connections between different layers* is more complex. Part of the difficulty is related to the discretization on grids composed of quadrilateral elements, whose size is doubled when coarseness is increased. Considering the geometry (see figure 2.4), it's apparent that there is no immediate definition of the LPs *above* or *below* a given one.

Here is one possible solution to the problem. First the coarse-to-fine influence is defined according to a *minimum distance* criterion: the updating of a given LP depends on the activation values of the LPs in the coarser scale that are at minimum distance (in the $x - y$ plane) from it. The only problem with this definition is that some LPs will have *two* LPs above with minimum distance, while others will have *one*. This slight asymmetry can be corrected by adjusting the connection weights so that the combined effect of the *two* activated minimum distance LPs (defined as *weak* influence) will be the same as the influence of the *single* LP in the other case (defined as *strong* influence), as will be shown in the following section.

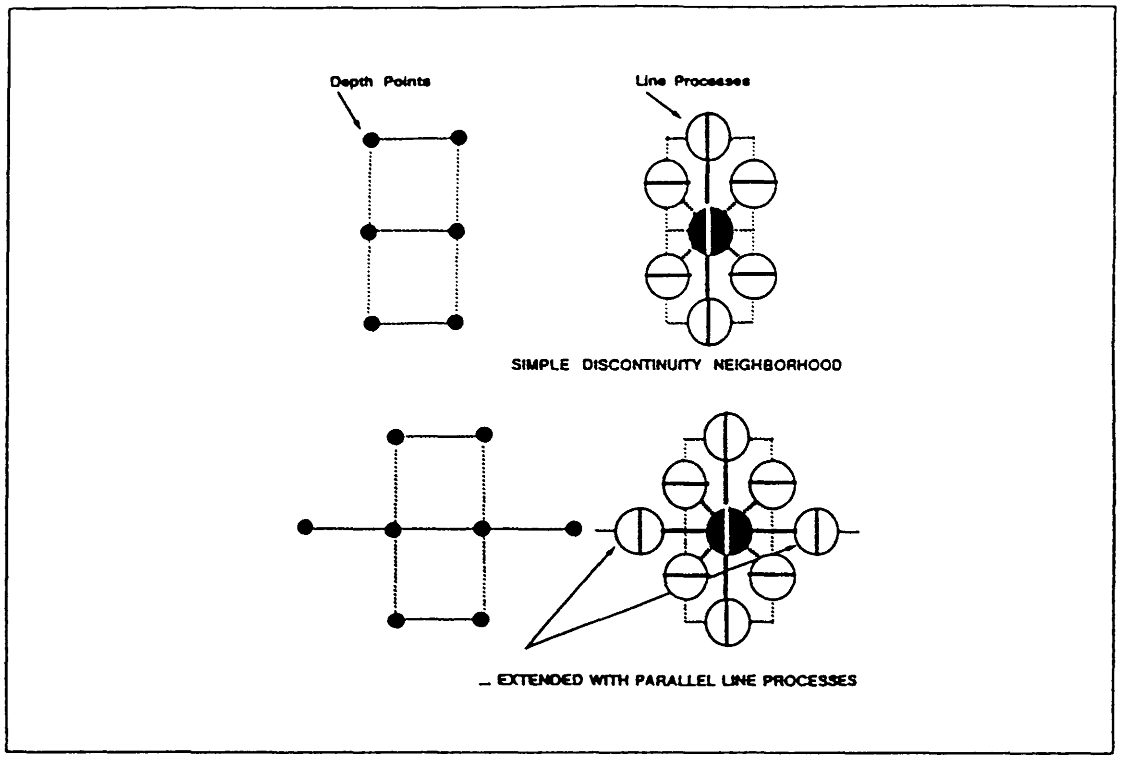

Depth Points
Line Processes
SIMPLE DISCONTINUITY NEIGHBORHOOD
_ EXTENDED WITH PARALLEL LINE PROCESSES

Figure 2.3: Discontinuity neighborhood inside a given layer. Depth points (left) and corresponding line processes (right) are shown. Influence from the "parallel" line processes is used to prevent duplication of lines going form coarser to finer scales.

Then the fine-to-coarse influence is defined by a symmetry requirement: if process x in scale X influences y in scale Y, then conversely y will be influenced by x. In this way the fine $\rightarrow$ coarse influence is determined uniquely after defining the coarse $\rightarrow$ fine one.

2.3.2 Updating Rule and Look-up Table

As we have seen before, starting from partial "visual" information, the dynamical system of the line processes and depth points on the different scales must evolve in time to a state corresponding to a faithful *reconstruction* of the three-dimensional structure and a *perceptual grouping* of it into "meaningful" pieces. Creation of discontinuities

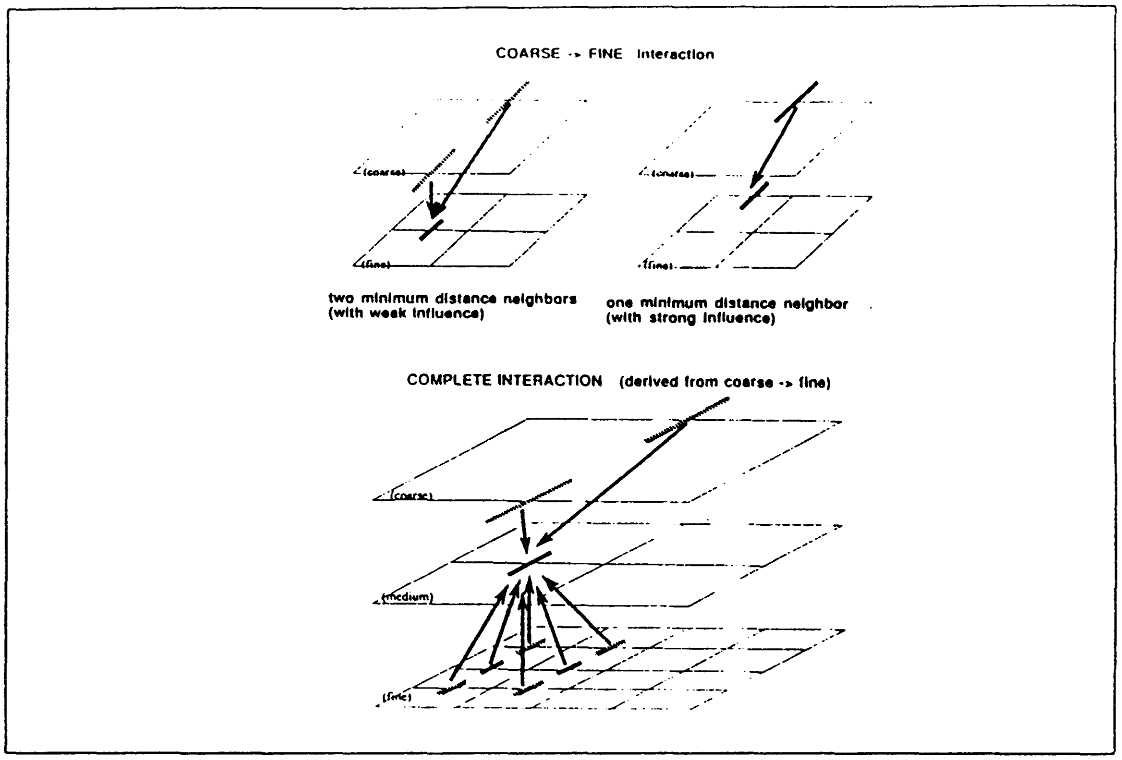

COARSE -> FINE interaction
(coarse)
(fine)
two minimum distance neighbors (with weak influence)
one minimum distance neighbor (with strong influence)
COMPLETE INTERACTION (derived from coarse -> fine)
(coarse)
(medium)
(fine)

Figure 2.4: Discontinuity neighborhood between different layers. First coarse → fine influence is derived based on the minimum distance prescription. Then using symmetry the complete interaction is derived .

therefore must be favored either by the presence of a large difference in the z values of the nearby DPs [5] or by the presence of a partial discontinuity structure that can be improved. Because usually the perceptual grouping *corresponds* to the underlying physical structure, these two driving forces cooperate to create the desired results.

Let us define as *benefit* the square of the derivative at a given point, because introduction of one discontinuity is "beneficial" when this quantity is large

[5]During this work a "membrane" energy term in the functional is considered. "Thin plate" and higher order terms that may be necessary for some reconstruction problems can clearly be accommodated in the suggested scheme.

$$Benefit = (\partial z/\partial x)^2 \approx \frac{(z_{i+1,j} - z_{i,j})^2}{h_k^2} \quad \text{for a vertical discontinuity} \tag{2.10}$$

and let's introduce a *cost* for a discontinuity in a given environment

$$Cost = f(\text{LPs} \in \text{SSN}; \underline{\text{LPs} \in \text{DSN}})$$

The local effect of other activated or inactivated discontinuities is given by the dependence of the *Cost* function on the LP values in the neighborhood at the same scale and at different scales (variables for activation values at different scales are underlined in the above notation). *Cost* is therefore a function of binary variables and will be defined in the next section. The updating rule for a LP is given by

$$LP \leftarrow 1 \quad \text{iff} \quad Cost < Benefit \tag{2.11}$$

Because the *Cost* is a positive quantity, discontinuities will be switched on only when there is a sufficient difference in nearby z values. Moreover, because the *Cost* depends on the LPs neighborhood, a good discontinuity structure can be favored by "discounting" *Cost* if the local structure is improved by activating the given LP.

Cost is a function of a limited number of binary variables, therefore to increase simulation speed and to provide the flexibility that is convenient for simulating different interaction schemes, a *look-up table* approach was used.

As shown in figure 2.5, an index into the table containing the *Cost* values [6] is obtained by reading the activation values (0 or 1) of nearby LPs and considering them as bits in the binary representation of the index.

2.3.3 Invariance, Scale, and Topology: a "Natural" Parametrization

Clearly segmentation should not depend on the physical scale of the structure [7]. If the z values of a surface are multiplied by a given factor, one should still get the same

[6] For 8 neighbors one gets a 256 entry table when considering only the SSN. To consider the DSN, a 64k entry table is needed.

[7] Unless we want this to happen.

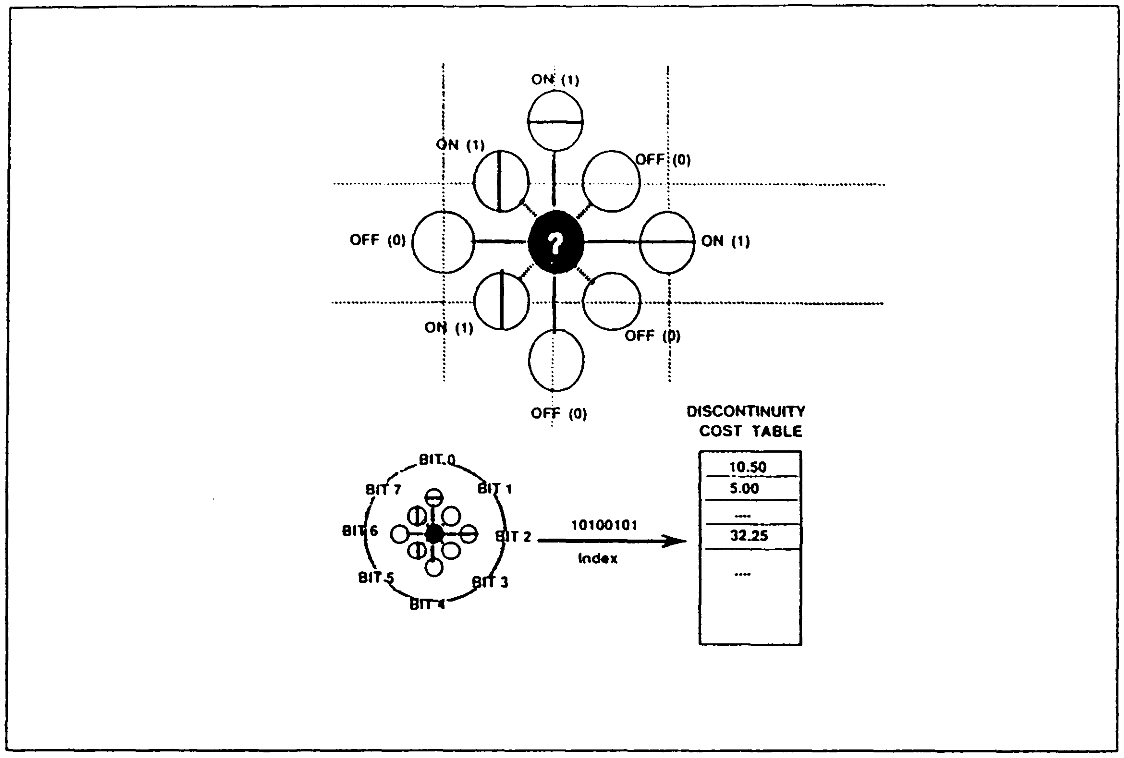


Figure 2.5: Look-up table for discontinuities. Activity values are used as bits of an index into the "cost" table.

distribution of line processes by scaling the *Cost*'s appropriately. Besides, the "topological" influence (enforcement of good discontinuity structure) should be independent of scale.

To *separate the effects of scale and topology* we decided to isolate the scale factor into one parameter *dh*, corresponding to the typical size of $\partial z/\partial x$ and $\partial z/\partial y$ that we want to be detected by our LPs. Because the comparison implied by eqn. 2.10 and eqn. 2.11 is with the square of these quantities, let's define the cost for a LP in the absence of other active LPs in the neighborhood as

$$Cost_0 \equiv f(0,...,0;\underline{0,...,0}) = dh^2$$

It would be of little practical use to allow 256 degrees of freedom in the definition of *Costs* for the SSN. First *rotational invariance* must be valid. If a given configuration is rotated by multiples of 90 degrees, *Cost* must remain equal. Moreover, because of the discretization of direction involved in the quadrangular grid, it seems reasonable to extend the notion of rotational invariance for cases like the one in figure 2.6, corresponding to a more general rotation.

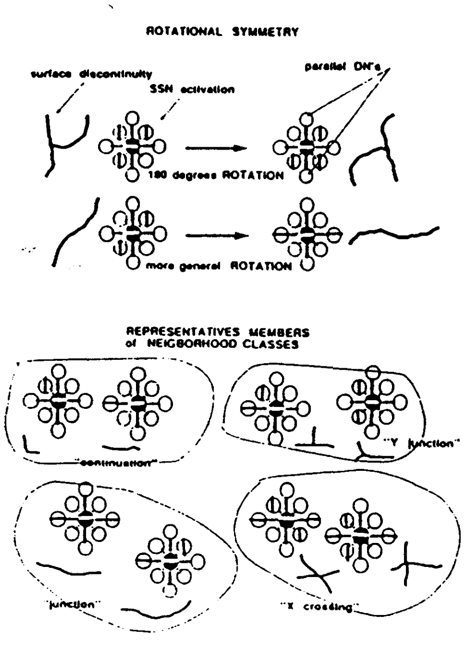


Figure 2.6: Rotational invariance leads to a small number of "topological classes" for the possible neighborhood structures.

We decided finally to classify all possible SSN* configurations (let's postpone consideration of the effect of the parallel LPs for the moment) into groups, depending on the number of regions in which the surface is divided at the location of the discontinuity. For some examples, see again figure 2.6. The *Cost* for a neighborhood with n cuts is multiplied by an associated factor a_n, to be selected by the user. If the number of cuts is too large, *Cost* is set to a very large value (to penalize formation of "tangled" lines).

$$Cost_n \equiv f(\text{LPs} \in \text{SSN}^*,0,0; \underline{0,...,0}\ ) = Cost_0 \times a_n$$

if local surface patch is cut into n pieces by the SSN* structure.

$$Cost = \infty \quad \text{if } n \geq 5.$$

The "inhibitory" influence of parallel lines is described by factor a_i (greater than one), with

$$Cost(\text{LPs} \in \text{SSN}; \underline{0,...,0}\ ) = Cost(\text{LPs} \in \text{SSN}^*,0,0; \underline{0,...,0}\ ) \times a_i^{np}$$

where $np \equiv$ number of parallel LPs $\in$ SSN.

Last but not least, presence of lines at the coarser or finer scale will reduce *Cost* by factors r_u or r_d respectively (smaller than one), in the *strong* influence case. In the *weak* influence case the factors become $\sqrt{r_u}$ or $\sqrt{r_d}$ [8].

$$Cost(\text{LPs} \in \text{SSN}; \underline{\text{LPs} \in \text{DSN}}\ ) = Cost(\text{LPs} \in \text{SSN}; \underline{0,...,0}\ ) \times r_u^{na} \times r_d^{nb}$$

where $na \equiv$ number of *above* LPs $\in$ DSN ($\times 1/2$ if *weak* influence).

and $nb \equiv$ number of *below* LPs, similarly.

This choice turned out to be a convenient method for "programming" the computational system in order to obtain a desired segmentation structure. For example, if the occurrence of crossings of type X in one class of images is believed to be rare, a large a_X parameter will do the job.

2.4 Combining Discontinuity Detection and Surface Reconstruction in Time and Scale

Our proposal for approaching the "priority problem" between smooth reconstruction and discontinuity detection is to *combine both phases in time and scale* .

[8] Let' s remember that the combined *weak* influence of two LPs (equal to $\sqrt{r_u} \times \sqrt{r_u}$) must be equal to the *strong* influence of a single LP (equal to r_u).

The reconstruction treats the computational units of the two types (line processes and depth points) on an equal footing, assigning them equal priority and equal time.

The general flow of control for the multiscale algorithm is similar to that given in [78] (here it is described using a pseudocode derived from the *C* language). The essential difference in this case is that each relaxation step in the initialization part and in the recursive multiscale call is associated with a line detection step that uses information about line elements in finer and coarser scales, as follows:

```
int fmg( )
{
    int i,layer ;
    layer= coarsest;
    i=naa;while(i--) {update_line_processes(layer);relax(layer);}
    update_line_processes(layer);
    for(layer = immediately finer;layer <= finest;layer++)
                {down(layer-1);mg(layer);}
}

int mg(layer) int layer;
{
    int i;
    if(layer== coarsest){ update_line_processes(layer);relax(layer);}
    else{
         i=na;while(i--){update_line_processes(layer);relax(layer);}
         i=nb;{up(layer);while(i--)mg(layer-1);down(layer-1);} /*recursion*/
         i=nc;while(i--){update_line_processes(layer);relax(layer);}
    }
    update_line_processes(layer);
}
```

The function `relax()` performs the relaxation step, while `update_line_processes()` updates the discontinuity values. `up()` and `down()` are, respectively, the injection operator (for fine-to-coarse restriction) and the bilinear extension operator (for coarse-to-fine extension).

Summarizing, first an initial number of relaxations are performed on the coarsest scale, then the approximate solution is extended to finer scales and the recursive multigrid call is applied to each of these.

In the present implementation, relaxation is based on the Gauss-Seidel (sequential) method [9]. A given z value is updated as follows:

$$z(x,y) \leftarrow \frac{z_{sum} + \beta \times h^2 \times d(x,y)}{n_{sum} + \beta \times h^2}$$

$$\text{where } \beta \equiv \frac{1}{\lambda}\,;\; h \equiv \text{ grid step.}$$

$$z_{sum} \equiv \text{ sum of neighboring DPs } \textit{not} \text{ separated by an active discontinuity ;}$$

$$n_{sum} \equiv \text{ number of terms in the sum ;}$$

$$z_{sum} = \sum_{dx=\pm h; dy=\pm h} \overline{LP}(x+dx, y+dy) \times z(x+dx, y+dy);$$

$$n_{sum} = \sum_{dx=\pm h; dy=\pm h} \overline{LP}(x+dx, y+dy);$$

As we will show in the following, this coordination scheme not only greatly improves convergence speed (the typical multigrid effect) but also produces a more consistent reconstruction of the surface at different scales.

2.5 Results of Multiscale Algorithm

Detailed performance tests have been made using noisy data for z values corresponding to "Randomville" structures. These are obtained by constructing quadrilateral blocks with random coordinates, heights, slants and tilts and placing them in the image plane. The data are then corrupted by noise and loaded as constraints in the algorithm.

[9] The choice of the relaxation method can be modified depending on the desired surface properties or the amount of parallelism in the computation, while maintaining the proposed coordination strategy with the discontinuity detection step.

2.5.1 Performance with Dense Constraining Data

In this case constraints are present on *all* grid points. Initial values for the LPs are equal to the corresponding constraints, where these are given, or to zero, where these are missing. All DP values are initially zero. Border conditions are obtained by "clamping" depth points to zero.

All timing results are given in terms of *work units*, where a work unit is defined as the amount of computation required to perform one iteration on the finest grid in the hierarchy. As far as absolute timing is concerned, a work unit corresponds to less than one minute for 129×129 images using a microcomputer [10] and to approximately 800ms using a simple parallel computer [11].

For 129×129 "images" and noise values corresponding to 25% of the highest structure, a faithful reconstruction of the surface (within a few percent of the original one) is normally obtained after one single multiscale sweep (with V cycles) on four layers [12]. The total reconstruction time is 3.43 work units.

Because of the asymptotic optimality of multiscale methods, time increases linearly with the number of image pixels (i.e., time $\propto n^2$ for an image with size n).

User interface examples and results from some tests are shown in the following figures. Figure 2.7 shows the simulation environment on the SUN workstation. Active line processes are shown in the higher half of the screen, for the different layers. Depth values of the surface are encoded using a proportional gray value and displayed in the lower part.

The first screen displays an intermediate state of the algorithm, where many spurious discontinuities due to noise are still present. The second screen shows the final result.

Figure 2.8 and figure 2.9 show the final result for a typical "Randomville" image. The original surface , the surface corrupted by noise (25 %), and reconstruction on different scales are shown in this order.

[10] SUN 386i by SUN Microsystems.

[11] Definicom board with 4 Transputers with Parasoft software.

[12] In other words, parameters na,nb,nc in mg() are equal to one.

2.5.2 Performance with Sparse Sampling Rate

In order to assess the degradation in performance with increasing sparseness of the randomly placed depth constraints (corrupted by 25% noise), the sampling rate was reduced down to 10% of the image points in the finest scale.

Constraints for coarser scales are then obtained by *averaging* the constraints for the finer scales, starting with the layer immediately "above" the finest one and repeating the averaging operation until the coarsest layer is encountered. In all tests we managed to reconstruct a "correct" surface using the same basic algorithm (in the same computational time). Figure 2.10 displays the data used for the tests (representing two slanted and rotated quadrilateral surfaces). Both the original data and the randomly-placed noise-corrupted constraints are shown.

As shown in figure 2.11, performance degradation is hardly noticeable, even after the sampling rate has been reduced to 10% . For larger amounts of noise or fewer constraining data it is useful to increase the number of iterations on each layer. Typically, performance reaches its limits for a value of three iterations per layer.

2.5.3 Speed-up with Respect to Single-Layer Approach

Some tests have been done in order to assess the gain in speed obtained by using more than a single scale in the algorithm. With this purpose, the 129×129 data set described in the previous section was used as input to a single-scale algorithm . This consists of relaxation and discontinuity detection on the scale corresponding to the finest grid used by the multiscale algorithm.

Reconstruction with a quality similar to the one obtained with the multiscale approach can be obtained only at the price of a large increase in the number of iterations. Furthermore, the number of relaxation steps for each discontinuity detection step has to be large (50 : 1 heuristically, for the test problem). This is a consequence of the slow propagation of information during the relaxation steps. If discontinuities are detected more frequently, spurious discontinuities at the border between regions with and without constraints will be activated, and these in turn will affect reconstruction in an almost irreversible way. The resulting computation times are larger by two orders of magnitude with respect to the multigrid times.

Because reconstruction with fixed parameters was not satisfactory, we decided to use an heuristic described in [72] : formation of discontinuities is penalized at the beginning, to favor a smooth interpolation except at very steep depth gradients, and then gradually encouraged, so that the surface will break at smaller gradients. In the present implementation the parameter *dh* is linearly decreased in the course of the computation.

Figure 2.12 shows the evolution of the single-scale algorithm up to 300 iterations (300 work units because iterations were done at the finest scale).

In this case the shallowest part of the contour has been lost because of the smoothing effect of relaxation and is not recovered even after drastically decreasing *dh* (*dh* cannot become too small otherwise spurious LPs will be activated).

2.6 Summary

We showed that the extension of multiscale methods to discontinuity detection can be done in an effective way, combining surface reconstruction and discontinuity detection *in time and scale.*

This reduces total computational time by orders of magnitude with respect to single scale methods and provides a better coordination between the two requirements of faithful reconstruction and good discontinuity structure.

We strongly believe that the presented method can be easily adapted for similar problems in early vision (for example, optical flow, shape from shading ,...) and more general problems in which a two-dimensional distribution of data must be reconstructed and segmented into smooth regions.

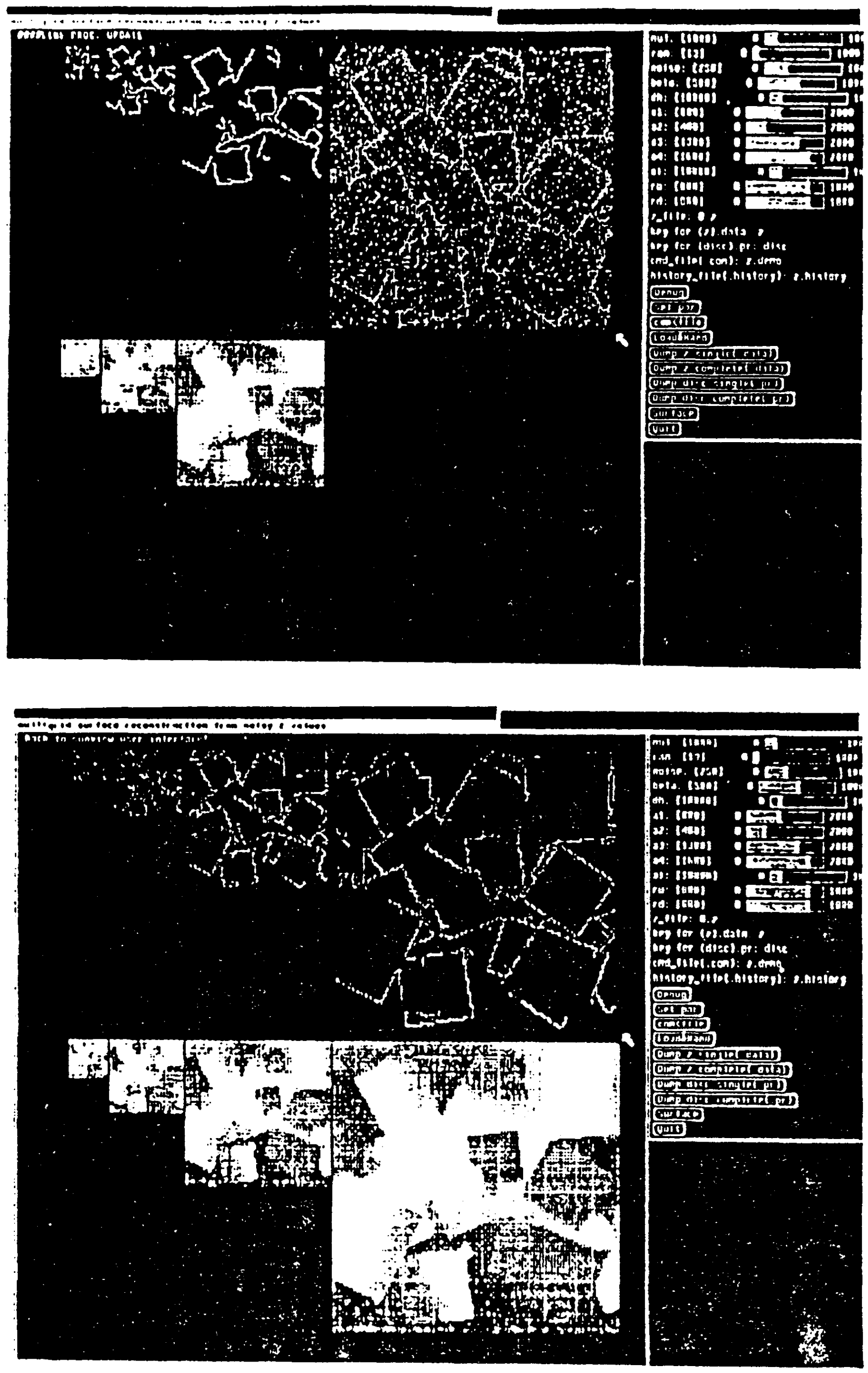


Figure 2.7: Simulation environment: results during computation and final results. In the upper part of the screen are activated discontinuities, in the lower part are gray-encoded depth values (on the different scales).

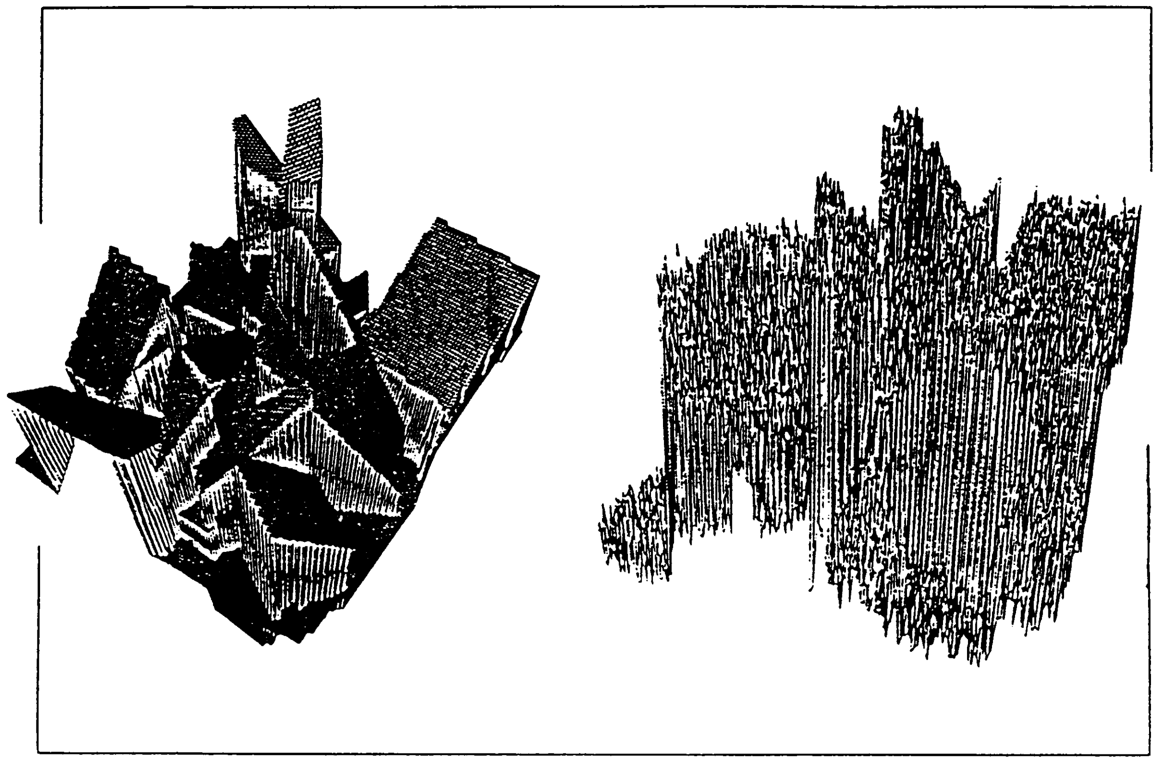


Figure 2.8: "Randomville" landscape: original and noisy images (25% random noise).

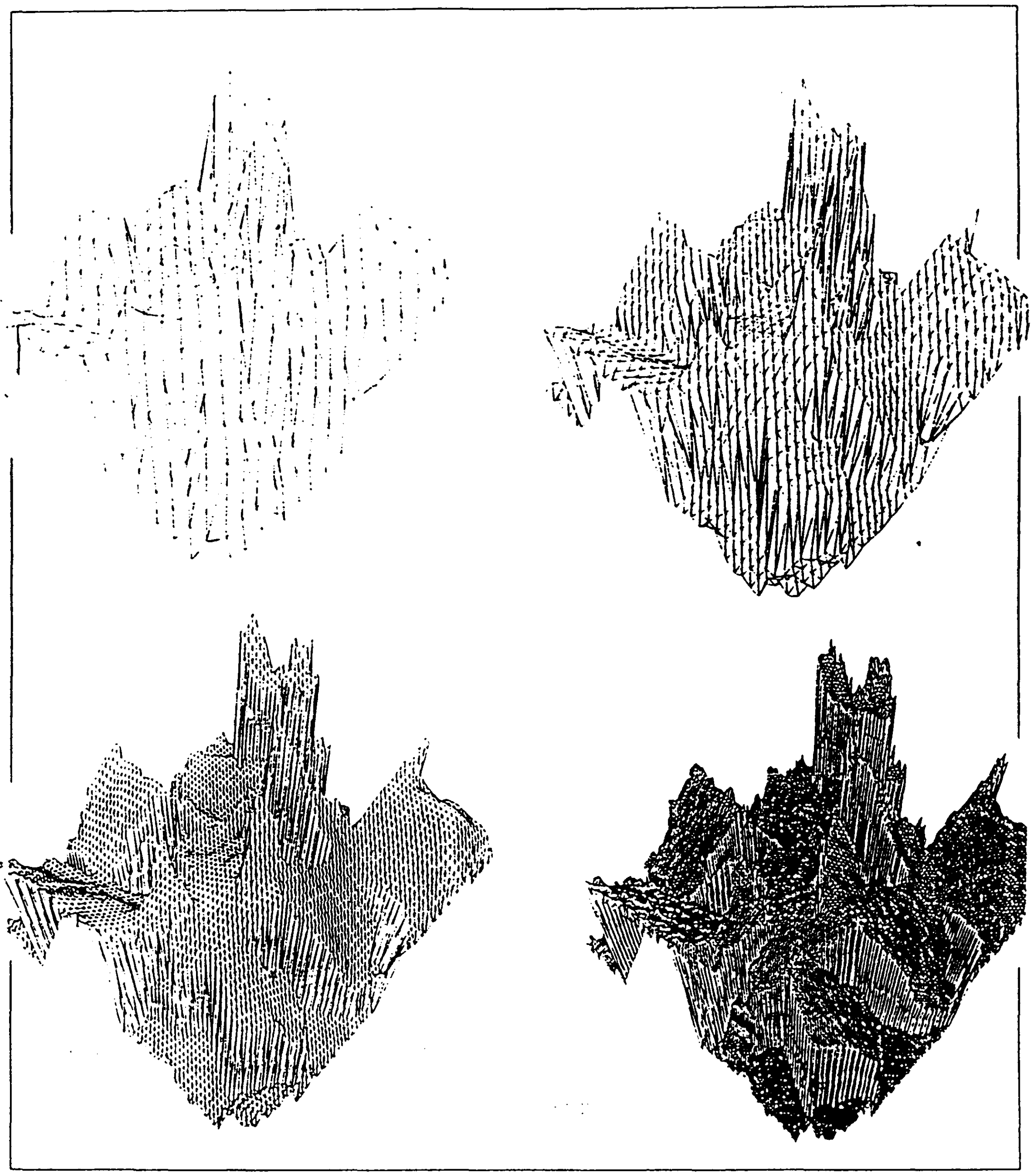


Figure 2.9: Multiscale reconstruction of "Randomville" landscape from dense constraints: results on different scales.

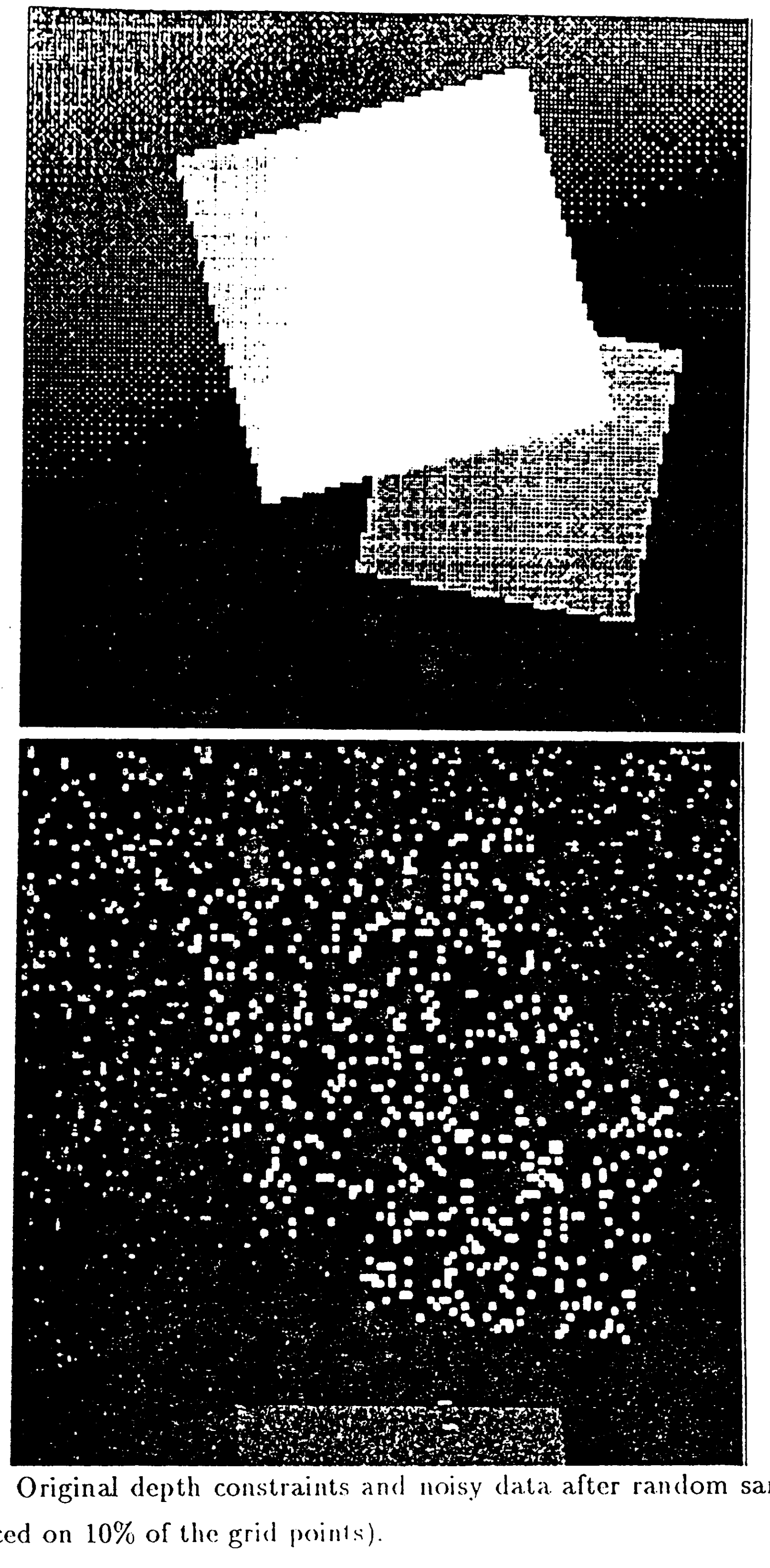


Figure 2.10: Original depth constraints and noisy data after random sampling (constraints placed on 10% of the grid points).

