1-1-1992

Image processing workstation software development and feature size measurement methods for NDE X-ray images

Richard Ali Brown
Iowa State University

Image processing workstation software development and feature size measurement methods for NDE X-ray images

by

Richard Ali Brown

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Electrical Engineering

Iowa State University
Ames, Iowa

1992

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The author acknowledges the assistance of his advisor, Dr. John Basart, in helping define the scope of the work contained herein and in the preparation of the thesis. Thanks are extended to the substitute committee members, Dr. Satish Udpa, and Dr. Joseph Gray, for reading an unusually long thesis in a short amount of time. Portions of this work were supported in part by United Stated Department of Commerce Grant ITA 87-02.

CHAPTER 1: INTRODUCTION

1.1 Overview of NDE

Nondestructive Evaluation, or NDE, is the science of detecting and characterizing flaws in engineered materials, individual components, and final assemblies of manufactured items without damaging or destroying them. NDE is becoming increasingly important to modern society for reasons discussed below, and its importance is becoming increasingly recognized by industry, governments, and the public. The field is highly interdisciplinary in nature, and has no clearly defined boundaries. Its methods range from simple visual inspection by human eyes to the use of sophisticated energy-generating and sensing devices, the data from which may be fed through signal conditioning equipment to powerful computer systems for processing by algorithms based on theories in such diverse fields as optics, physiology, electrical engineering, artificial intelligence & neural networks, statistics, and computer science.

NDE is important to modern society because of its roles in maintaining economic vitality and public safety. It has become widely discussed in recent years that the cost of servicing or replacing a defective manufactured item increases dramatically with the delay in manufacturing process between the time that defective components or materials are introduced into an assembly or sub-assembly and the time that the defects are found. When re-work is impractical or impossible on a defective finished item, significant amounts of economic, material, and labor resources are wasted by the inadequacy or

failure of components or materials representing a small fraction of the item's total cost. Manufacturing process and product quality are increasingly cited as being crucial to economic competitiveness and vitality; NDE can make important contributions to attaining this quality.

Modern society is dependent upon a wide variety of large, complicated, powerful, and potentially dangerous machinery such as airplanes, trains, cars and trucks, oil refineries, and nuclear power plants. Major malfunctions in this machinery can be catastrophic. Thus, the possibility of putting such machinery into service with potentially dangerous defects must be minimized. Destructive testing of *samples* of materials and components can tell a manufacturer something about the probability distributions of those materials' and components' properties (destructive testing of *all* materials and components would obviously result in nothing being manufactured). However, even though we may know these distributions accurately (and this is very seldom the case), we are still betting human lives on the odds that we calculate. A manufacturer of high-liability machinery must inspect as thoroughly as is practical *every single* critical component or piece of material used in every product that goes out the door; statistical outliers can not be tolerated. Also, because such machinery is inevitably affected by the enormous forces it generates and absorbs and the hostile environments in which it often serves, it is necessary to perform periodic inspections in the field; the purpose of these is to detect damage before it becomes critical and to predict remaining safe lifetime. The in-service inspection techniques must obviously be non-destructive in nature. In most cases, the design lifetime of a complex machine is highly empirical and is often "fudged", or extremely

conservatively estimated. The benefits of continued service from a machine and the cost of its replacement make it far more economical to continue with periodic inspections than to take the machine out of service simply because it is past its design lifetime. Since the destructive-testing-of-samples approach is unacceptable for high-liability machines, and because of the safety and economic benefits of in-service inspections, NDE techniques have become increasingly important for public safety.

In general, an NDE inspection system consists of an energy source, an energy sensor (which may not necessarily sense the same kind of energy produced by the energy source), signal conditioning and analysis equipment (optional, depending on the application), and a display device. This is shown schematically in Figure 1.1. Conclusions about the specimen under test are drawn from the sensor measurements and an understanding of the interaction between the source energy and the specimen. The most widely used NDE methods may be distinguished by the energy sources they employ: electromagnetic, ultrasonic, and X-ray. Other methods include, but are not limited to, fluorescent dye penetrants, nuclear magnetic resonance, holography, and thermography. Often, as is the case with x-ray radiographic methods, a two-dimensional array of data, or *image*, representing a projection of the spatial distribution of some property of the specimen under test, is produced by the sensor. This image may be used by an inspector to visualize the size and location of a flaw within the specimen, and to determine its nature and severity.

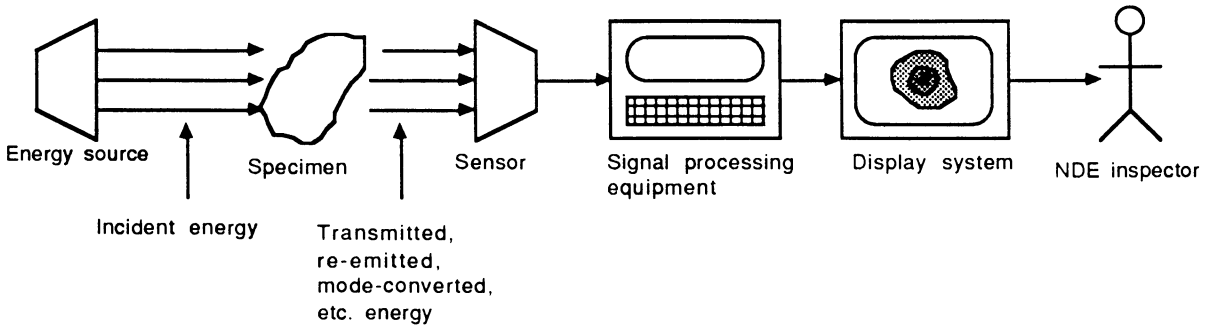


Figure 1.1. General NDE inspection system

1.2 Image Processing in NDE

Image processing is the science of manipulating two-dimensional arrays of data for purposes of representation, storage, transmission, and extraction of information from the data. Image processing may be used to great advantage in NDE applications. The image formed by a sensor will typically contain both useful and useless information. "Useful" information is directly related to specimen properties we wish to measure; all other information produced by the sensor is useless "noise" and may hamper the perception and interpretation of the information of interest. The broad and interdisciplinary nature of NDE is due to the vast variety of materials (and their unique shapes and properties) that require inspection. Image processing techniques for NDE, in turn, are driven by this variety, and are consequently quite varied and specialized. No single processing algorithm or small group of algorithms can be regarded as generally applicable to all NDE image processing needs.

We may roughly divide image processing techniques into three levels of robustness: (1) qualitative enhancement, (2) quantitative measurement and estimation, and (3) automated flaw detection, classification, and measurement. At the simplest level are techniques which qualitatively enhance an image. These algorithms often are based on physiological considerations of human visual perception, and may or may not preserve the relationship between the specimen's physical properties and the image intensity level; their purpose is mainly to make it easier for a human inspector to locate suspicious areas in the image. Examples of such algorithms are thresholding, pixel inversion, histogram equalization, and adaptive histogram equalization. More robust, quantitative techniques of measurement and estimation may be employed when consistent and objective information is to be extracted from an image. The image is manipulated with well-defined mathematical operations, usually based on theory which respects the relationship between the image intensity and the specimen properties, and the result is often a well-defined quantity that is not open to different interpretations by different inspectors. Examples of such techniques are statistical noise filtering, stereographic flaw depth reconstruction, and some of the flaw sizing techniques used in this work. At the highest level, automatic flaw detection, classification, and measurement techniques attempt to mimic the experienced eye of the NDE inspector, using information from qualitative and quantitative analyses as input to higher-level algorithms based on pattern recognition principles. These techniques tend to be the most highly specialized and are developed for very specific NDE inspection situations. The work discussed in this thesis pertains to techniques at the first two levels of robustness discussed above, qualitative enhancement

and quantitative measurement and estimation. The reader is referred to the textbooks by Pratt (1991), Gonzalez and Wintz (1987), and Jain (1989) for a general introduction to image processing techniques.

1.3 HAPPI: An Integrated NDE Image Processing Environment

HAPPI is an integrated image processing software environment developed in the Electrical and Computer Engineering Department under sponsorship of the Center for Advanced Technology Development at Iowa State University. (HAPPI is an acronym for "Here's A Program for Processing Images".) The author was part of the group which produced HAPPI. Many of the image processing routines included in HAPPI were developed by the X-ray Image Processing Group in the Electrical and Computer Engineering Department at ISU, under sponsorship of the Center for NDE (also at ISU).

The functions provided in HAPPI include a large repertoire of image processing, measurement, and analysis routines, image data acquisition and image data management functions, macro-related functions, and various operating system access functions. The user interface is based on a graphical pointing device, in this case a mouse, and a set of graphical windows, or areas on the host computer's display screen which serve as I/O channels between the user and the program.

The majority of this thesis deals with HAPPI's design. The finished package is evaluated with respect to several criteria and in the context of other commercially available image processing software. The contemporary image processing software development environment and its influence on

how HAPPI was written, as well as future trends in this development environment, is addressed. Also, the more salient design features of the software package are discussed, and in this context, a step-by-step procedure for compiling and linking new functions to HAPPI, with access to the functions through the HAPPI user interface, is given. The remainder of the thesis discusses the effects of HAPPI's processing routines on image feature size.

1.4 Effects of Processing Routines on Feature Size

It is for the designer to determine the size and types of flaws that can be safely tolerated by his/her design. Depending on a manufactured item's application and operating environment, a particular flaw may be perfectly harmless or may invite catastrophe. The designer considers these factors and his/her knowledge of the materials and components used in his design to arrive at an educated estimate of what constitutes a significant flaw. It is for NDE engineers and inspectors to provide measurements of a flaw's properties. The accuracy of these measurements must be sufficient for a rational course of action to be taken with respect to a suspected flawed component.

In reviewing the NDE literature, it appears that there is not a standard set of methods for determining flaw sizes in x-ray radiographs, especially with digital imaging techniques. (The literature searched included most of the last decade or so of: *NDT International, Journal of NDE, International Advances in NDT, Review of Progress in QNDE, British Journal of NDT, Soviet Journal of NDT, Research Techniques in NDT, and Materials Evaluation.*) However, some of the

literature develops pertinent theory. A number of authors discuss the effects of various radiographic system parameters on attainable flaw size resolution and on the theoretical film density profile for certain simple flaw geometries (Fishman, *et al.*, 1981), (Notea, 1983), (Segal and Trichter, 1988). Also, simple yet practical methods have been proposed for measuring flaw through-thickness dimensions (i.e., the flaw dimension perpendicular to the plane of the image) (Halmshaw, 1979). The theory and methods are not without their limitations, and have been developed using assumptions of rather ideal conditions. It is under the non-ideal image conditions of high noise, low contrast, and non-uniform background so often encountered in NDE radiography that image processing techniques are used to try to bring out information about a suspected flaw.

Where image processing is used to improve an image, the processing may produce artifacts and/or distort the size of a flaw. In many images, the flaw information is mixed in with the noise in a way that does not allow complete separation of the two. Also, many robust signal processing methods that can produce very impressive results are heavily dependent upon the accuracy of estimates of signal and noise properties. When the property estimates are not accurate, results can be worse than those produced by less robust methods.

In this work, we apply a simple set of size metrics to raw and processed images of simulated flaws, with the goal of assessing the effects of several of HAPPI's image processing routines on the measured size of image features. The metrics used represent reasonable, but not necessarily optimal, ways of

measuring flaw size. The effects of noise, flaw shape, and contrast are included in the study.

CHAPTER 2: CONTEMPORARY IMAGE PROCESSING TRENDS AND HAPPI's DESIGN

2.1 Introduction

In this chapter, we put HAPPI in perspective by discussing the contemporary image processing software development environment in which it was written, as well as projected future trends in this environment. We touch on the need for and the state of software standards, and on current and projected hardware performance. The influence of these factors on how HAPPI was written is discussed. The general design objectives and top-level structure and functionality of HAPPI are laid out as foundation for the next two chapters.

2.2 Contemporary Imaging Trends

HAPPI was written at a time in which the image processing industry had yet to mature. At this writing, cost-effective commodity solutions to diverse and demanding industrial image processing problems are few and far between. In this section, we discuss the current and projected trends in hardware platforms and software standards that are important to the maturation of the image processing industry, as well as a sampling of contemporary image processing software packages.

2.2.1 Hardware Capabilities and Costs

Digital image processing has been made practical by recent gains in computer hardware performance which have been driven by the advances in microcircuit integration made in the last couple of decades. At this writing, it is practical to perform rudimentary image processing operations on personal computer-based systems costing less than $10,000, and more compute-intensive image processing operations on graphics workstation-based systems in the $10,000 to $50,000 range. However, these systems by themselves may not always be adequate to satisfy the needs of an NDE radiographic inspection operation. A prerequisite for image processing is image digitization. The hardware needed to digitize radiographs at the high spatial and intensity resolutions required for critical applications is still cost-prohibitive for many potential users at this writing. A rudimentary digitization system, consisting of an imaging tube-based or charge-coupled-device- (CCD) based video camera, high-quality lens, camera stand, lightbox (for illuminating radiographs), and frame grabber (video signal digitizer) may be put together for $10,000 to $15,000. State-of-the-art image scanning devices with spatial resolution down to 25 microns and 8-bit intensity resolution currently are sold for $40,000 to $60,000; systems with 12-bit resolution for more demanding applications are priced yet higher. Also, increased image resolution means larger volumes of data to process, which leads to increased speed requirements (and thus increased cost) for the processing system to keep overall inspection times reasonable. Consequently, demanding applications still tend to be served by

special-purpose expensive hardware. One NDE radiograph image processing system receiving much attention at this time is the Scan IV system by DuPont. This system consists of a high-resolution (35 microns spatial resolution and 3.5 decades light intensity dynamic range) digitizer, a workstation-class computer augmented by several add-on image processing boards, high-capacity (~2 gigabyte), high-speed optical disk drives, a video signal digitizer for incorporating real-time video images into the system, 3 image display CRTs, and a high-resolution digital film recorder for film hardcopy output (Eizember, 1990). This system is presently sold for hundreds of thousands of dollars and reportedly requires several person-months of time to set up and get running. Other radiographic image processing systems have been developed at Ohio State University, the Army Materials Technology Laboratory, and the Electric Power Research Institute, among others (Sheppard, 1987). The high price of such systems keeps their sales volume low, and so high-performance radiograph image processing is presently not a commodity. Development of software products for these systems tends to proceed slowly, with custom work being done for each customer and with software not being portable between different high-end systems. There seems to be widespread agreement that the "traditional approach of using custom hardware and software to address the imaging applications has actually retarded the growth of new imaging technology by keeping prices high and not addressing the issue of standards conformity required to spur application development" (Pfeiffer, 1990, p. 36).