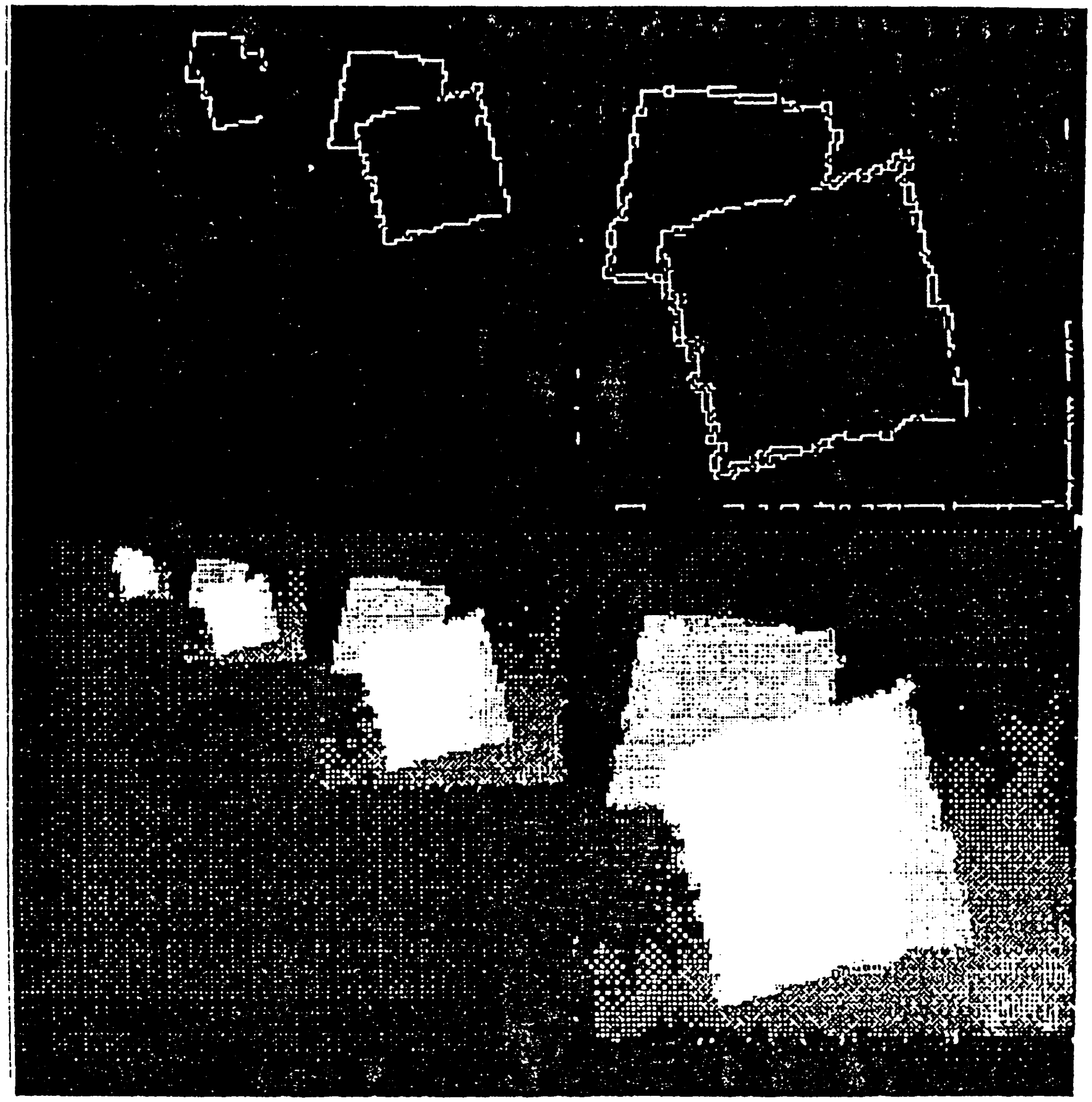


Figure 2.11: Reconstruction with 10% sampling rate: multiscale algorithm.

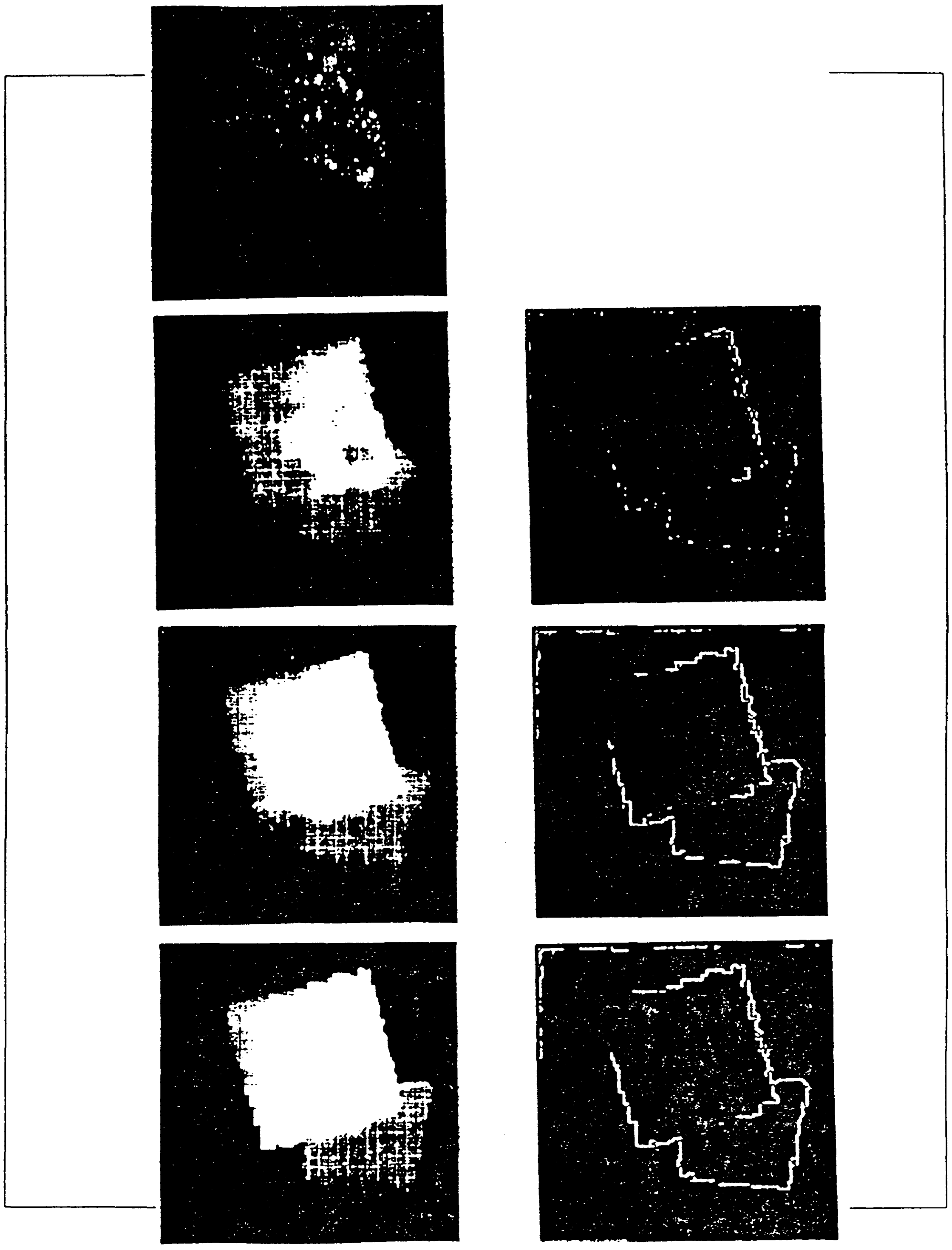


Figure 2.12: Reconstruction with 10% sampling: evolution of single scale algorithm. State after 20, 50, 200, and 300 iterations is shown. Discontinuities are activated only after 50 preliminary relaxation steps.

Chapter 3

Adaptive Multiscale Scheme for Optical Flow

3.1 Introduction: Reliable Estimation of the Optical Flow

During the last decade there has been increasing interest in analyzing sequences of time-varying images and in particular in determining the 2-D motion or velocity field, which is the projection of the 3-D velocity field onto the image plane (see [37] for a review). In particular situations, the apparent motion of the brightness pattern, known as the optical flow, provides a sufficiently accurate estimate of the motion field [1]. The two main approaches that have been proposed for determining the optical flow are *differential* [56,28] or based on *matching* of tokens or intensity values [44,34]. The former estimates the flow field from spatial and temporal variations of the image brightness while the latter involves an explicit matching of the low-level (intensity values) or high-level features or tokens across successive frames.

Both approaches make a basic assumption about the *scale* of the image patterns and of the motion to be determined. In differential methods, reliable derivative estimation requires that the space-time variation of the intensity pattern is small with respect to the discretization steps. Similarly, in single-scale matching methods the search for correspondence is limited to a local neighborhood defined by the expected motion amplitude.

The multigrid algorithm with the "full approximation storage" scheme has been suggested as a way to solve the differential equation in Horn and Shunck's method of deriving the optical flow [61]. This algorithm converges in a time proportional to the number of pixels in the image, is computationally efficient, and produces in addition a consistent result at different spatial scales. Unfortunately, both the multigrid method and simpler coarse-to-fine continuation schemes tend to suffer from their homogeneous computational structure. In some cases this may cause the optical flow detection process to oscillate between different estimates at different scales or even to converge to a wrong solution [52,55]. Indeed, if no explicit direction is given in order to select locally the appropriate scale, different scales will, in general, provide conflicting information.

[1] For example, use of the "brightness constancy" assumption to derive the optical flow is correct (produces a flow coincident with the motion field) if the scene is illuminated by one fixed light source at infinity, the surfaces are Lambertian, strong intensity gradients are present, and the motion is in planes parallel to the image plane.

We propose a method for tuning the discretization grid to a measure of the reliability of the information derived from a given scale. This measure will be based on a *local estimate of the errors* due to noise and discretization.

The flow of control is from coarse to fine scale, and we assume that the largest motion in the image can be estimated by one of the used scales.

Binary discontinuities in the optical flow are introduced explicitly, both because they prevent smoothing across regions corresponding to different moving objects and because they provide a compressed representation that can be used by subsequent visual modules.

Since our long term interest is in achieving real time processing, some thought is given to selecting methods and algorithms with low computational complexity.

The chapter is organized as follows. First, we summarize some concepts about differential methods for optical flow, then we introduce and discuss some fundamental shortcomings of multiscale versions of these approaches. Next, we describe our scheme with adaptive discretization and explicit discontinuities. It requires the derivation of an estimate for the relative error in the flow field at a given scale. Finally, we present some experimental results obtained with synthetic and real world image sequences.

3.2 Differential Methods

In order to estimate the optical flow from a series of time-varying images one needs to make some assumptions about the temporal evolution of the image brightness. Let $E(x,y,t)$ be the image brightness at point (x,y) at the time t and suppose that E varies smoothly with respect to space and time. Many differential methods assume that the brightness of patches in the image remains approximately constant over small time intervals [56]. This simple assumption leads to the constraint equation

$$\frac{dE}{dt} = E_x u + E_y v + E_t = 0 \tag{3.1}$$

relating the change in image brightness at an image point (x,y) to the two components $u = \frac{dx}{dt}$ and $v = \frac{dy}{dt}$ of the flow field. E_x, E_y, and E_t denote the spatial and temporal brightness derivatives that must be estimated from successive image frames.

Clearly, this *brightness constancy* equation does not determine both components of the optical flow field uniquely. From eqn. 3.1 one can recover directly only the component of the optical flow in the direction of the brightness gradient[2]. This is known as the aperture problem and it has often been taken as evidence that the estimation of the optical flow is a fundamentally ill-posed problem. Further assumptions leading to additional constraints are needed in order to retrieve the flow component along the isobrightness contours [76].

Horn and Schunck [56] regularized the problem by assuming that the optical flow varies smoothly almost everywhere in the image, and they proposed to minimize the quadratic energy functional

$$\Phi = \int\int_{Image} (E_x u + E_y v + E_t)^2 + \alpha^2(u_x^2 + u_y^2 + v_x^2 + v_y^2)dxdy, \tag{3.2}$$

where the first term expresses the rate of change of the image brightness along the flow line and the second one the departure from smoothness. The weighting factor α is proportional to the expected noise in the estimates of the spatial and temporal brightness derivatives. The appropriate Euler-Lagrange equations

$$(E_x u + E_y v + E_t)E_x = \alpha^2 \Delta u \tag{3.3}$$

$$(E_x u + E_y v + E_t)E_y = \alpha^2 \Delta v \tag{3.4}$$

give a necessary condition for an extremum of Φ. After discretization, this leads to a very large algebraic system (a pair of equations for each point in the image) that can be solved using local and iterative "relaxation" methods. The solution method used throughout this work is *Gauss-Seidel lexicographic relaxation.* During an updating cycle the grid points are considered in a fixed order (for the considered images one starts from the north-west pixel and follows the image lines), and the new approximation (u^{n+1}, v^{n+1}) of the flow field can be determined from the estimated brightness derivatives and from the local average $(\overline{u}^n, \overline{v}^n)$ of the previous flow estimate by

[2]The solution to eqn. 3.1 is: $(u, v) = \frac{-E_t \nabla E}{\|\nabla E\|^2} + \mathbf{a}$ where $\mathbf{a}$ is perpendicular to ∇E

$$u^{n+1} = \overline{u}^n - \frac{E_x(E_x\overline{u}^n + E_y\overline{v}^n + E_t)}{(\alpha^2 + E_x^2 + E_y^2)} \tag{3.5}$$

$$v^{n+1} = \overline{v}^n - \frac{E_y(E_x\overline{u}^n + E_y\overline{v}^n + E_t)}{(\alpha^2 + E_x^2 + E_y^2)}, \tag{3.6}$$

Natural boundary conditions are given by zero normal derivative.

The smoothness assumption regularizes the problem but it constrains the field to vary smoothly across the occluding boundaries, where flow discontinuities can be expected. Several approaches have been suggested to prevent smoothing over discontinuities (see [31] for example). This question will be addressed in section 3.7 where we will present our coordination scheme for the smoothing and discontinuity detection phases.

Recently, Uras et al. [45] argued that the estimation of the optical flow is not, in general, an underconstrained problem since the image brightness satisfies other natural assumptions besides the brightness constancy equation. They proposed to consider the vector equation

$$\frac{d}{dt}\nabla E = 0 \tag{3.7}$$

which involves second order brightness derivatives and which is verified exactly for a parallel translation in the image plane and when the light source is distant and fixed. In general, this *gradient constancy* equation (as well as any other pair of scalar equations among eqn. 3.1 and eqn. 3.7) determines the flow field uniquely. This makes the optical flow problem only ill-conditioned but not ill-posed (except when only straight edges exist and $E_{xx}E_{yy} - E_{xy}^2 = 0$). The optical flow cannot be recovered when the two equations are linearly dependent (in general it cannot be recovered reliably if the linear system is not well conditioned). At these locations there is not enough local information and it is therefore necessary to impose some local smoothing.

3.3 Towards an Adaptive Multiscale Approach

All differential or feature-based methods for recovering the optical flow working at a single spatial scale share a fundamental limitation. This limitation stems from the fact that an optical flow algorithm needs to solve, at least implicitly, a matching problem. Indeed, any single scale method faces an ambiguity when it must bring into correspondence image intensities or image brightness features in successive frames. This is especially true when the motion amplitude becomes similar to the spatial intensity wavelength[3].

In the differential context, this problem can easily be shown for a one-dimensional sinusoidal intensity profile $sin\frac{2\pi}{L}(x - vt)$ of wavelength L moving with velocity v. In the one-dimensional case, the brightness constancy equation determines the optical flow uniquely and the measured velocity $\tilde{v}$ is given by

$$\tilde{v} = -\frac{\tilde{E}_t}{\tilde{E}_x} = \frac{\frac{sin\frac{2\pi}{L}(x+vt+v\Delta t)-sin\frac{2\pi}{L}(x+vt-v\Delta t)}{2\Delta t}}{\frac{sin\frac{2\pi}{L}(x+vt+\Delta x)-sin\frac{2\pi}{L}(x+vt-\Delta x)}{2\Delta x}} = \frac{sin(\frac{2\pi}{L}v\Delta t)}{sin(\frac{2\pi}{L}\Delta x)}\frac{\Delta x}{\Delta t}, \tag{3.8}$$

where $\tilde{E}_x$ and $\tilde{E}_t$ are the three-point approximations of the spatial and temporal brightness derivatives obtained using the spatial and temporal discretization steps Δx and Δt. Three-point derivatives provide a better estimate $(O(h^2))$ with respect to the forward difference formula. In addition, the temporal and spatial derivatives are estimated at the same point (no phase shift is present, as explained in [35]).

Figure 3.1 shows some characteristic graphs of the measured velocity and relative error in the velocity as a function of the true velocity v for different wavelengths L. For clarity, the curves are plotted versus the dimensionless ratios $\frac{v\Delta t}{L}$ and $\frac{\Delta x}{L}$, where Δx is the discretization step in space and Δt in time.

While in the limit $L \longrightarrow \infty$ eqn.3.8 converges to the correct velocity v, the relative error in the computed velocity becomes of the order of 100% even for small velocities when the wavelength is smaller than approximately five spatial sampling steps ($\frac{\Delta x}{L} \geq 0.2$). Note that we take into account only the error due to the approximation of the brightness derivatives.

[3]Reducing the interframe movement by increasing the acquisition frequency diminishes this ambiguity but increases the computational burden because more images have to be treated. Since the motion scale is not known in advance, this frequency must be high and noise problems due to intensity quantization or other sources can arise. A too small interframe movement will, for instance, be undistinguishable from zero after quantization.

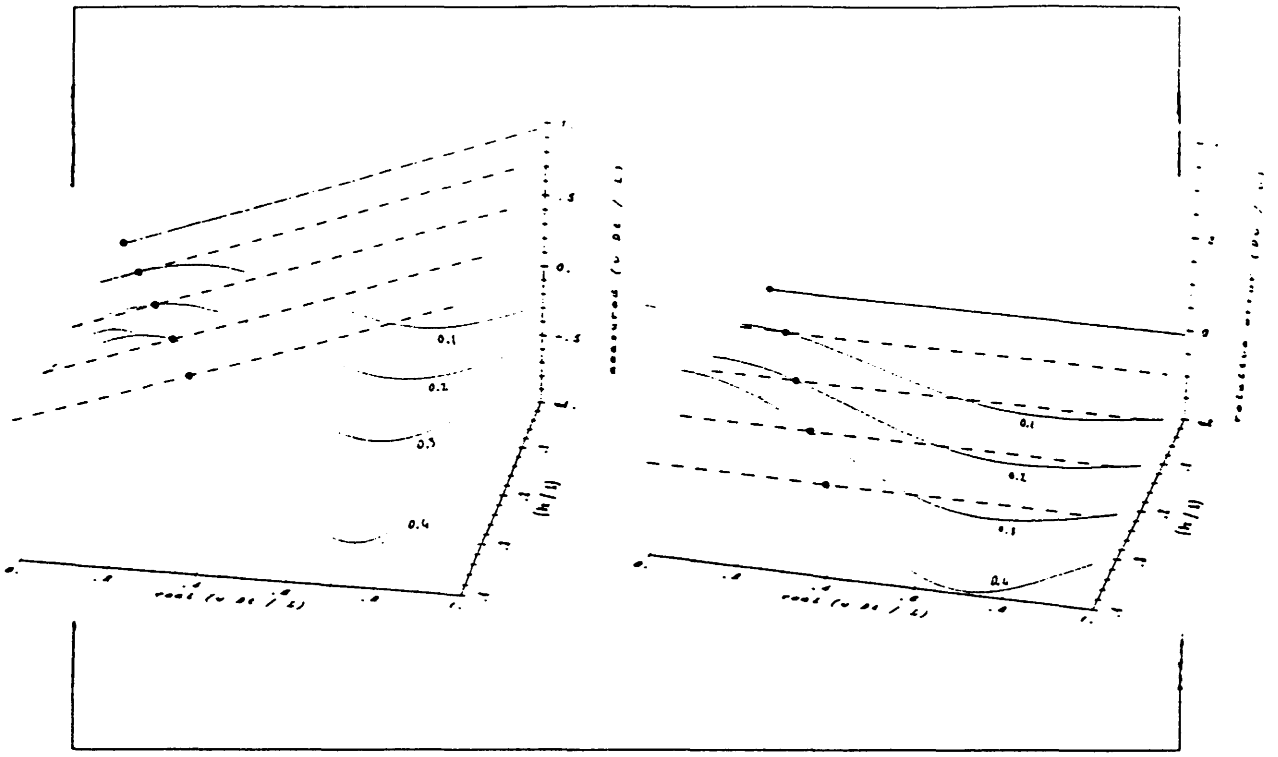


Figure 3.1: Measured velocity (a) and relative error in measured velocity (b), defined as $\delta v/|v| = |(v-\tilde{v})|/|v|$, for moving sinusoidal pattern. Values obtained with discretization (dots) are compared with correct values (dashed line). Three point approximation for derivatives is used. Curves for different values of $\frac{\Delta x}{L}$ are shown.

In the case of the Horn and Schunck method a basic problem arises when the motion amplitude is too large with respect to given resolution (i.e., is more than a few spatial sampling steps)[4]. The problem is caused by the use of *discretized formulas* for the estimation of the temporal and spatial derivatives of the image brightness. As said before, the accuracy of these formulas decreases when the brightness changes rapidly on the scale given by the discretization step, because in this case the step cannot be considered infinitesimal.

To deal with the matching ambiguity one can consider a resolution pyramid and

[4]In this work we assume that the brightness constancy equation 3.1 is valid.

work at different scales ([49] and the contained references). Since the high frequencies are attenuated at lower resolution, the spatial and temporal derivatives of the brightness are smoothed and their estimate is more informative. In addition, a multiscale approach with an effective coordination scheme between the different resolutions reduces the computational effort. It has been shown that the *multigrid* algorithm ([64] for the general theory) converges in a time proportional to the number of pixels (i.e., to n^2 for an n x n image) and is furthermore efficient (reducing computation by two orders of magnitude for images with 129X129 pixels). This effect can be explained when one considers that in local iterative procedures information propagates only to nearest neighbors during an updating step. Now, since points that are neighbors at lower resolutions are separated by many fine resolution steps, fewer iterations at lower resolutions are sufficient to spread information between areas that are distant in the original image.

There has been some previous work on multiscale determination of the optical flow [52,55,61]. Terzopoulos [61] applied, for instance, the multigrid algorithm to the Euler-Lagrange equations 3.3 and 3.4. The idea of the multigrid methods [77] consists in starting from an approximation with smoothed error obtained by relaxation on the fine grid and in determining a correction of this approximation on the coarser grid. This is computationally less expensive and it can be done recursively by relaxation on the coarse grid and correction on the next coarser grid. The fine-to-coarse and coarse-to-fine intergrid transfers are realized using, respectively, restriction and interpolation operators with local averaging properties. Note that the starting approximation itself can be obtained in a coarse-to-fine fashion, using nested iterations.

Terzopoulos reported, for the case of an expanding Lambertian sphere, a substantial speed-up with respect to the single scale relaxation [61]. It is important to point out that this result applies to an image that contains a unique dominant spatial frequency (related to the sphere diameter). Since in this special case the velocity is perpendicular to the brightness gradient, the first iteration is already sufficient (in the absence of noise) to recover the correct optical flow. Indeed, the multigrid method turns out to be much less effective for more complex images with superposed frequencies, or even for single frequencies if, as will be shown, a grid coarser than the finest one provides a better estimate.

This difficulty has also been encountered by Glazer [55] and Enkelmann [52] and is relevant to any multiscale scheme, when conflicting information is present at different scales.

An example is given in one dimension by considering two scales with a 2 : 1 resolution and an intensity profile composed of two sine waves of different wavelengths L_1 and L_2. Suppose that in terms of the fine grid spatial step: $L_1 = 3$, $L_2 = 6$ and the intensity profile velocity is equal to 2 (in the following, for simplicity, Δt is equal to 1). On the coarse grid the higher frequency will be almost completely suppressed [5] and the measured velocity is equal, according to eqn. 3.8, to the true velocity $v = 2$.

Figure 3.2 shows that on the fine scale there is at least a 50% error in the velocity for any combination of the two frequencies. In particular, the measured velocity is equal to 1 for an intensity profile with only the low frequency and it has even an opposite sign when the ratio between the high and low frequencies is greater than 0.5. It is worth noting that if $v = 1$ the correct velocity would clearly be recovered at the fine scale.

Typically, the image brightness is a superposition of different frequencies corresponding to the different objects and textures in the scene. Thus, a multiscale scheme *à la multigrid*, involving a (systematic) bidirectional information flow from high-to-low and low-to-high resolution, is not appropriate because it is likely to mix incoherent information from the different scales. The scheme may not converge[6] or it may converge to an incorrect result.

The previous examples and considerations suggest a new strategy. It starts by estimating the overall flow field at a reasonably coarse scale. This approximation is then improved on successive finer scales only in regions of the image *where its estimated error is greater than a predefined threshold.* A *local inhomogeneous* approach is thus obtained, where areas of the images characterized by different spatial frequencies or by different motion amplitudes are processed at the appropriate resolutions, avoiding corruption of good estimates by inconsistent information from a different scale.

A simple local criterion to evaluate the local reliability of the flow field is based

[5] By the smoothing operation preceding the subsampling process.

[6] It may oscillate between two different grids with conflicting information, for example.

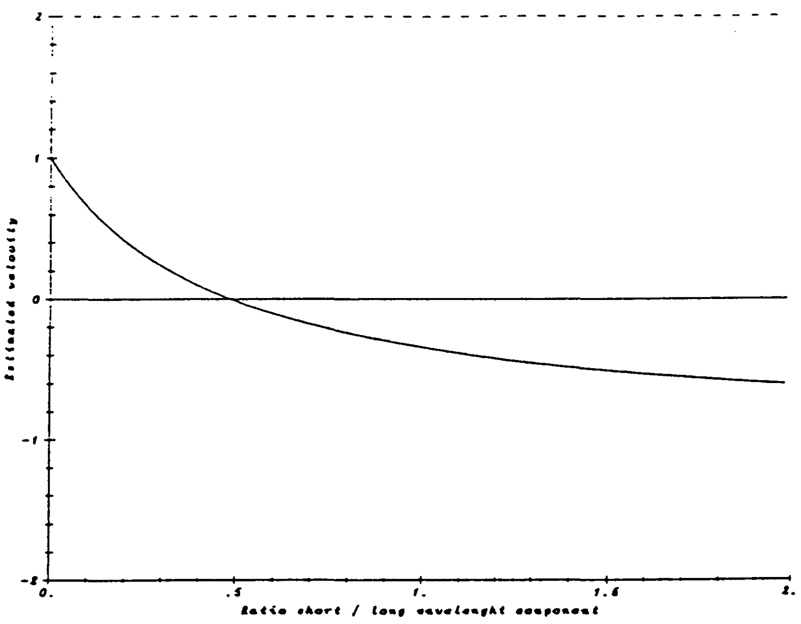


Figure 3.2: Measured velocity for superposition of sinusoidal patters as a function of the ratio short / long wavelength component. 3pt approximation for derivatives is used. The correct velocity is equal to 2 ($\Delta t = 1$).

on measuring the amplitude of the brightness gradient in different areas. In fact, in areas with small gradient the optical flow estimate is mainly obtained by filling in flow information from areas with larger brightness gradient [56].

This criterion however is not only insufficient but it may also lead to deterioration of an initially correct estimate. In fact, areas with large gradient values tend to contain high frequencies and therefore to be plagued by discretization errors, which in this way will be propagated to nearby regions. A better criterion should consider both the discretization and the quantization (noise) errors.

In section 3.4 we shall derive a local estimate for the *overall relative error* in the optical flow that takes into account these two contributions. This estimate will then be used as a *local* criterion to choose the appropriate scale for the estimation of the optical flow on a given part of the image, as will be proposed in section 3.5.

3.4 Estimation of the Flow Field Error

We shall now derive an estimate for the relative error in the flow field. It is worth noting that this estimate does not depend on the algorithm used for recovering the optical flow as long as it assumes the brightness constancy equation.

The error will be derived in the one-dimensional case and then extended to two dimensions using rotational invariance.

We first consider the contribution to the flow field error due to the approximation of the brightness derivatives. Let $f(x - vt)$ be a one-dimensional translating brightness profile. Taylor's expansion yields the three-point approximation for the first order brightness derivatives

$$\frac{f(y+h) - f(y-h)}{2h} = f'(y) + \frac{f'''(y)h^2}{6} + O(h^3). \tag{3.9}$$

In 1-D, the *brightness constancy equation* 3.1 reduces to $E_x v + E_t = 0$, where $E(x,t) = f(x - vt)$. It is easy to show (see appendix B) that, neglecting higher order terms, the three-point approximations of the temporal and spatial derivatives are given by

$$\tilde{E}_t \approx f_t - \frac{vf'''(y)}{6}(v\Delta t)^2 \quad \text{and} \quad \tilde{E}_x \approx f_x + \frac{f'''(y)}{6}(\Delta x)^2, \tag{3.10}$$

where Δx and Δt are the spatial and temporal sampling steps. By substitution in the brightness constancy equation, we obtain an approximated expression for the measured flow field

$$\tilde{v} = -\frac{\tilde{E}_t}{\tilde{E}_x} \approx -\frac{f_t - \frac{vf'''(y)}{6}(v\Delta t)^2}{f_x + \frac{f'''(y)}{6}(\Delta x)^2} \tag{3.11}$$

which leads (by second order Taylor's expansion) to the relative error

$$\frac{\delta v}{v} = \frac{\tilde{v} - v}{v} \approx \frac{f'''(y)}{6f'(y)}((v\Delta t)^2 - (\Delta x)^2). \tag{3.12}$$

Thus, provided higher order terms can be neglected, the relative error in the flow field due to the three-point approximation of the brightness derivatives is close to zero when the interframe motion $v\Delta t$ is of the order of the spatial sampling step Δx. In particular, for a sinusoidal brightness profile $sin\frac{2\pi}{L}(x - vt)$ of wavelength L, we have

$$\frac{\delta v}{v} \approx \frac{2\pi^2}{3L^2}((\Delta x)^2 - (v\Delta t)^2). \tag{3.13}$$

In practice, the image brightness is corrupted by quantization error and by noise so that we cannot expect the constancy equation to hold exactly. We shall now estimate the flow field relative error due to the quantization of the intensity levels. Clearly, this provides a lower bound on the relative error due to the noise in the image brightness. The one-dimensional constancy equation $v \approx -\frac{E_t}{E_x}$ leads, when v is not null, to the expression for the relative error

$$\frac{\delta v}{|v|} \approx \sqrt{(\frac{\delta E_x}{E_x})^2 + (\frac{\delta E_t}{E_t})^2}, \tag{3.14}$$

where δE_x and δE_t are the errors on the temporal and spatial derivatives respectively[7].

If we consider the errors induced by the quantization process, we have (assuming that the image intensity is an integer going from 0 to a maximum value n, so that the minimum amount of "observable" intensity difference is 1, see also figure 3.3):

$$\delta E_x \approx \frac{1}{\Delta x} \quad \text{and} \quad \delta E_t \approx \frac{1}{\Delta t}. \tag{3.15}$$

[7]To derive eqn. 3.14 we made the assumption $\delta v \approx \sqrt{(\frac{\partial v}{\partial E_x})^2 \delta E_x^2 + (\frac{\partial v}{\partial E_t})^2 \delta E_t^2}$, valid for a Gaussian distribution of the errors.

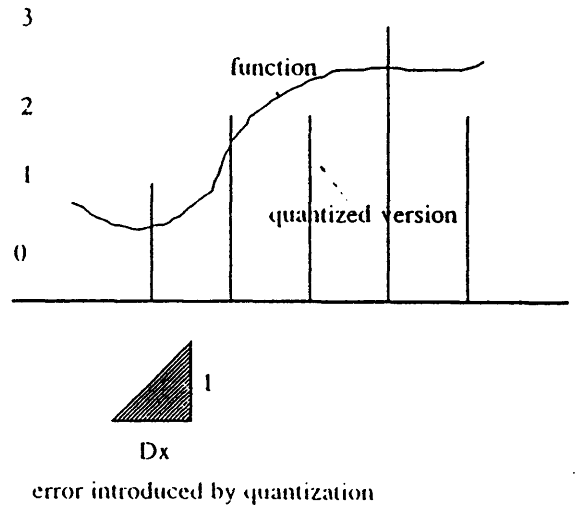


Figure 3.3: Error in derivative estimation due to quantization of intensity values.

This crude approximation is sufficient because the error estimate will be compared with a user-defined threshold in the adaptive scheme. A better average precision in the estimated relative error can be obtained by using the *average discretization error*. This average can be calculated using a statistical model for the considered images, as explained in [32].

Since $|v| \approx | \frac{E_t}{E_x} |$, we can rewrite

$$\frac{\delta v}{|v|} \approx \frac{\sqrt{(\delta E_x)^2 + (\frac{\delta E_t}{v})^2}}{| E_x |} \approx \frac{\sqrt{\frac{1}{(\Delta x)^2} + \frac{1}{(v \Delta t)^2}}}{| E_x |}. \tag{3.16}$$

In the following we shall denote the spatial and temporal differences with $\Delta_x E$ and $\Delta_t E$, respectively. Now $|E_x| \approx | \frac{\Delta_x E}{\Delta x} |$ and therefore

$$\frac{\delta v}{|v|} \approx \sqrt{\frac{1}{(\Delta_x E)^2} + \frac{1}{(vE_x\Delta t)^2}}. \tag{3.17}$$

By the constancy equation, which holds approximately,

$$\frac{\delta v}{|v|} \approx \sqrt{\frac{1}{(\Delta_x E)^2} + \frac{1}{(E_t\Delta t)^2}} \tag{3.18}$$

and since $E_t \approx \frac{\Delta_t E}{\Delta t}$,

$$\frac{\delta v}{|v|} \approx \sqrt{\frac{1}{(\Delta_x E)^2} + \frac{1}{(\Delta_t E)^2}}. \tag{3.19}$$

Finally, we get an overall estimation of the flow field relative error due to the three-point approximation of the derivatives and to the quantization of the image intensity[8]:

$$\frac{\delta v}{|v|} \approx C(x) \mid (\Delta_t E)^2 - (\Delta_x E)^2 \mid + \sqrt{\frac{1}{(\Delta_x E)^2} + \frac{1}{(\Delta_t E)^2}}, \tag{3.20}$$

where the function $C(x)$ depends on the first and third brightness derivatives at the image point x under consideration.

The first term refers to the approximation of the derivatives and can be derived from 3.12 using the constancy equation and the two basic expressions $E_t \approx \frac{\Delta_t E}{\Delta t}$ and $E_x \approx \frac{\Delta_x E}{\Delta x}$. Since this term does not depend on the number n of brightness quantization levels and since $\Delta_t E$ as well as $\Delta_x E$ are proportional to n, the function $C(x)$ must be proportional to $\frac{1}{n^2}$. This relation can be shown for a sinusoidal intensity profile. In that case, the first term of eqn. 3.20 can be rewritten, according to 3.13, as

$$\frac{2\pi^2(\Delta x)^2}{3L^2}((\frac{\Delta_t E}{\Delta_x E})^2 - 1) \tag{3.21}$$

[8]Given the approximated nature of the estimate, a simple summation of the quantization and discretization errors is used in this final step.

Let's introduce the parameter ρ (*fractional range* of intensity values in a given image), defined by ρ = (maximum_intensity - minimum_intensity)/n. The typical scale for the value of the brightness derivative is given by the range of intensity values of the sinusoid ρn, divided by the wavelength L (i.e., $\frac{\Delta_x E}{\Delta x} \approx \frac{\rho n}{L}$). This latter relation implies that $(\frac{\Delta x}{L})^2 \approx (\frac{\Delta_x E}{\rho n})^2$, which leads (by substitution in eqn. 3.21) to the inverse relation between $C(x)$ and n^2. After completing the cited substitution, the equation for the error estimation that was used in the tests is the following:

$$\frac{\delta v}{|v|} \approx \frac{C}{\rho^2 n^2} \mid (\Delta_t E)^2 - (\Delta_x E)^2 \mid + \sqrt{\frac{1}{(\Delta_x E)^2} + \frac{1}{(\Delta_t E)^2}}, \tag{3.22}$$

where the value for C is $\frac{2\pi^2}{3}$ as suggested by the above argument. For a general image, the the fractional range of the image ρ was estimated using the standard deviation σ in the distribution of intensity values ($\rho = \sigma/n$).

It is clearly difficult to determine the third derivative of the intensity at every point in the image but our tests show that, as a working hypothesis, we can consider it as a constant independent of the image position. In practice, we shall use the constant estimated for sinusoidal gratings given in eqn. 3.22. The "difference" terms (like $\Delta_x E$) grow linearly with the number of discretization levels n. Therefore, while the first term of the overall relative error does not depend on n, the second term, which expresses the contribution due to the quantization process, decreases with n and can be eliminated (at a price!) by increasing the number of quantization levels.

The quantity in expression 3.22 is clearly only an approximation of the overall relative error. Note that approximations are necessary since it is (clearly) not possible to evaluate the error in the optical flow precisely without knowing precisely the optical flow itself.

It is important to point out that the final result in eqn. 3.22 presents in a concise way the tradeoff between the two kinds of errors introduced. According to this criterion, the "close-to-optimal scale" is the one that locally minimizes this relative error.

The two-dimensional estimate of the overall relative error is obtained from eqn. 3.22 by rotational invariance, substituting $(\Delta_x E)^2$ with the sum of the squared differences in the two dimensions $(\Delta_x E)^2 + (\Delta_y E)^2$. This amounts to measuring the field unreliability according to the error on the component of the velocity that is normal to the brightness gradient.

3.5 The Error-Based Adaptive Multiscale Scheme

Our proposed strategy is based on a low-to-high resolution scheme. The resolution pyramid used will be described in section 3.6 and the locally adaptive discretization strategy is implemented with the use of an ***inhibition flag*** associated with each point in the image.

Preliminary processing consists in building the Gaussian pyramid associated with the image [49]. The Laplacian pyramid data are then obtained at every scale by expanding the intensity values at the coarser scale and subtracting them from those at that scale (details about the procedure are in [49]). This procedure is equivalent to performing at each level a difference of Gaussians (DOGs) that are a reasonable approximation of Laplacians of Gaussians [36].

Computational time is reduced with respect to filtering with masks with large sizes, while the produced zero crossings are hardly distinguishable from those obtained by filters with a large mask in all our test images.

When the above data are obtained, the Horn and Schunck relaxation algorithm described in equations 3.5 and 3.6 (with Gauss-Seidel lexicographic updating) is applied starting from the lowest resolution for a selected number of cycles. As will be shown in section 3.7, if the image contains different moving objects it is important to activate *line processes* in order to avoid smoothing over discontinuities [74,69,31].

After the relaxation cycles are finished, the overall estimation of the flow field relative error (according to eqn. 3.22) is calculated for every pixel at the given resolution. This quantity is then used to decide about the local reliability of the optical flow. For every pixel a test is done to see whether the error is below a defined threshold T_{err} or if the pixel is already inhibited. If the test is positive, the grid point corresponding to this pixel at the finer resolution in the pyramid and its immediate four neighbors (in the east,west,north,south directions) are inhibited.

The optical flow values are then interpolated (with bilinear interpolation) to the next finer scale where they are used as initial approximation for further local relaxations. *Inhibited* pixels will not participate in the relaxation process and will maintain the optical flow values interpolated from coarser resolutions, preventing loss of the reliable estimation due to incorrect derivatives at the new scale (as explained before and in the final tests).

This procedure is then repeated iteratively, where relaxation occur only in the regions where the approximation obtained at the coarser scale is not yet satisfactory.

The optimal grid structure for a given image is translated into a pattern of active and *inhibited* grid points in the pyramid, as illustrated in figure 3.4.

Final result of the computation is a reconstruction of the optical flow at the different resolutions with an explicit indication of the motion discontinuities, with an associated measure of the optical flow reliability. This information will be used by subsequent visual modules.

In the present scheme, computation starts from a field equal to zero on the coarsest scale, while in a real-time continuous scheme it should start from the previously determined field.

3.6 The Resolution Pyramid

In this work we consider a 2:1 resolution pyramid built from a sequence of three images. The coarser versions of these images are obtained by local averaging using the the 5-point and one-parameter mask proposed by Burt [49]. The mask is essentially an approximation of a Gaussian filter with support given by five points [49].

The procedure is then repeated iteratively to construct the other low resolution versions of the three images. For an appropriate choice of the parameter the result closely approximates the convolution of the original images with Gaussian filters of appropriate width [49]. If the original images are noisy an additional preliminary filtering with a Gaussian filter whose size can be selected by the user is applied to the sequence of raw images. In the tests that we carried out, the finest scales contained 129x129 pixels. The number of layers depends on the image size (a pyramid with 3 different resolutions is used for the chosen size).

The spatial and temporal derivatives of the brightness are calculated independently at each level of the pyramid using the three-point approximations [35]. Besides the higher accuracy, the three-point formula has the advantage of estimating the derivatives at the intermediate frame instead of between the frames like the two-point one. This fact will turn out to be very useful in the discontinuity detection process (as will be illustrated in the following section). The "flow of information" from the three images to the estimated derivatives and error is shown in figure 3.5.

3.7 Discontinuity Detection

For typical scenes the optical flow is piecewise smooth so that discontinuities are necessary for a faithful recovery of the flow field. Some authors suggested using "oriented smoothness" constraints [38], adapting the constraints to the local differential structure of the intensity "surface."

Others succeeded in considerably improving the effectiveness of the Horn and Schunck algorithm by introducing explicit binary discontinuity elements (line processes) [31] on a grid halfway between the grid formed by the image pixels. The line processes are either "on" or "off" depending on whether the smoothness term between the corresponding points of the image has to be neglected or not. Some additional terms are added to the energy Φ (which is no longer quadratic) in order to control the spread of the line processes activation. The line processes are then updated to minimize Φ. To effectively combine the estimation of the optical flow and the discontinuity detection, their updating cycles are separated in time by a few relaxation cycles. It is worth noting that even though it is no longer guaranteed to converge to a global minimum, the system reaches good minima of Φ [31].

A positive feature of the last method is that the additional information contained in the discontinuities can be used by following high-level vision modules.

The method presented in this work detects discontinuities at different resolutions by introducing line processes at every scale. The type of neighborhood considered for a line process (LP) within a given level of the pyramid is shown in figure 2.3. The line process activation must clearly be favored by a large difference in flow field magnitude between the two closest points in the image and by the possibility of improving the local discontinuity structure. We define the *benefit* of the LP between pixels (i,j) and $(i+1,j)$ as

$$benefit = ((u_{i+1,j} - u_{i,j})^2/h_k^2 + (v_{i+1,j} - v_{i,j})^2/h_k^2),$$

where h_k is the spatial step at the kth scale. This is proportional[9] to the amount by which the activation of the line element can decrease the energy Φ. A similar expression defines the benefit of a horizontal LP between pixels (i,j) and $(i,j+1)$.

The *cost* of a line process depends on the local discontinuity structure and it is defined from a basic cost C that corresponds to the typical size of the flow field discontinuity that we want to detect. The possible local structures are then classified depending on the number n of continuous regions in which they are divided and a parameter C_n is associated to each class. Now the cost of a line process with a local structure composed of n regions is given by $cost = C * C_n$ and a line process is activated if and only if $benefit > cost$.

In order to combine the discontinuity detection at the different scales, the line processes can also interact with contiguous ones in the two adjacent levels (see figure 2.4). The state of a line process at a level k, $0 \leq k \leq L$, influences also the cost of its neighboring line elements in the levels $k-1$ and $k+1$. If this element is "on" it will decrease by a certain factor F_{ud} (where "ud" is for "up or down") the cost of its neighboring line elements in the two adjacent levels. The cycles of line process updates are then combined at every scale with the relaxation sweeps. This type of coordination scheme gave good results when applied to surface reconstruction and the look-up table approach was very efficient [62].

The detection of optical flow discontinuities can be improved further by using information on the intensity discontinuities [24]. As in [31], we prevent the activation of line process where there are no zero-crossings of the Laplacian-of-a-Gaussian filter (at the different scales) unless there is strong evidence for the flow discontinuity.

This is realized by choosing a basic cost C that is large with respect to the typical size of the flow field and a very large factor F_{zc} by which the presence of a zero-crossing

[9]Because of the α^2 factor in eqn. 3.2.

decreases the cost of the corresponding line processes.

Zero crossings of $\nabla^2 G$ have been chosen because of their useful properties. For example, they are not created as the scale increases and they form close contours, unless they intercept the boundaries. A detailed description of other properties is in [43].

3.8 Experimental Results

Tests of the proposed algorithm have been done for images of varying complexity. To measure in a quantitative way the correctness of the derived optical flows, images have been generated in a controlled environment, combining different parts with different textures (both "natural' and artificial). The image generating "tool" allows the user to select different movements for the different parts, producing a sequence of three images (the data for the algorithm) with the associated motion flow (used for comparison).

Test results are presented both in visual form (display of the obtained optical flow) and in graphical form (graph of the root mean square (r.m.s.) error between the correct motion flow and the obtained optical flow).

Finally the results of tests with a sequence of video-acquired images of a natural scene are presented.

3.8.1 Two-Dimensional Sinusoidal Patterns

These examples demonstrate the necessity of an adaptive scheme based on a measure of the optical flow reliability.

The images show a "plaid" pattern, a superposition of sine waves of different wavelengths in the vertical and horizontal directions. The intensity of a pixel with coordinates (i, j) is obtained according to the following formula:

$$I(i,j) = \frac{255}{4(1+R)^2}(1+R+\sin(\frac{2\pi}{L}i)+R\sin(\frac{2\pi}{l}i))(1+R+\sin(\frac{2\pi}{L}j)+R\sin(\frac{2\pi}{l}j)) \quad (3.23)$$

The relative amount of short versus long wavelength component is determined by the parameter R, the intensity is normalized to obtain values in the range (0-255).

The first example illustrates the basic difficulty arising in a multiscale strategy. The parameter R is 1.0, the long and short wavelengths are 7.5 and 3.2[10]. The resulting image is in figure 3.6.

Movement is a translation in the plane in the north-east direction with velocity equal to (1 , 2).

Comparison of the results of the homogeneous versus the adaptive coarse-to-fine strategy are shown in figure 3.7. Ten iterations are done on every discretization grid, bilinear interpolation of results is applied before relaxation is initiated on a finer grid.

Relaxation on the coarsest grid produces an optical field whose difference with the correct motion flow *increases* as a function of the iteration number. This is caused by large errors in derivative estimation on this grid (the intensity is almost constant and discretization errors, which are the dominant error term in this case, are large).

The situation is better on the intermediate grid. In spite of incorrect initial values obtained from the coarser grid, the error is rapidly reduced after the first relaxations. Error in derivative estimation reaches in this case the minimum value (motion on this scale is less than the dominant wavelength).

After interpolation to the finest grid, the homogeneous scheme continues the relaxation process, driving the result to a *worse* solution. This is again caused by bad derivative estimation (motion on this scale is not small in comparison with the shorter wavelength). On the contrary, the adaptive scheme recognizes that the error on the intermediate scale is lower than the given threshold T_{err} (0.4 in this case) in most of the image pixels, so that the computational units corresponding to these pixels at the finer scale are "inhibited" (no relaxation is done) and the error in the obtained optical flow is similar to that on the middle scale.

The difference in the qualitative structure of the derived optical flow can be appreciated in figure 3.8.

Finally figure 3.9 shows a display of the *estimated* error (according to eqn. 3.20) on the different scales.

The quantization error is larger at the coarser scale, while the derivative estimation

[10]These represent "generic" wavelengths (not multiples of the grid step to avoid particular effects), chosen to give different "dominant" components at different scales.

error is larger at the finest scale. The total error reaches the minimum on the middle scale, in agreement with the results about the r.m.s. measured error.

In more complex images, errors will be greatly different in different part of the image so that the reconstructed optical flow will be "frozen" at different scales, as will be shown in following examples.

3.8.2 Expanding Sphere

These examples illustrate the difficulties due to propagation of velocity field information across boundaries between different moving objects.

A set of ray-traced images of an expanding sphere was chosen because it was used in Terzopoulos [61] as an example of the speed-up that can be obtained with the multigrid algorithm. The spheres are superimposed on a fixed "natural" background, in order to provide derivatives different from zero on the background[11]. These images contain a unique dominant spatial frequency of the order of magnitude given by the sphere diameter.

If we do not consider the effect of quantization (255 intensity levels) and assume that the motion amplitude is very small with respect to the radius, one iteration is sufficient to recover the correct optical flow, as can be seen from equations 3.5 and 3.6 in the special case of a velocity vector perpendicular to the brightness gradient. The function of relaxation is, in this case, to provide a better estimate by averaging noisy derivative estimations on neighborhoods with a size that increases with the number of relaxations applied[12].

Unfortunately, this is true only if one assumes that the occluding boundary is known *a priori* and if the correct velocity is given on this boundary, as was the case in Terzopoulos' work [61]. In the general case (no initial information) different results are possible. As will be shown in the following tests, the r.m.s. error increases for small spheres (because noisy information on the boundary is propagated in both directions), while for larger spheres it first decreases (for the averaging effect) but after a few iterations increases (an average over very large neighborhoods becomes worse than the

[11]If derivatives are zero, all motion fields minimize the Horn and Schunck functional.

[12]With a "Gaussian" weighting of the different derivative estimates.

original estimate) with a speed proportional to the parameter α in the cited equations.

The graphs in figure 3.10 show the behavior of the r.m.s. error as a function of the work units, for two different values of the sphere radius (55 and 95 pixels). Movement is an expansion (3 pixels per frame on the border of the sphere). Both the single scale and the multiscale algorithms are tested.

For the smaller sphere, single scale relaxations make the error *worse*. The multiscale algorithm does not improve the result. The r.m.s. error as a function of work units in not monotonic (see graph), and the last fine scale iterations show an increasing error. The effect of the boundary is particularly bad at the coarsest scale because the ratio boundary / internal points is large.

For the larger sphere (the sphere boundary is now outside the visible window of 129x129 pixels) the situation is different. Single scale relaxations improve the r.m.s. error at the beginning. The error reaches its minimum when 4 work units are completed, then it increases. In this case the multiscale approach reduces the error faster (the minimum is reached after 1.06 work units). But the minimum value is reached on the middle scale and error becomes larger on the finest scale.

The following figure shows the optical field obtained with the multiscale algorithm in the two cases.

These examples show that the effect of the boundary conditions on the result is indeed an important one. Going from an exact *a priori* knowledge of the occluding boundary with their velocity values to a situation where the only boundary conditions are the "natural" boundary conditions at the border of the image, can lead to very different results.

If an exact knowledge of the occluding boundary information is missing, incorporation of a boundary detection step in the algorithm is essential in order to avoid smoothing across regions corresponding to different moving objects, as will be shown in the next section.

3.8.3 Occluding Objects

This test compares the result obtained with or without discontinuity detection. It shows that the optical flow near an occluding boundary may be reconstructed with large errors unless the smoothing process is blocked by line processes.

The images contain two spheres of different sizes (radius are 35 and 24 pixels), translating with velocities (0.0 , -1.0) and (0.8 , 0.2) against a natural background. Their reflectance patterns are sinusoidal grids (L is 13.3) of a different intensity range mapped onto them using polar projection (in order to obtain a wide range of Fourier components in the different regions of the spheres), while illumination is coming from a source at infinity orthogonal to the image plane.

The parameters for the discontinuity detection process are $C = 1.0$, $C_n = (0.6, 0.5, 2.0, 5.0, 100.0)$, and $F_{zc} = 0.25$. While the parameter C is essential and is related to the typical scale of the motion discontinuities, it is important to emphasize that the precise value of the other parameters is not as important as their relative sizes (which are chosen in order to favor continuation of existing lines and to discourage formation of junctions with more than two line processes and parallel lines)[13].

The following figures illustrate the optical flow obtained with the adaptive multiscale process, using 6 relaxations on each of three scales. The first image shows the result obtained without discontinuity detection, while the second one shows the result when a discontinuity detection step has been done every 2 relaxations.

The qualitative results are confirmed by the graph of the measured r.m.s. error in the optical field for the two cases.

Although zero crossings are dense for this image (as shown in figure 3.14), the final activation of the motion discontinuities corresponds in a reasonable way to the real motion discontinuities. This is an indication that the effect of parameter F_{zc} is only that of guiding the discontinuity detection process, while the final placement is dictated by the presence of a real motion discontinuity.

3.8.4 Tests with Natural Images

The images used for this test show a pine cone moving in the upward direction. They were acquired with a S-VHS video camera and a Targa frame grabber. Movement was

[13]In other terms the C_n parameters could be given as values of a function $f(n, \phi, \chi)$ of a fixed qualitative form and dependent on one or two parameters.

executed by adjusting a tripod sustaining the object by 0.25cm every frame. Measured velocity in pixels is 1.6 pixel / frame. Tests have been done for sets of three images taken every one, two, and three frames. The average velocity (on a window centered on the pine cone) obtained with the homogeneous multiscale algorithm is compared with that obtained with the adaptive version. While this second version always produces a better estimate, the difference is particularly significant for large motion amplitudes, as shown in figure 3.15.

In this case the fine scale derivative information is completely erroneous. This is recognized by the adaptive scheme that *freezes* the solution obtained at coarser grids, producing a better final estimate.

3.9 Summary and Conclusion

We have shown that a simple estimation of the relative error in the flow field can lead to an effective adaptive multiscale scheme for recovering the optical flow. This method provides a more accurate flow field reconstruction by dealing locally with the different types of motions and textures in a generic image. Contradictory derivative estimations on different scales do not cause incorrect optical flow reconstruction. The multiscale strategy finds the first scale (starting from the coarsest one) that produces a reliable estimate and locally freezes the result.

The suggested scheme is especially necessary for scenes with multiple motions and/or multiple patterns and textures.

It is worth noting that the strategy used in this approach is general, in the sense that it does not depend on the single scale algorithm used to recover the optical flow. Other algorithms need to change the error estimation equation in a way that is appropriate for their derivative estimation.

While for the presented tests the parameters involved in the discontinuity detection process have been chosen with a trial and error process, automatic tuning of the parameters is suggested for practical implementations of the presented algorithm.

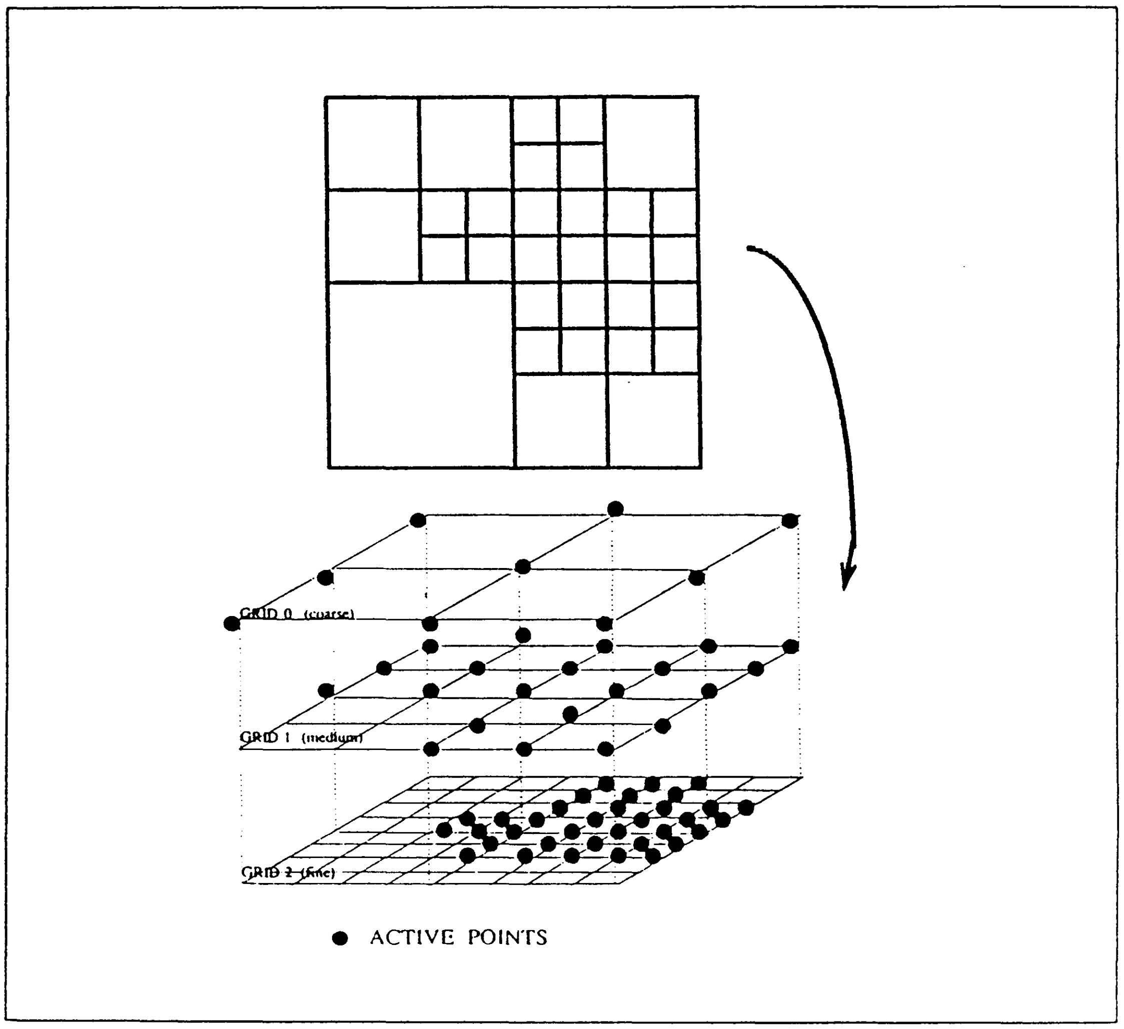


Figure 3.4: Adaptive grid and activity pattern in the multiresolution pyramid.

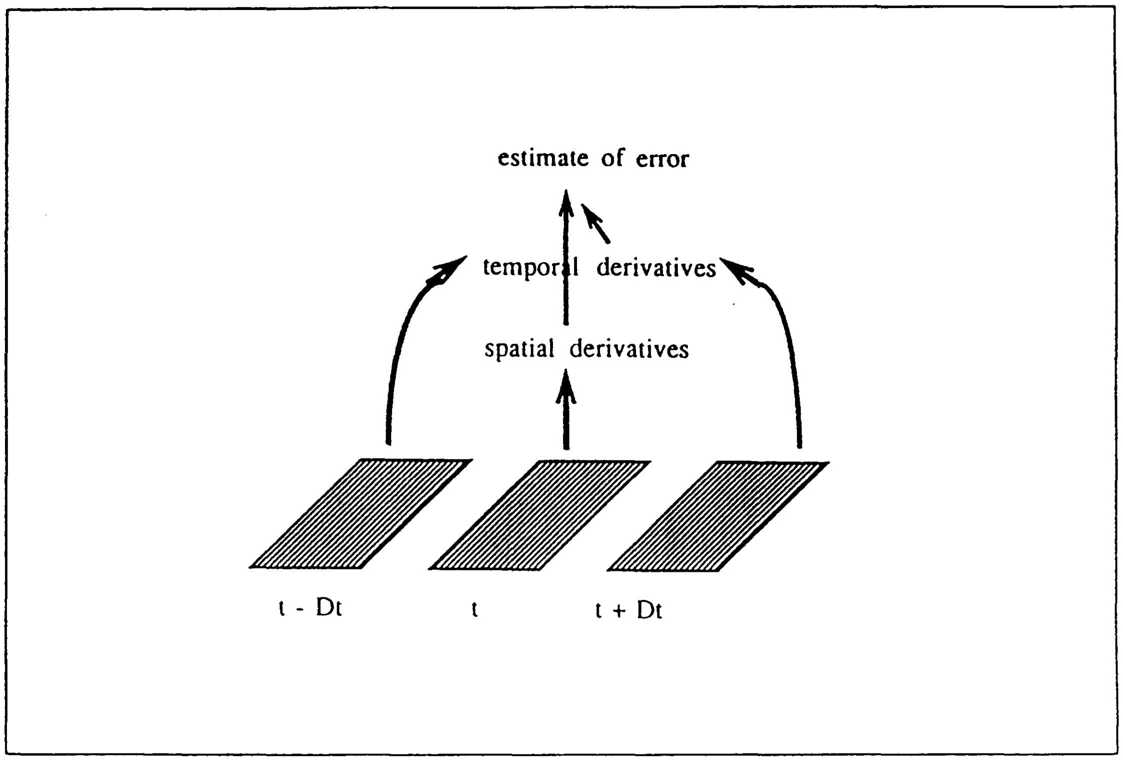


Figure 3.5: Information flow (at each level of the Gaussian pyramid) for the estimation of spatial and temporal derivatives and errors.

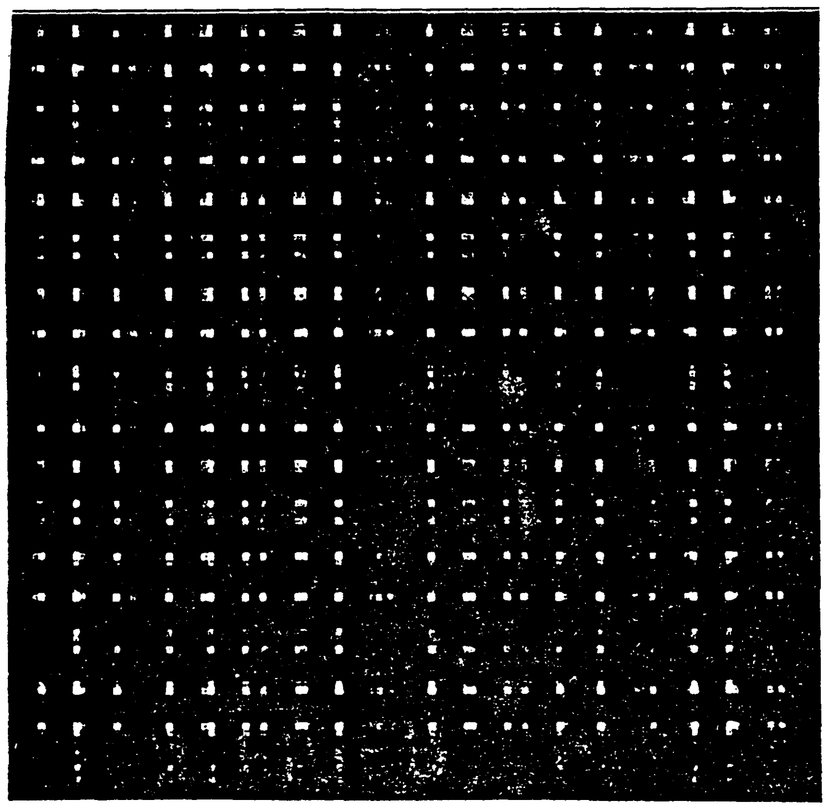


Figure 3.6: "Plaid" image: two-dimensional pattern with long and short wavelengths.

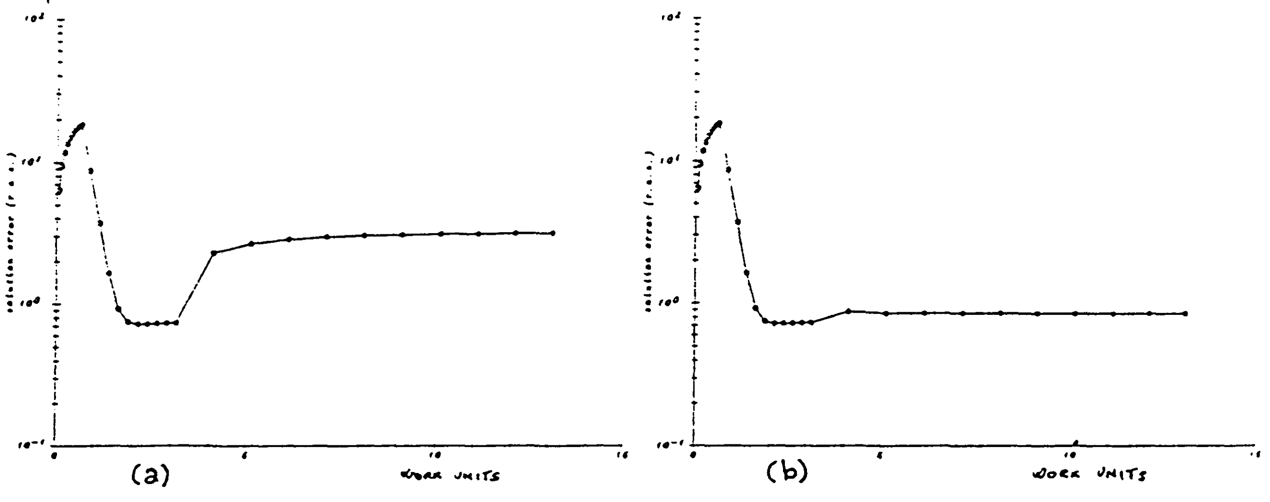


Figure 3.7: Comparison of homogeneous versus adaptive multiscale strategy: translating "plaid" pattern. Graphs show r.m.s. error in the optical flow as a function of *work units* (i.e., as a function of actual computing time, because a *work unit* is defined as the amount of computation used for a complete relaxation at the finest scale). Fig.(a): multiscale homogeneous strategy (with no adaptation). Fig.(b): multiscale adaptive strategy. The adaptive algorithm "freezes" the result at the intermediate grid because the error measure is below threshold T_{err} and interpolates to finest grid.

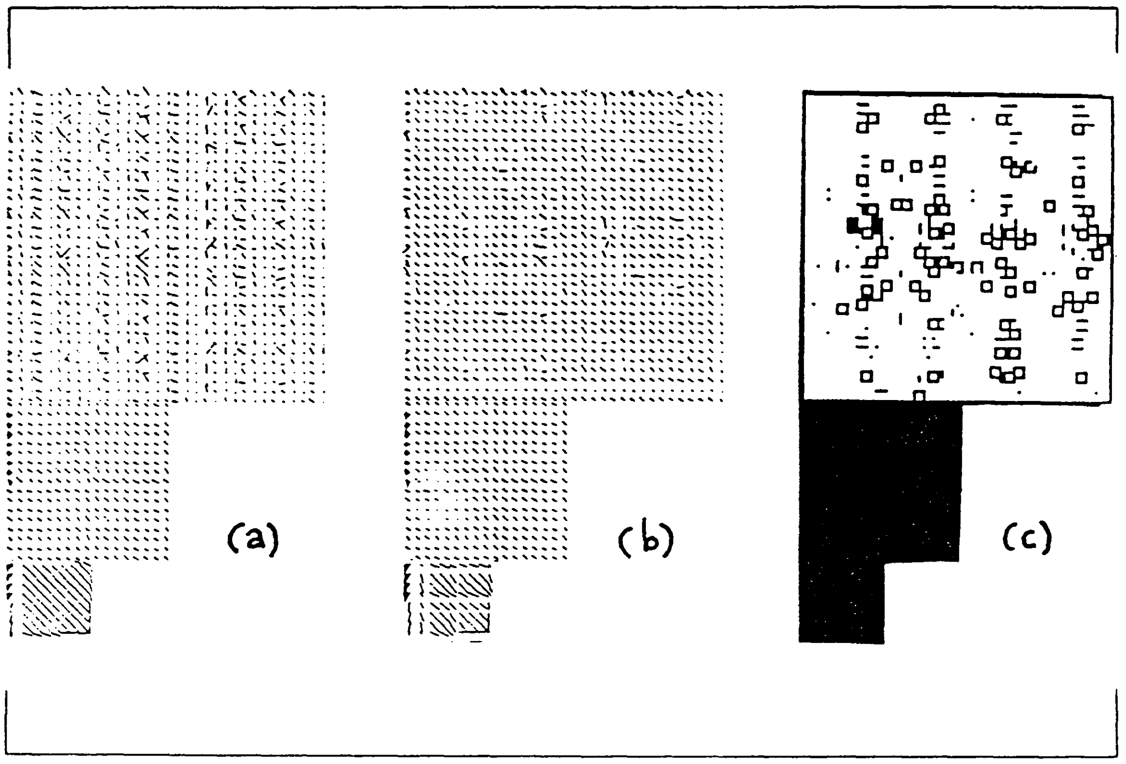


Figure 3.8: Reconstructed optical flow for translating "plaid" pattern. Fig.(a): homogeneous multiscale strategy. Fig.(b): adaptive multiscale strategy. Fig.(c): active (black) and inhibited (white) points.

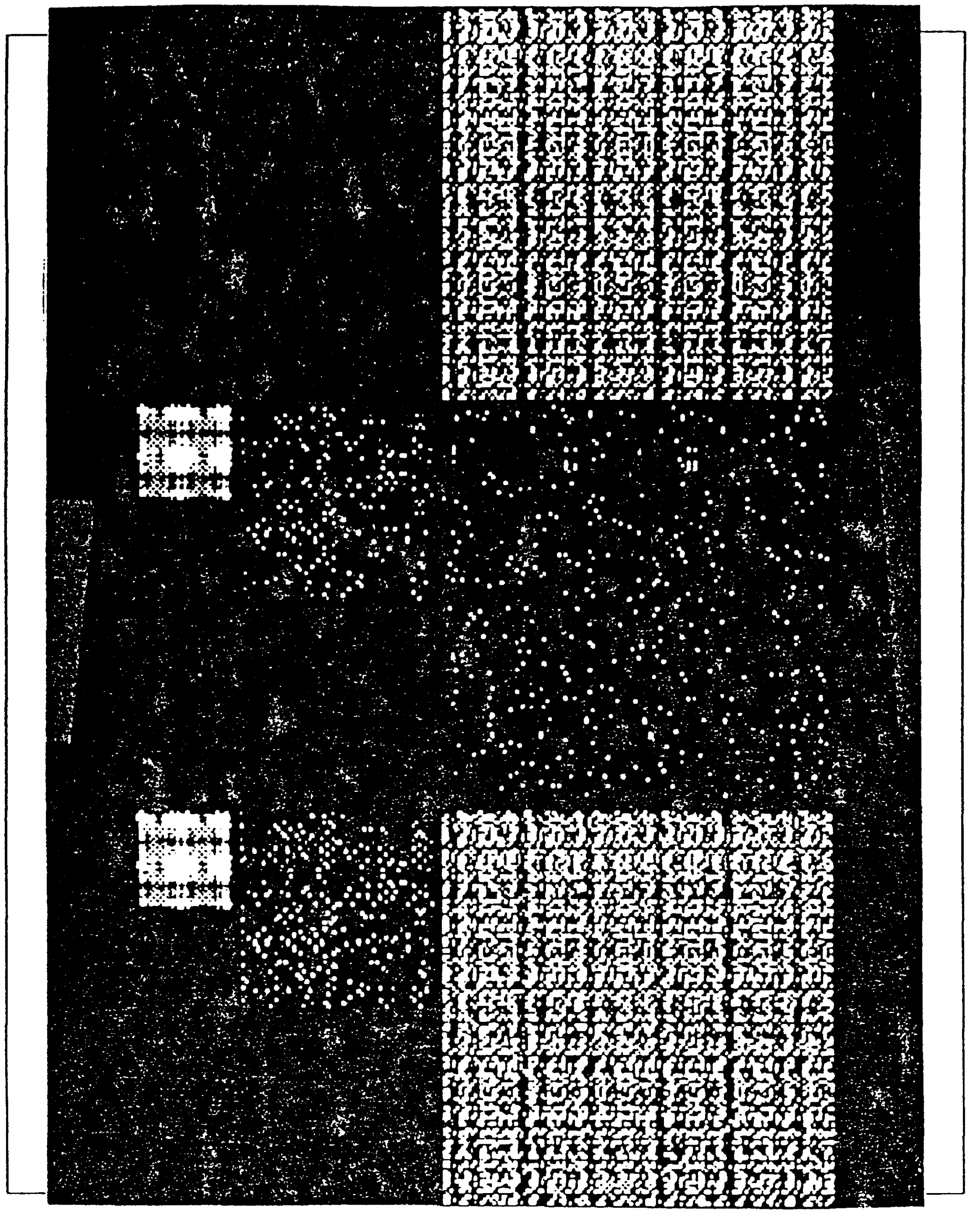


Figure 3.9: Estimated error on different scales. Intensity value is proportional to error. Derivative estimation, quantization, and total error are shown in this order.