The imaging industry has begun to respond to the difficulties presented by high-priced custom image processing systems. Pfeiffer (1990) argues that

the increasing availability of image data, the continued improvement in price-performance ratios of desktop computers, and emergence of software standards in the form of Application Program Interfaces (APIs), will lead to high-performance image processing capability being embedded in the workstations of the future, in much the same way that high-performance graphics capabilities have been integrated into current workstations. A tightly integrated "visualization environment" is foreseen wherein the image processing software development environment is but a part of a larger, comprehensive environment which includes high-level graphics tools and a customizable system-user interface. As of the early 90's several major workstation vendors had in fact begun to embed image processing capability in their products (Yencharis, Oct. 1990), although some industry observers felt that some of these efforts were not yet very well thought out (Mazor, 1990). Considered particularly significant are the increasing appearance of DCT (Discrete Cosine Transform, used to compress image data for storage and transmission) chips in workstations and the widespread use of the Intel i860 RISC processor in new parallel supercomputers (Mazor 1990).

In the recent past and near future, those requiring a relatively modest image processing capability have and will continue to develop solutions by integrating hardware from various vendors and patching together application software from whatever development tools and libraries are provided with the hardware. Factors such as open computer bus architectures and graphics standards presently make development of such solutions a relatively easy task when performance requirements are not particularly demanding. However, until such time as the workstation "visualization environment" envisioned by

Pfeiffer develops and matures, commodity off-the-shelf systems will not be available to satisfy many of the diverse and often demanding applications in NDE image processing.

2.2.2 Software Standards

The image processing market has fallen short of expectations for the 1980's, according to industry observers (Yencharis Aug. 1990 and Oct. 1990, Schwarz 1990, and Mazor 1990). A primary reason for this is cited as being the lack of turnkey solutions (i.e., complete hardware-software systems which users buy, turn on, and immediately begin using to solve their problems), the development of which has been hampered by the lack of software standards. Image processing is seen by some as being not a market per se, but rather a broad and diverse set of applications within existing markets (Schwarz 1990). Others who may speak of an actual "market" for image processing nevertheless also see it as being broad, diverse, and shallow, with many potential customers needing only one or two processing systems to use as tools to get their job done. It is not economical for software developers to attempt to address the needs of such a market without the "enabling technology" provided by a good set of software standards.

It is traditional for software development to lag hardware development in all areas of computerized data processing, and the lag has been noted for some time in the field of image processing (Frei, 1985). This lag is, to some degree, natural and expected; software developers want to be confident that there will be significant demand (in the form of an installed base of users of

the target hardware platform) for their product before committing resources to the product's development. And in any case, a working prototype of the target hardware must be available for any appreciable software development to take place. However, the software lag, when large compared with the rate of progress in hardware capability, can retard the growth of computer markets. By the time software products that fully exploit the capabilities of a given generation of hardware are on the market, the next generation of hardware is out, and the prospective buyer of a system must choose between a hardware platform that is already becoming obsolete but for which there is useful software available, and a state-of-the-art hardware platform which will probably not have useful software available until it, too, is becoming obsolete. Under these circumstances, many potential buyers may simply decide not to buy anything. Some industry observers believe that the image processing software development lag is steadily getting worse (Mazor 1990), and that this is keeping customers away. Software standards are an important way of dealing with the negative effects of the software development lag. By hiding the hardware-specific details from the software developer, these standards make it possible for the developer to write software that runs on multiple hardware platforms and/or more than one generation of a given hardware platform, and to do so in less time than would be required without standards. The software developer's costs are greatly reduced and potential earnings increased, and thus his/her risk is lowered.

Serious attempts at developing image processing software standards have appeared only since the late 1980's. One notable early effort is the Imaging Kernel System, or IKS, developed at the University of Lowell. IKS was

designed as a device-independent application program interface (API) which would allow programmers to, without detailed knowledge of the target hardware architecture, develop image processing applications programs that were portable to any hardware platform supporting the standard and that would automatically take full advantage of any special image processing capabilities of each hardware platform. Features of IKS included object-oriented design, use of virtual devices and virtual device tables for translating application requests from the API level down to the appropriate hardware, data abstraction of various data items and structures used in image processing, and a client-server design model in which client programs (i.e., application programs) requested processing services of the IKS server through the API. While IKS itself was not adopted as an industry-wide standard, its developers went on to sit on the American National Standards Institute (ANSI) committee X3H3.8, which, along with the International Standards Organization (ISO) committee SC24, began working to develop an ANSI standard image processing API known as the Programmer's Imaging Kernel, or PIK (Pfeiffer 1990).

Another API, under development at a company called VITec, is known as "Programmer's Image Computing Environment Software (PICES). PICES developers also sat on the ANSI committee developing PIK, and as of late 1990 claimed that the then currently available version of PICES would conform to the PIK standard when it is finalized (Pfeiffer 1990). PICES has many features in common with IKS, such as memory management, support for user-defined algorithms and data types, and virtual I/O device interfaces; its developers also claim that its design will facilitate interoperability with other APIs (e.g.,

graphics APIs such as PHIGS and GKS), leading to the tightly integrated "visualization environment" foreseen by Pfeiffer.

The formation of the ANSI committee to develop PIK represents widespread recognition of the need for an industry standard image processing API. The situation with PIK in early 1991 was as follows: Most of the major workstation vendors and many of the major vendors of special-purpose image processing hardware subsystems are represented on the committee. The stated goals of the PIK committee are much the same as those of other, previous image processing API developers: software portability and extensibility, hardware platform independence, compatibility with other standard APIs, window systems, and image file formats, and provision of data management utilities. While PIK does not address system performance issues, not excluding real-time applications is also a stated goal (Stephenson 1990). PIK contains a large and diverse library of image processing algorithms and utilities, which reflects the broad, shallow nature of the image processing market and the broad-based makeup of the ANSI PIK committee. Most image processing applications developers will likely deal with only a small portion of this library. The reader is referred to the article by Stephenson (1990) for a summary of PIK operators, but is cautioned that the only final word on PIK will be the ANSI standard itself. Several issues are yet to be resolved with PIK, and others will attend the finalized version. There is not yet agreement on the implementations of all image processing algorithms in the standard's library. The internal (i.e., machine) representation format of image pixel data types is not specified by the standard; neither are storage formats or conventions for image data memory management specified. These issues will affect efforts to

verify a PIK implementation's conformance to the standard, and, since verifiability of a standard is an important requisite for its acceptance, could slow its acceptance. Also, PIK does not address performance issues; this encourages the migration of the standard to the largest number of "price/performance points", from low-cost personal computers to expensive supercomputers. However, coupled with other non-specified system characteristics such as memory management conventions, the lack of performance specifications could hold pitfalls for applications developers (Stephenson 1990). In any case, the PIK standard will more than likely have a positive impact on image processing application development, spurring growth in the image processing industry as a whole.

In early 1991, an official ISO project, titled Image Processing and Interchange (ISO/IEC Project 1.24.10), was begun to develop an international standard integrating an image processing API as well as an image interchange facility (Clark 1992). Previous work on PIK is to form the basis for the API, which is now called PIKS (for Programmer's Imaging Kernel System). It is intended that the image processing API (i.e., PIKS) and the image interchange facility (IIF) will work independently of each other, although there will be an interface between the two. To address the problems presented by the broad, shallow nature of image processing markets, there will be a number of conformance levels for both PIKS and the IIF. Less demanding applications will only need to meet lower conformance levels of the standard. The PIKS standard is currently planned to specify about 200 image processing operators in the following categories: image analysis, classification, color processing, detection and registration, edge, line and spot detection, enhancement,

filtering, geometric, morphological, point operations, restoration, segmentation, shape, unitary transformation, and 3-D specific operators. Details about PIKS have not yet been made available to the general public; the schedule for the standard places its completion date in early 1994 (Clark, 1992).

Another standard still under development and receiving much attention is the JPEG compression standard (this standard is more a specification of a set of algorithms than of a software interface, but is nevertheless important to the image processing industry). The acronym JPEG stands for the Joint Photographic Experts Group, a collaborative effort between the CCITT (International Telegraph and Telephone Consultative Committee), and ISO (International Standards Organization). JPEG's purpose is to develop a robust standard for compression of virtually any type of continuous-tone digital source image; the draft standard compression method is based on the Discrete Cosine Transform, or DCT (Wallace 1991). The JPEG compression standard is seen as another extremely important enabling technology for image processing applications. Though image capture and display devices suitable for a multitude of applications are now quite affordable, many of these applications are still not yet viable due the enormous amounts of data required to represent digital images and the attendant storage and transmission costs. JPEG's stated goals are as follows: 1) To achieve state-of-the-art or nearly state-of-the-art compression rates for a wide range of image quality ratings, while allowing the application or user to set the desired compression/image quality tradeoff, 2) to be applicable to virtually all continuous-tone digital source images, 3) to have tractable computational complexity, allowing software implementation with good performance on general-purpose CPUs as well as

low-cost hardware implementations, and 4) to have sequential, progressive, lossless, and hierarchical encoding modes of operation (the reader is referred to the article by Wallace for details on these modes). It is predicted that if JPEG's goals are substantially met, many image processing applications will flourish, widespread exchange of image databases between different application areas will take place, and performance-sensitive applications inhibited by high storage and transmission costs will be served by high-volume, low-cost VLSI implementations.

2.2.3 Other Contemporary Image Processing Packages

Many types of image processing software products are currently available, and we may only expect more to appear. These products range from simple algorithm libraries to complete, end-user application programs, such as HAPPI. In this section, we briefly discuss a few image processing software packages which are contemporaries of HAPPI. The intent here is to look at a sampling of the different types of available products; a comprehensive analysis of the image processing software market is beyond the scope of this document.

At one end of the spectrum of image processing products is Paragon IL/F, from Paragon Imaging. The IL/F product is simply a FORTRAN subroutine library of image processing algorithms. The IL/F library is large and robust, with functions ranging from simple image data management utilities, arithmetic (add, subtract, multiply, divide) operations on images, and statistical analyses of images, to more advanced image restoration algorithms

such as Wiener filtering. Use of Paragon IL/F requires programming; IL/F is not meant as an application. The IL/F user is responsible for specifying the desired behavior of his/her image processing application, and for implementing it through the subroutine library. This product is somewhat primitive, as it only offers algorithms for processing functionality; it does not provide user-interface building tools. The choice of FORTRAN for the library is a handicap for development of applications with graphical user interfaces (GUIs); most modern GUIs are written in a more powerful language, such as C, and difficulties would likely be encountered in interfacing a GUI with this particular processing software.

A much more sophisticated application development software product, also from Paragon, is known as Visualization Workbench. This product not only provides an extensive algorithm library like that found in IL/F, but in addition has facilities for creating combinations of graphical and command line-based user interfaces. Most importantly, the application programmer can develop an application without writing actual source code; Visualization Workbench provides a "visual programming" feature, wherein the application developer creates a prototype by manipulating graphical icons on the computer display. Visualization Workbench is designed to run on the host processor of a workstation-class computer, and thus does not support any special-purpose accelerator boards. However, its algorithm library is claimed to support the PIK draft standard, which means that it should automatically take advantage of any PIK-compliant accelerators once the standard is finalized and such products begin to appear.

Another application development environment with slightly different features is Environment36, from Gems of Cambridge. Unlike Visualization Workbench, Environment36 supports an optional hardware accelerator card, also manufactured by Gems. Environment36 consists of this hardware accelerator and an application development software package called Gemsoft36. Gemsoft36 contains both an algorithm library (which appears to be less robust than Paragon's) and programmer's toolkit for building a graphical user interface. In this respect, prototyping with Gemsoft36 is most certainly easier than with Paragon IL/F, though not likely to be as effortless as with Visualization Workbench. Like Visualization Workbench, Gemsoft36 runs on a workstation-class host computer.

Representative of personal computer-based image processing is Image-Pro from Media Cybernetics. Originally available only for PCs, this end-user application package has also been implemented on workstation-class computers. As all but the most rudimentary point transformations are often unwieldy to perform on a PC's host processor, a number of image processing accelerator boards are available for PCs; Image-Pro supports several of these boards. Many relatively primitive image analysis functions are available in Image-Pro (e.g., histograms); the few processing functions are also fairly elementary, consisting mostly of convolution-based and lookup table (LUT) transformations. A separate Image-Pro module which performs Fourier frequency-domain processing can be purchased, but, in this author's opinion, the base package is of such limited use that it would greatly benefit from having the Fourier module integrated into it. The consensus of Image-Pro users, including both those quoted in trade journals and users at Iowa State's

Center for NDE, is that while it may be useful as an exploratory tool for newcomers to image processing, Image-Pro's utility is severely limited for more experienced practitioners with more demanding applications (Mazor 1990).

Several image processing packages have been developed at U.S. universities and research laboratories; many have been placed in the public domain and are thus available free of charge. One such package is called View, and is co-funded by the Lawrence Livermore National Laboratory, the Strategic Defense Initiative Organization, and the Rome Air Development Center. View is written to run on workstation-class computers with a window-based user interface. Unlike the Paragon and Gems products, View is an end-user application. The extent and diversity of its image processing capability is much greater than that of Image-Pro, but somewhat less than that of the Paragon products. Distinguishing features of View are that it supports three-dimensional data set processing and visualization, has a basic image simulation capability, and includes some traditional filters not found in other packages (e.g., Bessel and Butterworth filters). View is maintained as an ongoing project by the Lawrence Livermore National Laboratory.

The Scan IV system from DuPont, mentioned earlier in the chapter, represents the other end of the spectrum from simple algorithm libraries. It consists of special-purpose hardware plus end-user application software written specifically for that hardware; it is truly a bundled, turnkey system for NDE radiography. Its distinguishing features are its very high-resolution scanner, large image storage capacity, and film hardcopy capability. Literature for the product indicates that the system's software is not as state-

of-the-art as its hardware, and that because of the relatively low volume of sales, custom software work is often done for individual customers. However, the product literature also indicates that more specific applications software (e.g., image processing for weld flaw classification) is planned.

As may be seen from the above examples, many types of image processing software, ranging from toolkits to hardware-specific application programs are available to meet different needs. This is a reflection of the nature of the image processing market. It is hoped that the above discussion has given the reader a feel for this market that will provide some perspective for assessing HAPPI. In the next section, we discuss the design objectives, program features, and the top-level structure and functionality of HAPPI.

2.3 HAPPI Design Objectives and Program Features

HAPPI's design objectives were based on perceived user needs gathered through interaction with the industrial sponsors of the Center for NDE at ISU. These sponsors indicated that they would like to have an integrated hardware-software system capable of both capturing and processing digital images of radiographs, with a large repertoire of image processing algorithms accessed through a friendly, intuitive graphical user interface. The most likely user of the system was to be a radiographic technician responsible for inspecting parts; other possible users included NDE engineers responsible for developing inspection methods for new products. Also, HAPPI was intended to be a prototype for a commercial package to be developed by an industrial partner

of the Center for Advanced Technology Development. The design objectives of HAPPI as enumerated at the beginning of the project are listed below:

1) To provide an easy-to-use interface between the NDE radiographer and image processing software particularly useful for NDE.

2) To allow the user to produce useful results (i.e., detected flaws) without requiring him/her to embark on a long, detailed study of image processing theory.

3) To provide the user with a wide range of utilities such as image file format conversions, audit trails of image processing steps, and macro building (where a "macro" in this sense is a specific series of image processing steps performed in sequence on an image or set of images).

4) To provide an interface for users who wish to add their own processing algorithms to the package.

5) To make the software as device-independent as possible.

6) To make modification and enhancement of the package by programmers other than the original authors straightforward.

HAPPI features a graphical user interface based on the X Window network-based graphical window system developed at MIT. The user interface consists of a graphical hierarchical menu structure through which all program functions are accessed using the mouse. A "Main Menu" window and an "Information" window are displayed at all times while HAPPI is running. Other types of windows appear only when the functions they serve are activated by the user through the Main Menu window. These other windows

include submenus activated by making a main menu selection, a "Value" window through which the user enters parameters for processing routines, an "Acknowledge" window in which the user is advised of unusual or dangerous situations (e.g., the user attempting to delete a newly created image which has not yet been saved to disk), image windows in which images are displayed, a system window which gives the user access to an operating system shell, and graphics windows in which image histograms and one-dimensional image slice data are displayed. HAPPI's main menu selections are: "Image Processing", which contains all of HAPPI's image processing functions; "Acquisition", which contains image acquisition functions; "Images", which contains functions for loading and saving images from and to disk storage; "Macros", which contains macro processing functions (to be explained below); "Special Functions", which contains functions to access operating system services, including an operating system shell; "Buffer", which displays a list of all images currently in memory and their display status (i.e., displayed vs. hidden); and "Quit", which exits the user from the program. For further details on HAPPI's menu hierarchy, the reader is referred to the HAPPI documentation listed in the bibliography.

An important feature of HAPPI is its built-in macro language. This feature allows the execution of package functions normally accessed through the menu structure to be performed with no user interaction between the beginning and end of the sequence of macro instructions. The macro language contains all common features found in other high-level languages, such as variable declarations, conditional and looping constructs, and user-defined subroutines.

It was assumed that the targeted user of HAPPI (i.e., the NDE radiographer) is not an experienced programmer, and also that he/she is not familiar with image processing theory or techniques. Default input parameters for each processing routine are provided to enable a new user to get a feel for the results produced by a particular algorithm without being concerned with how the results were obtained. Also, when a user alters the default input parameters, the values used are preserved and become the new default parameters for remainder of the processing session. It was also assumed that the user may need to do both routine inspection of parts from a production line as well as occasional inspection of a part in the prototype stage. The menu-based interface thus accommodates interactive processing for exploratory prototype inspection while the macro language facilitates batch processing of multiple production line radiographs once a processing scheme has been optimized for a particular part.

To make HAPPI as portable as possible, it was necessary to adhere to all existing software standards. However, at the time of HAPPI's design, the image processing APIs discussed in the previous section were either not widely adopted or not complete. In essence, there were no *image processing* software standards upon which HAPPI could be built; each hardware vendor had a unique, nonstandard interface. Consequently, it was not possible for HAPPI to make use of any high-performance computer architectures for image processing while remaining highly portable. It was thus decided that HAPPI would be implemented on a graphics workstation-class computer typical of those used for a wide range of engineering tasks. The operating system of

choice for these computers is UNIX[1], and the graphics standard used on these machines is X Windows (or simply "X"). The most widely used and documented programmer's interface to X is through the C programming language. At the beginning of the project which produced HAPPI, X was the de facto industry standard; X has since gained universal acceptance. The C programming language is a natural choice for applications running under UNIX, as the UNIX operating system itself is written in C. The language is relatively small, which allows the programmer to regularly use most of its features and makes applications extremely portable; C also gives the programmer access to powerful low-level hardware functions.

The X Window system was developed at MIT in cooperation with a consortium of corporate sponsors. It provides high-performance, device-independent, network transparent graphics, and features a client-server programming model. In this model, application programs act as "clients" which request the network services of an X server. The X server is a program running on a user's display which controls that display's hardware and provides I/O services to applications, and which maintains its own local data structures to minimize network traffic between it and its clients. The fact that clients may request X services across a network means that compute-intensive applications may run on a powerful central host computer while displaying sophisticated graphical output on one or several low-cost X display stations. On invocation, X applications must request and make a connection with an X server and initialize any data structures to be maintained by the server. The foundation layer of X is the X network protocol; this is the mechanism by

[1]UNIX is a trademark of AT&T.

which servers and clients communicate. The application programmer's lowest level interface to X is a set of C language function calls, built on top of the protocol, and referred to as Xlib. The authors of X intended that most applications be written using a higher-level programming interface, called a toolkit, than Xlib. However, at the time the HAPPI project began, no standard toolkit had emerged (O'Reilly 1989 and Nye 1990, p.10), and so application programmers could not be certain of writing extremely portable code using any of the toolkits available at that time. For this reason, a sort of "custom toolkit" was written for HAPPI using Xlib; this set of routines was used extensively throughout HAPPI to create and destroy windows and exchange information between the program and the user. Several of the routines are discussed in Chapter 4 on extending HAPPI; additional information is available from the HAPPI documentation in the bibliography. The reader is referred to, as an example, the text by Scheifler *et al.* (1988) for further information on X.

CHAPTER 3: EVALUATION OF HAPPI

3.1 Introduction

As with most projects with a finite time budget, HAPPI is not everything that it could be. The program was written not by experienced software designers but by students, and of necessity, much on-the-job learning took place during the course of the project. In this chapter, we evaluate HAPPI's strengths and weaknesses, with the hope that the experience gained will influence both the maintenance and extension of the X-ray Image Processing Group's local version of HAPPI and the future design of other image processing software by the group.

3.2 Strengths of HAPPI

HAPPI's strengths are the robustness of its library of processing routines, the ease of use of its user interface, and its portability, extensibility, and programmability. This is not to say that HAPPI is perfect in all of these areas, but rather that it addresses them well. We will see in the next section where HAPPI could be improved in these and other areas.

HAPPI's repertoire of image processing routines consists both of common, well-known techniques, as well as more specialized techniques which have been developed over the past few years in the X-ray Image Processing Group. The more common techniques are found in many commercial image processing software packages, and thus constitute a

minimum amount of functionality that HAPPI needs in order be competitive with such software packages. Most of these common techniques have been implemented in HAPPI. (The gaps in HAPPI's repertoire of basic processing routines are discussed in the next section.) The more specialized techniques developed by the X-ray Image Processing Group, such as the routines found under the "Flaw Detection" menu, round out HAPPI's processing capability and distinguish it from other, more generic image processing packages. These routines were developed using NDE images as test data, and are, to varying extents, better "tuned" to certain NDE applications than the more common routines. On occasions when HAPPI has been presented to the industrial sponsors of the Center for NDE at ISU, NDE practitioners have indicated that HAPPI's library of processing routines is quite robust compared with that of other commercially available software. One consequence of this is that HAPPI may often provide many more processing functions than are needed for a particular NDE application.

HAPPI's menu-driven, graphical user interface has proven to be easy for first-time users to experiment with. During the last phase of the project, a complete demonstration system, including a workstation, frame grabber, light box, and camera, was taken to the NDE lab of one of the Center for NDE's industrial sponsors. Personnel at the sponsor's site were able to load, process, and store images with minimal help from the developers of HAPPI and without reading a manual. The routines which fetch user input for HAPPI's processing routines guide the user's choice of input parameters, indicating and enforcing any parameter constraints, and proposing default values that meet these constraints. Default parameter values for processing routines with

similar input parameters are shared between such processing routines. Also, the last value entered by the user for any given parameter is preserved and used as the default parameter value at the next invocation of any function which uses that parameter. These features help the user to easily experiment with the effects of each processing routine on images without burdening him/her with the responsibility of remembering parameter constraints and previously used parameter values.

HAPPI's portability has been demonstrated by successful ports (with minor modifications) to computers other than the Stellar GS1025 on which it was developed. The program is based on standards that were stable at the time it was written: C, UNIX, and the Xlib interface to X Windows. HAPPI is thus, in theory, portable to any system that adheres to these standards. The C language is itself inherently portable by virtue of the "smallness" of the language; it is relatively easy to write portable programs in C by following a few simple conventions (Kelley and Pohl, 1984, p.2). UNIX is the operating system of choice on the workstation-class computers for which HAPPI was designed. As HAPPI's graphics routines were written using the Xlib low-level interface to X Windows, HAPPI does not require the support of any particular X toolkit to port to a particular workstation.

The code structure underlying HAPPI's image processing functionality is fairly regular and repetitive. This makes HAPPI readily extensible by C programmers. In Chapter 4, we give a procedure for adding new image processing routines to HAPPI, discussing in detail the code structure and tools available to the programmer modifying HAPPI. A central piece of code examined in Chapter 4 is the Image Processing Manager. The Image

Processing Manager enhances HAPPI's extensibility and maintainability by providing a versatile interface to HAPPI's image processing routines that accommodates both menu-driven and macro-driven access to the processing routines.

HAPPI's built-in macro language provides programmability to the HAPPI user. The macro language can execute most of HAPPI's image processing and image I/O functions, and also implements many features common to general-purpose computer programming languages, such as variable declarations, looping and decision constructs, and procedure definitions. HAPPI's "convert history to macro" feature allows the user to create a macro from the processing history of an image without typing a single line of macro language code. The macro language is useful for doing repetitive processing of many similar images. It may also be used in an exploratory processing situation to determine the most useful processing routines and input parameters to use on a particular image or class of images.

3.3 Areas for Further Improvement to HAPPI

There are many ways in which HAPPI could be improved. Some of the possible improvements involve adding desirable features that were identified later in the project but were not implemented for lack of time. Others involve more extensive changes to the underlying structure of the program. For purposes of discussion, the proposed improvements to HAPPI in this section are grouped into user interface enhancements, overall program behavior

enhancements, additional functionality, and enhancements to performance and code maintainability.

3.3.1 User Interface Enhancements

Perhaps the most useful improvement to HAPPI's user interface would be the addition of a command-line interface concurrent with the existing menu-based graphical user interface. With such an interface, the user would be able to access any of HAPPI's functions by typing alphanumeric text in a command entry window and possibly by using programmable function keys. Experience has shown that software users tend to favor mouse-and-menu-based interfaces when first learning how to use a new program, as the menus guide their choice of input. However, as a user becomes more experienced with a program, and begins to memorize the various commands and command parameters and options, a command-line interface becomes more desirable, as it generally facilitates faster user interaction and results in less screen clutter than a mouse-and-menu-based interface; this is especially evident when there are many nesting levels in the menu hierarchy. HAPPI's built-in macro language is the most logical starting point for implementing a command-line interface; the language already provides access to HAPPI's most-used and most important functions. The degree of difficulty of adapting the macro language to a command-line interface would depend on how much of the macro language is implemented in the command language. Making the macro language's programming constructs available from the command window would require more effort than simply making the image I/O and processing

calls available. It should be noted that a large part of the task of implementing a command-line interface for HAPPI lies in parsing, analyzing, and interpreting the command line; much of this has been taken care of in implementing the macro language.

A possible extension to a command-line interface to HAPPI would be the execution of HAPPI commands and/or macros directly from the operating system prompt. This would involve invoking HAPPI without creating or displaying any windows, loading the remainder of HAPPI's (non-windowing) code and executing the command and/or macro, then returning to the operating system. Such an extension to HAPPI's interface would be useful for users who, after experimenting with different processing techniques, have identified and standardized particular methods that they use frequently. These users may wish to implement their standard processing methods on a large number of images without being obliged to clutter their computer screens with HAPPI's graphical interface at a time when they are not using that interface. Such an extension to HAPPI would also provide processing capability to users who do not have X Windows display capability at their particular terminals.