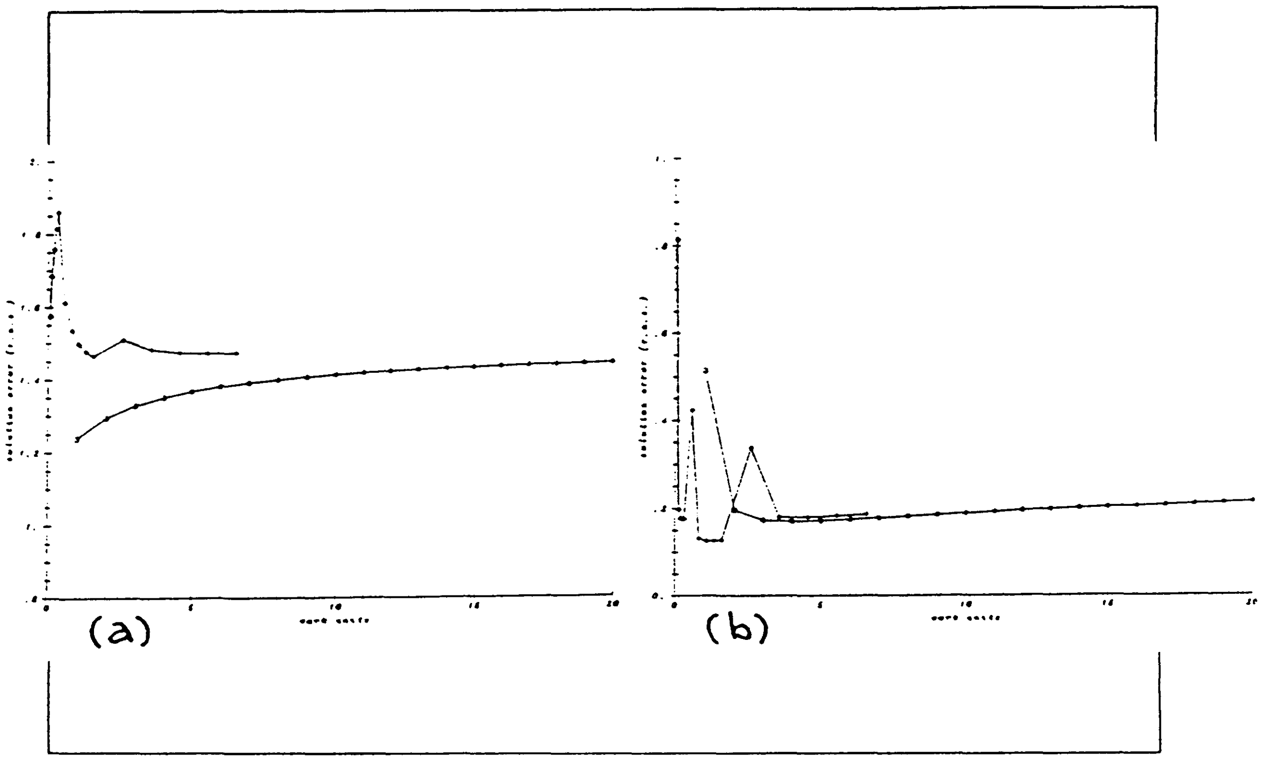


Figure 3.10: Expanding sphere: r.m.s. error as a function of the amount of computation in the multiscale scheme. Fig.(a): small sphere (radius=55). Fig.(b): larger sphere (radius=95). Single scale results (circles) and multiple scale results (diamonds). Interpolation to finer scales increases temporarily the r.m.s. error. Algorithm is terminated after the given number of work units because r.m.s. error is increasing.

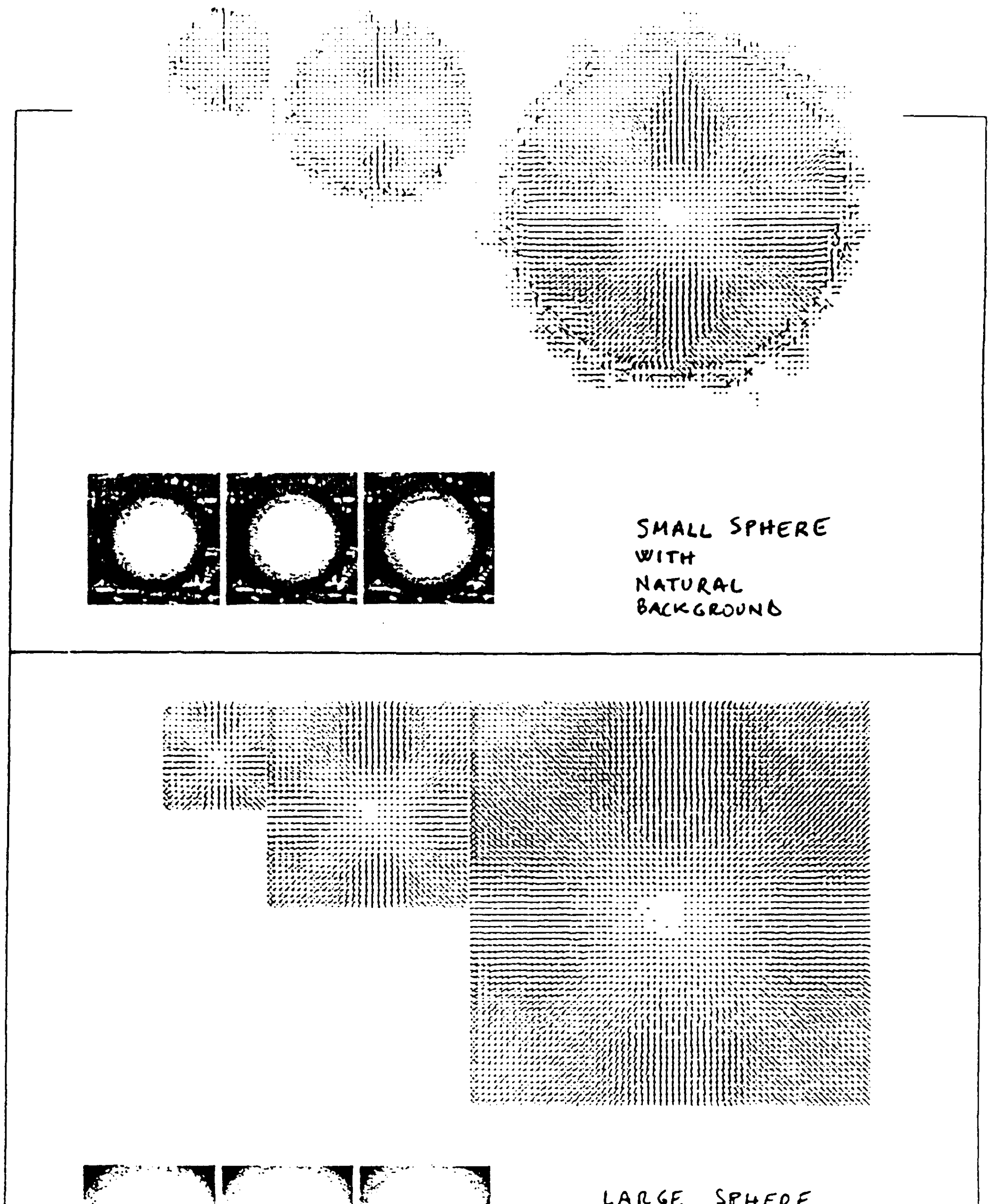


Figure 3.11: Multiscale optical flow for spheres of different sizes. The effect of the sphere boundary on the result is visible for the smaller sphere.

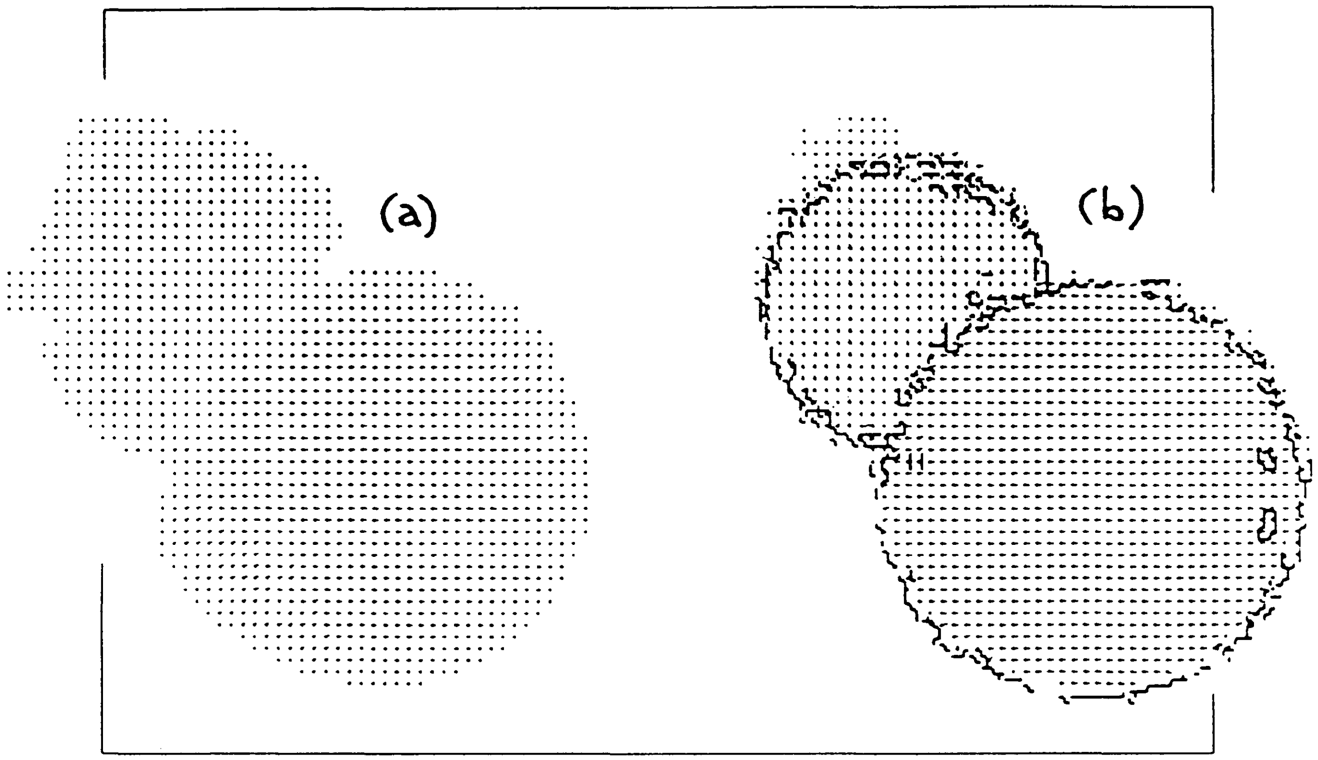


Figure 3.12: Occluding moving spheres: optical flow obtained without (a), and with the concurrent discontinuity detection process (b).

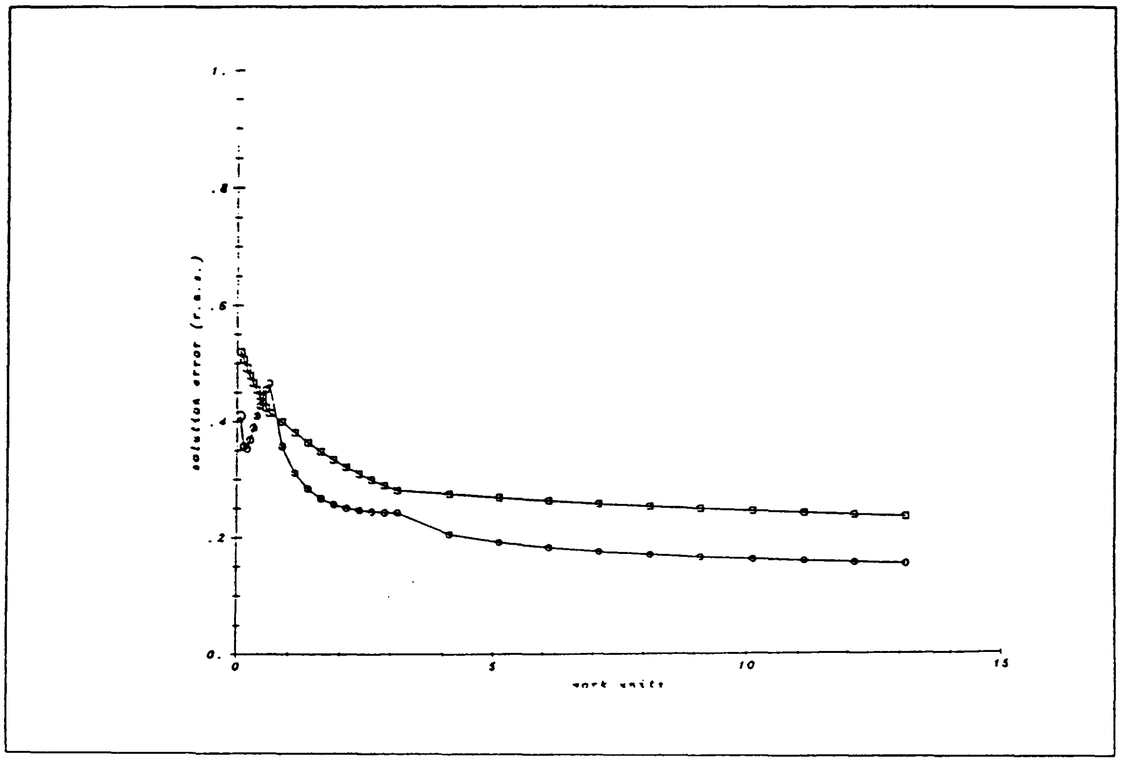


Figure 3.13: Occluding moving spheres: graphs of the r.m.s. error with (o) and without (□) discontinuities.

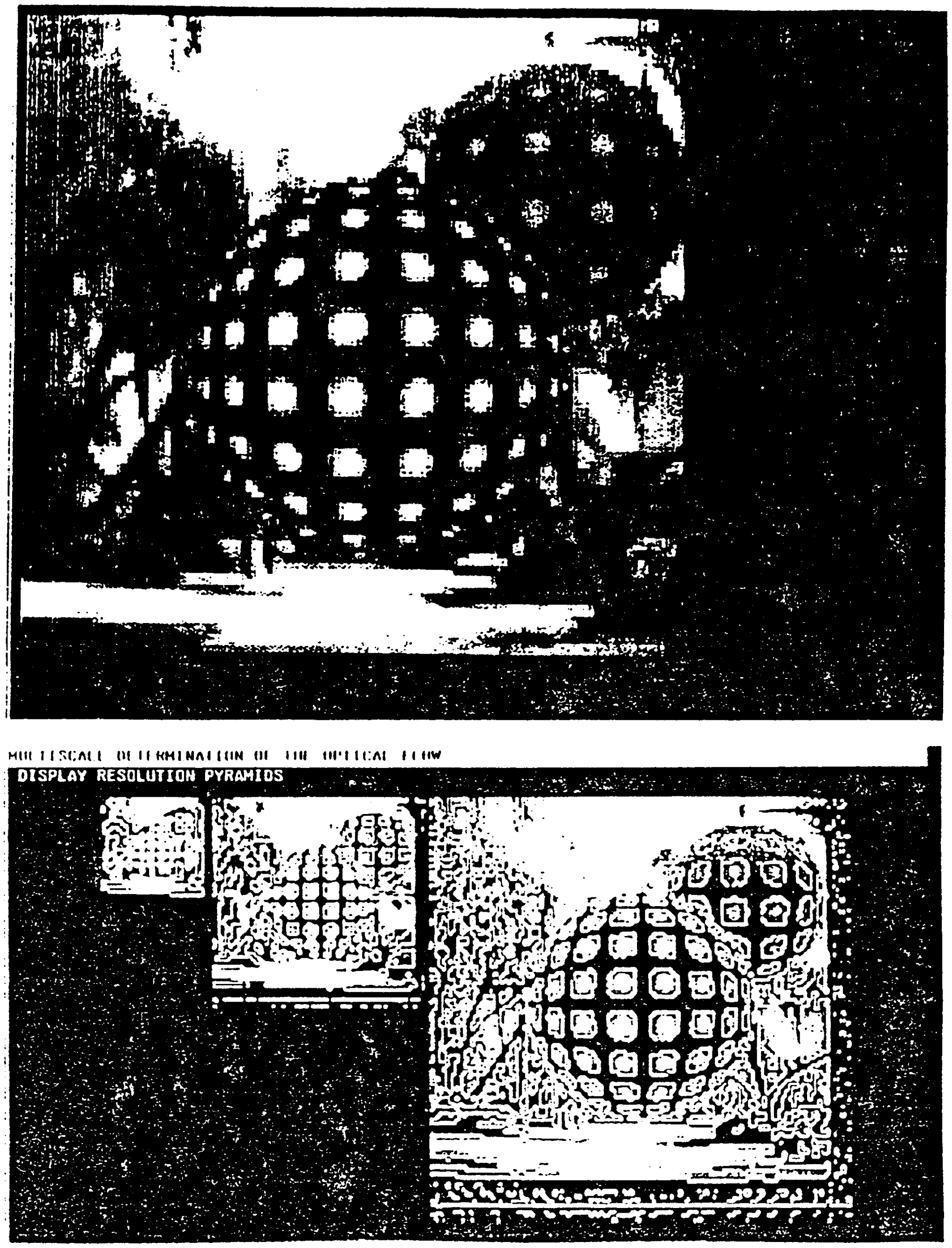
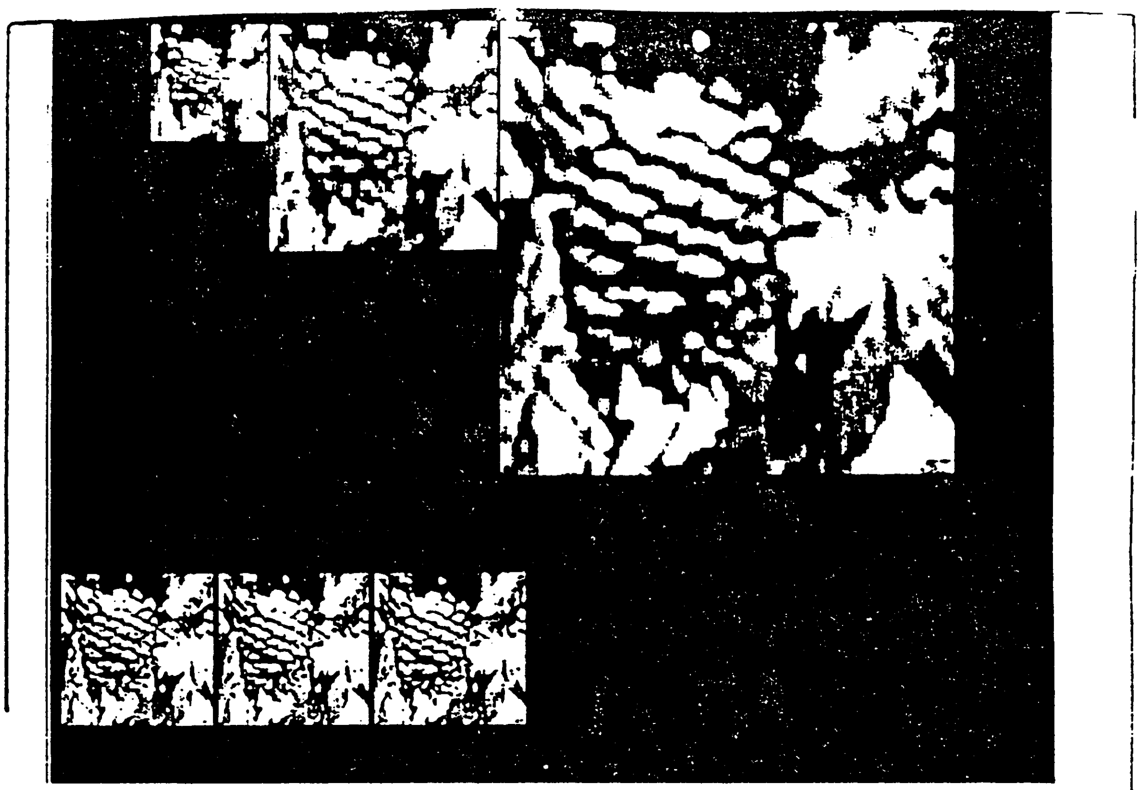

MULTISCALE DETERMINATION OF THE OPTICAL FLOW
DISPLAY RESOLUTION PYRAMIDS

Figure 3.14: Occluding moving spheres: image and derived zero crossings.



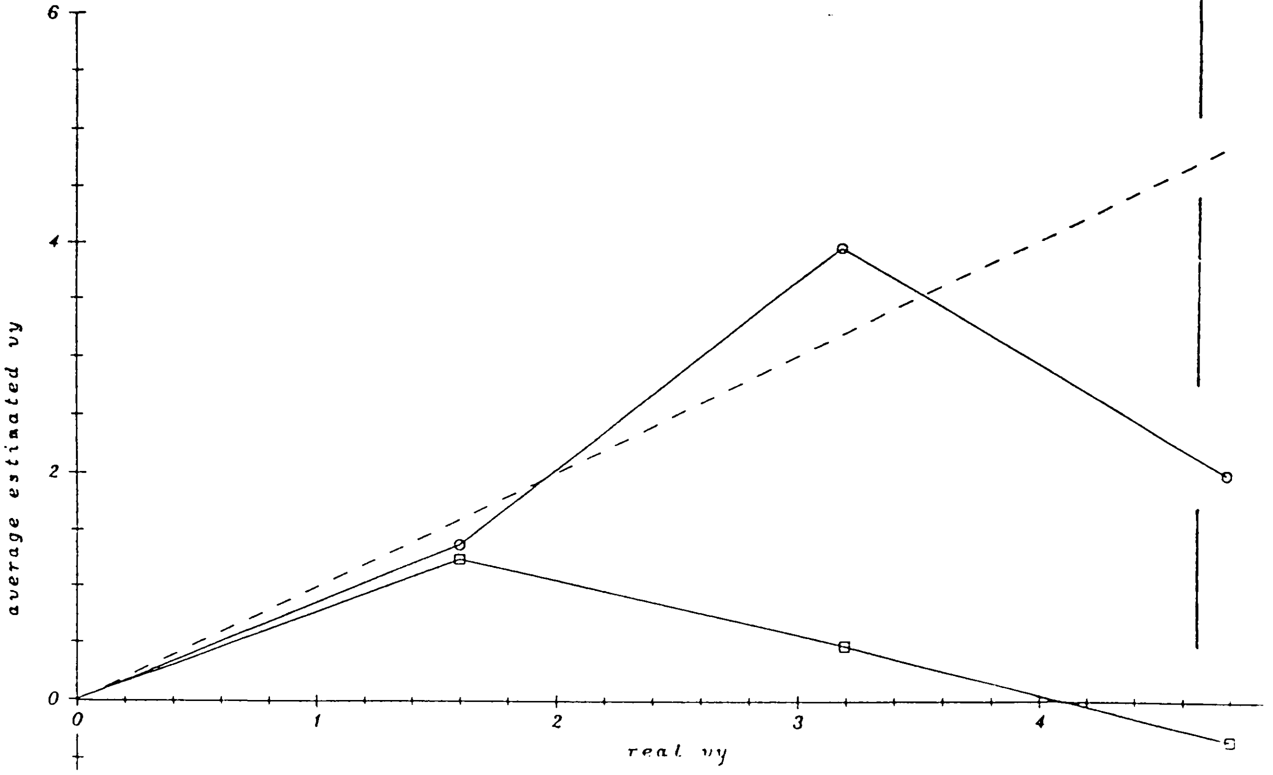


Figure 3.15: Test image and comparison of results. Average velocities (v_y component in the down-up direction) obtained with the homogeneous and adaptive methods are compared with the correct velocity.

Part III

Implementation of the Algorithms on the Hypercube Concurrent Processor

Chapter 4

Benchmark of Applications on the Hypercube

4.1 Introduction: SIMD vs. MIMD Approach

From our experience MIMD computers with powerful processors (≈ 1 Mflop), sufficient distributed memory (≈ 1 Mbyte), and two-dimensional internode connections[1] are a close-to-optimal choice for implementing medium-level vision algorithms (see also [68, 51] and [60] for a general discussion).

In this case a simple two-dimensional *domain decomposition* can be used efficiently: a slice of the image with its associated pyramidal structure is assigned to each processor.

More complex schemes with dynamic load balancing are not needed because a real-time scheme is supposed to produce a solution in the given time in the worst possible case, when all grid units are active (this situation corresponds to images with fine details in all regions of the scene).

All nodes are working all the time, switching between different levels of the pyramid. No modification to the sequential algorithm is needed for points in the image belonging to the interior of the assigned domain. On the contrary, points *on* the domain boundary need to know values of points assigned to nearby processors. With this purpose the assigned domain is extended and a communication step before each iteration on a given layer is used, as described in [87,58,63]. The communication overhead is a "surface effect" proportional to the linear dimension of the domain.

Considering now implementations on a SIMD parallel computer with a large number of processors, the maximum amount of parallelism is obtained assigning one processor to each grid point [65,57]. SIMD implementations, given the small grain size of the processors, are subject to some efficiency and portability problems. As an example of these problems, if the implementation is on a fine grain hypercube parallel computer and if the mapping is such that all the communication paths in the pyramid are mapped into communication paths in the hypercube with length bounded by two [65], a fraction of the nodes is never used (one third for two-dimensional problems encountered in vision). Furthermore, if the standard multigrid algorithm is used, when iteration is on a coarse scale all the nodes in the other scales (i.e., the majority of nodes) are idle and the efficiency of computation is in part compromised.

[1]In particular, Hypercube computers support a two-dimensional mesh.

Details about the different applications are given in the following chapters.

4.2 Domain Decomposition

If one defines as one *work unit* the amount of computation required by a complete relaxation and discontinuity detection on the finest grid, execution of the presented algorithms require from 3 to 10 work units [2], depending on the number of relaxations used on each layer (i.e., depending on the precision required on the solution).

Given the regularity of the algorithms and the locality of communication between different computational elements, they can be parallelized in a straightforward manner. The decomposition scheme depends on the *technical constraints* imposed by characteristics of the individual processors and of the hardware connections between them. One essential distinction that has to be done is related to the number of processors available and the "size" of a single processor.

If implementation is done on a SIMD parallel computer with a number of processors comparable to the number of computational units, one strategy assigns one processor to each unit (see [65,70]). In this manner the maximum amount of parallelism is obtained. The drawback of this approach is that if the implementation is on a hypercube parallel computer and if the mapping is such that all the communication paths in the pyramid are mapped into communication paths in the hypercube with length bounded by two [65], a fraction of the nodes is never used (one third for two-dimensional problems encountered in vision). A detailed presentation of mapping techniques available for hypercube multiprocessors with small grain nodes [3] has been given in [65]. These techniques are based on the assignment of a one-dimensional array of pixels to processors ordered according to the binary reflected Gray code. The maximum Hamming distance between processors assigned to grid points that need to communicate data is in this case limited by two.

Furthermore, if the standard multigrid algorithm is used, when iteration is on a coarse scale all the nodes in the other scales (i.e., the majority of nodes) are idle and

[2]Using a SUN 386i workstation and C language, this corresponds to approximately one minute, if memory is large enough to contain the entire pyramidal structure.

[3]The Connection Machine is an example.

the efficiency of computation is in part compromised. To ameliorate this problem, intrinsically parallel multiscale algorithms must be considered [66].

Fortunately, if a MIMD computer with powerful processors, sufficient distributed memory, and two-dimensional internode connections (clearly the hypercube contains a two-dimensional mesh) is available the above problems do not exist. The individual processors are powerful and capable of containing data corresponding to the large group of pixels assigned to them [4]. Assuming that a two-dimensional grid can be enbedded in the parallel architecture [5], a two-dimensional *domain decomposition* assigns to every processor a rectangular patch of the image with its "slice" of pyramidal structure (containing elements at all scales corresponding to the assigned patch).

The two mapping strategies are illustrated in figure 4.1.

All nodes are working all the time, switching between different levels of th pyramid (as required by the multiscale algorithm) as illustrated in figure 4.2.

No modification to the sequential algorithm is needed for points in the image belonging to the interior of the assigned domain. On the contrary, points *on* the domain need to know values of points assigned to a nearby processor. With this purpose the assigned domain is extended to contain points assigned to nearby processors and a communication step before each iteration on a given layer is responsible for updating this strip so that it contains the correct (most recent) values. Every processor operates at *all levels* of the pyramid, alternating computation and communication steps to exchange the data on the borders of the assigned domain, as illustrated in figure 4.3. Only two exchanges of data in the two dimensions are necessary.

4.3 Communication Overhead and Complexity

Multigrid algorithms are optimal in the sense that they can compute a solution in time proportional to the number of unknowns. Let's suppose that complexity for the

[4]This is indeed the case for the Definicom board with Transputers used during the development phase. Each Transputer contains up to 1 Megabyte of memory and produces approximately 10 Mips.

[5]For a hypercube multiprocessor the cross product of two one-dimensional Gray codes can be used.

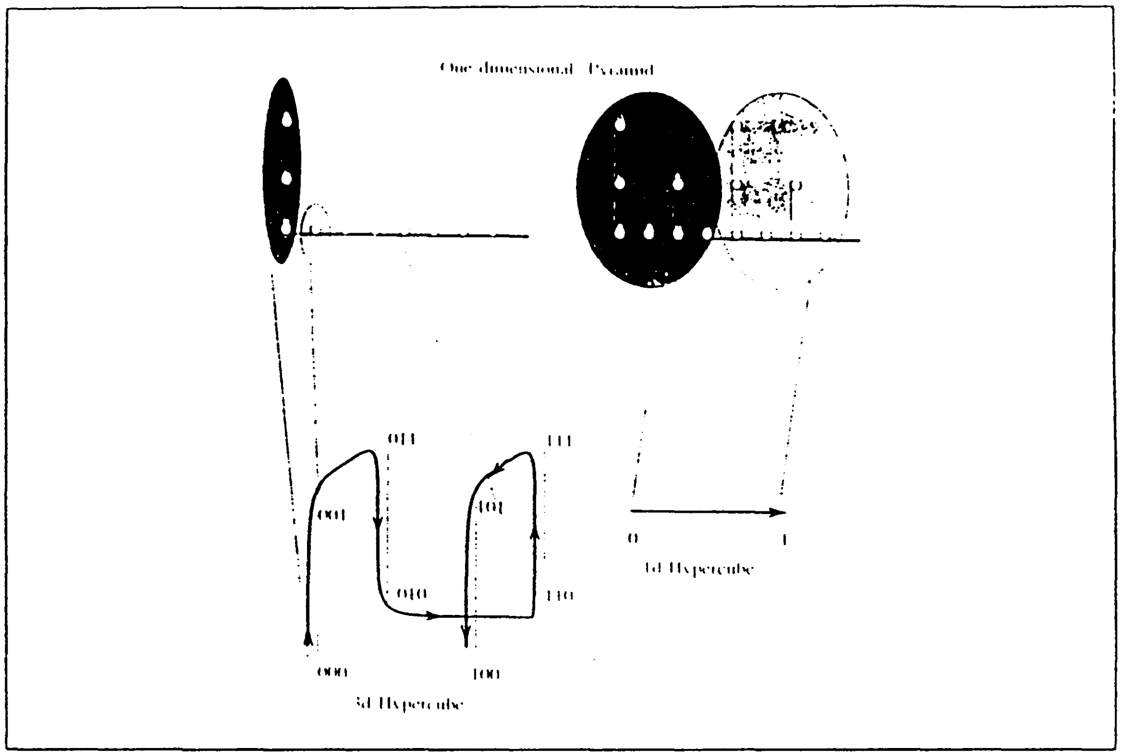


Figure 4.1: Grid points of a pyramidal structure can be mapped to Hypercubes in different ways, depending on technology constraints. Left: mapping using Gray code. Right: mapping using domain decomposition.

standard algorithm is (asymptotically) $Time = \gamma_{comp}\ n$, where n is the number of pixels and γ_{comp} depends on the number of relaxations used in the algorithm.

Since all processors are active most of the time and since the communication overhead is a "surface effect" proportional to $1/\sqrt{n}$, where n is the number of pixels assigned to a given processor, the parallel implementation brings a speed-up that is approximately linear in the number of available processors. Taking both computation and communication into account, complexity for the suggested parallel version is

$$Time = \gamma_{comp}\ \frac{n}{D} + \gamma_{comm}\sqrt{\frac{n}{D}} \tag{4.1}$$

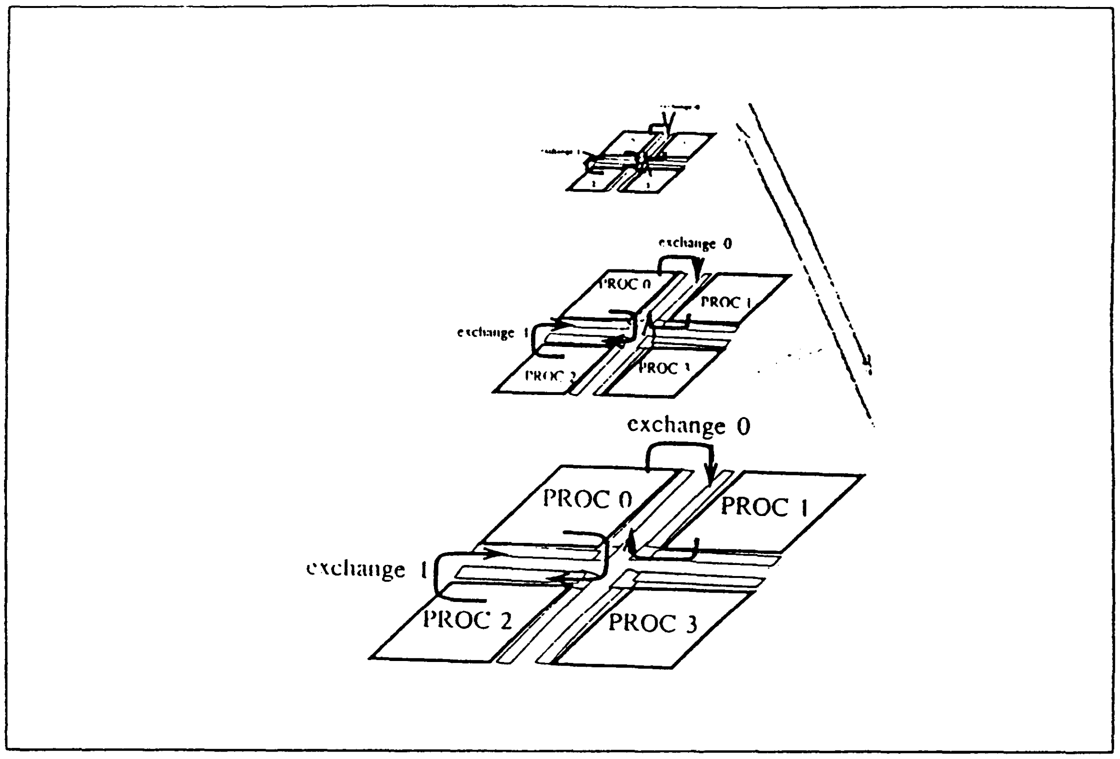


Figure 4.2: Domain decomposition for multigrid computation. Processor communication is on a two-dimensional grid, each processor operates at all levels of the pyramid.

where D is the number of domains (equal to the number of processors). The proportionality factor γ_{comm} depends on the communication speed, on the number of iterations, and on the height of the pyramidal structure.

Preliminary timing has been done using a board with four processors [6] obtaining times of 600-900ms for 65x65 images on all the considered problems. Each node spends approximately 20% of its time in internode communication. In addition some time is required to load the data and read results. Results are illustrated graphically in figure 4.4.

Given the approximately linear speed-up, a configuration with 8x8 nodes should be able to "solve" a 256x256 image in less than one second (excluding input-output

[6] Definicom board with Transputers, software from Parasoft.

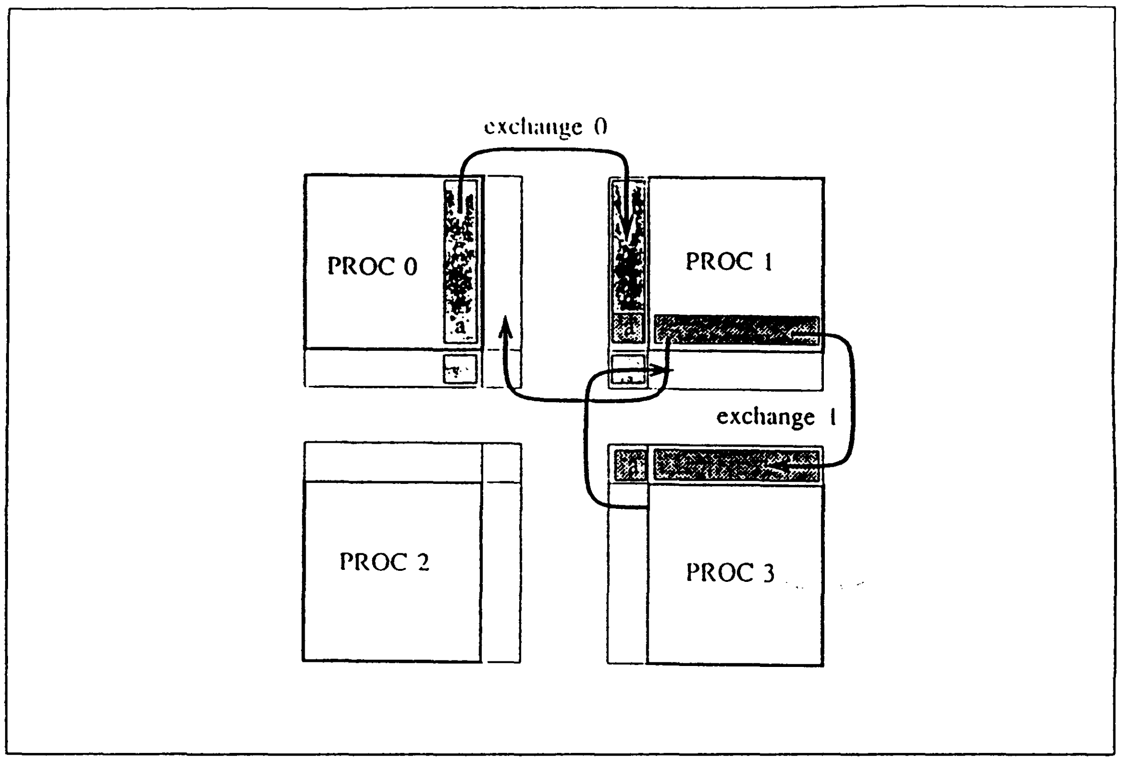


Figure 4.3: Communication strategy for two-dimensional domain decomposition. Data of the assigned domain are bordered by data received from nearby processors. Two exchanges are sufficient.

time).