When a processing routine is executed in the present version of HAPPI, the user must always first select the processing routine to be executed, and then select the input image(s) for the routine. An alternative mode of operation would be to allow the user to designate an image as the "currently selected image", and have all processing routines selected by the user automatically operate on the currently selected image. This mode of operation would save the user unnecessary mouse motion and button clicks when he/she

is experimenting with the effects of different processing routines on the same image. Providing this alternative mode of operation would not be difficult given the present state of HAPPI's structure. There would, however, need to be a method of indicating graphically which image on the screen is the "currently selected image". A further step would be a "multiple image select" mode, wherein several images could be selected and a common processing routine automatically applied to all of them, using the same processing parameters for each of them.

Another enhancement to HAPPI's user interface which would help reduce unnecessary mouse motion and button clicks involves placing "active" or "smart" borders around HAPPI's image and graphics windows. In the present version of HAPPI, the user must select a menu item to delete or hide an image or graphics window, and must re-select the same menu item for every window on which he/she wishes to perform the action. After selecting the menu item, the user must then select the window on which to perform the action, resulting in two clicks per window per operation. A more efficient way to remove or hide image and graphics windows would be to place graphical borders around these windows, with graphical "buttons" for deleting and hiding the window. The user could then delete an image from the screen and the computer's memory with a single mouse button click on the appropriate spot on the image window border.

Finally, the method of entering mask values for large user-defined convolution masks needs to be streamlined. The current version of HAPPI requires the user to manipulate a graphical "value window" to enter every single mask value. This can be very slow and time-consuming for large masks.

A better method would be to allow the user to type in all mask values directly from the keyboard.

3.3.2 Program Behavior Enhancements

A very significant enhancement to HAPPI's overall behavior would be the addition of some sort of multitasking capability. In the present version of HAPPI, the user may not access any of the program's functions while an image processing routine is running. Depending on the speed of the host computer system on which HAPPI is running, the size of the image being processed, the parameters passed to a processing routine, and other factors, the execution time of a processing routine can be anywhere from a few seconds to several minutes or tens of minutes. The longer processing times can detract from the advantages the user gains from the interactive processing environment provided by HAPPI. A multitasking version of HAPPI would allow the user to execute more than one of HAPPI's functions at once, allowing the user to be, on the average, more productive. The allowed number of concurrently running tasks in a multitasking version of HAPPI is a design parameter that would need to be studied. With each additional concurrent task, some computational overhead is incurred, and at some point the overhead would begin to offset the benefits of multitasking.

One way of implementing a multitasking version of HAPPI is to create a separate UNIX process for every HAPPI function whenever that function is invoked and perform *interprocess communication* between the function and the main program via *pipes*. Pipes are first-in-first-out (FIFO) data structures

which serve as I/O channels for interprocess communication. Such an architectural modification to HAPPI would be a significant undertaking. Other methods of interprocess communication that could be considered in building a multitasking version of HAPPI include *messages, semaphores, shared memory,* and *remote procedure calls* (RPC's) (Stellar Computer Inc., 1988b, p.15-1). These are all implemented through UNIX system calls; the C language itself has no multiprogramming features (Kernighan and Ritchie, 1986, p.2).

Another possible enhancement to HAPPI's overall behavior is the ability to read "startup files", which would allow individual users to customize the program's behavior to their preferences. In the present version of HAPPI, this is not an important issue, as the number of items which could be customized is small. Future versions of HAPPI with more overall system behavior options would benefit more from such an enhancement.

3.3.3 Additional Functionality

A number of functions could be added to HAPPI which would increase its utility. Some of these are commonly found in commercial software; other less common functions were inspired by experience with HAPPI itself. We discuss here several of these functions, while recognizing that the list is not exhaustive; most people who use any particular piece of software for a long time can think of endless enhancements they would like to see.

HAPPI could benefit from the addition of more data visualization tools. Many scientific software packages provide extensive plotting and graphing capabilities, such as contour plots, 3-d hidden line plots, and others. Addition

of these capabilities to HAPPI would enhance its image analysis power, since the "best" data display method depends both on the specific application and on the tastes of the user.

HAPPI lacks, and should have, full support for real and complex-valued images. HAPPI can currently represent such images internally, but does not allow the user to manipulate them. As a consequence, the inverse Fast Fourier Transform (FFT) is not accessible to the user. (The forward transform is accessible, but currently only provides the magnitude of the complex-valued frequency-domain image, and scales the floating-point magnitude values to the 0-255 grey scale range.) This is an oversight, as the inverse FFT is an essential function, and full support for operations on real and complex-valued images should have been planned for earlier in the project. A large group of image data manipulation routines embedded in HAPPI's source code, known as the "image operation routines", or "iops", supports user functions which manipulate grey scale and binary images. Writing a corresponding set of support routines for real and complex-valued images would facilitate the addition of user functions to manipulate these images as well. No changes to HAPPI's overall architecture would be needed; the new support routines would simply be grouped with, and accessed in the same way as, the existing ones. Methods and policies for displaying the inherently 4-dimensional data of a complex-valued image on a 2-dimensional, 8-bit computer display would need to be developed, and would have to address both the higher dimensionality and the large dynamic range (compared with grey scale images) of complex-valued images. Another, related capability commonly found in commercial image processing software that is missing from HAPPI is user-definable

frequency-domain filtering. Full support for complex-valued images and the inverse FFT would set the stage for implementing such a frequency-domain filtering capability as well.

Support for additional image data structures may be advantageous for future versions of HAPPI. The program was originally intended for use on X-ray NDE images. Future versions may include features for processing images formed with other NDE inspection techniques, including thermographic, electromagnetic, and ultrasonic methods. One can envision combining these images into a composite image that could yield much more information about a part under test than could any of the images formed with the individual inspection techniques. Routines for manipulating such a composite image data structure, similar to those which handle binary and grey scale images, would need to be written to support processing of the composite images. Another variation on this idea is to provide support for processing and manipulation of 1-d arrays as data objects similar to images. Many of HAPPI's image processing routines could be used to advantage on 1-d data sets from, say, ultrasonic scans. Each of these routines would need to be examined and modified if necessary to adapt to 1-d inputs. New display routines would need to be written to display 1-d arrays as graphs rather than as light intensities.

HAPPI would benefit from having its own hardcopy capability. Images created in HAPPI can be printed by first saving them to disk in PostScript format and printing them from the UNIX operating system using a PostScript printer. (PostScript is a page description language, or PDL, and is a device-independent standard supported on a large number of laser printers. PDL's such as PostScript can be used to produce very high-quality hardcopy.)

However, this obviously requires more steps than would be needed if the system calls necessary to print directly from HAPPI were built in to the program. Also, images and macro files are the only data objects created by HAPPI that can currently be saved to disk and printed from the operating system. It would be more desirable to allow the user to print any of HAPPI's data objects, including images, image history data, image statistics data, image display lookup tables, graphs, and even the entire screen, directly from HAPPI.

Other I/O functions that would benefit HAPPI include the ability to read and write any of HAPPI's above-mentioned data objects to and from disk, and the ability to do so in different file formats (e.g., native HAPPI formats, PostScript format, and other image formats from various hardware and software vendors). These capabilities would allow printing of saved HAPPI data objects from outside the program when only a printout is needed, and would allow HAPPI data objects to be read into other application software, such as desktop publishing packages, for purposes of report generation.

Once the user of HAPPI has identified a flaw or suspected flaw in an image, he/she may wish to annotate the image with a graphical indicator of the flaw's location and perhaps with explanatory text. HAPPI currently lacks, and would benefit from having, this capability. One issue to be addressed in implementing graphical and textual annotation is how to keep the annotation information with the image data without overwriting any image data. If the annotation information is stored in a separate file from the image, it is possible for either the image or the annotation file to become "orphaned" if the other file is deleted, moved, or renamed. On the other hand, the annotation

data should not be simply written over pixel intensity data in the image file, as the overwritten data may be needed later. One possible solution is to make the annotation information part of the image data structure, keeping it separate from the pixel data but saving it in the same file as the image. This solution would involve revising HAPPI's image load and save routines and any image file format conversion routines to handle the revised image file format.

Some of HAPPI's routines, such as the white noise generator currently found under the "Noise Filters" menu, are meant for experimentation by the user and not for filtering the user's image data to make it somehow more desirable. These routines allow the user to add *known* degradations to images, and experiment with the effects of these degradations on subsequent processing steps. A useful extension to this capability would be the addition of a more comprehensive set of flaw simulation functions. This set could include routines for generating images of voids and cracks, adding noise with user-specified probability distributions, convolving simulated images with transfer functions characteristic of various imaging systems, adding a slowly varying intensity "trend" to simulated images, and composing a test image from extracted portions of other images (real or simulated).

A new trend in user interfaces for signal processing, VLSI design, Computer Aided Software Engineering (CASE), and many other types of engineering software is the ability to do "visual programming." With these software packages, the user programs a complicated process by interconnecting graphical objects representing the various operations that make up the process. Each object may have several inputs and outputs, as well as feedback, depending on the application. Examples of software with such

capabilities are the Visualization Workbench product from Paragon Imaging (Paragon Imaging, Inc.), and the development system software for a video signal processor marketed by Silicon & Software Systems (Blagden and Scanlan, 1990). HAPPI's macro language could gain a new level of user friendliness if it were implemented with a visual programming interface. Adding this capability would be a significant undertaking; while major architectural modifications to HAPPI would probably not be necessary, the additional functions needed to draw the graphical symbols and translate graphical information to actual sequences of macro instructions would require careful design and many lines of code.

While HAPPI makes a large number of functions available to the user, it seems that there are always more that would be nice to have. A number of miscellaneous tools and functions proposed for addition to HAPPI near or since the end of the project's funding are briefly discussed in the following paragraphs.

A useful function that could be readily implemented in HAPPI is the "in-place" processing of a region-of-interest (ROI) in an image, with the output data being overlaid at its original location within the image from which it was extracted. This would allow the user to reduce processing time by processing a smaller data set while retaining the visual context of the processed data for image interpretation.

HAPPI presently is capable of generating colormaps, or image display look-up tables (mappings of an image's numerical pixel values to light intensities on the computer display) which have a single linear segment. This

capability could be extended to allow piecewise linear colormaps with multiple linear segments.

One of HAPPI's particularly useful analysis tools is the "Pixel Analyzer", which, as the user moves the mouse cursor within an image, dynamically displays a magnified region of the image and the coordinates and numerical value of the pixel to which the mouse cursor points. This tool could be enhanced by calibrating its readout in terms of engineering units, such as centimeters in place of pixel coordinates, and optical density in place of numerical pixel value.

Another of HAPPI's analysis tools is the "real-time slice." To use this tool, the user drags a "slice cursor" (a vertical or horizontal line) across an image with the mouse, and the row or column of the image currently under the slice cursor is dynamically displayed in a separate "slice window" as a graph of grey level vs. position along the row or column. As the user moves the slice cursor across an image, the row or column of the image graphed in the slice window is continuously updated. The real-time slice capability could be further enhanced by allowing the user to take a real-time "slice" of the image at any arbitrary angle. This would help the user in analyzing long crack-like image features oriented at any angle.

3.3.4 Performance and Code Maintainability Enhancements

As mentioned in Subsection 3.3.2, some of HAPPI's image processing routines can take tens of minutes to complete, and given HAPPI's current inability to do multitasking, the user cannot do any useful work with HAPPI

while a processing routine is executing. Thus, a user who often needs to perform the more time-consuming processing tasks will find processing with HAPPI to be an inefficient use of his/her time. Part of the solution to this problem is, as previously discussed, to provide a multitasking capability within HAPPI. The other part of the solution is to make HAPPI run as fast as possible. In this section, we discuss issues related to increasing HAPPI's processing speed.

The Stellar GS1025 graphics supercomputer on which HAPPI was developed contains a *multistream processor* with a *synchronous pipeline multi-processor architecture*, which can concurrently execute up to four instruction "streams", and also contains four identical vector/floating-point processor units which can work independently or in tandem (Stellar Computer Inc. 1987, p. 7, 15). Depending on how a program is compiled (i.e., what compiler options are specified), and on how busy the computer system is with other tasks, a program may run in a parallel and vectorized fashion, using from one to four of the available instruction streams and from one to four of the vector/floating-point processors. At the time that HAPPI was written, the C compiler provided with the GS1025 system did not have full support for all optimization options, and so the present version of program has not been compiled with these options. Thus, HAPPI does not take advantage of much of the computing power available on this system. In one test of HAPPI's processing speed on the Stellar GS1025, the Abingdon Cross image processing benchmark (Preston, 1990) was performed, with quite unimpressive results (Doering, 1990). At this writing, the latest operating system release for this machine, with full support for the C compiler's optimization options, is soon to

be installed, making compilation of a vectorized and parallelized version of HAPPI possible on the GS1025. A fully optimized version of HAPPI will be considerably more useful for the X-ray Image Processing Group than the present version. (Note, however, that the speedup from optimization discussed here only applies to the Stellar machine.)

Although the optimizing compiler on a multiprocessing vector machine such as the Stellar does most of its work of vectorizing and parallelizing code automatically, the programmer must sometimes intervene and provide explicit instructions to the compiler to get the most performance out of a program. Certain code constructions inherently cannot be optimized. For example, loop vectorization is inhibited whenever the compiler detects a real or apparent *recurrence* in a loop. A recurrence, in the sense used here, is "an assignment to a variable in one loop iteration, followed by a use of that variable in a subsequent iteration" (Stellar Computer Inc., 1988a, p. 2-15). An autoregressive calculation is an example of a recurrence in this sense of the word, and as it is an inherently serial calculation, it cannot be vectorized. Since the compiler can not know everything about a program's execution in advance, it is sometimes unclear whether a certain piece of code can be safely optimized, and the compiler will refrain from optimizing some optimizable sections of code out of caution. The programmer may insert special instructions to the compiler, or *compiler directives*, in his/her code that tell the compiler it is "safe" to optimize a section of code. Several directives are available to enable the different types of optimizations. Obviously, the programmer should use the optimization directives with care; unpredictable and elusive errors can arise if the compiler is given license to optimize non-

optimizable code. There are also ways in which the programmer can write program statements to make the optimizing compiler's task easier. For example, one optimization technique known as "tree height reduction" attempts to break an expression into as many as possible sub-expressions as can be concurrently evaluated. Sometimes a programmer will use unnecessary parentheses in writing an expression simply to make the intended order of operations in the expression more clear at a casual glance. While this practice may result in more readable code, it can also unnecessarily constrain the compiler's choices as to the order of operations in an expression, thus overriding the more optimal choice the compiler would have made in the absence of the unnecessary parentheses (Stellar Computer Inc., 1988a, p. 2-5).

Other speed enhancements to certain of HAPPI's routines requiring no code modifications are possible through the use of special routine libraries. Workstation vendors sometimes provide special optimized versions of standard C libraries, such as the math library, which take advantage of any special architectural features of their workstations. The Stellar GS1025 has such a math library (the "fastmath" library), which features fast-executing vectorized implementations of trigonometric, inverse trigonometric, logarithmic, exponential, and hyperbolic functions in both single and double precision versions. Using the fast math library in place of the regular library is as simple as changing one line of text in HAPPI's source code files. (Note: The "fastmath" library was not accessible from C in the version of the compiler used for the HAPPI project; the new version soon to be installed has full support for "fastmath.")

The speed enhancements discussed thus far are mostly accomplished by tools supplied as part of a workstation's software development environment, and require relatively little input from the programmer. However, the resulting speedup in program execution will only be as good as the tools themselves, and also can only do so much to speed up inefficient code. It is up to the programmer to analyze and revise his/her code in ways that the tools cannot. A first step in doing this is code profiling. Code profiling is the "running of a program in such a way that is can be analyzed to determine where it spends most of its time" (Christian, 1988, p. 145). This is usually accomplished by compiling a program using a compiler option which inserts additional instructions into the program to allow the monitoring of control flow. The program is then typically run under control of another, special program called a *profiler*, which reports on what percentage of its total execution time the program spends in each routine. The most time-consuming sections of code are thus identified, and the programmer can then maximize the speedup in execution time gained per unit time spent rewriting inefficient code. One particular routine in the current version of HAPPI that is known to need rewriting for speed (though its unusual slowness was not identified with code profiling in this particular case) is the "Row/Col Fit" routine under the "Trend Removal" menu. This routine runs at least an order of magnitude slower in HAPPI than its original stand-alone version, for reasons unknown at this writing. To date, HAPPI has not been profiled to identify problem code.

One way in which HAPPI might be sped up after a more detailed analysis of its code is through the judicious choice of control parameters to the fast version of the memory allocation routine *malloc()*. The way in which this

routine divides up available blocks of memory, and thus the speed at which it can satisfy memory allocation requests, is determined by these parameters.

A possible performance enhancement whose potential benefit has not yet been quantified is dynamic memory management, or "garbage collecting". HAPPI must ask the operating system to dynamically allocate memory space for many of its data structures. The operating system services each request by searching a "memory map" for the next available chunk of contiguous memory locations of the appropriate size and returning the address of the beginning of that chunk to HAPPI. When a function within HAPPI completes, the memory allocated for that function is deallocated, or released back to the operating system. However, between the time that memory is initially allocated for a function and the time it is deallocated, other requests for memory may have been made by other functions. Also, each memory allocation request asks for a certain size chunk of *contiguous* memory locations, and so pieces of contiguous memory smaller than the requested size are skipped over (and thus left unallocated) by the operating system in servicing the memory allocation request. These conditions can result in what is known as "memory fragmentation." Since contiguous chunks of memory are not necessarily deallocated in the exact reverse order that they are allocated, there will be "holes" of unallocated memory in the memory map; this may make future memory allocation requests more difficult to satisfy. If the memory map becomes extremely fragmented, it may become impossible for the operating system to satisfy the next memory allocation request, and, depending on how well it was written to handle such a situation, a program may crash. A solution to this problem is to periodically move all data in

allocated memory to contiguous locations, so that there are no holes in the memory map. The extent to which HAPPI's performance degrades due to memory fragmentation has not been analyzed. The fragmentation phenomenon is dependent upon such things as which of HAPPI's functions the user exercises and in what order they are exercised, and also on the amount of memory available on the host computer system.

A wide spectrum of steps could be taken to enhance the maintainability of HAPPI's source code. In Chapter 4, a detailed procedure is given for adding image processing routines to the program. As will be seen in that chapter, the procedure involves duplication and modification of several code structures, resulting in redundant code in places. In particular, the routines which fetch user input for the image processing routines are all very similar in structure. An alternative to this redundant code is the implementation of a universal parameter fetching routine, which would be passed the number, names, and data types of the input parameters for each routine along with any constraints on their allowed values, and would adapt as necessary to display the appropriate input parameter menus for each processing routine. This would ease the programmer's task of adding new routines to HAPPI by eliminating the tedious and error-prone step of copying and modifying a piece of code, thereby allowing him/her to concentrate on the more important task of describing the input parameter data requirements correctly. Also, a universal parameter fetching routine would slow the rate at which HAPPI's code size grows with each new processing routine added.

The UNIX programming environment provides a set of programs known as the Source Code Control System (SCCS). Although HAPPI was not developed

using SCCS, future versions of the program (and any other future large software projects in the X-ray Image Processing Group, for that matter) would be much easier to maintain under the SCCS system, as it automates many administrative tasks in software development. For example, SCCS keeps track of previous versions of source files in an incremental fashion (only the *changes between versions* are stored, so as to conserve disk space); this feature allows the programmer to return to any previous version of a program, and also provides an audit trail of changes to source code files. The system also can be used to control who may edit which source code files, and to protect against two or more programmers simultaneously editing the same file (such a situation can result in programmer A losing all his/her changes to the file when programmer B saves his/her changes after programmer A's changes have been saved).

In Chapter 2, we briefly discussed the former lack of image processing program interface standards and the beginnings of such standards that are only now emerging. HAPPI's maintainability and portability will be enhanced by supporting such standards in the future. Maintainability is enhanced because standards tend to "hide" implementation details from programmers who may inherit HAPPI, allowing new tools and functions to be built onto HAPPI quickly and with confidence of portability. Standards such as the JPEG image compression standard will be implemented in special-purpose hardware on future workstations; thus, support of such standards will also have the desirable effect of greatly increasing HAPPI's performance.

HAPPI's user interface was built from the low-level Xlib interface to X windows. As mentioned before, this had the desirable effect of making the

program independent of any particular X toolkit supplied by a workstation vendor. However, one disadvantage of this approach is that the graphics routines that were custom written for HAPPI are now deeply embedded in the code; calls to these routines are used in every function that requires interaction with the user. This could make it quite difficult to change the "look and feel", of HAPPI's user interface were it decided such a change is needed. A way around the problem is to replace all of HAPPI's custom-written graphics routines with translation routines that would interface to a different graphics routine library such as one of the many X toolkits now available. The numerous calls the to present graphics routines could then be left in the code. It should be noted, however, that this would be just a "patch", a temporary and inelegant way to change the appearance of HAPPI's interface.

The maintainability of HAPPI from the user's perspective could bear improvement as well. HAPPI's present method of interacting with image processing routines written by an end-user is not very sophisticated. The user-written program is not really integrated into HAPPI's interface at all. Rather, the user must communicate with HAPPI through file I/O, which means, quite simply, that he/she must write stand-alone programs that read and write images in HAPPI's image file format. It is not a trivial task to write a program that can easily integrate the functionality of a user-written program into its interface. However, we anticipate that the significant architectural modifications (e.g., creation of separate UNIX processes for each function, and connection of these via UNIX interprocess communication mechanisms) necessary to implement a multi-tasking version of HAPPI will put in place

much that is required for smoothly integrating user-written programs into HAPPI's interface.

3.4 HAPPI 2.0

Since the initial draft of this chapter, a second version of HAPPI has been written. This version, known as HAPPI 2.0, runs on a Sun SPARCstation[1] IPC workstation, and is installed at this writing on the SPARCstation host picard.ee.iastate.edu in the X-ray Image Processing Group's laboratory. HAPPI 2.0 was funded by the Center for Advanced Technology Development (CATD) at Iowa State University. At this writing, CATD has exclusive control of the source code; the original version of HAPPI is the only one whose source code is available for modification by students in the X-ray Image Processing Group. In this appendix, we outline the differences between the original HAPPI and version 2.0.

The most significant improvement to HAPPI in version 2.0 is the implementation of a multitasking capability. Separate UNIX processes handle the menu functions and image processing functions. Version 2.0 still only executes one processing routine at a time, but the user is able to use other HAPPI functions while an image is being processed, and may cue up several processing routines for sequential execution before the currently executing routine is completed.

---

[1] SPARCstation is a trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc.

HAPPI 2.0 has a revised image data structure, which now includes information on the parent image, and, if applicable, the coordinates within the parent image from which an image was extracted. All images are now internally represented with floating-point pixel values (using the C *float* data type) for purposes of processing. Support for additional image data types and for the inverse FFT, as discussed in Subsection 3.3.3, is implemented in HAPPI 2.0. Also included is a new menu of image data type conversion routines for easy manipulation of the various formats.

HAPPI 2.0's user interface was written using the Open Look[2] user interface. As discussed in Subsection 3.3.4, the calls to the original version of HAPPI's custom-written windows toolkit have been removed in version 2.0, and replaced with calls to the Open Look toolkit. This significantly changes the look and feel of HAPPI. Other changes to the user interface include a reorganization of the menu structure and behavior; function groupings have been changed, and pop-up menus are no longer allowed to obscure images. Also, the sometimes awkward "value window" discussed in Subsection 3.3.1 has been eliminated in favor of a simpler "dialog box" into which the user simply types the desired parameters.

Many desirable to changes to HAPPI identified in Section 3.3, and a few existing features of the original version, were not implemented in version 2.0 because of time constraints. No command-line interface was added. The pixel analyzer, histogram, and image slice graphics functions, and the macro language of the original version are not present in version 2.0. Also not implemented were: execution of HAPPI commands from the operating system

---

[2] Open Look is a trademark of AT&T

shell, entry of user-defined mask values, user-specific startup files, frequency-domain filtering, hardcopy direct from the program, additional data plot types, graphical/textual image annotation, visual macro programming, piecewise linear colormapping, and additional support for integrating end-user processing routines.

CHAPTER 4: EXTENDING HAPPI

4.1 Introduction

To be able to interpret results correctly, the image processing researcher developing a new algorithm must be in complete control of the algorithm's implementation, and so must write it completely from scratch or build it from subroutine libraries whose inputs and outputs are well-defined. However, to use the algorithm in a robust way, as a tool in an overall image processing scheme, it is useful for the researcher be able to use common algorithms (other than the one under development) as pre-, post- or intermediate processing steps without having to also write his/her own version of these common algorithms. Also, for the researcher to make his/her algorithm accessible to colleagues working in related research areas, it is helpful to have an easy-to-use interface to the algorithm; such an interface may propose default input parameters for the user, guide the user in selecting from the proper range of values for input parameters, and check the user's choice of input parameter values for correctness. These capabilities can be provided by integrating the researcher's algorithm into a pre-existing image processing software environment. To integrate his/her algorithm into an existing image processing environment and provide a robust user interface to the algorithm requires additional programming on the part of the researcher above and beyond the minimum requirement of writing the algorithm itself. The researcher might ask: How much of this programming overhead is justified to reap the benefits?

Experience with HAPPI has shown that, provided they are written using a few simple conventions, new algorithms may be added to HAPPI in anywhere from 45 minutes to 4 hours, depending on the complexity required of the user interaction with the algorithm and other factors, addressed in later sections of this chapter. (Note that this estimate does not include compile time; compile time is addressed in Section 4.11, "Putting it All Together".)

There is a continuing effort to develop image processing algorithms for NDE applications in the Electrical and Computer Engineering Department's X-ray Image Processing group at Iowa State University. Thus, this group has a need for an image processing software environment that is extensible and that provides the programmer with software tools for building a friendly user interface onto newly added algorithms. Extensibility was one of the primary design objectives of HAPPI, and as such, HAPPI addresses these needs for the X-ray Image Processing Group.

This chapter gives a procedure for integrating image processing routines into HAPPI. As HAPPI is a large program, the procedure given cannot anticipate all possibilities, and in practice will need to be supplemented by referring to the HAPPI Technical Manuals (Volumes 1 through 4) located in the X-ray Image Processing Group's laboratory. These manuals provide detailed information about individual tools available to the HAPPI programmer, and include all of the source code for the program as it stood at the end of the project's funding on June 30, 1990. At this writing, the source code referenced in this chapter resides in hard disk storage on the Stellar GS1025 graphics supercomputer in the X-ray Image Processing

Group's laboratory. The hostname of this computer is "fuji.ee.iastate.edu" (with Internet address 129.186.5.211), and the source code is located in the directory */home/catd/src*. As this directory's access is restricted, it will be necessary for system users who wish to modify HAPPI to contact a system administrator to request write priveledges for the directory.

4.2 Required Programming Background

The main requirements for adding image processing code to HAPPI are proficiency in the C programming language and basic familiarity with the UNIX operating system (e.g., ability to log on to the system, edit, move, copy and rename files, and traverse the directory hierarchy). In particular, a good grasp of the following programming concepts is essential to extending HAPPI: Data types and type conversions, C function declarations, function return values, looping and decision constructs (especially the *switch* construct), pointers and arrays, the C preprocessor, structures, unions, enumerated data types, and dynamic memory allocation/deallocation. Brief explanatory remarks summarizing important concepts are included throughout to help the reader unfamiliar with C follow the discussion. However, it is beyond the scope of this document to provide a tutorial on C that will bridge the gap for the non-C programmer. The reader is referred to the bibliography for a sample of the many C and Unix texts available. These references, particularly Kernighan and Ritchie (1988), should be consulted for formal definitions of the C concepts mentioned in this document. The Kernighan and Ritchie text is considered to be the de facto specification of

the C language. Basic familiarity with the UNIX source-code-level debugger *dbx* is helpful for, but not essential to, the integration of new routines into HAPPI. A brief example of how to use *dbx* is given in Section 4.11. In addition, experience with using HAPPI will help the programmer to better understand the flow of program control and to anticipate the effects of each line of code in his/her programs.