4.4 Results for Shape from Shading

In this section I present the results of the parallel implementation of the shape from shading algorithm proposed in [71]. They proposed an iterative scheme for solving the shape form shading problem. A preliminary phase recovers information about orientation of the planes tangent to the surface at each point by minimizing a functional containing the image irradiance equation and an *integrability constraint*, as follows:

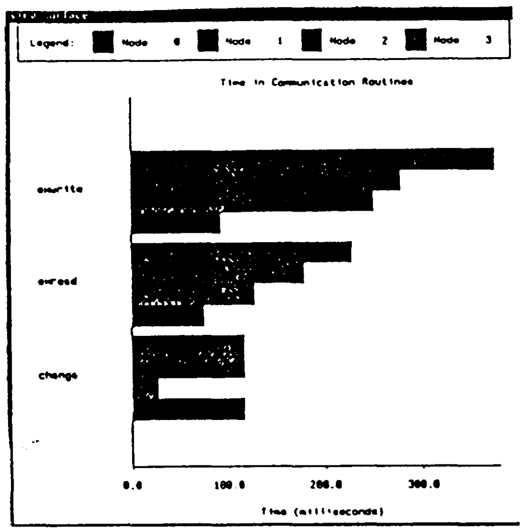


Figure 4.4: Timing results. Above: time spent exchanging data (change) and communicating with the host (read-write).

$$E(p,q) = \int_{Image} (I(x,y) - R(p,q))^2 + \lambda(p_y - q_x)^2 dxdy \tag{4.2}$$

$$\text{where } p = \partial z/\partial x$$
$$q = \partial z/\partial y$$
$$I = \text{measured intensity}$$
$$R = \text{theoretical reflectance function}$$

After the tangent planes are available, the surface z is reconstructed minimizing the following functional:

$$E(z) = \int_{Image} (z_x - p)^2 + (z_y - q)^2 dxdy \tag{4.3}$$

Euler-Lagrange differential equations and discretization are left as an exercise to the reader.

figure 4.5 shows the reconstruction of the shape of a hemispherical surface starting from a ray-traced image [7]. Above is the result of standard relaxation after 100 sweeps, below the "minimal multigrid" result [8] whose total solution time is equivalent to approximately four iterations on the finest grid.

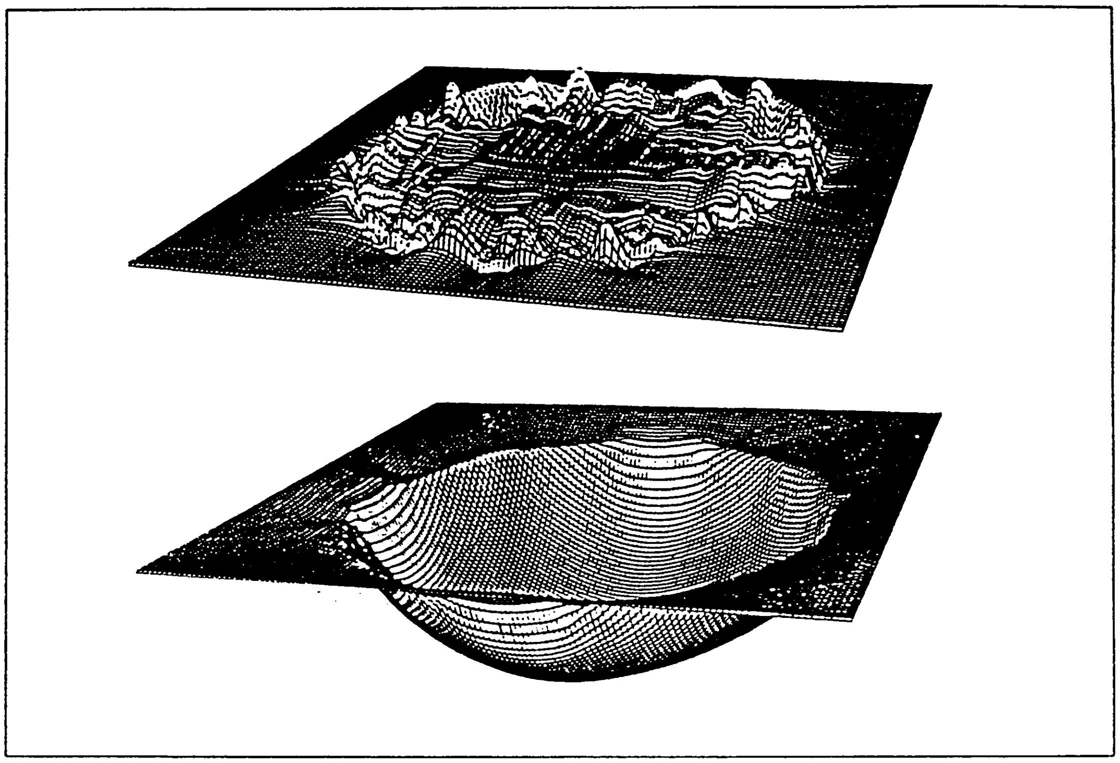


Figure 4.5: Reconstruction of shape from shading : standard relaxation versus multigrid.

This case is particularly hard for a standard relaxation approach. The image can be interpreted "legally" in two possible ways: *either as a concave or a convex hemisphere.* Starting from random initial values, after some relaxations some image patches will

[7]A simple Lambertian reflection model is used.

[8]V cycles with one relaxation on each level

typically "vote" for one or the other interpretation and try to extend the local interpretation to a global one. This not only takes time (given the local nature of the updating rule) but encounters an endless struggle in the regions that mark the border between different interpretations. The multigrid approach solves this "democratic impasse" on the coarsest grids (much faster because now information spreads over large distances) and propagates this decision to the finer grids, which will now concentrate their efforts on refining the initial approximation.

Another example is show in figure 4.6 , where the algorithm tried to reconstruct the three-dimensional structure of the Mona Lisa face painted by Leonardo [9].

Discontinuities were not considered for the two previous tests.

4.5 Results for Surface Reconstruction from Depth Constraints

For the surface reconstruction problem (with membrane energy term) the energy functional is

$$E(z(x,y)) = \int_{Image} (z(x,y) - d(x,y))^2 + \lambda(z_x^2 + z_y^2)dxdy \tag{4.4}$$

A physical analogy is that of fitting the data $d(x,y)$ with a membrane pulled by springs connected to them. A given z value is updated as follows:

$$z(x,y) \leftarrow \frac{z_{sum} + \beta \times h^2 \times d(x,y)}{n_{sum} + \beta \times h^2} \tag{4.5}$$

where $h \equiv$ grid step.

$$z_{sum} = \sum_{dx=\pm h; dy=\pm h} \overline{DN}(x+dx, y+dy) \times z(x+dx, y+dy);$$

[9]Anticipating the reader's unhappiness with her aesthetic appearance, let's remember that the Lambertian reflectance model is clearly a very naive approximation of the artistic shading used by Leonardo

$$n_{sum} = \sum_{dx=\pm h; dy=\pm h} \overline{DN}(x+dx, y+dy);$$

The effect of active discontinuities (DN=1) is clearly that of confining the smoothing action inside the detected borders.

Detailed performance tests have been made using noisy data for z values corresponding to "Randomville" structures. These are obtained by generating random coordinates, heights, slants and tilts for quadrangular blocks and placing them in the image plane. The data are then corrupted by noise and loaded as constraints in the algorithm.

For 129 x 129 "images" and noise values corresponding to 25% of the highest structure, a faithful reconstruction of the surface (within a few percent of the original one) is normally obtained after one single multiscale sweep (with V cycles) on four layers [10].

The total computational time corresponds approximately to the time required by 3 relaxations on the finest grid. Because of the optimality of multiscale methods, time increases linearly with the number of image pixels.

User interface examples and results from some tests are shown in chapter 2. Figure 2.7 shows the simulation environment on the SUN workstation. The reconstruction of a typical "Randomville" image has been presented in chapter 2 (see figures 2.8 and 2.9).

A simplified version of the used parallel program for MIMD machines, using the Express communication routines (a commercial version of the communication environment developed within the Caltech Concurrent Computation Program) is listed in chapter A in the appendix.

4.6 Summary and Discussion

The presented multiple scale algorithms can be efficiently executed on a parallel computer with medium grain size and a two-dimensional domain decomposition is suggested as a simple but effective approach. Given the computational load per pixel of the algorithms (approximately 100 floating point operations per pixel), the communication

[10] In other words, parameters na,nb,nc in mg() are equal to one.

time between neighboring processors is not a critical parameter[11].

The loading and unloading time (i.e., the time required to load the image data into the different nodes and to get the results back to the host processor) has been the limiting factor in the "close-to-real-time" implementation[12].

Richer connectivity (for example, use of an additional channel for direct transmission of the image data and results to and from each processing node) or faster channels have to be used if 30 images per second (with 512x512 pixels) have to be transferred to the nodes for processing.

[11]A bandwidth of 1Mbyte/sec is enough for efficiency greater than 90% using processors of 1Mflop with 1Mbyte of memory.

[12]For example, the bandwidth for image loading obtained with Tranputer boards and Parasoft software is less than 100Kbytes/sec.

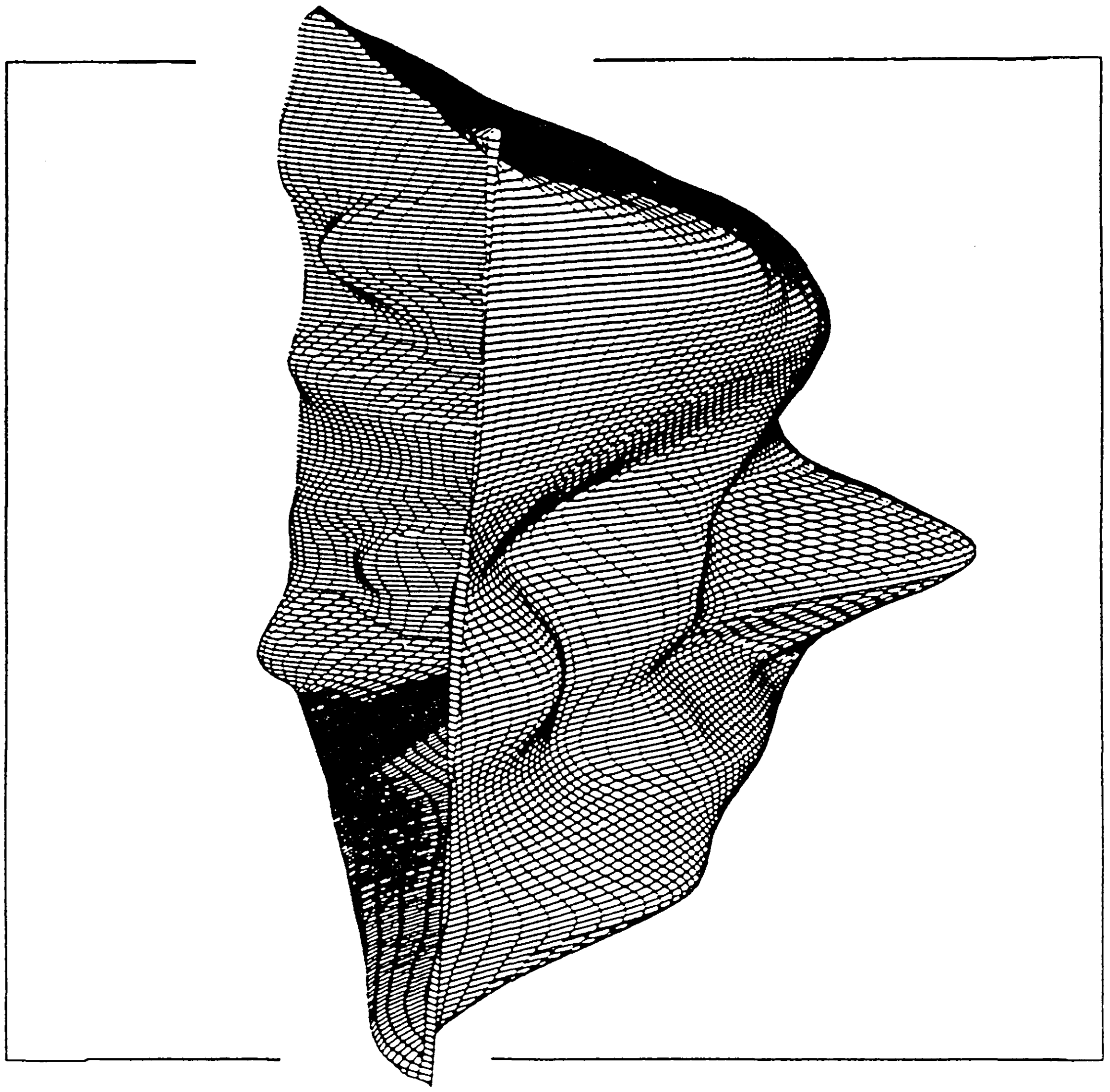


Figure 4.6: Mona Lisa in three dimensions.

Part IV

Teaching Multilayer Perceptrons with Optimization Methods

Chapter 5

Fast Neural Net Teaching

5.1 Introduction: Teaching Neural Networks with the Memoryless Quasi-Newton Method

Multilayer feedforward "neural" networks have been shown to be a useful tool in diverse areas, such as pattern classification, multivariable functional approximation, and forecasting over time [82,83,89,93,95,91,85]. For example, in the character recognition field, many applications have already been developed using neural network approaches (see [92] and the contained references).

In the "back-propagation" (BP) learning procedure a network with a fixed structure is programmed using gradient descent in the space of the weights, where the *energy function* to be minimized is defined as the sum of squared errors [95].

A common difficulty encountered in using back-propagation is that the number of iterations required for convergence tends to increase rapidly with the size of the problem. Significant problems require the use of supercomputers, while other learning tasks are beyond current computational power.

Now, it is well known from the optimization literature that pure gradient descent methods can be, and usually are, very inefficient, as will be explained in the following section. In addition, there are *no general prescriptions* for selecting the parameters in the learning algorithm (like the learning and momentum rate in BP). It is usually left to the user to find a good or optimal combination of these parameters that leads to avoidance of local minima and fast convergence times. This is surely interesting from the point of view of theoretical research ([100] is an example), but leads to a waste of time and computational resources during this *meta-optimization* phase (optimization of the behavior of the optimization method).

The focus of this work has been on transferring some *meta-optimization* techniques, usually left to the user, to the learning algorithm itself. Since this involves measuring optimization performance and correcting some parameters while the optimization algorithm is running, some *global* information is required (typically in the form of scalar products of quantities distributed over the network).

In all cases the "standard" back-propagation algorithm is used to find the values of the energy and the negative gradient for a given configuration. The differences are in the definition of the *search direction* and/or in the selection of a *step size* along the selected direction.

In the first method proposed, the search direction remains equal to the negative gradient but the (scalar) step size is adapted during the computation. This strategy has been suggested independently in [101] and is here summarized for convenience before using it in the test problems. In the second one both the search direction and the step size are changed in a way that is suggested by standard techniques used in optimization. In both cases the network is updated only after the entire set of patterns to be learned has been presented to it.

The description of the two proposed methods (see also [80]) is preceded by a brief discussion about the limits of back-propagation and about some heuristics that have been proposed for accelerating its convergence.

5.2 Limits of Back-propagation

In a given iteration n of back-propagation, the *search direction* $\mathbf{d}_n$ is given by the negative gradient of the energy, while the step along this direction is taken to be proportional to $\mathbf{d}_n$ with a fixed constant ϵ (*learning rate*), as follows:

$$\mathbf{d}_n = -\nabla E(\mathbf{w}_n) \tag{5.1}$$

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \epsilon\ \mathbf{d}_n \tag{5.2}$$

The learning rate is usually chosen by the user to be "as large as possible without leading to oscillations" ([95]).

The inefficiency of gradient descent methods is well known in the optimization literature. For example, if the steepest descent method is applied to a quadratic function $F(\mathrm{x}) = \mathrm{c}^T\mathrm{x} + \frac{1}{2}\ \mathrm{x}^T G\mathrm{x}$ (G symmetric and positive definite) using an *exact line search* to determine the step length, it can be shown that

$$F(x_{n+1}) - F(x^\star) \approx \left(\frac{\lambda_{max} - \lambda_{min}}{\lambda_{max} + \lambda_{min}}\right)^2 (F(x_n) - F(x^\star)) \tag{5.3}$$

where $x^\star$ is the optimal point and λ_{max} and λ_{min} are the largest and smallest eigenvalues of G. This means that the asymptotic error reduction constant can be *arbitrarily close* *to unity* ([88]). A case in which this happens is when "the search space contains long ravines that are characterized by sharp curvature across the ravine and a gently sloping floor"([95]).

The situation can be ameliorated in part by modifying the search direction with the introduction of a *momentum* term, as follows:

$$\mathbf{d}_n = -\nabla E(\mathbf{w}_n) + \left(\frac{\alpha}{\epsilon}\right) \triangle \mathbf{w}_{n-1} \tag{5.4}$$

Assuming that the parameter α is chosen appropriately, convergence is in this case faster, although the obtained performance is far from optimal.

Recently an overview of heuristics employed to accelerate back-propagation has been presented in [94], where it is suggested that each weight should be given a different learning rate, changing over time during the computation.

5.3 The "Bold Driver" (BD) Method

This method is an example of an heuristic solution to the problem of selecting an appropriate learning rate in BP [1]. It requires only a limited change to standard back-propagation, based on the following intuitive argument.

In general, the number of steps to convergence for BP is a decreasing function of the learning rate up to a given point, where oscillations in the weights are introduced, the energy function does not decrease steadily and good local minima are missed. Performance degradation in this case is usually rapid and unpredictable. The proposed solution is to start with a given learning rate (any value greater than zero will work) and to monitor the value of the energy function $E(\mathbf{w}_n)$ after each change in the weights. If E decreases, the learning rate is then increased by a factor ρ. Vice versa if E increases, this is taken as an indication that the step made was too long, the learning rate is decreased by a factor σ, the last change is canceled and a new trial is done. The process of reduction is repeated until a step that decreases the energy value is found (this will be found if the learning rate is allowed to tend to zero, given that the search direction is that of the negative gradient).

[1] For clarity of comparison the momentum rate is set to zero.

Heuristically, ρ has to be close to unity (say $\rho \approx 1.1$) in order to avoid frequent "accidents," because the computation done in the last back-propagation step is wasted in these cases. Regarding the parameter σ a choice of $\sigma \approx 0.5$ can be justified with the reason that if the local "ravine" in the search space is symmetric on both sides this will bring the configuration of the weights close to the bottom of the valley.

The exponential increase in the learning rate ($\epsilon = \epsilon_0 \rho^n$) is preferred to a linear one because it *will* typically cause an "accident" after a limited number of steps, assuming that the proper learning rate for a given terrain increases less rapidly. Now, such accidents are productive because after them the learning rate is reset to a value appropriate to the local energy surface configuration.

An example for the size of the learning rate as a function of the iteration number is given in figure 5.1.

The performance of this apparently "quick and dirty" method (considering both the number of iterations required and the quality of the local minimum found) is close and usually better than that obtainable by optimizing a learning rate that is to remain fixed during the procedure. Besides the momentum term, there are now no learning parameters to be tuned by the user on each problem. The given values for ρ and σ can be fixed once and for all and, moreover, performance does not depend critically on their choice, provided that the heuristic guidelines given above are respected. Given the above reasons, we decided to use this method in order to obtain a meaningful comparison with the method suggested in the following section.

5.4 The BFGS Memoryless Quasi-Newton Method

We will use the term "conjugate gradient method with inexact linear searches" as a synonym for "one-step BFGS memoryless quasi-Newton method", (BFGS for short) leaving some technical details and a brief explanation in appendix C.

Shanno [98] reviews several variations of the conjugate gradient method and suggests one method using inexact linear searches and a modified definition of the search direction that "substantially outperforms known conjugate gradient methods on a wide class of problems".

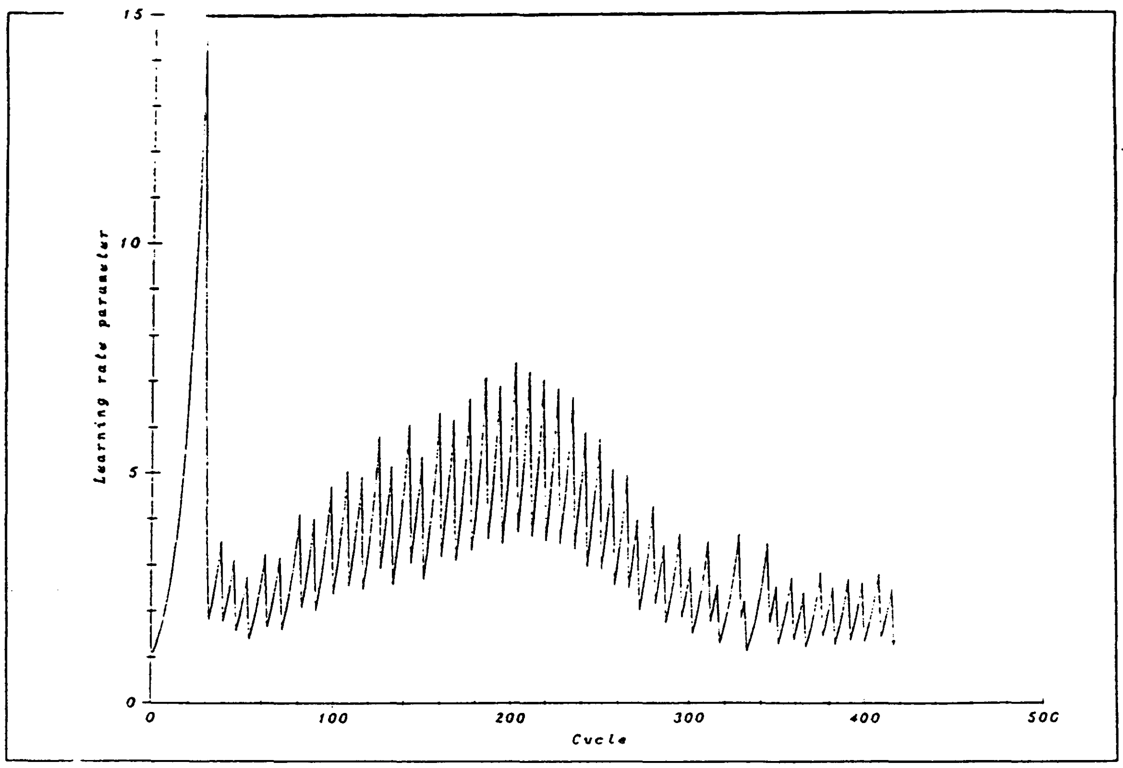


Figure 5.1: Example of learning rate behavior as a function of the iteration number for the "bold driver" network. "Accidents" during the search cause a rapid decrease in the learning rate, followed by exponential increase during normal operation.

Let's define the following vectors: $\mathbf{g}_n = \nabla E(\mathbf{w}_n)$, $\mathbf{p}_n = \mathbf{w}_n - \mathbf{w}_{n-1}$ and $\mathbf{y}_n = \mathbf{g}_n - \mathbf{g}_{n-1}$. In the suggested strategy, successive approximations to the minimizer $\mathbf{w}^\star$ of a function $E(\mathbf{w})$ are generated iteratively in the following way:

$$\mathbf{d}_0 = -\mathbf{g}_0 \tag{5.5}$$

$$\mathbf{d}_n = -\mathbf{g}_n + A_n \mathbf{p}_n + B_n \mathbf{y}_n \tag{5.6}$$

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \epsilon_n \ \mathbf{d}_n \tag{5.7}$$

$$\text{where} \quad \epsilon_n = \min_{\epsilon} E(\mathbf{w}_n + \epsilon \ \mathbf{d}_n) \tag{5.8}$$

The coefficients A_n and B_n are combinations of scalar products of the vectors defined at the beginning of this section, as follows:

$$A_n = -\left(1 + \frac{\mathbf{y}_n \cdot \mathbf{y}_n}{\mathbf{p}_n \cdot \mathbf{y}_n}\right) \frac{\mathbf{p}_n \cdot \mathbf{g}_n}{\mathbf{p}_n \cdot \mathbf{y}_n} + \frac{\mathbf{y}_n \cdot \mathbf{g}_n}{\mathbf{p}_n \cdot \mathbf{y}_n} \tag{5.9}$$

$$B_n = \frac{\mathbf{p}_n \cdot \mathbf{g}_n}{\mathbf{p}_n \cdot \mathbf{y}_n} \tag{5.10}$$

Every N steps (N being the number of weights in the network) the search is restarted in the direction of the negative gradient. It is worth noting that if the function $E(\mathbf{w})$ is quadratic in an N-dimensional space ($E(\mathbf{w}) = \mathbf{c}^T\mathbf{w} + \frac{1}{2}\ \mathbf{w}^T G \mathbf{w}$, where G is a positive definite symmetric matrix), this method is guaranteed to converge to the minimum in at most $N+1$ function and gradient evaluations. Correction of the search direction based on previous steps is in part reminiscent of the use of a *momentum* term introduced in [95], with the added feature that a definite prescription is given for the choice of the various factors.

A critical issue to consider when applying conjugate gradient methods to back-propagation is that the computation required during the exact one-dimensional optimization implied by eqn. 5.8 is expensive because every function and gradient evaluation involves a complete cycle of pattern presentation and error back-propagation; therefore efficient approximate one-dimensional optimization have to be used. The one-dimensional minimization used in this work is based on quadratic interpolation and tuned to back-propagation where in a single step both the energy value and the negative gradient can be efficiently obtained. Details on this step are contained in appendix D.

The above method, while requiring only minor changes to standard back-propagation, is capable of reducing the number of steps to convergence by orders of magnitude on some problems with a large number of weights. Three example problems and the obtained results are described in the two following sections. A similar optimization approach, using Polak-Ribiere optimization, is presented in [90]. They also obtain a sizable speed-up with respect to standard back-propagation, although they do not optimize its parameters[2]. Now, a major difficulty with the Polak-Ribiere algorithm is that the search directions obtained are not necessarily *descent* directions, and numerical instability can result. This happens because the matrix used to obtain the search direction from the gradient is not symmetric and hence not positive definite. Another problem caused by this lack of symmetry is that the quasi-Newton equation is not satisfied (see [98]). The cited difficulties are not present in the BFGS method.

5.5 Test: the Parity Function

Recently Tesauro and Janssens [100] measured optimal averaging training times and optimal parameter settings for standard back-propagation with momentum term. Their training set contains binary strings as input and their parity as target output.

In order to benchmark the two new proposed methods, the same network is used (n input units, $2n$ hidden units, one output) and weights are initialized randomly using the same scale parameter r_{opt} and momentum rate parameter α_{opt} as those given in [100].

The results of 100 simulations for each problem show first that back-propagation with adaptive learning rate (BD) produces results that are close to those obtained by optimizing parameters in back-propagation with fixed learning rate, second that the memoryless quasi-Newton method brings a sizable speedup on both previous methods. Visual and numerical displays of results are in figure 5.2 and in table 5.1. Results in [100] are given both as number of iterations and as number of cycles. This last number is more significant for the comparison, since in the other cases weights are corrected

[2]We agree with them that "finding parameters (for BP) that result in fast progress and stable behavior is a black art, at best".

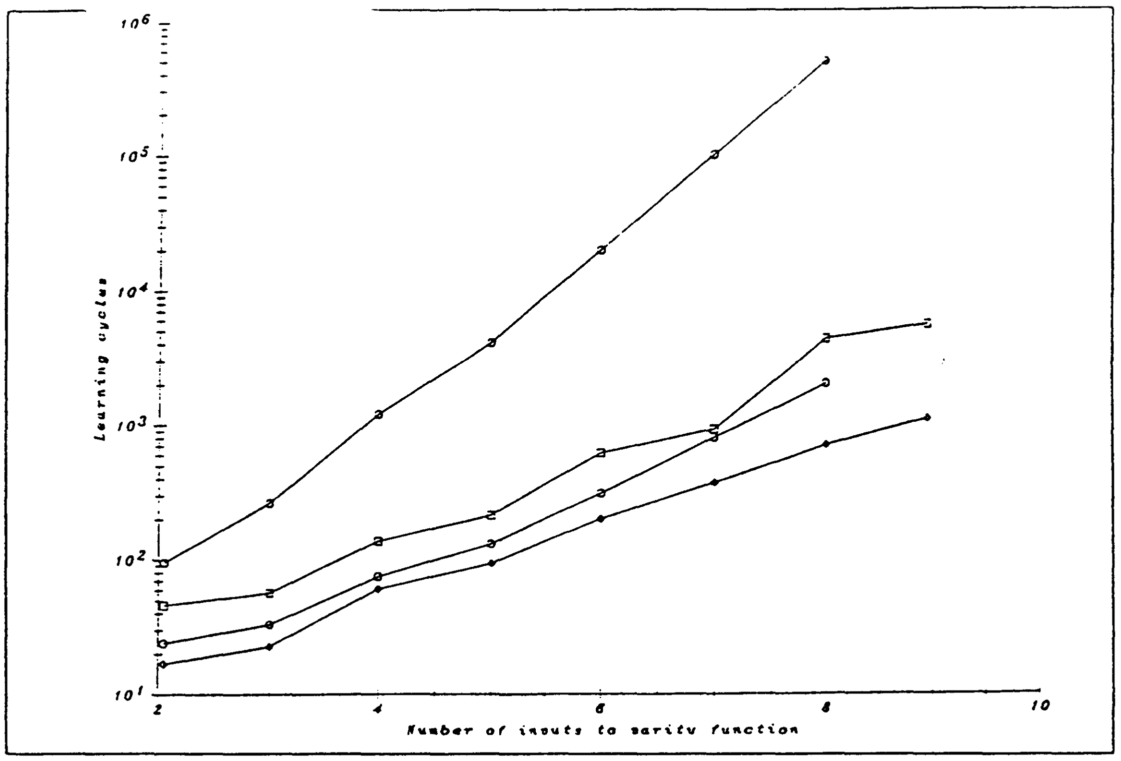


Figure 5.2: Performance comparison: standard back-propagation with optimal parameters from Tesauro-Janssens (Upper curve (o): number of iterations. Lower curve (o): iterations divided by training patterns), back-propagation with adaptive learning rate (□) and memoryless quasi-Newton method (◇).

after one cycle of pattern presentations. Since the number of local minima is small in this case, only data regarding correct convergence are shown.

5.6 Test: the Dichotomy Problem

This problem consists in classifying a set of randomly generated patterns in two classes. It has been demonstrated in [81] that an arbitrary dichotomy for any set of N points in general position in d dimensions *can* be implemented with a network with one hidden layer containing $\lceil N/d \rceil$ neurons. This is in fact the smallest such net, as dichotomies that cannot be implemented by any net with fewer units can be constructed. In this

patterns	BP	BD	BFGS	speedup (BD/BFGS)
	cycles (s.d.)	cycles (s.d.)	cycles (s.d.)	
2	24 (N/A)	46 (11)	16 (8)	2.8
3	33 (N/A)	57 (17)	22 (10)	2.6
4	75 (N/A)	137 (57)	68 (58)	2.0
5	130 (N/A)	213 (115)	93 (69)	2.3
6	310 (N/A)	616 (835)	199 (127)	3.0
7	800 (N/A)	875 (359)	371 (300)	2.3
8	2000 (N/A)	4310 (3088)	700 (368)	6.1

Table 5.1: Results for parity problem. Timing comparison between standard back-propagation with optimal parameters (from Tesauro-Janssens) and the two methods suggested in the article.

test the pattern coordinates are random values belonging to the [0-1] interval.

A dichotomy problem is defined by the number of patterns generated. The dimension of the space and the number of inputs is two, the number of middle-layer units is $\lceil N/2 \rceil$ by the above criterion and one output unit is responsible for the classification.

Simulation runs have been made starting from small random weights (to break symmetry), with maximum size r equal to 0.1. Correct performance is defined as coming within a margin of 0.1 of the correct answer. Results of the "bold driver" and the memoryless quasi-Newton methods are compared in figure 5.3.

The capability of the network does not avoid the problem of local minima. These points are detected in an approximate way by terminating the search when the modulo of the gradient or of the weight change becomes less than 10^{-6}. In fact the results show that their number is increasing as a function of the dimension of the search space (i.e., the number of weights in the network). Results of different tests (the random number generator seed is changed) are given in table 5.2.

5.7 Summary and Discussion

The main object of this work has been that of comparing standard back-propagation with the memoryless quasi-Newton method (BFGS) for optimization. This method has

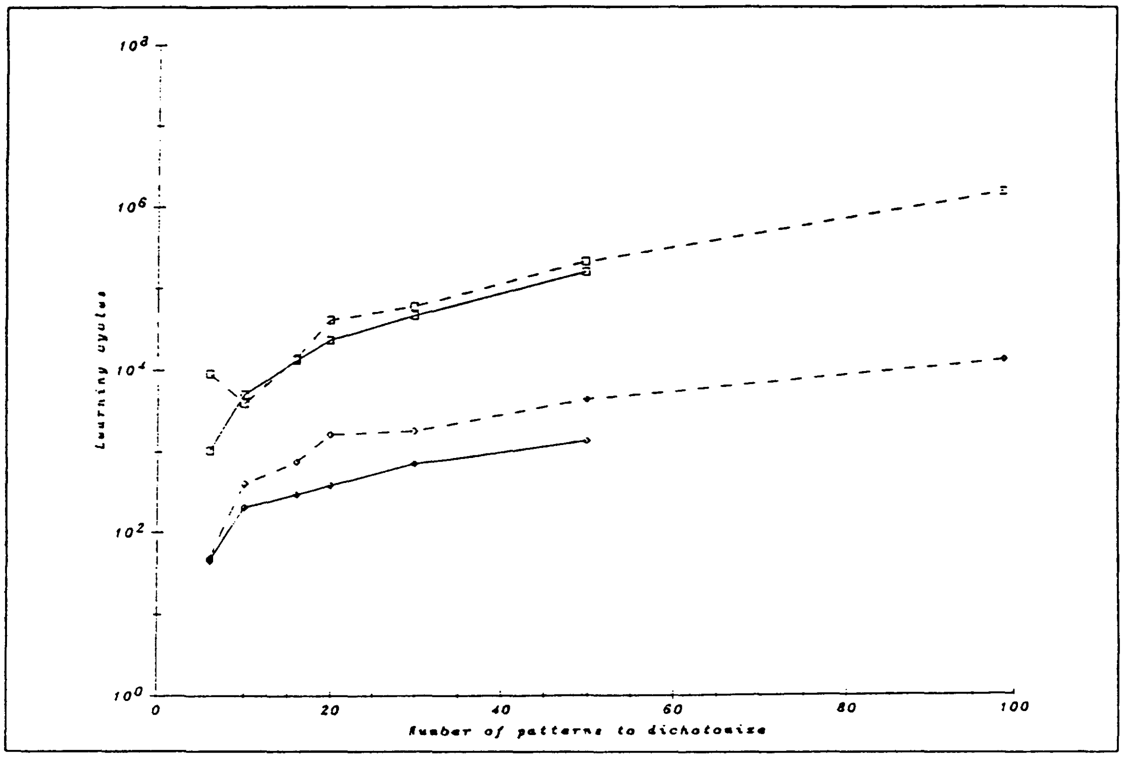


Figure 5.3: Performance comparison: back-propagation with adaptive learning rate (squares) versus memoryless quasi-Newton method (diamonds). Continuous lines show the average number of cycles for convergence to correct solution, dashed lines for convergence to local minimum.

patterns	BD	BFGS	speedup (BD/BFGS)
6	cases: cycles (s.d.)	cases: cycles (s.d.)	
correct loc.min.	124: 1040 (1458) 4: 9032 (10403)	115: 44 (56) 13: 49 (74)	23.6
10			
correct loc.min.	104: 5044 (6870) 24: 3923 (4914)	90: 204 (368) 38: 404 (1005)	24.7
16			
correct loc.min.	106: 13245 (10572) 22: 14116 (11960)	94: 295 (513) 34: 755 (1605)	44.8
20			
correct loc.min.	111: 23293 (16792) 17: 41000 (28583)	87: 380 (433) 41: 1632 (3021)	61.3
30			
correct loc.min.	44: 46265 (20761) 20: 59843 (16555)	36: 710 (418) 28: 1800 (1300)	65.1
50			
correct loc.min.	4: 157296 (36837) 4: 211292 (59424)	13: 1347 (600) 51: 4307 (2159)	116.7
100			
correct loc.min.	0: 8: 1435950 (560974)	0: 64: 12645 (4161)	N/A

Table 5.2: Results for dichotomy problem. Back-propagation with adaptive learning rate ("bold driver" method, or BD) vs. memoryless quasi-Newton method (BFGS). Number of test cases and average number of cycles (and standard deviation) for convergence to correct solution or local minimum are shown. Speedup is given only for convergence to correct solution.

been selected because its memory and computation requirements during each step grow only linearly with the number of weights. Furthermore numerical stability is assured and the number of steps for convergence has been shown to be small in many test problems. The strategy for the one-dimensional search is based on quadratic interpolation and requires a limited number of (expensive) function evaluations.

Since back-propagation requires a choice of its learning rate, a fair comparison brought us to consider an adaptive version of back-propagation (BD), where the learning rate is adapted to the structure of the energy surface. From some tests, BD produces results close to those obtainable by optimizing BP (with parameters that are to remain fixed during the learning phase) and therefore can be considered a good candidate for a fair comparison against BFGS. In addition, the user-driven optimization of parameters is avoided.

For the considered test problems, the memoryless quasi-Newton method converges in a time that is from one to two orders of magnitude smaller than that required by BD (or by a session of BP with optimized parameters). The BFGS method therefore should be considered as an effective modification to standard BP, especially for problems that require an expensive training phase.

One possible objection to using standard optimization techniques for BP is that they require some sort of *global* computation. Now *locality* is a concept that depends on the mapping between a given algorithm and the processors (VLSI hardware, biological neurons, ...) responsible for the computation. In this sense back-propagation is *local* if different processors are assigned to the different weights and "neurons" and if the chain rule for partial derivatives is used in calculating the gradient ,"back-propagating" the errors through the network. A concept related but different from locality is that of *parallelism*, where a given computation can be done by more computational units working concurrently on different partial tasks (with a speedup in the time required to complete it). Despite the fact that networks performing local computation are usually easier to implement in parallel architectures, it is nonetheless true that parallelism of computation can be obtained also in certain cases where a global information exchange is required (see [87] for many examples in both areas).

Both proposed methods have indeed been implemented on parallel hardware [3] (assigning different patterns to be learned to different processors) and require only one global exchange of information during each learning cycle, in order to choose the next learning rate and search direction. Efficiency of implementation is close to 100% for problems with a large number of patterns to be learned (this is one case in which computation time tends to prevail over communication time).

[3] A hypercube built with 16 Transputers.

Part V

Conclusion and Appendices

Chapter 6

Real-Time Vision Machines?

6.1 Brief Conclusion

The design of real-time computer vision systems is facilitated if fast special purpose visual modules are integrated using parameter optimization (i.e., learning techniques).

Nonetheless, since the learning task is difficult and requires a computing time that increases rapidly with the size of the problem, it is imperative to use all the available algorithmic tools before applying learning only to select a restricted number of essential and difficult to determine or unknown parameters.

In this thesis, efficient techniques based on multiple scale processing with adaptive grids and discontinuities have been implemented on parallel computers, showing that real-time performance for low and intermediate level computer vision modules is within the reach of available digital computing technology. In the near future the same operations may be implemented at a still lower cost using analog VLSI vision chips.

In addition, learning techniques based on optimization have been shown to converge in some minutes of CPU time using standard microprocessors (for problems with a few hundreds of parameters to be determined).

The next challenge (left as an excercise for the reader...) is that of integrating the available visual modules in an optimal way in order to open the road to a widespread use of computer vision for the different applications.

Appendix A

Listing of Hypercube Program

A.1 Header with Basic Data Structures and Macros

A.2 Host Program and Graphics

A.3 Node Program

Appendix B

Three Point Approximation of Derivatives

We shall derive the third-order expressions for the three-point approximations of the temporal and spatial brightness derivatives. Let $f(x - vt)$ be a one-dimensional translating brightness profile. Taylor's expansion provide the three-point formula for the first order brightness derivatives:

$$\frac{f(y+h) - f(y-h)}{2h} = f'(y) + \frac{f'''(y)h^2}{6} + O(h^3). \tag{B.1}$$

The approximation of the temporal derivative is given by

$$\tilde{E}_t = \frac{f(x - v(t + \Delta t)) - f(x - v(t - \Delta t))}{2\Delta t} \tag{B.2}$$

which becomes by setting $y = x - vt$,

$$\tilde{E}_t = \frac{f(y - v\Delta t) - f(y + v\Delta t)}{2\Delta t} \tag{B.3}$$

and which is, according to (22), equal to

$$- vf'(y) - \frac{vf'''(y)(v\Delta t)^2}{6} + O(v(v\Delta t)^3). \tag{B.4}$$

Since $f_t = f'(x - vt)(-v) = -vf'(y)$, we arrive at

$$\tilde{E}_t = f_t - \frac{vf'''(y)}{6}(v\Delta t)^2, \tag{B.5}$$

where the higher order terms are neglected. A similar expression holds for the approximation of the spatial derivative

$$\tilde{E}_x = f_x + \frac{f'''(y)}{6}(\Delta x)^2, \tag{B.6}$$

where Δx is the spatial sampling step.

Appendix C

Memoryless Quasi-Newton Methods and Conjugate Gradient Methods

The *Newton's method* for minimization (see the comprehensive description in [88]) is based on a local *quadratic model* for the function F to be minimized, obtained using Taylor expansion:

$$F(\mathbf{x_n} + \mathbf{d}) \approx F_n + \mathbf{g_n}^T\mathbf{d} + \frac{1}{2}\ \mathbf{d}^T H_n \mathbf{d} \tag{C.1}$$

where H_n is the Hessian matrix.

Newton's method defines the step $\mathbf{d}$ to be the minimizer of the Taylor expansion, or the solution of the linear system obtained from eqn. C.1:

$$H_n\mathbf{d} = -\mathbf{g_n} \tag{C.2}$$

If H_n is positive definite, only *one* iteration is required to reach the minimum of the model function.

The key to the success of Newton type methods is in the curvature information provided by the Hessian matrix. Now, *quasi-Newton* methods *build up* curvature information as the iterations of a descent method proceed, without explicitly forming the Hessian matrix. Let's define as K_n the iterative *approximation* to the Hessian matrix.

Since for the Hessian

$$\mathbf{g_n}(\mathbf{x_n} + \mathbf{d_n}) \approx \mathbf{g_n} + H_n\mathbf{d_n} \tag{C.3}$$

or

$$H_n\mathbf{d_n} \approx (\mathbf{g_n}(\mathbf{x_n} + \mathbf{d_n}) - \mathbf{g_n}) = \mathbf{y_n} \tag{C.4}$$

the new Hessian approximation $K_{n+1} = K_n + U_n$ is required to satisfy the following *quasi-Newton condition*:

$$K_{n+1}\mathbf{d_n} \approx \mathbf{y_n} \tag{C.5}$$

Different quasi-Newton methods vary in their prescription for the updating of K_n. The *Broyden-Fletcher-Goldfarb-Shanno (BFGS)* update

$$K_{n+1} = K_n - \frac{1}{\mathbf{d_n}^T K_n \mathbf{d_n}} K_n \mathbf{d_n} \mathbf{d_n}^T K_n + \frac{1}{\mathbf{y_n}^T \mathbf{d_n}} \mathbf{y_n}\mathbf{y_n}^T \tag{C.6}$$

is believed to be the most effective update formula. It has *hereditary symmetry* (K_{n+1} is symmetric if K_n is) and *hereditary positive-definiteness* for "sufficiently accurate" linear searches along the search direction given by

$$\mathbf{d_n} = -K_n^{-1}\mathbf{g_n} \tag{C.7}$$

For large-scale problems the computation used in matrix-vector multiplications ($O(N^2)$; where N is the number of variables of the function to be minimized) and the memory requirements make this method inefficient.

The "one-step" *memoryless* BFGS update is given by the BFGS formula for the *inverse* Hessian, with the previous approximation taken as the identity matrix (eqn. 5.6). Only vectors need to be stored and computation is reduced to $O(N)$, because only scalar products are involved.

If *exact linear searches* are made, memoryless quasi-Newton methods generate *mutually conjugate* directions. Shanno illustrates this relationship in detail ([98]).

Appendix D

One-Dimensional Minimization

Let us write $E(\epsilon)$ for $E(\mathbf{x}_{n-1} + \epsilon\mathbf{d}_n)$ where $\mathbf{d}_n$ has been defined in eqn. (11). First $E(0)$ and $E(\epsilon = 4\epsilon_{n-1})$ are calculated.

If $E(\epsilon = 4\epsilon_{n-1})$ is greater or equal to $E(0)$, the parameter ϵ is divided by four until $E(\epsilon)$ is less than $E(0)$. Since **d** is guaranteed to be a descent direction this point will be found. Then the minimizer ϵ_{min} of the parabola going through the three points is found. The process is then repeated with the three points obtained after substituting ϵ_{min} for one of the three previous points, to reobtain the configuration with the function value at middle point less than that at either end. The process continues until the difference in the last two approximations to the minimum value is less than 10^{-6}.

On the contrary, if $E(\epsilon = 4\epsilon_{n-1})$ is less than $E(0)$, the parameter ϵ is multiplied by four until $E(\epsilon)$ is greater than $E(0) + \epsilon E'(0)$ (to assure existence of a minimum in the quadratic minimization step). If this is found the final ϵ is set either to the quadratic minimizer of the parabola through $E(0)$ and $E(\epsilon)$ with initial derivative $E'(0)$ or to $4\epsilon_{n-1}$, depending on the minimum value of the energy function for these two points. If this is not found after a reasonable number of trials (5 in our case), the final ϵ is set to $4\epsilon_{n-1}$.

The efficiency of the method is due to the fact that only a very limited number of iterations are actually done in the two cases . Furthermore, in the second case the derivative $E'(0)$ is obtained rapidly with the scalar product of $\mathbf{d}_n$ and $\mathbf{g}_n$, which in turn are found together with the value $E(0)$ during the last back-propagation step.

Bibliography

References mostly for Chapter 2

[1] R. Battiti, "Collective Stereopsis on the Hypercube," in *Proc. III Conf. on Hypercube Conc. Comp. and Appl.* **Vol. II**, 1000–1006, (Pasadena, CA, 1988).

[2] R. Battiti, "Surface Reconstruction and Discontinuity Detection: a Fast Hierarchical Approach on a Two-Dimensional Mesh," in *Proc. of the IV Conf. on Hypercube Concurrent Computers and Applications*, (John Gustafson at. al. (eds.), Monterey, 1989).

[3] A. Blake and A. Zisserman, *Visual Reconstruction*, (MIT Press: Cambridge, MA, 1987).

[4] A. Brandt, "Multi-Level Adaptive Solutions to Boundary-Value Problems," *Math. Comput.* **31**, 333–390, (1977).

[5] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, *Solving Problems on Concurrent Processors*, (Prentice Hall, New Jersey, 1988).

[6] P. Frederickson, O. A. McBryan, "Intrinsically Parallel Multiscale Algorithms for Hypercubes," in *Proc. III Conf. on Hypercube Conc. Comp. and Appl.* **Vol. II**, 1726–1734, (Pasadena, CA, 1988).

[7] W. Furmanski and G. Fox, "Integrated Vision Project on the Computer Network," *Caltech C3P report* **623**, (Caltech Concurrent Computation Project, Pasadena, CA 91125, 1988).

[8] S. Geman and D. Geman, "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images," *IEEE Trans. Pattern Analysis Machine Intelligence* **6**, 721–741, (1984).

[9] T. Kanade, *Three-Dimensional Machine Vision*, (Kluver: Boston, MA, 1987).

[10] C. Koch, J. Marroquin, A. Yuille, "Analog Neuronal Networks in Early Vision," *Proc. Natl. Acad. Sci. USA* **83**, 4263–4267, (1986).

[11] D. Marr and T. Poggio, "Cooperative Computation of Stereo Disparity," *Science* **195**, 283–287, (1976).

[12] J. Marroquin, "Surface Reconstruction Preserving Discontinuities," *M.I.T. Artif. Intell. Lab. Memo* **792**, (MIT, Cambridge, MA, 1984).

[13] J. Marroquin, "Optimal Bayesian Estimators for Image Segmentation and Surface Reconstruction," *M.I.T. Artif. Intell. Lab. Memo* **839**, (MIT, Cambridge, MA, 1985).

[14] T. Poggio and C. Koch, "Ill-Posed Problems in Early Vision: from Computational Theory to Analogue Networks," *Proc. R. Soc. Lond. B* **218**, 303–323, (1985).

[15] T. Poggio, V. Torre and C. Koch, "Computational Vision and Regularization Theory," *Nature* **317**, 314–319, (1985).

[16] K. Stüben and U. Trottenberg, "Multigrid Methods: Fundamental Algorithms, Model Problem Analysis and Applications," in *Multigrid Methods Proc.*, 1–176, (Springer-Verlag: Berlin, BRD, 1982).

[17] D. Terzopulos, "Multilevel Computational Processes for Visual Surface Reconstruction," *Comp. Vis., Graph., and Image Proc.* 24, 52–96, (1983).

[18] D. Terzopulos, "Image Analysis Using Multigrid Relaxation Methods," *IEEE Trans. Pattern Analysis Machine Intelligence* 8, 129–139, (1986).

[19] D. Terzopulos, "Regularization of Inverse Visual Problems Involving Discontinuities," *IEEE Trans. Pattern Analysis Machine Intelligence* 8, 413–424, (1986).

References mostly for Chapter 3

[20] R. Battiti "Surface reconstruction and discontinuity detection: A hierarchical approach," *Caltech C3P Report* **676**, (Caltech Concurrent Computation Project, Pasadena, CA 91125, 1988).

[21] A. Brandt "Multi-level adaptive solutions to boundary-value problems," *Math. Comput.* **31**,333–390, (1977).

[22] P.J. Burt "The pyramid as a structure for efficient computation," Rosenfeld A.(ed) *Multiresolution image processing and analysis*, 6–35, (Springer-Verlag 1984).

[23] W. Enkelmann "Investigations of multigrid algorithms for the estimation of optical flow fields in image sequences," *Computer Vision, Graphics and Image Processing* **43**, 150–177, (1988).

[24] E. Gamble and T. Poggio "Integration of intensity edges with stereo and motion," *MIT Artificial Intelligence Lab. Memo No.970,*, (1987).

[25] S. Geman and D. Geman "Stochastic Relaxation,Gibbs Distributions, and the Bayesian Restoration of Images," *IEEE Trans. Pattern Analysis Machine Intelligence* **6**, 721–741, (1984).

[26] F. Girosi, A. Verri and V. Torre "Constraints for the computation of the optical flow," *Proceedings of the IEEE Workshop on Visual Motion*, 116–124, (Irvine, CA, March 1989).

[27] F. Glazer "Multilevel relaxation in low-level computer vision," Rosenfeld A.(ed)*Multiresolution image processing and analysis*, 312–330, (Springer-Verlag, 1984).

[28] E.C. Hildreth "Computations underlying the measurement of visual motion," *Artificial Intelligence* **23**, 309–354, (1984).

[29] B.K.P. Horn *Robot Vision*, (MIT Press, McGraw-Hill, 1986).

[30] B.K.P. Horn and G. Schunck "Determining Optical Flow," *Artificial Intelligence* **17**, 185–203, (1981).

[31] J. Hutchinson, C. Koch, J. Luo and C. Mead "Computing Motion Using Analog and Binary Resistive Networks," *IEEE Computer*, 52–63, (March 1988).

[32] B. Kamgar-Parsi and B. Kamgar-Parsi "Evaluation of Quantization Error in Computer Vision," *IEEE Trans. Pattern Analysis Machine Intelligence* **11(9)**, 929–940, (1989).

[33] C. Koch, J. Marroquin, A. Yuille "Analog neuronal networks in early vision," *Proc. Natl. Acad. Sci. USA* **83**, 4263–4267, (1986).

[34] J. Little, H.H. Bülthoff, and T. Poggio "Parallel optical flow using local voting," *Proceedings of the Int. Conf. on Comp. Vision* (Tarpon Springs, Florida, Dec. 1988).

[35] J. Little and A. Verri "Analysis of differential and matching methods for optical flow," *Proceedings of the IEEE Workshop on Visual Motion*, 173–180, (Irvine, California, March 1989).

[36] D. Marr *Vision*, (Freeman, NewYork, 1982).

[37] H.H. Nagel "Analysis techniques for image sequences," *Proc. 4th Int. Joint Conf. on Pattern Recognition*, (Kyoto, Japan, Nov 1978).

[38] H.H. Nagel and W. Enkelmann "An Investigation of Smoothness Constraints for the Estimation of Displacement Vector Fields from Image Sequences," *IEEE Trans. Pattern Analysis Machine Intelligence* 8(5), 565–593, (1986).

[39] H. Nishihara "Practical real-time imaging stereo matcher," *Opt. Eng.* **23(5)**, 536–545, (1984).

[40] T. Poggio, V. Torre and C. Koch "Computational vision and regularization theory," *Nature* **317**, 314–419, (1985).

[41] Stüben K. and Trottenberg Multigrid Methods: Fundamental Algorithms, Model Problem Analysis and Applications," *Multigrid Methods Proc.*, 1–176, (Springer-Verlag, Berlin, 1982).

[42] D. Terzopoulos "Image analysis using multigrid relaxation methods," *IEEE Trans. Pattern Analysis Machine Intelligence* 8, 129, (1986).

[43] V. Torre, T. Poggio "On Edge Detection," *IEEE Trans. Pattern Analysis and MAchine Intelligence* 8(2), 147–163, (1986).

[44] S. Ullman "Analysis of Visual Motion by Biological and Computer Systems," *IEEE Computer*, 57–69, (August 1981).

[45] S. Uras, F. Girosi, A. Verri and V. Torre "A computational approach to motion perception," *Biological Cybernetics* **60**, 79–87, (1988).

References mostly for Chapter 4

[46] R. Battiti "Surface Reconstruction and Discontinuity Detection: a Fast Hierarchical Approach on a Two-Dimensional Mesh," *Proc. of the IV Conf. on Hypercube Concurrent Computers and Applications* (John Gustafson at. al. (eds.), Monterey, 1989).

[47] R. Battiti *Caltech C3P Report*, in preparation.

[48] A. Brandt "Multi-level adaptive solutions to boundary-value problems," *Math. Comput.* **31**, 333–390, (1977).

[49] P.J. Burt "The pyramid as a structure for efficient computation," Rosenfeld A.(ed)*Multiresolution image processing and analysis*, 6–35, (Springer-Verlag, 1984).

[50] T.F. Chan and Y. Saad "Multigrid Algorithms on the Hypercube Multiprocessor," *IEEE Trans. on Computers* **Vol. C-35, No. 11**, (1986).

[51] H. Embrechts , D. Roose "Efficiency and Load Balancing Issues for a Parallel Component Labelling Algorithm," *Proc. of the IV Conf. on Hypercube Concurrent Computers and Applications*, (John Gustafson at. al. (eds.), Monterey, CA, 1989).

[52] W. Enkelmann "Investigations of multigrid algorithms for the estimation of optical flow fields in image sequences," *Computer Vision, Graphics and Image Processing* **43**, 150–177, (1988).

[53] G. Fox , M. Johnson , G. Lyzenga ,S. Otto , J. Salmon , D. Walker *Solving Problems on Concurrent Processors*, (Prentice Hall, New Jersey, 1988).

[54] W. Furmanski and G. G. Fox "Integrated vision project on the computer network," *Caltech C3P report* **623**, (Caltech Concurrent Computation Project, Pasadena, CA 91125, 1988).

[55] F. Glazer "Multilevel relaxation in low-level computer vision," Rosenfeld A.(ed)*Multiresolution image processing and analysis*, 312–330, (Springer-Verlag, 1984).

[56] B.K.P. Horn and G. Schunck "Determining Optical Flow," *Artificial Intelligence* **17**, 185–203, (1981).

[57] H.A.H. Ibrahim , J.R. Kendler , D.E. Shaw "Low-level image analysis tasks on fine-grained tree-structured SIMD machines," *J. Par. Distr. Comput.* 4, 546–574, (1987).

[58] O. McBryan and E. Van de Velde "Hypercube Algorithms and Implementations," *SIAM J. Sci. Stat. Comput.* **8**, 227–287, (1987).

[59] T. Poggio, V. Torre, and C. Koch "Computational vision and regularization theory," *Nature* **317**, 314–319, (1985).

[60] Q.F. Stout "Supporting divide-and-conquer algorithms for image processing," *J. Par. Distr. Comput.* 4, 95–115, (1987).

[61] D. Terzopoulos "Image analysis using multigrid relaxation methods," *IEEE Trans. Pattern Analysis Machine Intelligence* 8, 129, (1986).

[62] R. Battiti "Collective Stereopsis on the Hypercube," *The III Conf. on Hypercube Conc. Comp. and Appl.* Vol II, 1000-1006, (ACM Press, New York, 1988).

[63] R. Battiti "Surface Reconstruction and Discontinuity Detection: a Hierarchical Approach," *Caltech C3P Report* **676** - B, (Caltech Concurrent Computation Project, Pasadena, CA 91125, 1988).

[64] A. Brandt "Multi-level adaptive solutions to boundary-value problems," *Math. Comput.* **31**, 333–390, (1977).

[65] T.F. Chan and Y. Saad "Multigrid Algorithms on the Hypercube Multiprocessor," *IEEE Trans. on Computers* Vol. C-35, No. 11, (1986).

[66] P. Frederickson and O. A. McBryan "Intrinsically Parallel Multiscale Algorithms for Hypercubes," *The III Conf. on Hypercube Conc. Comp. and Appl.* Vol II, 1726–1734, (ACM Press, New York, 1988).

[67] G. Fox, M. Johnson, G. Lyzenga, S. Otto , J. Salmon , D. Walker *Solving Problems on Concurrent Processors*, (Prentice Hall, New Jersey, 1988).

[68] W. Furmanski and G. C. Fox "Integrated vision project on the computer network," *Caltech C3P report* **623**, (Caltech Concurrent Computation Project, Pasadena, CA 91125, 1988).

[69] S. Geman and D. Geman "Stochastic Relaxation,Gibbs Distributions, and the Bayesian Restoration of Images," *IEEE Trans. Pattern Analysis Machine Intelligence* **6**, 721 (1984).

[70] J. G. Harris "A new approach to surface reconstruction: the coupled depth/slope model," *Proc. IEEE First Int. Conf. on Computer Vision* 277–283, (London, 1987).

[71] B.K.P. Horn, M.J. Brooks "The Variational Approach to Shape from Shading," *MIT A.I. Memo* **813**, (1985).

[72] C. Koch, J. Marroquin, A. Yuille "Analog neuronal networks in early vision," *Proc. Natl. Acad. Sci. USA* **83**, 4263–4267, (1986).

[73] D. Marr and T. Poggio "Cooperative computation of stereo disparity," *Science* **195**, 283–287, (1976).

[74] J.L. Marroquin "Surface Reconstruction Preserving Discontinuities," *MIT A.I. Memo* **792**, (1984).

[75] T. Poggio and C. Koch "Ill-posed problems in early vision: from computational theory to analogue networks," *Proc. R. Soc. Lond. B* **218**, 303–323, (1985).

[76] T. Poggio, V. Torre and C. Koch "Computational vision and regularization theory," *Nature* **317**, 314–319, (1985).

[77] K. Stüben and Trottenberg "Multigrid Methods: Fundamental Algorithms, Model Problem Analysis and Applications," *Multigrid Methods Proc.*, 1–176, (Springer-Verlag, Berlin, 1982).

[78] D. Terzopoulos "Image analysis using multigrid relaxation methods," *IEEE Trans. Pattern Analysis Machine Intelligence* 8, 129, (1986).

[79] D. Terzopoulos "Regularization of inverse visual problems involving discontinuities," *IEEE Trans. Pattern Analysis Machine Intelligence* 8, 413 (1986).

References mostly for Chapter 5

[80] R. Battiti "Accelerated Back-propagation Learning: Two Optimization Methods," *Complex Systems* accepted for publication (1990).

[81] E.B. Baum "On the Capabilities of Multilayer Perceptrons," *Journal of Complexity* **4**, 193–215, (1988).

[82] A. Borsellino and A. Gamba "An outline of a mathematical theory of PAPA," *Nuovo Cimento Suppl. 2* **20**, 221–231, (1961).

[83] D. S. Broomhead and D. Lowe "Multivariable Functional Interpolation and Adaptive Networks," *Complex Systems* **2**, 321–355, (1988).

[84] P. Collet and J.P. Eckmann,*Iterated map on the Interval as Dynamical Systems*, (Birkhauser, Boston, 1980).

[85] J. Denker, D. Schwartz, B. Wittner, S. Solla, R. Howard, L. Jackel and J. Hopfield "Large Automatic Learning, Rule Extraction, and Generalization," *Complex Systems* **1**, 877–922, (1987).

[86] M. Feigenbaum, J. Stat. Phys. **19**, 25, (1978).

[87] G. Fox, M. Johnson, G. Lyzenga, S. Otto , J. Salmon , D. Walker *Solving Problems on Concurrent Processors*, (Prentice Hall, New Jersey, 1988).

[88] P. E. Gill, W. Murray and M. H. Wright, *Practical Optimization*, (Academic Press, 1981).

[89] R. P. Gorman, T. J. Seinowski, "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets," *Neural Networks* **1**, 75–89, (1988).

[90] A. H. Kramer, A. Sangiovanni-Vicentelli, "Efficient Parallel Learning Algorithms For Neural Networks," *Advances in Neural Information Processing Systems* Vol. 1, 75–89, (Morgan Kaufmann, CA, 1988).

[91] A. Lapedes and R. Farber, "Nonlinear signal processing using neural networks: Prediction and system modeling," Los-Alamos Preprint LA- UR-87-1662.

[92] A. Rajavelu, M. T. Musavi, and M. V. Shirvaikar "A Neural Network Approach to Character Recognition," *Neural Networks*, **2**, 387–393, (1989).

[93] T. J. Seinowski and C. R. Rosenberg, "Parallel Networks that learn to pronounce English Text," *Complex Systems*, **1**, 145–168, (1987).

[94] R. A. Jacobs, "Increased Rates of Convergence Through Learning Rate Adaptation," ,*Neural Networks* **1**, 295–307, (1988).

[95] D. E. Rumelhart and J. L. McClelland (eds.),*Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*, (MIT Press, 1986).

[96] D. E. Rumelhart and J. L. McClelland (eds.),*Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 2: Psychological and Biological Models*, (MIT Press, 1986).

[97] R. Serra and G. Zanarini, *Tra Ordine e Chaos*, (Clueb, Bologna, Italy, 1986).

[98] D. F. Shanno, "Conjugate gradient methods with inexact searches," *Mathematics of Operations Research* **3 - 3**, 244–256, (1978).

[99] G. Tesauro, "Scaling relationship in back-propagation learning: dependence on the training set size," *Complex Systems* **1**, 367–372, (1987).

[100] G. Tesauro and B. Janssens, "Scaling relationship in back-propagation learning," *Complex Systems* **2**, 39–44, (1988).

[101] T.P. Vogl, J.K. Mangis, A.K. Rigler, W. T. Zink and D. L. Alkon, "Accelerating the Convergence of the Back-Propagation Method," *Biological Cybernetics* **59**, 257–263, (1988).

[102] P. J. Werbos, "Generalization of Back-propagation with Application to a Recurrent Gas Market Model," *Neural Networks* **1**, 339–356, (1988).

[103] R. D. Williams, "Finite Elements for 2D Elliptic Equations with Moving Nodes," *Caltech C3P Report 423*, (Concurrent Computation Project, Caltech, Pasadena, CA 91125, 1987).

PARALLEL SURFACE RECONSTRUCTION (header)

mg.h

```
/*********************************************************************/
/*mg.h:         program for multigrid surface reconstruction        */
/*              header with data structures and macros              */
/*              written by Roberto Battiti, Feb 1988                */
/*              uses Express communication environment (Parasoft)   */
/*              version with floats, for 65X65 images               */
/*********************************************************************/

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

#define PROFILE 0                       /*1 for profile information    */

#define LAY     7                       /*defines ext memory available */
#define NEU     5722                    /*neurons in LAY layer pyramid */
#define LP      11176                   /*tot of discontinuities       */
#define II      64                      /* i.e. (1<<(LAY-1))           */
#define ID      65                      /* i.e. (II +1) image dimension */
#define STEP(lay)               (II >>(lay))
#define NEU_IN_ROW(lay)         ((1<<(lay))+1)
#define LP_IN_LAY(lay)          (2*NEU_IN_ROW(lay)*(NEU_IN_ROW(lay)-1))
/* neurons inside a given layer:        */
#define NEU_IN_LAY(lay) ( NEU_IN_ROW(lay) * NEU_IN_ROW(lay) )
/*from physical (x,y) to offset in layer*/
#define OFFSET(x,y,l)     ( ((x)/STEP(l)) + ((y)/STEP(l))*NEU_IN_ROW(l) )
/*from offset in layer to physical (x,y)*/
#define PHYSX(o,l)        ( ((o)%NEU_IN_ROW(l)) *STEP(l) )
#define PHYSY(o,l)        ( ((o)/NEU_IN_ROW(l)) *STEP(l) )

/*parallel version stuff||||||||||||||||||||||||||||||||||||||||||||||||||*/
#include "express.h"
#define NODES 4
#define NX 2
#define NY 2
#define NODE_NEU  1797  /*neurons in LAY layer pyramid,for each node, plus border*/
#define NODE_LP   3432  /*tot of discontinuities        */
#define NODE_II 32
#define NODE_ID 33      /*35 including border*/
#define COMM_SIZE (NODE_ID*NODE_ID*4)   /*communication size = image patch in node*/
/*depends on DATA_TYPE*/

#define QUIT  0         /*commands sent to nodes by host*/
#define LOAD_IMAGE 1
#define BACK_IMAGE 2
#define DEBUG   3
#define FMG     4

#define NODE_NEU_IN_ROW(lay)     (NEU_IN_ROW(lay-1) +2)  /*2 for BORDER pixels*/
#define NODE_LP_IN_LAY(lay)      (2*NODE_NEU_IN_ROW(lay)*(NODE_NEU_IN_ROW(lay)-1))
/* neurons inside a given layer:          */
/* neurons inside a given layer:          */
#define NODE_NEU_IN_LAY(lay)     ( NODE_NEU_IN_ROW(lay) * NODE_NEU_IN_ROW(lay) )
/*from relative (x,y) to offset from node_layer*/
/*(x,y)=(00) for top-left OWNED pixel*/
#define NODE_OFFSET(x,y,l)       ( (1+((x)/STEP(l))) + (1+((y)/STEP(l)))*NODE_NEU_IN_ROW(
l) )
/*from offset in layer to physical (x,y)*/
#define NODE_PHYSX(o,l) ( (((o)%NODE_NEU_IN_ROW(l)) -1) *STEP(l) )
#define NODE_PHYSY(o,l) ( (((o)/NODE_NEU_IN_ROW(l)) -1) *STEP(l) )

extern int T_command_to_node();
extern char fromnodes[COMM_SIZE];
extern int host_back_image();
/*parallel version stuff||||||||||||||||||||||||||||||||||||||||||||||*/
```

```
#define MAX_RAND 2147483647             /*2^31 -1                    */
#define MAX_STRLEN 56
/*values for mg_flag:           */
#define SHAPE_INTEGR     0              /* shape enforcing integrability*/
#define SHAPE_SMOOTH     1              /*  ...............smoothness   */
#define SURFACE          2              /* surface reconstruction       */
#define LINE_PROC        3              /* line processes               */

#define LP_METHOD_TABLE 0               /*methods for fill_lp_table()   */
#define LP_METHOD_CONST 1
#define LP_METHOD_WOJTEK 2
#define LP_METHOD_CHRISTOF 3
#define DOWN_METHOD_SIMPLE 0            /*methods for interpolation     */
#define DOWN_METHOD_STAR 1
#define DOWN_METHOD_STAR_DISC 2

#define INFINITY        1000000000000.0

#define STOP     {while(fgetc(stdin)==EOF){}}

#define BYTE unsigned char
typedef float DATA_TYPE;                /*basic data type: float or double?*/

/*******************************************************************/
typedef struct _MGPAR{                  /*state variables and parameters*/
        int lp_method,down_method;
        int nlay,naa,na,nb,nc,ran;
        int history,film,zoom,ld;
        DATA_TYPE max_h;
        DATA_TYPE mul;
        DATA_TYPE  alpha,beta,noise;
        DATA_TYPE dh,a1,a2,a3,a4,ai;    /*lp price list */
        DATA_TYPE ru,rd;        /*reduction for up or down lines*/
        char z_out_file[MAX_STRLEN];
        char disc_out_file[MAX_STRLEN];
        char cmd_file[MAX_STRLEN];
        char z_file[MAX_STRLEN];
        char pr_file[MAX_STRLEN];
        char history_file[MAX_STRLEN];
} MGPAR;                /* alpha = 1/lambda             */
#define EX_LP_METHOD    LP_METHOD_TABLE
#define EX_DOWN_METHOD  DOWN_METHOD_STAR
#define EX_NLAY         4
#define EX_NAA          1
#define EX_NA           1
#define EX_NB           1
#define EX_NC           1
#define EX_RAN          13
#define EX_HISTORY      1
#define EX_FILM         0
#define EX_ZOOM         1
#define EX_ALPHA        0.005
#define EX_BETA         0.25            /* data agreement spring        */
#define EX_NOISE        0.25
#define EX_MAX_H        10.0
#define EX_MUL          1.0
#define EX_DH           10.0
#define EX_A1           .8              /* a1...ai are multiplying price*/
#define EX_A2           .4
#define EX_A3           1.3
#define EX_A4           1.6
#define EX_AI           10.0
#define EX_RU           0.5
```

```
#define EX_RD          0.5
#define EX_Z_OUT_FILE "z"
#define EX_DISC_OUT_FILE "disc"
#define EX_CMD_FILE "demo.com"
#define EX_Z_FILE "ran3.z"
#define EX_PR_FILE "film.pr"
#define EX_HISTORY_FILE "demo.history"

#define INIT_MGPAR       \
{EX_LP_METHOD,EX_DOWN_METHOD,                                      \
 EX_NLAY,EX_NAA,EX_NA,EX_NB,EX_NC,EX_RAN,                          \
 EX_HISTORY,EX_FILM,EX_ZOOM,0,                                     \
 EX_MAX_H,                                                         \
 EX_MUL,                                                           \
 EX_ALPHA,EX_BETA,EX_NOISE,                                        \
 EX_DH,EX_A1,EX_A2,EX_A3,EX_A4,EX_AI,                              \
 EX_RU,EX_RD,                                                      \
 EX_Z_OUT_FILE,                                                    \
 EX_DISC_OUT_FILE,                                                 \
 EX_CMD_FILE,                                                      \
 EX_Z_FILE,                                                        \
 EX_PR_FILE,                                                       \
 EX_HISTORY_FILE                                                   \
}
/*******************************************************************/
typedef struct _NEURON{                /*basic neuron with connections */
        BYTE n_type;
        DATA_TYPE z,*znoisy;
        BYTE bt;        /*additional byte                */
        struct _NEURON *n, *s, *e, *w;
        struct _NEURON *ne, *se, *nw, *sw;
        struct _NEURON *next;
        struct _NEURON *on, *os, *oe, *ow;      /*pt to others  */
        struct _NEURON *u0,*u1,*d0,*d1,*d2,*d3,*d4,*d5; /*disc. conn. betw. layers*/
} NEURON;
#define INIT_Z 0.0
#define INIT_ZNOISY (DATA_TYPE *)0
#define INIT_BT 0
#define VOID ((NEURON *) 0)
#define IN_NEU  1                      /*values for n_type: internal   */
#define BO_NEU  2              /* on the REAL border    */
#define NODE_BO_NEU  0         /*for pixels on the "communication border in NODES*/

#define INITIAL_NEU     {                             \
                        NODE_BO_NEU,                          \
                        INIT_Z,INIT_ZNOISY,           \
                        INIT_BT,                      \
                        VOID,VOID,VOID,VOID,          \
                        VOID,VOID,VOID,VOID,          \
                        VOID,                         \
                        VOID,VOID,VOID,VOID,          \
                        VOID,VOID,VOID,VOID,VOID,VOID,VOID,VOID \
                        }
#define LP_TABLE_SIZE   256
/*******************************************************************/

/*access and movement in pyramid structure macros                  */
#define gett(val)  this->val
#define getn(val)       this->n->val
#define gets(val)       this->s->val
#define gete(val)       this->e->val
#define getw(val)       this->w->val

#define getne(val) this->ne->val
```

```
#define getse(val) this->se->val
#define getnw(val) this->nw->val
#define getsw(val) this->sw->val

#define geton(val)      this->on->val
#define getos(val)      this->os->val
#define getoe(val)      this->oe->val
#define getow(val)      this->ow->val

#define getd0(val)      this->d0->val
#define getd1(val)      this->d1->val
#define getd2(val)      this->d2->val
#define getd3(val)      this->d3->val
#define getd4(val)      this->d4->val
#define getd5(val)      this->d5->val
#define getu0(val)      this->u0->val
#define getu1(val)      this->u1->val


#define H_INDEX(i)      \
{ i=0;if(getn(bt))i+=64;if(getne(bt))i+=128;if(gete(bt))i+=1;if(getse(bt))i+=2;  \
  if(gets(bt))i+=4;if(getsw(bt))i+=8;if(getw(bt))i+=16;if(getnw(bt))i +=32;}

#define V_INDEX(i)      \
{ i=0;if(getn(bt))i+=1;if(getne(bt))i+=2;if(gete(bt))i+=4;if(getse(bt))i+=8;    \
  if(gets(bt))i+=16;if(getsw(bt))i+=32;if(getw(bt))i+=64;if(getnw(bt))i+=128;}


/****************************extern declarations************************/
extern MGPAR mgpar;
extern int read_commandfile();
extern int dump_disc();
extern int dump_image();
extern int rand();
extern int sw_header();
extern int sw_vis_feedback(); /*visual feedback */
extern DATA_TYPE z[ID][ID];
```