4.3 Flow of Control in HAPPI

As previously discussed in Chapter 2, HAPPI's user interface is organized into a Main Menu and an Information Window (both of which are always displayed), and a set of Submenus and other various graphical windows (which are only displayed when activated by the user). We now describe the flow of program control behind HAPPI's user interface, particularly for HAPPI's *Image Processing* Main Menu item.

When HAPPI is started, HAPPI's X Windows environment is set up, and all static data structures (those that remain constant for the entire time that the program is running), such as menu text, are initialized. HAPPI's main routine then draws the Main Menu and Information Window, and enters a loop waiting for mouse input from the user. The user must select one of the Main Menu items by positioning the mouse cursor over the item and clicking the left mouse button. Associated with each of the Main Menu items is a corresponding *C function* (i.e., a block of C code to which arguments may be passed; see the references on C for a formal definition of a C function) known as a "manager", which draws and removes submenus under the Main Menu

items, accepts and evaluates user input, and manages the different HAPPI functions grouped under that Main Menu item. We will refer to these managers as "menu managers" to distinguish them from a different type of manager to be discussed later in this section. Thus, the *Image Processing* Main Menu item has associated with it an "Image Processing menu manager", and similarly for the other Main Menu items. When the user selects a Main Menu item, program control is transferred to the appropriate menu manager. This chapter will discuss only the Image Processing menu manager and its various subordinate managers, as these are the only managers that need concern the programmer adding new image processing algorithms to HAPPI. The flow of control at the highest level in HAPPI is illustrated in Figure 4.1.

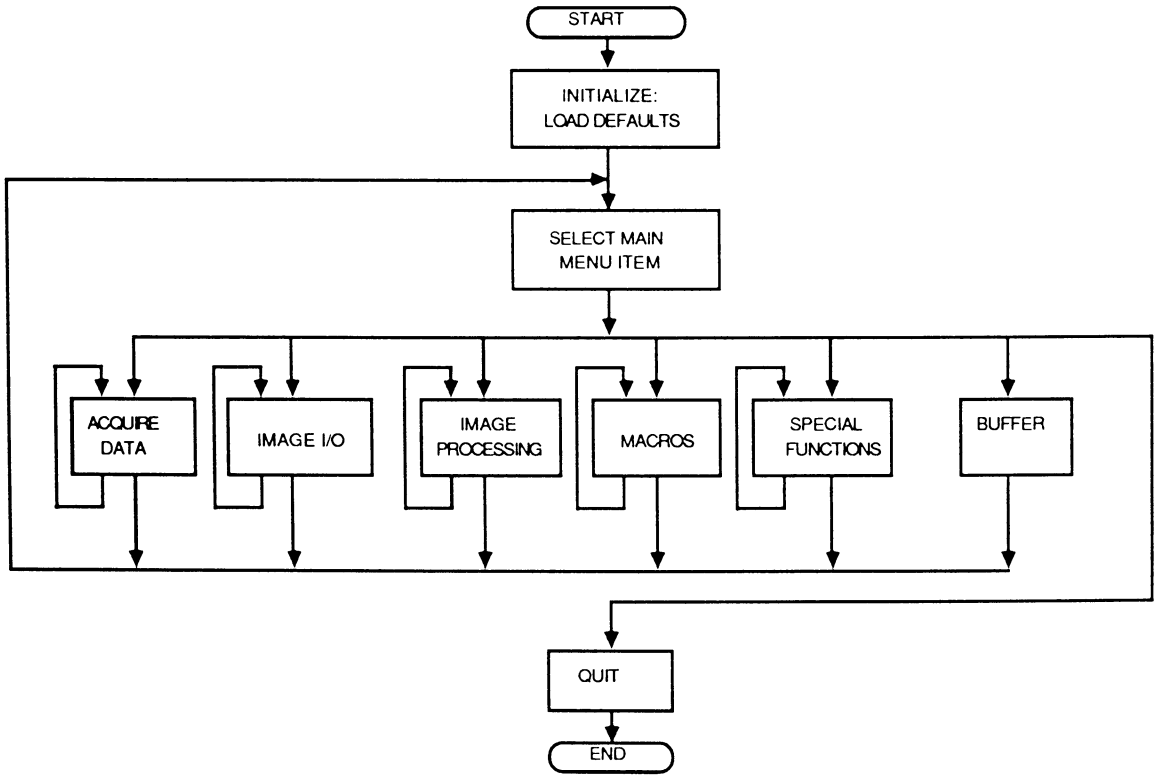


Figure 4.1. Flow of control in HAPPI at the highest level

HAPPI's image processing functionality is built in several layers. In the present version of the source code, there is an unfortunate similarity between the name of the Image Processing menu manager and the name of one of its subordinate manager functions which does not manage menus, but rather executes calls to individual image processing algorithms, and the programmer is cautioned against getting the two confused. The top layer of HAPPI's image processing functionality, the Image Processing menu manager, is a function called *Img_Process_Manager()* (note: the parentheses appended to the function name are how the C language indicates that something is a function as opposed to, say, a variable), which is called from the main program loop when the user selects the *Image Processing* Main Menu item. The menu manager *Img_Process_Manager()* displays the "Image Processing" submenu under the *Image Processing* Main Menu item and enters a loop waiting for additional mouse input from the user.

The user may then select one of several categories, or *classes*, as termed in HAPPI's source code, of image processing algorithms from the Image Processing submenu. There is a separate menu manager which in turn handles each of the image processing classes. For example, if the user selects the *Noise Filters* Image Processing submenu item, control is passed from *Img_Process_Manager()* to the subordinate menu manager *Noise_Filters_Manager()*. Each menu manager for an image processing class displays a sub-submenu whose items are the image processing routines for its particular class, and similarly enters a loop waiting for additional mouse input for a sub-submenu selection.

For some of the managers, there are additional submenu layers. For example, when the user selects the *Image Analysis* menu item from the Image Processing Menu, control transfers to the function *Img_Analysis_Manager()*, which displays the sub-submenu for the class Image Analysis and waits for mouse input from the user for a particular sub-submenu selection. If the user selects *Image Measurement* from this sub-submenu, control is then passed to another subordinate menu manager, *Img_Measurement_Manager()*, which displays a sub-sub-submenu of image measurement menu selections and waits for mouse input for a particular selection. The extension to deeper nesting levels of additional submenus is similar.

The *switch* construct in C provides selective execution of multiple functional blocks of code based on a single condition (see the C references for a formal definition of *switch*) as follows: The integer expression in parentheses following the keyword *switch* is evaluated, and the list of *case labels* following the *switch* is examined one by one until the constant integer expression following the word "case" in the case label matches the value of the integer expression following *switch*, whereupon all code following the matching case label is executed. The function *Img_Process_Manager()* and all of its subordinate menu managers (e.g., *Noise_Filters_Manger()*, etc) taken together may be conceptualized as a large nested *switch* construct as illustrated in Figure 4.2. All text between "/*" and "*/" is a comment and not part of the code (e.g., /* This is a C comment */). The outer layer of this switch construct, *Img_Process_Manager()* "switches" on the user's selection from the Image Processing submenu, executing code following the matching

case label, which passes control to a subordinate menu manager that is itself essentially a switch construct and which in turn switches on the user's selection from the sub-submenu displayed by the subordinate menu manager. When the user selects a menu item that is at the bottom of the menu hierarchy (i.e., has no submenus beneath it), the appropriate image processing algorithm is called by the manager which currently has program control.

When the user selects a particular algorithm for execution, the menu manager which currently has program control, as part of the "Code to execute algorithm" shown in Figure 4.2, prompts the user for the input image(s), and, in some cases, performs some error checking on these images when the selected algorithm requires input images of particular dimensions or data types. In most cases, the manager then calls the selected algorithm through a call to the function *IP_manager()*. The reader is cautioned against confusing the function *IP_manager()* with the function *Img_Process_Manager()*, as mentioned above. *IP_manager()* is different from *Img_Process_Manager()* and its subordinate menu managers in that, among other things, *IP_manager()* does not itself display further menus and switch on user input. The function *IP_manager()* and all of its support functions (to be discussed below) will be collectively referred to as the "Image Processing Manager", as distinguished from the image processing *menu* manager, *Img_Process_Manager()*.

The Image Processing Manager is itself structured in three main layers. The outermost layer, *IP_manager()* itself, prepares for execution of an image processing algorithm by setting up a return location in its code to

which control will return in the event that an image processing algorithm is aborted. *IP_manager()* then calls, in sequence, two other functions, *GetParams()* and *CallIP()*, which constitute the second layer of *IP_manager()*. The third layer of *IP_manager()* consists of various support routines called by *GetParams()* and *CallIP()*.

```
/* Top level: Img_Process_Manager() */
        {
        :
        :
        switch(User selection from Image Processing submenu)
                {
                case (1st submenu item) :
/* 1st submenu manager */
                        switch(User selection from sub-submenu 1)
                                {
                                case (1st sub-submenu item) :
                                        {Code to execute algorithm}
                                case (2nd sub-submenu item) :
                                        {Code to execute algorithm}
                                :
                                :
                                case (i-th sub-submenu item) :
                                        {Code to execute algorithm}
                                default:
                                        {Default code}
                                }
                case (2nd submenu item) :
/*2nd submenu manager */
                        switch(User selection from sub-submenu 2)
                                {
                                case (1st sub-submenu item) :
                                        {Code to execute algorithm}
                                case (2nd sub-submenu item) :
                                        {Code to execute algorithm}
                                :
                                :
                                }
                                etc.
                }
        }
```

Figure 4.2. Nested switch construct structure of *Img_Process_Manager()*

The function *GetParams()* prompts the user for input parameters for the selected algorithm and places the user's inputs into a global "parameter block"; the parameter block is called *IPparam*, and is a C *structure* variable. Structure variables are compound data types that are custom-defined by the programmer. Structure variables are used to group together several pieces of data - usually when these data are of different types (e.g., integer, character, and floating point) - as a single entity. For example, in a program to keep track of hospital patients, a programmer might define a "patient" structure variable, which groups together different types of information about a patient, such as name, address, blood type, height, weight, etc. using a collection of integer, character, and floating point data. The fields, or "structure members", as they are termed in C, in IPparam are a series of character, integer, floating-point, image, and array variables which store the current default input parameters for all of HAPPI's image processing algorithms. The function *CallIP()* makes the call to the actual image processing algorithm, passing the input parameters from the global parameter block, IPparam, to the algorithm. This layered structure of the Image Processing Manager was chosen to facilitate flexibility in implementation of HAPPI's built-in macro language; the macro language can bypass the parameter fetching, calling the image processing algorithms directly. Two arguments, the *class* and *subclass* of the processing algorithm to be executed are passed to *IP_manager()*. The *class* refers to the submenu under which an algorithm is found, and the *subclass* refers to the particular algorithm from that submenu. Thus, for example, the "Mathematics" selection from the Image Processing menu is an example of a class, and the "Add

Images" selection from the Mathematics menu is an example of a subclass. An optional third argument, *subclass2*, may also be passed to *IP_manager()*, but it is currently not used. Both *GetParams()* and *CallIP()* are passed the *class*, *subclass*, and *subclass2* arguments from *IP_manager()*, and "switch" on these arguments in much the same way as the menu managers switch on the user's mouse input.

GetParams() switches on the *class* argument, and calls a support routine for that particular class, passing the value of *subclass* to the support routine. The names of the support routines are derived from the name of the particular class and prefixed with "P_" (the "P" derives from "parameter fetching"); for example, the support routine for the class mathematics is called *"P_Math()"*. The support routine called by *GetParams()* for the particular value of *class* then switches on the value of *subclass* passed to it and calls the specific parameter fetching routine for the selected class and subclass. The parameter fetching routine then displays menus and windows for the user to enter input parameters, and updates the global parameter block *IPparam* to reflect user input. When the user is satisfied with the values of the input parameters for the selected algorithm, and clicks "OK" on the parameter menu displayed by the parameter fetching routine, program control is transferred from the parameter fetching routine back up the hierarchy to the support routine, then to *GetParams()*, and back to *IP_manager()*.

Once control returns from *GetParams()*, *IP_manager()* calls *CallIP()*, again passing the value of the *class* and *subclass* arguments. Within *CallIP()*, a "destination image" is created and given a default name based on the name

of the "source image" for the routine. The "source image" is the image data structure that has been selected by the user as an input image for the algorithm to be executed, and the "destination image" is the image data structure in which the output image from the algorithm will be written. Until a processing routine has successfully (without encountering an error or being aborted by the user, for example) completed, the destination image is considered a temporary entity, to be immediately erased on unsuccessful completion of a processing routine. The variables *temp_img* and *temp_img2* are defined in the file Globals.h as global image pointer variables, and are used to point to the destination image(s) during image processing; their values are then assigned to destination image pointers in the global parameter block *IPparam* only after a processing routine completes successfully. The image pointer *temp_img* is assigned to point to the first (temporary) destination image within *CallIP()*. If a processing routine produces two output images, the second destination image is created in a support routine (to be discussed below) called by *CallIP()*, and in the support routine, the image pointer *temp_img2* is assigned to point to the second (temporary) destination image. After creating the first destination image and assigning its address to the pointer *temp_img*, *CallIP()* then "masks out", or suppresses mouse input and activates abort trapping; the mouse input masking is done to prevent buildup of useless mouse input during execution of an image processing algorithm. The activation of abort trapping allows the user to cancel an image processing operation in the event that, for example, the processing takes longer than the user is willing to wait, or the user accidentally started the processing with incorrect input.

Next, *CallIP()* switches on the *class* argument and calls a support routine for the particular value of *class*, passing the value of *subclass* to the support routine. The names of the support routines are derived from the names of the corresponding classes and prefixed with a "C_" (the "C" derives from "Call image processing routine"). For example, the support routine for the class mathematics is called *"C_Math()"*. The support routine called by *CallIP()* for the particular value of *class* then switches on the value of *subclass* passed to it and calls the specific image processing routine for the selected class and subclass. If the processing routine to be called produces two output images, the code following the case label for the particular routine creates the second destination image before calling the actual routine. Input parameters of the algorithm are passed to the routine as function arguments rather than having the individual routines read the parameters directly from the global parameter block; this was done to allow the macro language to work more flexibly with the image processing routines. On successful return from an image processing routine, control returns back up the hierarchy to the support routine, *CallIP()*, *IP_manager()*, the menu manager which called *IP_manager()*, *Img_Process_Manager()*, and hence back to the main loop.

Some additional tasks are performed on the way back up the hierarchy. For example, before *CallIP()* returns control to *IP_manager()*, it deactivates abort handling, "masks in" (i.e., stops suppressing) mouse input, and assigns the value(s) of the temporary image pointer(s) *temp_img* and *temp_img2* to the destination image pointers in the global parameter block IPparam. Also, before *IP_manager()* returns control to the menu manager which called it, it either scales or clips the destination image according to the values of two flag

variables, the overflow and underflow flags, in IPparam. The destination image is created with pixels of data type *int*, which is (on most computer systems) a 2-byte signed integer and may thus take values from approximately -32000 to +32000. This is done to prevent the wraparound errors that may occur when storing the results of operations on 1-byte numbers in a 1-byte variable. For display as a grey-scale image, the pixels of the destination image must be converted to the type *unsigned char* which is (on most computer systems) a 1-byte unsigned integer and may thus take values from 0 to 255. The conversion may be done either with scaling or clipping. Also, on successful return from *IP_manager()*, the menu manager which called *IP_manager()* adds the destination image to a data structure known as the "image buffer" which keeps track of all images in memory and whether or not they are currently displayed on the screen. The function which adds the destination image to the image buffer also automatically displays the image on the screen.

The hierarchical structure of HAPPI's image processing functionality is summarized by Figure 4.3. The figure is interpreted as follows: The first two layers represent the Image Processing menu manager and its subordinate menu managers, respectively. The remaining layers represent the structure of the Image Processing Manager. The top of the figure corresponds to the highest level of hierarchy. Program control passes both up and down between blocks, but does not cross vertical lines. Thus, for example, program control does not pass directly from *GetParams()* to *CallIP()*, but rather passes up to *IP_manager()* from *GetParams()* and then back down to *CallIP()*.

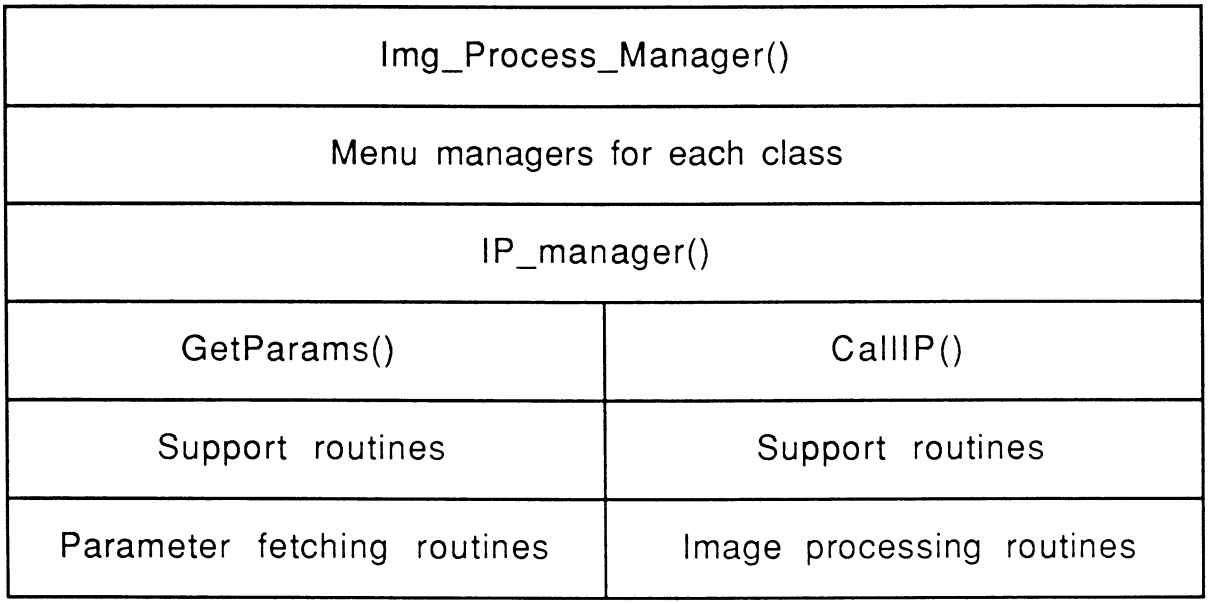


Figure 4.3. Hierarchical structure of HAPPI's image processing functionality

4.4 HAPPI Data Objects

In order to access, move, and alter data efficiently, HAPPI makes liberal use of the data type definition capabilities of the C language. Several *structure*, *union*, and *enumerated* data types are defined within HAPPI to allow the programmer to refer to complex data objects, such as images and convolution masks, with a single variable name. This section will discuss the data structures defined in HAPPI that are relevant to the addition of image processing routines to the program. These include images, templates, the global parameter block, the image buffer, and the "class" and "subclass" variables that are passed to *IP_manager()*.

HAPPI's image data structure includes not only pixel intensity values, but a variety of other information as well, including the image's name, its processing history (a data structure nested within the image data structure which records all processing steps that have been performed on the image since the raw image data was acquired), its height and width, the data type of its pixels (e.g., 1-byte unsigned integer, 2-byte signed integer, floating-point, etc.), its global statistics (e.g., max, min, mean, etc. of the entire image), a set of flags indicating which of the above fields are defined (i.e., have valid data), and a flag indicating whether the image has been saved to disk or if it exists only in volatile memory. The definition, or *structure template*, for HAPPI's image data structure is found in the file "images.h". The *typedef* statement in C is used to establish another name for a data type, and is most often used to define shorthand names for programmer-defined data types such as structure and union variables. A typedef statement is used in images.h to define the word IMAGE (note that C is case-sensitive, so that "IMAGE" is different from "image" in C) as data type "pointer to an image structure variable". This means that when we make a declaration such as:

```
IMAGE        s_image1;
```

in a piece of code we are writing for HAPPI, the variable *s_image1* is declared as a pointer to an image structure variable. A pointer is simply a variable which holds the *address* of a piece of data; thus, *s_image1* in the above example holds the address of, and thus "points to", an image structure variable. The reader is referred to the file images.h (or to a hardcopy of this

file in the section "Headers" of the HAPPI Technical Manual, Volume 1) for a complete definition of HAPPI's image data structure. All defined data types that are used as structure members of the image data structure are also defined in the file images.h, with the exception of the defined type HISTORY, which is defined in the file "history.h". The HISTORY defined data type is a special kind of structure variable known as a *linked list node*, and is used to store a single entry in the image structure's history structure member.

It is not necessary for the programmer to work directly with the structure members of an image structure variable, as a number of utility functions, discussed in the next section, are provided in HAPPI for reading from and writing to the image data structure. The reader should, however, refer to the image structure definition in the file images.h when there is any question as to the data types of the various structure members.

Convolution masks, or "templates" as they are called in HAPPI's code, are defined as structure variables in much the same way as images. The word TEMPLATE is defined with a typedef statement as data type "pointer to template structure variable", and so when we write a declaration such as:

```
TEMPLATE    lowpass;
```

we are declaring the variable *lowpass* as a pointer to a template structure variable. The structure definition for templates is also found in the file images.h. As HAPPI's template structure variables are not as large as its image structure variables and are less commonly used in the program, there are no utility routines for manipulating template structure variables; the

programmer must access the structure members of template structure variables directly in his/her code. For example, to reference the "hot_row" member of a template structure variable which is pointed to by the pointer variable *lowpass*, we would write "lowpass -> hot_row", where we have used the structure member notation *pointer_to_structure_variable -> structure member*. (Note the difference between this example and the structure member reference in the previous example using the "dot" notation; the dot is used with structure variables, while the "minus-sign-greater-than-sign" notation, ->, is used with *pointers* to structure variables.) The *hot_row* and *hot_col* members of the template structure refer to the row and column numbers, respectively, of the origin of the template and thus determine which template cell which will serve as the convolution sum accumulator in convolution operations. The *size* member refers to the number of rows or columns of the (square) template. The *kind* member refers to the data type of the template weights, integer or floating point. The *int_template* and *float_template* members are pointers to integer and floating-point matrices, respectively, of the template weights themselves. The *denom* member refers to "denominator"; in HAPPI's convolution routine, when a convolution sum is accumulated into the accumulator cell, it is divided by the value of *denom* before being written into the destination image.

As discussed in the previous section, input parameters for all of HAPPI's image processing algorithms are stored in the global parameter block *IPparam*. This structure variable is declared in the file "Globals.h". The programmer adding image processing routines to HAPPI will likely need to add to this structure definition. The *structure tag*, that is, the name used as

shorthand for the *structure template*, is called *"param"*. The *structure template* is simply the list of structure members and their data types. The two declarations shown in Figure 4.4 are excerpted from the file Globals.h, and are an abbreviation of the declaration of IPparam.

```
struct      param {
            IMAGE       s_image1,
                        s_image2,
                        d_image1,
                        d_image2;
            char        peak_name[20],
                        conv_name[20];
            int         max,
                        min,
                        :
                        :
                        bmsize;
            long int    seed;
            short       overflow_flag,
                        :
                        :
                        structure;
            char        logic_val,
                        :
                        :
                        fit_type;
            float       snratio,
                        :
                        :
                        bmvar;
            TEMPLATE    conv_temp1;
            };

extern struct  param  IPparam;
```

Figure 4.4. Abbreviated declaration of global parameter block in Globals.h

The line beginning with *struct* is the start of the structure declaration which assigns *param* as a structure tag for the structure template, which consists of everything between the left and right curly braces (e.g.,

{everything in here is a structure template}). The line beginning with *extern* declares IPparam as a structure variable using the template referred to by the tag *param*. The modifier *extern*, which stands for "external", makes the IPparam structure variable accessible from all parts of the program. Thus, from any point in HAPPI's code, we may refer to the image pointer *s_image1* in the global parameter block using the *structure member notation* "IPparam.s_image1". Similarly, we would refer to the integer *max* using the notation "IPparam.max".

The data types of the *class* and *subclass* variables which are passed to *IP_manager()* are "enum", that is, they are enumerated types. The data type *enum* allows the C programmer to conveniently assign a set of descriptive names to the integer values that may be taken on by an integer variable. These assigned names may then be used in relational tests and assignment statements involving the variable. As an example, consider the following declaration:

```
enum {no, yes} answer;
```

The variable *answer* is an integer variable, however, after writing the above declaration, we may compare it to or assign it the values 'yes' and 'no'. Thus, we may write:

```
answer = yes;
```

instead of:

```
answer = 1;
```

and:

```
if(answer == yes)
{execute this code}
```

instead of:

```
if(answer == 1)
{execute this code}
```

The C compiler assigns integer values to the enumerators 'yes' and 'no', but this is transparent to the programmer. The integer values in the *enumerator list* (everything between the curly braces) may be explicitly specified by the programmer, if desired. This option is used in HAPPI. Two *typedef* statements are used in the file Globals.h to define the words IP_CLASS and IP_SUBCLASS as enumerated types for the entire list of image processing algorithm classes and subclasses, respectively, and integers are explicitly assigned to each class and subclass name. The data type declarations:

```
IP_CLASS    class;
IP_SUBCLASS subclass;
```

thus declare the variables *class* and *subclass* as enumerated types with the enumeration lists defined in Globals.h. The programmer adding processing

routines to HAPPI will need to add to this enumeration list and so should familiarize him/herself with it.

The image buffer is a data structure in HAPPI which holds image structure pointers and keeps track of the display status of all images in memory (images loaded into the computer system memory by HAPPI may be displayed on the screen or "hidden" from view to reduce screen clutter). The image buffer is a structure variable named "buf", consisting of three arrays: an integer array, called *index[]*, of indices used to assign a unique number to each image in the buffer, an array of image pointers, called *images[]*, (of defined data type IMAGE), and a character array of status indicators, called *valid[]*. (In C, arrays are addressed using the array offset in square brackets following the array name.) As character variables are treated as 1-byte integers in C, the display status indicators in the image buffer may be assigned integer values between -128 and 127. A value of 0 is assigned to a status indicator if the entry in the buffer is empty or has been vacated by an image that has been removed from memory. A value of 2 is assigned to the indicator if the image is currently in memory and displayed; a value of 4 is assigned if the image is currently in memory but not displayed. The structure member notation buf.index[i] then refers to the index of the image in the i-th location of the image buffer; buf.images[i] refers to the image pointer in the i-th location of the image pointer, and buf.valid[i] refers to the status indicator of the image in the i-th location of the image buffer. The image buffer is a global variable, and is declared in the file Globals.h. The programmer adding image processing routines to HAPPI may occasionally find it necessary to search the image buffer to retrieve an image pointer

corresponding to an image index; image indices are returned by certain utility functions in HAPPI.

4.5 Tools for Manipulating Image Data

As mentioned above in the discussion of HAPPI's image data structure (Section 4.4), a number of utility functions are provided within HAPPI to manipulate image structure variables. This section will give a general discussion of these utilities; the reader is referred to the "Image Operations" sections of the HAPPI Technical Manual, Volume 1, for the details. The source code for all of these functions is found in the file "iops.c".