PARALLEL SURFACE RECONSTRUCTION (host program)
hmg.c

```
/*********************************************************************/
/*hmg.c:        program for multigrid surface reconstruction        */
/*              host program                                         */
/*              written by Roberto Battiti, Feb 1988                 */
/*              uses Express communication environment (Parasoft)    */
/*********************************************************************/

/*pr_  ...pixrect reference manual routines (SUN microsystems)       */
/*user interface: sunview, but easily portable  (sw_ routines)       */


#include "mg.h"
#include <pixrect/pixrect_hs.h>          /*image format is pixrect file  */

MGPAR mgpar = INIT_MGPAR;

DATA_TYPE z[ID][ID];              /*array containing complete "image"*/

char fromnodes[COMM_SIZE];        /* for communicating with nodes */
char tonodes[NODES*COMM_SIZE];    /* for communicating with nodes */
short coord[NODES][2];            /*x and y coordinate for each node*/

struct nodenv env;
int *ptoi;

main(argc, argv)
int argc;
char *argv[];
{
        T_load();                 /*load nodes and start command interpreter*/
        sw_init();                /* sunview user interface starts        */
#if PROFILE
        cprofcp();
#endif

        exit(0);
}

int T_load()                      /*load nodes and start command interpreter*/
{
        int dest;
        int pgind, nodes;
        int src, type = 123;
        int n;

        if(pgind=exopen("/dev/transputer", NODES, DONTCARE) < 0) exit(1);
        exload(pgind, "nmg");

        exparam(&env);                  /* Get system parameters */

        /*NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN*/
        for(n=0;n<env.nprocs;n++){
                src = DONTCARE;
                exread(fromnodes, COMM_SIZE, &src, &type);
                ptoi=(int *)fromnodes;
                coord[src][0]= *ptoi++;
                coord[src][1]= *ptoi++;
        }
        /*NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN*/
        printf("nodes are alive and well...\n");
        for(n=0;n<env.nprocs;n++){
                printf("node%d %d %d..\n",n,coord[n][0],coord[n][1]);
        }
```

```
int T_command_to_nodes(comm) int comm;
{
        int type=123;
        int command;

        command=comm;
        exbroadcast(&command, HOST, 4, ALLNODES, (int *)0, &type);
}

int dump_data()         /*dumps surface values as printable numbers*/
{

        FILE *z_out_file;
        static int serialn=0;
        char str[MAX_STRLEN];
        int i;
        DATA_TYPE *ptodt;

        sprintf(str,"%s_%d_%d.data",mgpar.z_out_file,serialn,NEU_IN_ROW(LAY-1));
        z_out_file=fopen(str,"w");
        i=ID*ID;
        ptodt=(DATA_TYPE *)z;
        while(i--){
                fprintf(z_out_file,"%f\n",*ptodt++);
        }
        fclose(z_out_file);
        serialn++;
}

int load_z_values()             /*from pixrect file to internal memory  */
{
        NEURON *this,*p;
        int i,j,fd,n,offx,offy;
        DATA_TYPE *ptodt;
        FILE *z_file;
        int src,dest,type=123;

        if((fd = open(mgpar.z_file,0))== -1)    {
                ERR(10);
                return(0);
        }
        while((i= read(fd,z,ID*ID*sizeof(DATA_TYPE)))!= ID*ID*sizeof(DATA_TYPE))
        {
                ERR(20);
                close(fd);
                fd=open(mgpar.z_file);
        }
        mgpar.ld=1;
        close(fd);

        mgpar.max_h=0.0;
        ptodt=(DATA_TYPE *)z;
        i=ID*ID;
        while(i--){
                if(*ptodt > mgpar.max_h)mgpar.max_h= *ptodt;
                ptodt++;
        }

        randomize_z();

        T_command_to_nodes(LOAD_IMAGE);
        ptodt= ((DATA_TYPE *)(tonodes+ COMM_SIZE*NODES)) -1;

        n=NODES;
        while(n--){
                offx=coord[n][0]*(NODE_II);
                offy=(NY-1-coord[n][1])*(NODE_II);
                j=NODE_ID;
                while(j--){
                        i=NODE_ID;
                        while(i--){
                                *ptodt-- =z[j+offy][i+offx];
                        }
                }
        }

        /*write image to nodes*/
        for(n=0;n<env.nprocs;n++){
                dest=n;
                exwrite(tonodes+n*COMM_SIZE,COMM_SIZE,&dest,&type);
        }

        sw_msg("image written to nodes...\n");

        sw_vis_feedback(LAY-1,SURFACE,"z values......");

        return(1);
}

int host_back_image()
{
        int n,src,type=123,offx,offy,i,j;
        DATA_TYPE *ptodt;

        T_command_to_nodes(BACK_IMAGE);
        for(n=0;n<env.nprocs;n++){
                src = DONTCARE;
                exread(fromnodes, COMM_SIZE, &src, &type);

                ptodt= ((DATA_TYPE *)(fromnodes+COMM_SIZE))-1;
                offx=coord[src][0]*(NODE_II);
                offy=(NY-1-coord[src][1])*(NODE_II);
                j=NODE_ID;
                while(j--){
                        i=NODE_ID;
                        while(i--){
                                z[j+offy][i+offx]= *ptodt--;
                        }
                }
        }
        sw_msg("image read from nodes...\n");

        sw_vis_feedback(LAY-1,SURFACE,"z values......");

        return(1);
}

/*#####################################################################################*/
int fmg(mg_flag) int mg_flag;           /* basic multigrid routine */
{
        int type=123;

        T_command_to_nodes(FMG);
        exbroadcast((char *)(&mgpar), HOST, sizeof(MGPAR), ALLNODES, (int *)0, &type);
```

```
/*****************************************************************************/

int dump_image(x,y,dx,dy,filename)      /*dumps region of screen in pixrect file*/
int x,y,dx,dy;
char *filename;         /*used for film-making                  */
{
        colormap_t *colormap=NULL;
        int type = RT_STANDARD;
        struct pixrect *screen,*icon;
        FILE *output;

        while((output=fopen(filename,"w"))==NULL)       {
                ERR(10);
                sleep(3);
        }

        screen=pr_open("/dev/fb");
        icon=pr_region(screen,x,y,dx,dy);
        while( pr_dump(icon,output,colormap,type,1)==PIX_ERR){
                ERR(13);
                fclose(output);
                sleep(3);
                output = fopen(filename, "w");
        }

        pr_close(screen);
        pr_close(icon);
        fclose(output);
}

int randomize_z()
#define MY_RAND (((DATA_TYPE) rand()/(DATA_TYPE) MAX_RAND)-.5)
{
        int i,j;
        DATA_TYPE maxnoise;

        srand(mgpar.ran);
        maxnoise = mgpar.noise * mgpar.max_h;
        for(i=0;i<ID;i++){
                for(j=0;j<ID;j++){
                        z[i][j] += maxnoise* MY_RAND;
                }
        }
}


ERR(number) int number;
{
        switch(number){
        case 10:
                sw_msg("\n error opening  file");
                break;
        case 11:
                sw_msg("\n error loading-creating raster file");
                break;
        case 12:
                sw_msg("\nwrong image size");
                break;
        case 13:
                sw_msg("\npr_dump failed");
                break;
        case 20:
                sw_msg("\n error in reading");
                break;
        case 100:
                sw_msg("bad cmd_file");
                break;
        default:
                break;
        }
}
```

PARALLEL SURFACE RECONSTRUCTION (graphics and user interface)

hmg_sw.c

```
/***********************************************************************/
/*hmg_sw.c:     Sunview routines (SUN) used for user interface im hmg.c */
/*              to be compiled with host program                        */
/*              written by Roberto Battiti, Feb 1988                    */
/*              uses Express communication environment (Parasoft)       */
/***********************************************************************/

#include "mg.h"
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/tty.h>
#include <suntool/panel.h>
#include <pixrect/pixrect_hs.h>         /*image format is pixrect file  */

#define PROG_LABEL "multigrid surface reconstruction from noisy z values"
#define LINE_WIDTH 3
#define SCREENX 1152
#define U_WIN_X 0                       /* U_ = user specified          */
#define U_WIN_Y 32
#define U_WIN_HEIGHT 850
#define U_WIN_WIDTH 850
#define U_WIN_ROWS  15
#define U_WIN_COLUMNS 32
#define MILLE 1000
#define PIX_COLOR(color) ((color)<<5)
#define MY_PIX 0x19                     /* no clipping                  */
#define LINE(line)  12*(line)
#define PIX     3                       /*size of a pixel in screen units*/
#define PIXX    5

Frame base_frame,setpar_frame;
Panel panel,setpar_panel;
Panel_item msg_item,fname_item,history_item,out_fname_item,disc_out_fname_item,cmd_fname
_item;
Canvas canvas;
Pixwin *pw;
Rect rect;
unsigned char colmap[3][256];

int bin_assoc_table[7]= {
        1,2,4,8,16,32,64};
int zeroten_assoc_table[11]= {
        0,1,2,3,4,5,6,7,8,9,10};
int yesno_assoc_table[2]={
        0,1};

typedef struct _SWPAR{
        int *history_assocp[2];
        int *film_assocp[2];
        int *zoom_assocp[2];
        int *lp_method_assocp[2];
        int *down_method_assocp[2];
        int *nlay_assocp[2];
        int *naa_assocp[2];
        int *na_assocp[2];
        int *nb_assocp[2];
        int *nc_assocp[2];
} SWPAR;

SWPAR sw_par = {
        {&(mgpar.history),yesno_assoc_table},
        {&(mgpar.film),yesno_assoc_table},
        {&(mgpar.zoom),yesno_assoc_table},
        {&(mgpar.lp_method),zeroten_assoc_table},
        {&(mgpar.down_method),zeroten_assoc_table},
        {&(mgpar.nlay),zeroten_assoc_table},
        {&(mgpar.naa),zeroten_assoc_table},
        {&(mgpar.na),zeroten_assoc_table},
        {&(mgpar.nb),zeroten_assoc_table},
        {&(mgpar.nc),zeroten_assoc_table}
};



sw_init()
{
        base_frame = window_create(0, FRAME,
            WIN_X, U_WIN_X,
            WIN_Y, U_WIN_Y,
            FRAME_INHERIT_COLORS,       TRUE,
            FRAME_LABEL, PROG_LABEL,
            0);

        canvas = window_create(base_frame, CANVAS,
            WIN_HEIGHT, U_WIN_HEIGHT,
            WIN_WIDTH, U_WIN_WIDTH,
            CANVAS_RETAINED,    FALSE,
            0);
        rect.r_left=rect.r_top=0;
        rect.r_width=U_WIN_WIDTH;
        rect.r_height=U_WIN_HEIGHT;
        pw = canvas_pixwin(canvas);

        panel = window_create(base_frame,        PANEL,
            WIN_RIGHT_OF,                canvas,
            0);
        sw_create_panel_items();
        window_fit(panel);

        window_fit(base_frame);
        sw_put_colmap();
        window_main_loop(base_frame);
}

int sw_create_panel_items()
{
        int i=0;
        int sw_quit(),sw_read_commandfile(),sw_load_z_values(),sw_dump_data();
        int sw_button(),sw_value(),sw_int_value(),sw_setpar();
        int sw_dump_disc();

        int sw_debug(),sw_host_back_image();
        panel_create_item(panel, PANEL_SLIDER,
            PANEL_LABEL_X,                          ATTR_COL(0),
            PANEL_LABEL_Y,                          ATTR_ROW(i++),
            PANEL_LABEL_STRING,          "mul:",
            PANEL_VALUE,                            (int)(mgpar.mul*MILLE),
            PANEL_MIN_VALUE,             0,
            PANEL_MAX_VALUE,             10*MILLE,
            PANEL_SLIDER_WIDTH,          100,
            PANEL_NOTIFY_LEVEL,          PANEL_DONE,
            PANEL_NOTIFY_PROC,           sw_value,
            PANEL_CLIENT_DATA,           (caddr_t)(&(mgpar.mul)),
            0);
        panel_create_item(panel, PANEL_SLIDER,
            PANEL_LABEL_X,                          ATTR_COL(0),
            PANEL_LABEL_Y,                          ATTR_ROW(i++),
            PANEL_LABEL_STRING,          "ran:",
    PANEL_VALUE,                          (int)(mgpar.ran),
    PANEL_MIN_VALUE,              0,
    PANEL_MAX_VALUE,              MILLE,
    PANEL_SLIDER_WIDTH,           100,
    PANEL_NOTIFY_LEVEL,           PANEL_DONE,
    PANEL_NOTIFY_PROC,            sw_int_value,
    PANEL_CLIENT_DATA,            (caddr_t)(&(mgpar.ran)),
    0);
panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                        ATTR_COL(0),
    PANEL_LABEL_Y,                        ATTR_ROW(i++),
    PANEL_LABEL_STRING,           "noise:",
    PANEL_VALUE,                          (int)(mgpar.noise*MILLE),
    PANEL_MIN_VALUE,              0,
    PANEL_MAX_VALUE,              MILLE,
    PANEL_SLIDER_WIDTH,           100,
    PANEL_NOTIFY_LEVEL,           PANEL_DONE,
    PANEL_NOTIFY_PROC,            sw_value,
    PANEL_CLIENT_DATA,            (caddr_t)(&(mgpar.noise)),
    0);
panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                        ATTR_COL(0),
    PANEL_LABEL_Y,                        ATTR_ROW(i++),
    PANEL_LABEL_STRING,           "beta:",
    PANEL_VALUE,                          (int)(mgpar.beta*MILLE),
    PANEL_MIN_VALUE,              0,
    PANEL_MAX_VALUE,              MILLE,
    PANEL_SLIDER_WIDTH,           100,
    PANEL_NOTIFY_LEVEL,           PANEL_DONE,
    PANEL_NOTIFY_PROC,            sw_value,
    PANEL_CLIENT_DATA,            (caddr_t)(&(mgpar.beta)),
    0);

panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                        ATTR_COL(0),
    PANEL_LABEL_Y,                        ATTR_ROW(i++),
    PANEL_LABEL_STRING,           "dh:",
    PANEL_VALUE,                          (int)(mgpar.dh   *MILLE),
    PANEL_MIN_VALUE,              0,
    PANEL_MAX_VALUE,              MILLE*100,
    PANEL_SLIDER_WIDTH,           100,
    PANEL_NOTIFY_LEVEL,           PANEL_DONE,
    PANEL_NOTIFY_PROC,            sw_value,
    PANEL_CLIENT_DATA,            (caddr_t)(&(mgpar.dh)),
    0);

panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                        ATTR_COL(0),
    PANEL_LABEL_Y,                        ATTR_ROW(i++),
    PANEL_LABEL_STRING,           "a1:",
    PANEL_VALUE,                          (int)(mgpar.a1   *MILLE),
    PANEL_MIN_VALUE,              0,
    PANEL_MAX_VALUE,              MILLE*2,
    PANEL_SLIDER_WIDTH,           100,
    PANEL_NOTIFY_LEVEL,           PANEL_DONE,
    PANEL_NOTIFY_PROC,            sw_value,
    PANEL_CLIENT_DATA,            (caddr_t)(&(mgpar.a1)),
    0);

panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                        ATTR_COL(0),
    PANEL_LABEL_Y,                        ATTR_ROW(i++),
    PANEL_LABEL_STRING,           "a2:",
    PANEL_VALUE,                          (int)(mgpar.a2   *MILLE),
    PANEL_MIN_VALUE,            0,
    PANEL_MAX_VALUE,            MILLE*2,
    PANEL_SLIDER_WIDTH,         100,
    PANEL_NOTIFY_LEVEL,         PANEL_DONE,
    PANEL_NOTIFY_PROC,          sw_value,
    PANEL_CLIENT_DATA,          (caddr_t) (& (mgpar.a2)),
    0);

panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                      ATTR_COL(0),
    PANEL_LABEL_Y,                      ATTR_ROW(i++),
    PANEL_LABEL_STRING,         "a3:",
    PANEL_VALUE,                        (int) (mgpar.a3  *MILLE),
    PANEL_MIN_VALUE,            0,
    PANEL_MAX_VALUE,            MILLE*2,
    PANEL_SLIDER_WIDTH,         100,
    PANEL_NOTIFY_LEVEL,         PANEL_DONE,
    PANEL_NOTIFY_PROC,          sw_value,
    PANEL_CLIENT_DATA,          (caddr_t) (& (mgpar.a3)),
    0);

panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                      ATTR_COL(0),
    PANEL_LABEL_Y,                      ATTR_ROW(i++),
    PANEL_LABEL_STRING,         "a4:",
    PANEL_VALUE,                        (int) (mgpar.a4  *MILLE),
    PANEL_MIN_VALUE,            0,
    PANEL_MAX_VALUE,            MILLE*2,
    PANEL_SLIDER_WIDTH,         100,
    PANEL_NOTIFY_LEVEL,         PANEL_DONE,
    PANEL_NOTIFY_PROC,          sw_value,
    PANEL_CLIENT_DATA,          (caddr_t) (& (mgpar.a4)),
    0);

panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                      ATTR_COL(0),
    PANEL_LABEL_Y,                      ATTR_ROW(i++),
    PANEL_LABEL_STRING,         "ai:",
    PANEL_VALUE,                        (int) (mgpar.ai  *MILLE),
    PANEL_MIN_VALUE,            0,
    PANEL_MAX_VALUE,            MILLE*100,
    PANEL_SLIDER_WIDTH,         100,
    PANEL_NOTIFY_LEVEL,         PANEL_DONE,
    PANEL_NOTIFY_PROC,          sw_value,
    PANEL_CLIENT_DATA,          (caddr_t) (& (mgpar.ai)),
    0);

panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                      ATTR_COL(0),
    PANEL_LABEL_Y,                      ATTR_ROW(i++),
    PANEL_LABEL_STRING,         "ru:",
    PANEL_VALUE,                        (int) (mgpar.ru  *MILLE),
    PANEL_MIN_VALUE,            0,
    PANEL_MAX_VALUE,            MILLE,
    PANEL_SLIDER_WIDTH,         100,
    PANEL_NOTIFY_LEVEL,         PANEL_DONE,
    PANEL_NOTIFY_PROC,          sw_value,
    PANEL_CLIENT_DATA,          (caddr_t) (& (mgpar.ru)),
    0);

panel_create_item(panel, PANEL_SLIDER,
    PANEL_LABEL_X,                      ATTR_COL(0),
    PANEL_LABEL_Y,                      ATTR_ROW(i++),
    PANEL_LABEL_STRING,         "rd:",
    PANEL_VALUE,                      (int)(mgpar.rd  *MILLE),
    PANEL_MIN_VALUE,            0,
    PANEL_MAX_VALUE,            MILLE,
    PANEL_SLIDER_WIDTH,         100,
    PANEL_NOTIFY_LEVEL,         PANEL_DONE,
    PANEL_NOTIFY_PROC,          sw_value,
    PANEL_CLIENT_DATA,          (caddr_t)(&(mgpar.rd)),
    0);


fname_item = panel_create_item(panel, PANEL_TEXT,
    PANEL_LABEL_X,                    ATTR_COL(0),
    PANEL_LABEL_Y,                    ATTR_ROW(i++),
    PANEL_VALUE_DISPLAY_LENGTH,       12,
    PANEL_LABEL_STRING,               "z_file:",
    PANEL_VALUE,                      mgpar.z_file,
    0);

out_fname_item = panel_create_item(panel, PANEL_TEXT,
    PANEL_LABEL_X,                    ATTR_COL(0),
    PANEL_LABEL_Y,                    ATTR_ROW(i++),
    PANEL_VALUE_DISPLAY_LENGTH,       12,
    PANEL_LABEL_STRING,               "key for (z).data:",
    PANEL_VALUE,                      mgpar.z_out_file,
    0);

disc_out_fname_item = panel_create_item(panel, PANEL_TEXT,
    PANEL_LABEL_X,                    ATTR_COL(0),
    PANEL_LABEL_Y,                    ATTR_ROW(i++),
    PANEL_VALUE_DISPLAY_LENGTH,       12,
    PANEL_LABEL_STRING,               "key for (disc).pr:",
    PANEL_VALUE,                      mgpar.disc_out_file,
    0);
cmd_fname_item = panel_create_item(panel, PANEL_TEXT,
    PANEL_LABEL_X,                    ATTR_COL(0),
    PANEL_LABEL_Y,                    ATTR_ROW(i++),
    PANEL_VALUE_DISPLAY_LENGTH,       12,
    PANEL_LABEL_STRING,               "cmd_file(.com):",
    PANEL_VALUE,                      mgpar.cmd_file,
    0);

history_item = panel_create_item(panel, PANEL_TEXT,
    PANEL_LABEL_X,                    ATTR_COL(0),
    PANEL_LABEL_Y,                    ATTR_ROW(i++),
    PANEL_VALUE_DISPLAY_LENGTH,       12,
    PANEL_LABEL_STRING,               "history_file(.history):",
    PANEL_VALUE,                      mgpar.history_file,
    0);


sw_create_setpar_popup();

panel_create_item(panel,PANEL_BUTTON,
    PANEL_LABEL_X,                    ATTR_COL(0),
    PANEL_LABEL_Y,                    ATTR_ROW(i++),
    PANEL_LABEL_IMAGE,panel_button_image(panel,"Debug",0,0),
    PANEL_NOTIFY_PROC,          sw_debug,
    0);

panel_create_item(panel,PANEL_BUTTON,
    PANEL_LABEL_X,                    ATTR_COL(0),
    PANEL_LABEL_Y,                    ATTR_ROW(i++),
    PANEL_LABEL_IMAGE,panel_button_image(panel,"Back image",0,0),
    PANEL_NOTIFY_PROC,          sw_host_back_image,
            0);

        panel_create_item(panel,PANEL_BUTTON,
            PANEL_LABEL_X,                      ATTR_COL(0),
            PANEL_LABEL_Y,                      ATTR_ROW(i++),
            PANEL_LABEL_IMAGE,panel_button_image(panel,"Set par",0,0),
            PANEL_NOTIFY_PROC,          sw_setpar,
            0);

        panel_create_item(panel, PANEL_BUTTON,
            PANEL_LABEL_X,                      ATTR_COL(0),
            PANEL_LABEL_Y,                      ATTR_ROW(i++),
            PANEL_LABEL_IMAGE,panel_button_image(panel,"com<file",4,0),
            PANEL_NOTIFY_PROC,          sw_read_commandfile,
            0);

        panel_create_item(panel, PANEL_BUTTON,
            PANEL_LABEL_X,                      ATTR_COL(0),
            PANEL_LABEL_Y,                      ATTR_ROW(i++),
            PANEL_LABEL_IMAGE,panel_button_image(panel,"Load&Rand",4,0),
            PANEL_NOTIFY_PROC,          sw_load_z_values,
            0);

        panel_create_item(panel, PANEL_BUTTON,
            PANEL_LABEL_X,                      ATTR_COL(0),
            PANEL_LABEL_Y,                      ATTR_ROW(i++),
            PANEL_LABEL_IMAGE,panel_button_image(panel,"Dump z single(.data)",4,0),
            PANEL_NOTIFY_PROC,          sw_dump_data,
            0);

        panel_create_item(panel, PANEL_BUTTON,
            PANEL_LABEL_X,                      ATTR_COL(0),
            PANEL_LABEL_Y,                      ATTR_ROW(i++),
            PANEL_LABEL_IMAGE,panel_button_image(panel,"Dump disc single(.pr)",4,0),
            PANEL_NOTIFY_PROC,          sw_dump_disc,
            0);

        panel_create_item(panel, PANEL_BUTTON,
            PANEL_LABEL_X,                      ATTR_COL(0),
            PANEL_LABEL_Y,                      ATTR_ROW(i++),
            PANEL_LABEL_IMAGE,panel_button_image(panel,"Surface",4,0),
            PANEL_CLIENT_DATA,          SURFACE,
            PANEL_NOTIFY_PROC,          sw_button,
            0);

        panel_create_item(panel, PANEL_BUTTON,
            PANEL_LABEL_X,                      ATTR_COL(0),
            PANEL_LABEL_Y,                      ATTR_ROW(i++),
            PANEL_LABEL_IMAGE,panel_button_image(panel,"Quit",4,0),
            PANEL_NOTIFY_PROC,          sw_quit,
            0);


}

int sw_create_setpar_popup()
{

        int i=0,sw_done();
        setpar_frame= window_create(base_frame, FRAME,
            WIN_X, U_WIN_WIDTH,
            WIN_Y, U_WIN_Y,
            FRAME_DONE_PROC,            sw_done,
    0);

setpar_panel= window_create(setpar_frame, PANEL,0);

msg_item = panel_create_item(setpar_panel, PANEL_MESSAGE,
    PANEL_ITEM_X,                      ATTR_COL(10),
    PANEL_ITEM_Y,                      ATTR_ROW(i++),
    PANEL_LABEL_STRING,                "User interface parameters",
    0);
/*CAREFUL in initializing panel value: see assoc_table  */

panel_create_item(setpar_panel, PANEL_CYCLE,
    PANEL_ITEM_X,                      ATTR_COL(0),
    PANEL_ITEM_Y,                      ATTR_ROW(i++),
    PANEL_VALUE,                       mgpar.history,
    PANEL_DISPLAY_LEVEL,               PANEL_CURRENT,
    PANEL_LABEL_STRING,                "history:",
    PANEL_CHOICE_STRINGS,              "No","Yes",0,
    PANEL_CLIENT_DATA,          (caddr_t)(sw_par.history_assocp),0,
    0);
panel_create_item(setpar_panel, PANEL_CYCLE,
    PANEL_ITEM_X,                      ATTR_COL(0),
    PANEL_ITEM_Y,                      ATTR_ROW(i++),
    PANEL_VALUE,                       mgpar.film,
    PANEL_DISPLAY_LEVEL,               PANEL_CURRENT,
    PANEL_LABEL_STRING,                "Film:",
    PANEL_CHOICE_STRINGS,              "No","Yes",0,
    PANEL_CLIENT_DATA,          (caddr_t)(sw_par.film_assocp),0,
    0);

panel_create_item(setpar_panel, PANEL_CYCLE,
    PANEL_ITEM_X,                      ATTR_COL(0),
    PANEL_ITEM_Y,                      ATTR_ROW(i++),
    PANEL_VALUE,                       mgpar.zoom,
    PANEL_DISPLAY_LEVEL,               PANEL_CURRENT,
    PANEL_LABEL_STRING,                "Zoom:",
    PANEL_CHOICE_STRINGS,              "No","Yes",0,
    PANEL_CLIENT_DATA,          (caddr_t)(sw_par.zoom_assocp),0,
    0);

panel_create_item(setpar_panel, PANEL_CYCLE,
    PANEL_ITEM_X,                      ATTR_COL(0),
    PANEL_ITEM_Y,                      ATTR_ROW(i++),
    PANEL_VALUE,                       mgpar.lp_method,
    PANEL_DISPLAY_LEVEL,               PANEL_CURRENT,
    PANEL_LABEL_STRING,                "lp_method:",
    PANEL_CHOICE_STRINGS,              "table","const","wojtek","christof",0,
    PANEL_CLIENT_DATA,          (caddr_t)(sw_par.lp_method_assocp),0,
    0);
panel_create_item(setpar_panel, PANEL_CYCLE,
    PANEL_ITEM_X,                      ATTR_COL(0),
    PANEL_ITEM_Y,                      ATTR_ROW(i++),
    PANEL_VALUE,                       mgpar.down_method,
    PANEL_DISPLAY_LEVEL,               PANEL_CURRENT,
    PANEL_LABEL_STRING,                "down_method:",
    PANEL_CHOICE_STRINGS,              "simple","star","star_disc",0,
    PANEL_CLIENT_DATA,          (caddr_t)(sw_par.down_method_assocp),0,
    0);
panel_create_item(setpar_panel, PANEL_CYCLE,
    PANEL_ITEM_X,                      ATTR_COL(0),
    PANEL_ITEM_Y,                      ATTR_ROW(i++),
    PANEL_VALUE,                       mgpar.nlay,
    PANEL_DISPLAY_LEVEL,               PANEL_CURRENT,
    PANEL_LABEL_STRING,                "nlay:",
        PANEL_CHOICE_STRINGS,               "0","1", "2","3","4","5","6",0,
        PANEL_CLIENT_DATA,          (caddr_t)(sw_par.nlay_assocp),0,
        0);
    panel_create_item(setpar_panel, PANEL_CYCLE,
        PANEL_ITEM_X,                       ATTR_COL(0),
        PANEL_ITEM_Y,                       ATTR_ROW(i++),
        PANEL_VALUE,                        mgpar.naa,
        PANEL_DISPLAY_LEVEL,                PANEL_CURRENT,
        PANEL_LABEL_STRING,                 "naa:",
        PANEL_CHOICE_STRINGS,       "0","1","2","3","4","5","6","7","8","9","10",0,
        PANEL_CLIENT_DATA,          (caddr_t)(sw_par.naa_assocp),0,
        0);
    panel_create_item(setpar_panel, PANEL_CYCLE,
        PANEL_ITEM_X,                       ATTR_COL(0),
        PANEL_ITEM_Y,                       ATTR_ROW(i++),
        PANEL_VALUE,                        mgpar.na,
        PANEL_DISPLAY_LEVEL,                PANEL_CURRENT,
        PANEL_LABEL_STRING,                 "na:",
        PANEL_CHOICE_STRINGS,       "0","1","2","3","4","5","6","7","8","9","10",0,
        PANEL_CLIENT_DATA,          (caddr_t)(sw_par.na_assocp),0,
        0);
    panel_create_item(setpar_panel, PANEL_CYCLE,
        PANEL_ITEM_X,                       ATTR_COL(0),
        PANEL_ITEM_Y,                       ATTR_ROW(i++),
        PANEL_VALUE,                        mgpar.nb,
        PANEL_DISPLAY_LEVEL,                PANEL_CURRENT,
        PANEL_LABEL_STRING,                 "nb:",
        PANEL_CHOICE_STRINGS,       "0","1","2","3","4","5","6","7","8","9","10",0,
        PANEL_CLIENT_DATA,          (caddr_t)(sw_par.nb_assocp),0,
        0);
    panel_create_item(setpar_panel, PANEL_CYCLE,
        PANEL_ITEM_X,                       ATTR_COL(0),
        PANEL_ITEM_Y,                       ATTR_ROW(i++),
        PANEL_VALUE,                        mgpar.nc,
        PANEL_DISPLAY_LEVEL,                PANEL_CURRENT,
        PANEL_LABEL_STRING,                 "nc:",
        PANEL_CHOICE_STRINGS,       "0","1","2","3","4","5","6","7","8","9","10",0,
        PANEL_CLIENT_DATA,          (caddr_t)(sw_par.nc_assocp),0,
        0);
    window_fit(setpar_panel);
    window_fit(setpar_frame);
}

int sw_setpar()
{
    window_set(setpar_frame, WIN_SHOW, TRUE,0);
}

int sw_done(frame) Frame frame;
{
    Panel_item item;
    int index, **assocp;

    if(frame == setpar_frame)
    {
        panel_each_item(setpar_panel,item)
            index=(int)panel_get_value(item);
        if(item != msg_item)
        {
            assocp = (int **)panel_get(item,PANEL_CLIENT_DATA,0);
            *(assocp[0]) = *(assocp[1]+index);
        }
        panel_end_each
    }
```

```
        window_set(frame, WIN_SHOW, FALSE,0);
}



sw_value(item,value,event) Panel_item item;
int value;
Event event;
{
        DATA_TYPE *dp;

        dp= (DATA_TYPE *)panel_get(item,PANEL_CLIENT_DATA,0);
        *dp= ((DATA_TYPE)value)/MILLE;

}

sw_int_value(item,value,event) Panel_item item;
int value;
Event event;
{
        int *ip;

        ip= (int *)panel_get(item,PANEL_CLIENT_DATA,0);
        *ip= value;

}


int sw_load_z_values()
{
        strcpy(mgpar.z_file, panel_get_value(fname_item));
        if(mgpar.history)sw_history("l");
        load_z_values();
}

int sw_dump_data()
{
        strcpy(mgpar.z_out_file, panel_get_value(out_fname_item));
        if(mgpar.history)sw_history("ds");
        dump_data();
}

int sw_dump_disc()
{
        strcpy(mgpar.disc_out_file, panel_get_value(disc_out_fname_item));
        if(mgpar.history)sw_history("es");
        dump_disc(LAY-1,mgpar.zoom);
}

int sw_quit()
{
        if(mgpar.history)sw_history("x");
        T_command_to_nodes(QUIT);
        window_destroy(base_frame);
}


int sw_read_commandfile()
{
        strcpy(mgpar.cmd_file, panel_get_value(cmd_fname_item));
        read_commandfile();
}

int sw_put_colmap()
```

```
        int i;
        Pixwin *pw_p;

        /*1,2,3= red,green,blue;0=white;255=black;4-254 gray*/
        i= 256;
        while(i--){
                colmap[0][i] = colmap[1][i] = colmap[2][i] = i;
        }
        colmap[0][1]=0 ;
        colmap[1][1]=0 ;
        colmap[2][1]=255 ;
        colmap[0][2]=0 ;
        colmap[1][2]=122 ;
        colmap[2][2]=122 ;
        colmap[0][3]=0 ;
        colmap[1][3]=255 ;
        colmap[2][3]=0 ;
        colmap[0][4]=122 ;
        colmap[1][4]=255 ;
        colmap[2][4]=0 ;
        colmap[0][5]=255 ;
        colmap[1][5]=255 ;
        colmap[2][5]=0 ;
        colmap[0][6]=255 ;
        colmap[1][6]=122 ;
        colmap[2][6]=0 ;
        colmap[0][7]=255 ;
        colmap[1][7]=0 ;
        colmap[2][7]=0 ;
        colmap[0][8]=122 ;
        colmap[1][8]=0 ;
        colmap[2][8]=122 ;
        colmap[0][9]=255 ;
        colmap[1][9]=255 ;
        colmap[2][9]= 255;

        pw_setcmsname(pw,"colmap");
        pw_putcolormap(pw,0,256,colmap[0],colmap[1],colmap[2]);
        pw_p = (Pixwin *)window_get(base_frame,WIN_PIXWIN);
        pw_setcmsname(pw_p,"colmap");
        pw_putcolormap(pw_p,0,256,colmap[0],colmap[1],colmap[2]);

        pw_p = (Pixwin *)window_get(panel,WIN_PIXWIN);
        pw_setcmsname(pw_p,"colmap");
        pw_putcolormap(pw_p,0,256,colmap[0],colmap[1],colmap[2]);
}

sw_msg(msg) char *msg;
{
        pw_text(pw,10,LINE(1),PIX_SRC,NULL,msg);
}

sw_button(item,event) Panel_item item;
Event event;
{
        int mg_flag;
        mg_flag=(int)panel_get(item,PANEL_CLIENT_DATA,0);

        if(mgpar.ld){
                if(mgpar.history) sw_history("z");
                fmg(mg_flag);
        }
}
```

```
sw_refresh(x0,y0,dx,dy) int x0,y0,dx,dy;
{
        pw_lock(pw,&rect);
        pw_write(pw,x0,y0,dx,dy,PIX_COLOR(10)| MY_PIX,NULL,0,0);
        pw_reset(pw);
}

sw_plot(lay,mg_flag,zoom,str) int lay,mg_flag,zoom;
char *str;
#define YEL (PIX_COLOR(5)|MY_PIX)
#define RED (PIX_COLOR(7)|MY_PIX)
{
        NEURON *this,*first;
        short pix,o,si,sj,i,j,inx,iny,col,nl,nr,n,nn,tnpo,x,y;
        DATA_TYPE th,mul;

        si= (zoom)? PIX : (PIX*((1<<lay)+ 2*lay));      /* (1<<lay) +(lay-1)+lay;*/
        pix= (zoom)? (STEP(lay)*PIXX) : PIX;
        nl=NEU_IN_LAY(lay);
        nr=NEU_IN_ROW(lay);
        pw_text(pw,10,LINE(1),PIX_SRC,NULL,str);

        switch(mg_flag){
        case LINE_PROC:
                nn=NEU_IN_ROW(lay);
                n=nn-1;
                tnpo=2*n +1;
                this=0;
                sj=20;
                sw_refresh(si,sj,nr*pix,nr*pix);
                pw_lock(pw,&rect);
                /**************alternate vert. horiz. ... last vertical********/
                y=sj;
                j=n;
                while(j--){
                        x=si+(pix-1);
                        i=n;
                        while(i--){
                                if(gett(bt))pw_write(pw,x,y,1,pix,RED,NULL,0,0);
                                x +=pix;
                                this++;
                        }
                        y += (pix-1);
                        x=si;
                        i=nn;
                        while(i--){
                                if(gett(bt))pw_write(pw,x,y,pix,1,RED,NULL,0,0);
                                x +=pix;
                                this++;
                        }
                        y ++;
                }
                x=si+(pix-1);
                i=n;
                while(i--){
                        if(gett(bt))pw_write(pw,x,y,1,pix,RED,NULL,0,0);
                        x +=pix;
                        this++;
                }
                /************************************************************/
                pw_reset(pw);
                break;
        case SURFACE:
                sj=420;
                mul=(mgpar.mul*255)/mgpar.max_h;
                pw_lock(pw,&rect);
                o=nl;
                first=0;
                while(o--){
                        this=first+o;
                        col=(gett(z) *mul+128);
                        if(col<10)col=10;
                        else if(col>254)col=254;
                        i= (o%nr)*pix+si;
                        j= (o/nr)*pix+sj;
                        pw_write(pw,i,j,pix,pix,PIX_COLOR((BYTE)col)|MY_PIX,NULL,0,0);
                }
                pw_reset(pw);
                break;
        }
}
int dump_disc(lay,zoom) int lay,zoom;   /*dumps surface values as printable numbers*/
{
        static int serialn=0;
        char str[MAX_STRLEN];
        int pix,si,sj,nr;

        si= (zoom)? PIX : (PIX*((1<<lay)+ 2*lay));      /* (1<<lay) +(lay-1)+lay;*/
        pix= (zoom)? (STEP(lay)*PIXX) : PIX;
        nr=NEU_IN_ROW(lay);

        sprintf(str,"%s_%d_%d.pr",mgpar.disc_out_file,serialn,NEU_IN_ROW(lay));
        dump_image(si,sj,pix*nr,pix*nr,str);
        serialn++;
}

int sw_header(mg_flag) int mg_flag;
{
        char str[MAX_STRLEN];

        switch(mg_flag){
        case SURFACE:
                sprintf(str,"%s","\nSURFACE RECONSTRUCTION                  ");
                break;
        }
        sw_msg(str);
}

int sw_image_show(img_pr) struct pixrect *img_pr;
{
        pw_write(pw,10,500,img_pr->pr_size.x,img_pr->pr_size.y,PIX_SRC,img_pr,0,0);
}

int sw_show_colors(x,y) int x,y;        /* quick and dirty */
{

        pw_write(pw,x,y,20,20,    PIX_COLOR(6)|MY_PIX,NULL,0,0);
        pw_write(pw,x+20,y,20,20,PIX_COLOR(5)|MY_PIX,NULL,0,0);
        pw_write(pw,x+40,y,20,20,PIX_COLOR(4)|MY_PIX,NULL,0,0);
        pw_write(pw,x,y-20,20,20,    PIX_COLOR(7)|MY_PIX,NULL,0,0);
        pw_write(pw,x+20,y-20,20,20,PIX_COLOR(9)|MY_PIX,NULL,0,0);
        pw_write(pw,x+40,y-20,20,20,PIX_COLOR(3)|MY_PIX,NULL,0,0);
        pw_write(pw,x,y-40,20,20,    PIX_COLOR(8)|MY_PIX,NULL,0,0);
        pw_write(pw,x+20,y-40,20,20,PIX_COLOR(1)|MY_PIX,NULL,0,0);
        pw_write(pw,x+40,y-40,20,20,PIX_COLOR(2)|MY_PIX,NULL,0,0);
}
```

```
int sw_debug()  /*retrieves messages from the nodes*/
{
        int dest, *ptoi;
        int src, type = 123;
        int n;

        T_command_to_nodes(DEBUG);

        for(n=0;n<NODES;n++){
                src = DONTCARE;
                exread(fromnodes, COMM_SIZE, &src, &type);
                ptoi=(int *)fromnodes;
        }
        sw_msg("debugged...\n");

}
int sw_host_back_image()
{
        host_back_image();
        sw_msg("image received...\n");
}

int read_commandfile()          /* interprets commands from file.com    */
#define GET(type,variable) {fgets(cmd,MAX_STRLEN, comf);sscanf(cmd,"%type ",&mgpar.varia
ble);}
{
        FILE *comf;
        char cmd[MAX_STRLEN];
        int lay;

        if((comf = fopen(mgpar.cmd_file,"r"))==NULL){
                ERR(100);
                return 0;
        }

        /*MSG*/ sw_msg("Reading commands from file.com\n");
        while(fgets(cmd,MAX_STRLEN,comf) != NULL){
                switch(cmd[0]){
                case 'l':
                        sscanf(&cmd[2],"%s",mgpar.z_file);
                        load_z_values();
                        break;
                case 'p':
                        sscanf(&cmd[2],"%s",mgpar.pr_file);
                        /*change dimensions appropriately
                         dump_image(?,?,?,?,mg_par.pr_file);*/
                        break;
                case 'c':
                        GET(d,lp_method);
                        GET(d,down_method);
                        GET(d,nlay);
                        GET(d,naa);
                        GET(d,na);
                        GET(d,nb);
                        GET(d,nc);
                        GET(d,ran);
                        GET(d,history);
                        GET(d,film);
                        GET(d,zoom);
                        GET(f,mul);
                        GET(f,beta);
                        GET(f,noise);
                        GET(f,dh);
                        GET(f,al);
                        GET(f,a2);
                        GET(f,a3);
                        GET(f,a4);
                        GET(f,ai);
                        GET(f,ru);
                        GET(f,rd);
                        break;
                case 'z':
                        if(mgpar.ld) fmg(SURFACE);
                        break;
                case 'd':
                        sscanf(&cmd[2],"%s",mgpar.z_out_file);
                        if(cmd[1]=='s')dump_data(LAY-1);
                        else if(cmd[1]=='c')for(lay=LAY-mgpar.nlay;lay<LAY;lay++)dump_dat
a(lay);
                        break;
                case 'e':
                        sscanf(&cmd[2],"%s",mgpar.disc_out_file);
                        if(cmd[1]=='s')dump_disc(LAY-1);
                        else if(cmd[1]=='c')for(lay=LAY-mgpar.nlay;lay<LAY;lay++)dump_dis
c(lay);
                        break;
                case 'x':
                        fclose(comf);
                        exit(1);
                        break;
                default:
                        break;
                }
        }
        fclose(comf);
        /*MSG*/ sw_msg("Back to sunview user interface! \n");
}
int sw_history(str) char *str;
#define PUT(type,variable) fprintf(fp,"\n%type  variable",mgpar.variable)
{
        FILE *fp;
        short save;
        static int serialn=0;
        fp=fopen(mgpar.history_file,"a");       /*append to save previous history*/
        switch(str[0]){
        case 'z':
                fprintf(fp,"\nc   (mgpar values)");
                PUT(d,lp_method);
                PUT(d,down_method);
                PUT(d,nlay);
                PUT(d,naa);
                PUT(d,na);
                PUT(d,nb);
                PUT(d,nc);
                PUT(d,ran);
                fprintf(fp,"\n%d  history",0);/* PUT(d,history);*/
                PUT(d,film);
                PUT(d,zoom);
                PUT(f,mul);
                PUT(f,beta);
                PUT(f,noise);
                PUT(f,dh);
                PUT(f,a1);
                PUT(f,a2);
                PUT(f,a3);
                PUT(f,a4);
                PUT(f,ai);
                PUT(f,ru);
                PUT(f,rd);
                fprintf(fp,"\nz   (multigrid surface rec.)");
                break;
        case 'l':
                fprintf(fp,"\nl %s ",mgpar.z_file);
                break;
        case 'd':
                fprintf(fp,"\nnd%c %s_%d ",str[1],mgpar.z_out_file, serialn);
                serialn++;
                break;
        case 'e':
                fprintf(fp,"\nd%c %s ",str[1],mgpar.disc_out_file);
                break;
        case 'x':
                fprintf(fp,"\nx (the end)");
                break;
        }

        fclose(fp);
}

int sw_vis_feedback(lay,mg_flag,str) int lay,mg_flag;
char *str;
/*visual feedback : quick&dirty*/
{
        int si,sj,i,j,o,pix,nr,nl;
        DATA_TYPE  *ptodt;
        int dt,mul;
        int maxi,maxj;

        si= (mgpar.zoom)? PIX : (PIX*((1<<lay)+ 2*lay));
        pix= (mgpar.zoom)? (2*STEP(lay)*PIXX) : PIX;    /* 2x for 65X65 images*/
        nl=NEU_IN_LAY(lay);
        nr=NEU_IN_ROW(lay);

        pw_text(pw,10,LINE(1),PIX_SRC,NULL,str);

        ptodt = (DATA_TYPE *)z;
        sj=20;
        mul=(int)((mgpar.mul*255.0)/mgpar.max_h);

        maxi=si+nr*pix;
        maxj=sj+nr*pix;
        pw_batch_on(pw);
        pw_lock(pw,&rect);
        for(j=sj;j<maxj;j += pix){
                for(i=si;i<maxi;i += pix){
                        dt= (int)((*ptodt++) *mul);
                        if(dt<11) dt= 11;
                        else if(dt>253)dt= 253;
                        pw_write(pw,i,j,pix,pix,PIX_COLOR((BYTE)dt)|MY_PIX,NULL,0,0);
                }
        }
        pw_reset(pw);
        pw_batch_off(pw);
}
```

PARALLEL SURFACE RECONSTRUCTION (node program)

nmg.c

```
/*****************************************************************/
/*nmg.c:        program for multigrid surface reconstruction        */
/*              node program (restricted version)                   */
/*              written by Roberto Battiti, Feb 1988                */
/*              uses Express communication environment (Parasoft)   */
/*****************************************************************/

#include "mg.h"

struct nodenv env;

DATA_TYPE node_z[NODE_ID][NODE_ID];     /*33*33*/
DATA_TYPE *ptodt;
char *ptoc;
int *ptoi;
int coord[2];

MGPAR mgpar = INIT_MGPAR;

DATA_TYPE lp_table[LP_TABLE_SIZE];

NEURON neuron[NODE_NEU];          /* memory containing all neurons        */
NEURON* node_layer[LAY];          /* entrance point in each layer for OWNED + BORDER*/
NEURON* layer[LAY];               /* entrance point in each layerfor OWNED pixels*/
NEURON initial_neu = INITIAL_NEU;

NEURON lp[NODE_LP];
NEURON* lp_lay[LAY];              /*entrance point for OWNED discontinuities*/
NEURON* node_lp_lay[LAY];         /*...          for OWNED + BORDER        */

#define VERT 1
#define HORIZ 0

int nnode,snode,wnode,enode;    /*north,south,west,east*/
char icomm[COMM_SIZE];  /*input communication array*/
char ocomm[COMM_SIZE];  /*output communication array*/

main()
{
        int n,i,j,k;
        int bytes_for_each;
        int type = 123, dest;
        int nprocs[2];
        int perbc[2];
        char *ibuf, *obuf;

#if PROFILE
        cprof_on();
#endif

        /* Read system parameters, number of nodes etc...... */
        exparam(&env);

        exgridsplit(env.nprocs,2,nprocs);
        exgridinit(2, nprocs);

        perbc[0]=perbc[1]=0;
        exgridbc(perbc);
        exgridcoord(env.procnum,coord);
        /*NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN*/
        for(n=0;n<env.nprocs;n++){
                if(env.procnum == n) {
                        ptoi=(int *)ocomm;
                        *ptoi++ = coord[0];
                        *ptoi++ = coord[1];
                        dest = HOST;
                        exwrite(ocomm, COMM_SIZE, &dest, &type);
                }
        }
        /*NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN*/

        nnode = exgridnode(env.procnum, VERT, +1);
        snode = exgridnode(env.procnum, VERT, -1);
        wnode = exgridnode(env.procnum, HORIZ, -1);
        enode = exgridnode(env.procnum, HORIZ, +1);

        node_sew_pyramid();

        server();       /*server for host: wait and interpret commands*/
#if PROFILE
        cprof_off();
        cprofelt("cprof.out");
#endif

        exit(0);
}

int node_sew_pyramid()  /* (xy)=(00) for the top-left OWNED point*/
/* lay is the layer in the COMPLETE image:lay=6=>65*65*/
/* hence highest lay is 1                               */
/* NODE_NEU_IN_LAY(lay) are pixels OWNED + BORDER       */
#define COARSEST 1
#define FINEST (LAY-1)
{
        int i,j,l,o,x,y,n,nn,tnpo;
        NEURON *p,*this,*pp,*pds,*ppds;
        DATA_TYPE *ptodt;

        /*****************************initialize pyramid structure of neurons*/
        j=0;
        for(l=COARSEST;l<LAY;l++){
                node_layer[l]= &(neuron[j]);
                j += NODE_NEU_IN_LAY(l);
        }
        i= NODE_NEU;
        while(i--) neuron[i] = initial_neu;

        for(l=COARSEST;l<LAY;l++){
                p=node_layer[l];
                for(o=0;o<NODE_NEU_IN_LAY(l);o++){
                        this = p+o;
                        x= NODE_PHYSX(o,l);
                        y= NODE_PHYSY(o,l);/*for macros */

                        if(o !=(NODE_NEU_IN_LAY(l)-1)){
                                gett(next)=(this+1);
                        } /*next          */
                        /*next will be MODIFIED at the END*/
                        if(l !=(LAY-1)){
                                pp=node_layer[l+1];                     /*down-up*/
                                gett(dQ) = (pp+NODE_OFFSET(x,y,l+1));
                                (pp+NODE_OFFSET(x,y,l+1))->u0 = this;
                        }
                        if(x != (NODE_II+1)){
                                gett(e) = (this+1);                     /*east-west*/
                                (this+1)->w = this;
                        }
                        if(y != (NODE_II+1)){
```

```
                        gett(s) = (this+NODE_NEU_IN_ROW(l));     /*north-south*/
                        (this+NODE_NEU_IN_ROW(l))->n = this;
                }
        }
}

for(l=COARSEST;l<LAY;l++){
        for(this=node_layer[l];this!=VOID;this=gett(next)){

                if(gett(n)) {
                        if(gett(n)->e) gett(ne)= gett(n)->e; /*ne,nw,se,sw. */
                        if(gett(n)->w) gett(nw)= gett(n)->w;
                }
                if(gett(s)){
                        if(gett(s)->e) gett(se)= gett(s)->e;
                        if((gett(s)->w)) gett(sw) = gett(s)->w;
                }

                if((gett(ne))&&(gett(sw)))gett(n_type) =IN_NEU;
        }
}

for(l=COARSEST;l<LAY;l++){      /*pixels = borders in the COMPLETE image*/
        this=node_layer[l]+NODE_OFFSET(((coord[0]==0)? 0: NODE_II),0,l);
        i=NODE_ID;
        while(i--) {
                gett(n_type)=BO_NEU;
                this=gett(s);
        }

        this=node_layer[l]+NODE_OFFSET(0,((coord[1]==0)?  NODE_II: 0),l); .
        i=NODE_ID;
        while(i--) {
                gett(n_type)=BO_NEU;
                this=gett(e);
        }
}

/**************************initialize pyramid structure of connections*/
/*strategy:   first connect disc. to neurons.... */
j=0;
for(l=COARSEST;l<LAY;l++){
        node_lp_lay[l]= &(lp[j]);
        j += NODE_LP_IN_LAY(l);
}
i= NODE_LP;
while(i--) lp[i] = initial_neu;

for(l=COARSEST;l<LAY;l++){
        p=node_layer[l];
        pds=node_lp_lay[l];
        nn=NODE_NEU_IN_ROW(l);
        n=nn-1;
        tnpo=2*n +1;
        for(o=0;o<NODE_NEU_IN_LAY(l);o++){
                this = p+o;  /*"this" points to "real" neurons*/
                if(gett(e)){
                        ppds = pds +((o/nn)*tnpo +(o%nn));
                        gett(oe)=ppds;
                        ppds->ow=this;
                        ppds->oe=this+1;
                        (this+1)->ow=ppds;
                }
                if(gett(s)){
```

```
                                        ppds = pds +(n+ (o/nn)*tnpo +(o%nn));
                                        gett(os)=ppds;
                                        ppds->on=this;
                                        ppds->os=this+nn;
                                        (this+nn)->on=ppds;
                                }
                        }
        }
        /*... then connect discontinuities among themselves*/
        for(l=COARSEST;l<LAY;l++){
                pds=node_lp_lay[l];
                nn=NODE_NEU_IN_ROW(l);
                n=nn-1;
                tnpo=2*n+1;
                for(o=0;o<NODE_LP_IN_LAY(l);o++){
                        this = pds+o;   /* points to discontinuity*/
                        if(o !=(NODE_LP_IN_LAY(l)-1)){
                                gett(next)=(this+1);
                        }
                        if(gett(oe)){
                                if(getoe(e)){
                                        gett(e)=this+1;
                                        (this+1)->w = this;
                                }
                                if(getoe(s)){
                                        gett(s)=this+tnpo;
                                        (this+tnpo)->n=this;
                                        gett(se)=this+nn;
                                        (this+nn)->nw=this;
                                        gett(sw)=this+n;
                                        (this+n)->ne=this;
                                }
                                if(getoe(n)){
                                        gett(n)=this-tnpo;
                                        (this-tnpo)->s=this;
                                        gett(ne)=this-n;
                                        (this-n)->sw=this;
                                        gett(nw)=this-nn;
                                        (this-nn)->se=this;
                                }
                        }
                        else {
                                if(getos(e)){
                                        gett(e)=this+1;
                                        (this+1)->w=this;
                                }
                                if(getos(s)){
                                        gett(s)=this+tnpo;
                                        (this+tnpo)->n = this;
                                }
                        }
                        if((gett(n))&&(gett(s))&&(gett(e))&&(gett(w)))gett(n_type) =IN_NE
U;
                }
        }
        /***********connect discontinuities in different layers********************/
        for(l=COARSEST;l<(LAY-1);l++){
                for(this=node_lp_lay[l];this!=VOID;this=gett(next)){
                        /*vert disc*/
                        if(gett(oe)){
                                p=getoe(d0)->w; /*z neuron below*/
                                gett(d0)=pds=p->oe;
                                pds->u0=this;
                                gett(d1)=pds=p->ow;
                        pds->u0=this;
                        if(pp=p->n){
                                gett(d2)=pds=pp->oe;
                                if(pds->u0)pds->u1=this;
                                else pds->u0=this;
                                gett(d3)=pds=pp->ow;
                                if(pds->u0)pds->u1=this;
                                else pds->u0=this;
                        }
                        if(pp=p->s){
                                gett(d4)=pds=pp->oe;
                                if(pds->u0)pds->u1=this;
                                else pds->u0=this;
                                gett(d5)=pds=pp->ow;
                                if(pds->u0)pds->u1=this;
                                else pds->u0=this;
                        }
                }
                /*hori disc*/
                else {
                        p=getos(d0)->n; /*z neuron below*/
                        gett(d0)=pds=p->os;
                        pds->u0=this;
                        gett(d1)=pds=p->on;
                        pds->u0=this;
                        if(pp=p->e){
                                gett(d2)=pds=pp->os;
                                if(pds->u0)pds->u1=this;
                                else pds->u0=this;
                                gett(d3)=pds=pp->on;
                                if(pds->u0)pds->u1=this;
                                else pds->u0=this;
                        }
                        if(pp=p->w){
                                gett(d4)=pds=pp->os;
                                if(pds->u0)pds->u1=this;
                                else pds->u0=this;
                                gett(d5)=pds=pp->on;
                                if(pds->u0)pds->u1=this;
                                else pds->u0=this;
                        }
                }
        }
}

/*next is MODIFIED: only OWNED pixels are connected, starting from layer[1]*/
/*p point to the west, pp to the east and both go from north to south*/

/*next for value neurons*/
for(l=COARSEST;l<LAY;l++){
        this=node_layer[l];
        layer[l]=gett(se);
        p=layer[l];
        pp=layer[l]+ NEU_IN_ROW(l-1) -1;
        i=NEU_IN_ROW(l-1) -1;
        while(i--){
                pp->next = p->s;
                p=p->s;
                pp=pp->s;
        }
        pp->next= VOID;
}
```

```
        /*next for discontinuity neurons*/
        for(l=COARSEST;l<LAY;l++){
                this=node_lp_lay[l];
                lp_lay[l]= gett(se)->se;
                p=lp_lay[l];
                pp=lp_lay[l]+ NEU_IN_ROW(l-1) -2;
                i=NEU_IN_ROW(l-1) -1;
                while(i--){
                        pp->next = p->sw;
                        p=p->sw;
                        pp=pp->se;
                        pp->next = p->se;
                        p=p->se;
                        pp=pp->sw;
                }
                pp->next= VOID;
        }

        /**************************adjust pointers to the initial data***************/
        ptodt = (DATA_TYPE *)node_z;
        for(this=layer[FINEST];this!=VOID;this=gett(next)){
                gett(znoisy)= ptodt++;
        }
        for(l=FINEST-1;l>=COARSEST;l--){
                for(this=layer[l];this!=VOID;this=gett(next)){
                        gett(znoisy) = getd0(znoisy);
                }
        }

}

int node_load_z_values() /*get patch of image from host*/
{
        int i,j,n;
        int dest,type=123;

        exread((char *)node_z,COMM_SIZE,(char *)0,(char *)0);
}

int node_back_z_values()
{
        int n,i;
        int dest,type=123;
        DATA_TYPE *ptodt,*ptosource;
        NEURON *this;

        ptodt=(DATA_TYPE *)ocomm;
        for(this=layer[LAY-1];this != VOID;this=gett(next)){
                *ptodt++ = gett(z);
        }

        for(n=0;n<env.nprocs;n++){

                if(env.procnum == n) {
                        dest = HOST;
                        exwrite(ocomm, COMM_SIZE, &dest, &type);
                }
        }
}
int node_debug()
{
        int n;
        int type = 123, dest;
        for(n=0;n<env.nprocs;n++){
                if(env.procnum == n) {
                        ptoi=(int *)ocomm;
                        *ptoi++ = 13;
                        *ptoi++ = 17;
                        dest = HOST;
                        exwrite(ocomm, COMM_SIZE, &dest, &type);
                }
        }

}

int server()
{
        int command;
        int type=123,i;

        for(;;){
                exbroadcast(&command, HOST, 4, ALLNODES, (int *)0, &type);
                switch(command){
                case QUIT:
                        return(1);
                        break;
                case FMG:
                        node_fmg();
                        break;
                case LOAD_IMAGE:
                        node_load_z_values();
                        break;
                case BACK_IMAGE:
                        node_back_z_values();
                        break;
                case DEBUG:
                        node_debug();
                        break;
                default:
                        break;
                }
        }

}

f_add(i,j, size)
int *i, *j;
int size;
{
        *i += *j;
        return 1;
}

/*****************************************************************************************/
int mg_init(mg_flag) int mg_flag;
{
        int i;

        switch(mg_flag){
        case SURFACE :
                i=NODE_NEU;
                while(i--){
                        neuron[i].z= *(neuron[i].znoisy);
                }
                /*ACH: znoisy may be VOID*/
                i=NODE_LP;
                while(i--){
                        lp[i].bt=0;
                    ;
                    break;
        }
        fill_lp_table(mgpar.lp_method);
}

int step(lay,mg_flag) int lay,mg_flag;
{
        exchange_borders(lay);

        lp_update(lay,mg_flag);
        mg_relax(lay,mg_flag);
}

int exchange_borders(lay) int lay;
#define LDZ   {*ptodt++ = gett(z);}      /*load z*/
#define UDZ   {gett(z) = *ptodt++ ;}     /*unload z*/
#define LDB   {*ptob++ = gett(bt);}
#define UDB   {gett(bt) = *ptob++ ;}
{
        BYTE *ptob;
        DATA_TYPE *ptodt;
        NEURON *this,*p;
        int num_bytes,i,nnr,type=123;

        nnr=NODE_NEU_IN_ROW(lay);
        num_bytes=(nnr+2)*sizeof(DATA_TYPE)+(2*nnr+1)*sizeof(BYTE);

        /*n->s*/
        ptodt=(DATA_TYPE *)ocomm;
        this=node_layer[lay]+(nnr-2)*nnr;/*starting point for loading comm. buffer*/

        LDZ;
        this=gett(n);
        LDZ;
        i=nnr-1;
        while(i--){
                this=gett(e);
                LDZ;
        }
        this=gett(s);
        LDZ;
        this=gett(ow);
        ptob=(BYTE *)ptodt;
        LDB;
        this=gett(ne);
        LDB;
        i=nnr-1;
        while(i--){
                this=gett(nw);
                LDB;
                this=gett(sw);
                LDB;
        }
        this=gett(se);
        LDB;

        exchange((char *)icomm, num_bytes, &nnode,  &type,
            (char *)ocomm,num_bytes,&snode,&type);

        ptodt=(DATA_TYPE *)icomm;
        this=node_layer[lay]+nnr; /*starting point for unloading*/
```

```
UDZ;
this=gett(n);
UDZ;
i=nnr-1;
while(i--){
        this=gett(e);
        UDZ;
}
this=gett(s);
UDZ;
this=gett(ow);
ptob=(BYTE *)ptodt;
UDB;
this=gett(ne);
UDB;
i=nnr-1;
while(i--){
        this=gett(nw);
        UDB;
        this=gett(sw);
        UDB;
}
this=gett(se);
UDB;

/*s->n*/
ptodt=(DATA_TYPE *)ocomm;
this=node_layer[lay]+nnr;

LDZ;
this=gett(s);
LDZ;
i=nnr-1;
while(i--){
        this=gett(e);
        LDZ;
}
this=gett(n);
LDZ;
this=gett(ow);
ptob=(BYTE *)ptodt;
LDB;
this=gett(se);
LDB;
i=nnr-1;
while(i--){
        this=gett(sw);
        LDB;
        this=gett(nw);
        LDB;
}
this=gett(ne);
LDB;

exchange((char *)icomm,num_bytes,&snode,&type,
     (char *)ocomm,num_bytes,&nnode  ,&type);

ptodt=(DATA_TYPE *)icomm;
this=node_layer[lay]+(nnr-2)*nnr;

UDZ;
this=gett(s);
UDZ;
i=nnr-1;
```

```
while(i--){
        this=gett(e);
        UDZ;
}
this=gett(n);
UDZ;
this=gett(ow);
ptob=(BYTE *)ptodt;
UDB;
this=gett(se);
UDB;
i=nnr-1;
while(i--){
        this=gett(sw);
        UDB;
        this=gett(nw);
        UDB;
}
this=gett(ne);
UDB;

/*e->w*/
ptodt=(DATA_TYPE *)ocomm;
this=node_layer[lay]+1;

LDZ;
this=gett(e);
LDZ;
i=nnr-1;
while(i--){
        this=gett(s);
        LDZ;
}
this=gett(w);
LDZ;
this=gett(on);
ptob=(BYTE *)ptodt;
LDB;
this=gett(se);
LDB;
i=nnr-1;
while(i--){
        this=gett(ne);
        LDB;
        this=gett(nw);
        LDB;
}
this=gett(sw);
LDB;

exchange((char *)icomm, num_bytes, &enode,  &type,
    (char *)ocomm,num_bytes,&wnode,&type);

ptodt=(DATA_TYPE *)icomm;
this=node_layer[lay]+nnr-2;

UDZ;
this=gett(e);
UDZ;
i=nnr-1;
while(i--){
        this=gett(s);
        UDZ;
}
```

```
this=gett(w);
UDZ;
this=gett(on);
ptob=(BYTE *)ptodt;
UDB;
this=gett(se);
UDB;
i=nnr-1;
while(i--){
        this=gett(ne);
        UDB;
        this=gett(nw);
        UDB;
}
this=gett(sw);
UDB;


/*w->e*/
ptodt=(DATA_TYPE *)ocomm;
this=node_layer[lay]+nnr-2;

LDZ;
this=gett(w);
LDZ;
i=nnr-1;
while(i--){
        this=gett(s);
        LDZ;
}
this=gett(e);
LDZ;
this=gett(on);
ptob=(BYTE *)ptodt;
LDB;
this=gett(sw);
LDB;
i=nnr-1;
while(i--){
        this=gett(nw);
        LDB;
        this=gett(ne);
        LDB;
}
this=gett(se);
LDB;

exchange((char *)icomm, num_bytes, &wnode,  &type,
    (char *)ocomm,num_bytes,&enode,&type);

ptodt=(DATA_TYPE *)icomm;
this=node_layer[lay]+1;

UDZ;
this=gett(w);
UDZ;
i=nnr-1;
while(i--){
        this=gett(s);
        UDZ;
}
this=gett(e);
UDZ;
this=gett(on);
```

```
        ptob=(BYTE *)ptodt;
        UDB;
        this=gett(sw);
        UDB;
        i=nnr-1;
        while(i--){
                this=gett(nw);
                UDB;
                this=gett(ne);
                UDB;
        }
        this=gett(se);
        UDB;
}

int node_fmg()                  /* basic multigrid routine */
{
        short i,j,lay,mg_flag;
        int type=123;

        /*get mgpar from host*/
        exbroadcast((char *)(&mgpar), HOST, sizeof(MGPAR), ALLNODES, (int *)0, &type);

        mg_flag=SURFACE;
        mg_init(mg_flag);

        lay= (LAY-mgpar.nlay);  /*coarsest layer                        */


        i=mgpar.naa;
        while(i--) {
                step(lay,mg_flag);
        }
        lp_update(lay,mg_flag);

        for(lay +=1;lay<LAY;lay++){
                mg_down(lay-1,mg_flag);
                mg(lay,mg_flag);
        }
}

int mg(lay,mg_flag) int lay,mg_flag;
{
        int i;


        if(lay==(LAY-mgpar.nlay)){
                step(lay,mg_flag);
        }/*coarsest layer*/
        else{
                i=mgpar.na;
                while(i--){
                        step(lay,mg_flag);
                }

                i=mgpar.nb;
                if(i!=0){
                        mg_up(lay);
                        while(i--)mg(lay-1,mg_flag);
                        mg_down(lay-1,mg_flag);
                }

                i=mgpar.nc;
                while(i--){
                        step(lay,mg_flag);
                }
        }

        lp_update(lay,mg_flag);
}

#define DOWN_STAR(val) pd = gett(d0); dif = (gett(val) - pd->val);
\
                pd->val += dif;
\
                dif *= .5;
\
                pd->n->val += dif; pd->s->val += dif; pd->e->val += dif; pd->w->val += d
if;\
                dif *= .5;
\
                pd->ne->val += dif; pd->nw->val += dif; pd->se->val += dif; pd->sw->val
+= dif;
#define DOWN_SIMPLE(val) getd0(val) = gett(val);
int mg_down(lay,mg_flag) int lay,mg_flag;  /* from lay (coarse) to lay+1 (fine) grid*/
{
        NEURON *this,*pd;
        DATA_TYPE dif;

        switch(mg_flag){
        case SURFACE:
                switch(mgpar.down_method){
                case DOWN_METHOD_STAR:
                        for(this=layer[lay];this!=VOID;this=gett(next)){
                                if(gett(n_type)==IN_NEU){
                                        DOWN_STAR(z);
                                }
                        }
                        break;
                case DOWN_METHOD_SIMPLE:
                        for(this=layer[lay];this!=VOID;this=gett(next)){
                                if(gett(n_type)==IN_NEU){
                                        DOWN_SIMPLE(z);
                                }
                        }
                        break;
                case DOWN_METHOD_STAR_DISC:
                        break;
                }
                break;
        }
}

int mg_up(lay,mg_flag) int lay,mg_flag; /* from lay (fine) to lay-1 (coarse)  grid*/
{
        NEURON *this;

        switch(mg_flag){
        case SURFACE:
                for(this=layer[lay-1];this!=VOID;this=gett(next)){
                        if(gett(n_type)==IN_NEU){
                                gett(z) = getd0(z);                     /* injection */
                        }
                }
                break;
        }
}
```

```
int mg_relax(lay,mg_flag) int lay,mg_flag;      /*Gauss-Seidel relaxation
{
        NEURON *this;
        DATA_TYPE i,step,hh,zb,b,b_hh,one_ov_four_b_hh ;
        char str[MAX_STRLEN];

        step = (DATA_TYPE) STEP(lay);
        hh= step*step;
        b=mgpar.beta;
        b_hh = b*hh;

        switch(mg_flag){
        case SURFACE:
                for(this=layer[lay];this!=VOID;this=gett(next)){
                        if(gett(n_type)==IN_NEU){

                                zb=0.0;
                                i=0;
                                if(geton(bt)!=1){
                                        i++;
                                        zb += getn(z);
                                }
                                if(getos(bt)!=1){
                                        i++;
                                        zb += gets(z);
                                }
                                if(getoe(bt)!=1){
                                        i++;
                                        zb += gete(z);
                                }
                                if(getow(bt)!=1){
                                        i++;
                                        zb += getw(z);
                                }
                                one_ov_four_b_hh = 1.0/((DATA_TYPE)i+ b_hh);
                                gett(z) = (zb + b_hh * (*(gett(znoisy)))  )*one_ov_four_b
_hh;
                        }
                }
                break;
        }
        sprintf(&(str[0]),"***relaxed layer %d        ",lay);
}

#define FIND_UP_DOWN_DISCOUNT(discount)         \
                {if(gett(ul))discount=((getul(bt))? rus: 1.0)*((getu0(bt))? rus: 1.0);  \
                else discount=((getu0(bt))? ru: 1.0);                                   \
                if(getd0(bt))discount *= rd;                 \
                if(getd1(bt))discount *= rd;                 \
                if(getd2(bt))discount *= rds;                \
                if(getd3(bt))discount *= rds;                \
                if(getd4(bt))discount *= rds;                \
                if(getd5(bt))discount *= rds;}
#define FIND_UP_DISCOUNT(discount)              \
                {if(gett(ul))discount=((getul(bt))? rus: 1.0)*((getu0(bt))? rus: 1.0);  \
                else discount=((getu0(bt))? ru: 1.0);}
int lp_update(lay,mg_flag) int lay,mg_flag;
{
        NEURON *this;
        int index,i;
        DATA_TYPE de,oneovhh;
        DATA_TYPE ru,rd,rus,rds,discount;           /*reduction up or down  */

        ru = mgpar.ru;
        rd = mgpar.rd;
        rus= sqrt(ru);
        rds= sqrt(rd);
        oneovhh= 1.0/(DATA_TYPE)(STEP(lay)*STEP(lay));

        if(lay<(LAY-1)){
                for(this=lp_lay[lay];this!=VOID;this=gett(next)){
                        if(gett(n_type)==IN_NEU){
                                if(gett(on))/*horizontal*/{
                                        H_INDEX(index);
                                        de= (geton(z) - getos(z));
                                        de *=de;
                                        de *=oneovhh;
                                        FIND_UP_DOWN_DISCOUNT(discount);
                                        de=lp_table[index]*discount -de;
                                }
                                else /*vertical*/{
                                        V_INDEX(index);
                                        de= (getoe(z) - getow(z));
                                        de *=de;
                                        de *=oneovhh;
                                        FIND_UP_DOWN_DISCOUNT(discount);
                                        de=lp_table[index]*discount -de;
                                }
                                if(de<0.0)gett(bt)=1;
                                else gett(bt)=0;
                        }
                }
        }
        else {  /*finest layer*/
                for(this=lp_lay[lay];this!=VOID;this=gett(next)){
                        if(gett(n_type)==IN_NEU){
                                if(gett(on))/*horizontal*/{
                                        H_INDEX(index);
                                        de= (geton(z) - getos(z));
                                        de *=de;
                                        de *=oneovhh;
                                        FIND_UP_DISCOUNT(discount);
                                        de=lp_table[index]*discount -de;
                                }
                                else /*vertical*/{
                                        V_INDEX(index);
                                        de= (getoe(z) - getow(z));
                                        de *=de;
                                        de *=oneovhh;
                                        FIND_UP_DISCOUNT(discount);
                                        de=lp_table[index]*discount -de;
                                }
                                if(de<0.0)gett(bt)=1;
                                else gett(bt)=0;
                        }
                }
        }
}

/*###############################################################################*/
int fill_lp_table(lp_method) int lp_method;
{
        DATA_TYPE price,price0,price1,price2,price3,price4,inhi;
        DATA_TYPE cl,cp,cw;
        short i,n,ne,e,se,s,sw,w,nw;
        short nlp,nup,ndown;

        price0=(mgpar.dh)*(mgpar.dh);
        inhi = mgpar.ai;
        price1=price0*(mgpar.a1);
        price2=price0*(mgpar.a2);
        price3=price0*(mgpar.a3);
        price4=price0*(mgpar.a4);
        c1=price2;
        cw=(price1-price2);
        cp=price2*(inhi-1.0);
        if((cw<0.0)||(cp<0.0));

        for(i=0;i<LP_TABLE_SIZE;i++){
                n=(i%2);
                ne=((i>>1)%2);
                e=((i>>2)%2);
                se=((i>>3)%2);
                s=((i>>4)%2);
                sw=((i>>5)%2);
                w=((i>>6)%2);
                nw=((i>>7)%2);
                switch(lp_method){
                case LP_METHOD_TABLE:
                        nlp=n+ne+se+s+sw+nw;    /*parallel lp excluded*/
                        switch(nlp){
                        case 0 :
                                price=price0;
                                break;
                        case 1 :
                                price=price1;
                                break;
                        case 2 :
                                price=price2;
                                break;
                        case 3 :
                                price=price3;
                                break;
                        case 4 :
                                price=price4;
                                break;
                        default :
                                price=INFINITY;
                                break;
                        }
                        if(w)price *= inhi;
                        if(e)price *= inhi;     /*price increase if parallel lp's*/
                        break;
                case LP_METHOD_CONST:
                        price =price0;
                        break;
                case LP_METHOD_WOJTEK:
                        nup= nw+n+ne;
                        ndown=se+s+se;
                        if((nup==0)&&(ndown==0)) price=price0;
                        else if((nup%2)==(ndown%2))price = ((nup%2)? (-price1): price1) +
price0;
                        else price=0.0;
                        break;
                case LP_METHOD_CHRISTOF:
                        nup= nw+n+ne;
                        ndown=se+s+se;
                        price= c1 +cp*(e+w) +cw*((1-nup)*(1-nup)+(1-ndown)*(1-ndown));
                        break;
                }
                lp_table[i]=price;
```