The first group of functions to be described will be referred to as the image structure member read/write/test functions, or "image structure access functions" for short. Recall that HAPPI's image data structure includes, among other things, the image's name, height and width, and a set of global image statistics. The image structure access functions are used to read from, write to, and test the validity of members of the image data structure. We may test the validity of the global image mean, for example, using the image structure test function *test_mean()*. This function is passed an image pointer (data type IMAGE) and returns a long (4-byte on most computer systems) integer equal to 0 if the image mean is not defined, and not equal to 0 if it is defined. Similarly, the function *get_mean()* is passed an image pointer, and returns the image mean as a float value. If the value of the image mean has not been calculated and initialized, the value returned by *get_mean()* will be meaningless (no pun intended). Thus, if we want to

retrieve the mean value of an image, we should first test its validity with *test_mean()*. The function *put_mean()* is passed an image pointer and a floating-point value, and writes the floating-point value to the image mean structure member. The image structure access functions are all named similarly, with prefixes "get_", "put_", and "test_" for the functions to read, write, and test, respectively, the various image structure members.

Another group of functions that accesses image data structures are the statistics calculating functions. These functions are passed an image pointer, and calculate the values of the various image statistics defined in HAPPI's image data structure, then write the calculated values to the appropriate image structure members. For example, the function *find_min_max()* is passed an image pointer, and finds the global minimum and maximum of the image and writes these values to the image structure members *min* and *max*, respectively.

Memory space for image structures is allocated and deallocated by the image allocation functions. The functions *grey_scale_image()* and *large_scale_image()* allocate memory for images with 1-byte unsigned integer pixels and 2-byte signed integer pixels, respectively, and initialize all fields to undefined values. The image thus created may then have data assigned to it, for example from the output of an image processing algorithm. The function *dispose_of_image()* is passed a pointer to an image pointer, and frees the memory used by the image, making it available to the host computer system again. This function does not take care of saving the image, so care should be exercised in its use. The functions *create_matrix()* and *remove_matrix()* allocate and deallocate, respectively, the memory for image

pixel data. Recall that the image pixel data may be one of several types; the structure member *kind* in the image structure indicates this data type. These two functions examine the *kind* structure member of the image, using an image structure access function called *get_kind()*, to determine how much memory to allocate/deallocate.

Three image pixel data type conversion routines, *convert_grey_to_large()*, *convert_large_to_grey_by_clip()*, and *convert_large_to_grey_by_scale()* are provided in HAPPI. Recall that before *CallIP()* makes the call to the image processing routine selected by the user, it creates a "destination image" where the routine writes its output. (Note: In some cases, a second destination image is required; the second destination image is usually created in the support routine called by *CallIP()*.) The destination image is created as what is called a "large scale image" in HAPPI; a large scale image is simply an image whose pixels are 2-byte signed integers (and may thus take values from approximately -32000 to 32000). This is done to avoid overflow and/or underflow within image processing algorithms. When a destination image is to be displayed, however, it must be converted to what is called a "grey scale image" in HAPPI; a grey scale image is simply an image whose pixels are 1-byte unsigned integers (and may thus take values from 0 to 255). The pixel data type conversion routines provide the conversion capabilities needed to work with the above two types of images.

Four image copy functions, *copy_image()*, *copy_image_header()*, *copy_partial_matrix()*, and *copy_matrix()*, are provided to copy various amounts of an image structure to another image structure. These functions

are currently only used by the macro language, but may be used within image processing algorithms as well.

4.6 Image Processing Support Functions

A number of functions routinely needed in image processing algorithms are provided in HAPPI by a set of image processing support routines. Such functions include memory allocation and deallocation for matrices and vectors, clearing and setting the overflow and underflow flags, random number generation, general n-dimensional forward and inverse Discrete Fourier transforms, matrix inversion, max and min operations, and sorting. This section gives a brief summary of these functions. All of the source code for these functions is located in the file IProutines.c, and documentation for these routines is in the "Support Routines" section of Volume 2 of the HAPPI Technical Manual.

The set of memory allocation and deallocation routines for matrices and vectors allows the user to create and destroy matrices and vectors of data types int, long (4-byte integer), and float. The allocation routines are named with the prefix "make_", followed by a letter 'i', 'l', or no letter (for int, long, and float, respectively), followed by the word "matrix" or "vector". For example, the routine *make_matrix()* allocates a matrix of float values; the routine *make_imatrix()* allocates a matrix of int values. The desired starting and ending indices of the matrix or vector are passed to the routines, and the routines return pointers to the appropriate data types. The ability to specify the starting and ending indices of matrices and vectors instead of just their

dimensions allows the programmer to use whatever array addressing scheme is most appropriate to the problem at hand. The names of the deallocation routines are similar to their allocation counterparts, with "make" replaced by "free"; the function *free_matrix()*, for example, frees a float-valued matrix. The return types of these functions are declared in the file "IProutines.h".

The four functions *clear_underflow()*, *clear_overflow()*, *set_underflow()*, and *set_overflow()* clear and set the underflow and overflow flags in the global parameter block. Recall that the values of these flags determine whether the large-scale destination image is clipped or scaled for display. The programmer would use these routines in an algorithm, for example, by examining the destination image for values outside the grey-scale image range of 0 to 255 and setting the flags according to the desired action before passing control back to the calling routine. If either the overflow or underflow flag is set, the destination image will be scaled for display; otherwise it will be clipped. If the destination image was required to always be scaled for display, the programmer would unconditionally set one or both flags in his/her code.

A group of mathematics routines rounds out HAPPI's image processing support functions. The routine *matrix_inverse()* calculates the inverse of a float-valued matrix of arbitrary dimensions using L-U decomposition and backsubstitution. The routine *fourn()* performs a general n-dimensional radix-2 forward or inverse fast Fourier transform. The routine *gamlog()* returns the natural log of the gamma function. The routine *gasdev()* returns zero-mean unity-variance gaussian deviates. The functions *max()* and *min()* return the maximum and minimum, respectively, of an arbitrary number of

integer arguments. The function *qcksrt()* implements the Quicksort algorithm. The functions *rand_u()* and *rand_p()* return uniform and Poisson deviates, respectively. Except for *max()* and *min()*, the above mathematics routines are taken directly from, or adapted from, Press *et al.* (1988).

4.7 A General Image Processing Routine Code Template for HAPPI

As may be seen from the preceding discussion, HAPPI's image processing routines lie near the bottom of the hierarchy of the program's flow of control. Because of this, they are largely isolated from, and function independently of, the rest of the program. As mentioned before, this independence of image processing routines from the rest of the program was designed into HAPPI for flexibility in implementing the built-in macro language. The independence of HAPPI's image processing routines from the rest of the program also makes it relatively straightforward to code an algorithm for integration into HAPPI. This section presents a general image processing routine code template for HAPPI.

The conventions, function calls, preprocessor control lines, and variable declarations used to code an algorithm for HAPPI are illustrated in the following code template in Figure 4.5. (The reader is cautioned not to confuse our use of the word "template" here with previous references to convolution templates. By "code template" we mean a generic, model piece of source code which is to be modified and added to by the programmer to generate image processing source code modules.) Note that the line numbers on the left side of the figure (and in all subsequent example code in this

document) are not part of the C source code file; they are included only so that each line of code may be referenced conveniently in the discussion. Also note that since lines of C code are terminated with a semicolon, a single long line

```
#include  "constants.h"
#include  "Globals.h"
#include  "errors.h"
#include <stdio.h>
#include <signal.h>
#include  "IProutines.h"

1       void    Newroutine(s_image1,s_image2,d_image,arg1,arg2,arg3)
        /*      Comments describing the routine     */
2       IMAGE           s_image1,
3                       s_image2,
4                       d_image;
5       int             arg1;
6       float           arg2;
7       char            arg3;
{
8       GREY_SCALE_PIXEL    **s_array1,
9                           **s_array2;
10      LARGE_SCALE_PIXEL **d_array;
11      int                 height,
12                          width,
13                          i,
14                          j;
15      s_array1  =  get_grey_matrix(s_image1);
16      s_array2  =  get_grey_matrix(s_image2);
17      d_array  =  get_large_matrix(d_image);
18      height  =  get_height(s_image1);
19      width  =  get_width(s_image1);
20      /*      Your code goes here          */
}
```

Figure 4.5. General image processing algorithm code template for HAPPI

We now discuss the code template line by line. The lines beginning with "#include" are preprocessor control lines which simply tell the C preprocessor to copy the contents of the named files into the source code at the location of the #include line before attempting to compile the code. The files named in the #include lines contain the data type definitions and function declarations necessary for the C compiler to make sense of the function calls and type declarations used in the rest of the source code. Line 1 begins the function header (i.e., the function name and the type declarations of its arguments); many of the image processing routines in HAPPI are declared as type *void* (meaning that the function itself does not return a value), however, if the programmer wishes to return a value, say, an error code, he/she should declare the routine as an *int*. The name of the routine is, appropriately, "Newroutine", and its argument list, including two source images, a destination image, and three algorithm parameters, follows in parentheses. The names of HAPPI's image processing routines begin with a single capital letter by convention; this is not a requirement, but helps programmers recognize an image processing routine as such. Lines 2 through 4 declare the first three arguments to *Newroutine()* as data type IMAGE, which, recall, is a pointer to an image structure variable. Lines 5 through 7 declare the algorithm input parameters *arg1*, *arg2*, and *arg3* as integer, floating-point, and character variables, respectively. The argument list of *Newroutine()* is representative rather than definitive. The programmer should add or delete arguments of the required data types as appropriate. The programmer should place all of the algorithm's required input parameters in the new routine's argument list; image processing

routines should neither need to access the global parameter block directly, nor should they prompt the user for inputs through the standard I/O device. Passing all algorithm parameters in the function argument list insures that the routine is truly a "black box" which, to do its job, needs only the set of arguments passed to it *from wherever in HAPPI* (e.g., Image Processing Manager or macro language) *it is called.* Exceptions to this rule include image processing routines which require the user to, for example, visually inspect the image and use the mouse to identify particular pixel coordinates to be used in the algorithm. In some of these cases, prompting for user input within the algorithm itself may be necessary; how to handle such cases is up to the programmer's judgement and creativity.

Lines 8 and 9 declare the variables s_array1 and s_array2 as data type "pointer-to-pointer to type GREY_SCALE_PIXEL". The defined data type GREY_SCALE_PIXEL is declared in Globals.h with a typedef statement as another name for the C data type *unsigned char.* This is the data type of a "grey scale image", as it is termed in HAPPI (Cf. Sections 4.4 and 4.5 above). Line 10 declares the variable d_array as data type "pointer to pointer to type LARGE_SCALE_PIXEL". LARGE_SCALE_PIXEL is also declared in Globals.h, as another name for data type *short int,* which is a signed integer of (at least, depending on the particular compiler) 2 bytes. This is the data type of a "large scale image".

Lines 11 and 12 declare integer variables to hold the values of image height and width, respectively. If the algorithm uses source and destination images that are all of the same dimensions, then only one set of such variables is needed; otherwise, the programmer should declare as many

additional such variables as the algorithm and input data dictate. Lines 13 and 14 declare a couple of general-purpose loop index variables; generally, at least one pair of these is needed for addressing the individual pixels of the source and destination image(s). Lines 15 and 16 make use of the image structure access function *get_grey_matrix()* to assign matrix pointers to the variables *s_array1* and *s_array2*, respectively. After execution of these lines, the programmer may reference the image pixel data in s_image1 and s_image2 using the notations s_array1[i][j] and s_array2[i][j], respectively (where *i* the row number and *j* is the column number, both indexed from zero). Line 17 performs a similar task, and after its execution, the programmer may reference pixel locations in the (large scale) destination image, d_image, using the notation d_array[i][j]. Lines 18 and 19 use the image structure access functions *get_height()* and *get_width()* to assign the height and width, respectively, of s_image1 to the variables *height* and *width*. At this point, we may use the variables *s_array1*, *s_array2*, *d_array*, *height*, and *width* to perform some image processing task, which may be generalized by the code fragment in Figure 4.6. The code fragment of Figure 4.6 would be placed in the code template of Figure 4.5 at line 20. This code fragment assigns to every pixel in the destination image a value that is some function of the input image data and the algorithm input parameters *arg1*, *arg2*, and *arg3*. In practice, most image processing routines will have more than three lines of processing code, but the code fragment of Figure 4.6 will likely be present in one form or another in any processing routine which returns a destination image.

```
for(i=0;i<height;i++)
   for(j=0;j<width;j++)
        d_array[i][j]  =  somefunction(s_array1,s_array2,arg1,arg2,arg3);
```

Figure 4.6. Code fragment generalizing image processing task

The coding of the algorithm itself past line 20 of the template of Figure 4.5 is, for the most part, independent of the rest of HAPPI, and is composed of pure C code and any functions written by the programmer. However, HAPPI's utility functions can and should be used to advantage to access image structure data fields and allocate/deallocate memory. The programmer is encouraged to examine the implementations of the algorithms already included in HAPPI for usage examples of the utility routines. The file "IProutines.c" contains the majority of HAPPI's image processing routines. The programmer is discouraged from defining any external (global) variables in his/her source code file; with more than one person modifying the program, it is easy to cause confusion when externals are declared in processing routines. If a global variable is deemed to be truly necessary, it should be declared in Globals.h.

To avoid unnecessarily recompiling existing image processing code, new image processing routines under test are usually placed in a separate source code file from the file containing routines already included in HAPPI. Three files currently in use are IPtest.c, IPtest2.c, and IPtest3.c. These files contain the #include lines of the code template of Figure 4.5, and the names of these files are included in the *"make file"* for HAPPI. The make file is part of the UNIX *make* utility program; once the programmer places his code in a file

that is specified in the make file for HAPPI, recompiling the program to test the new code is as simple as typing "make" at the UNIX prompt in the source code directory.

4.8 Handling Errors, I/O, and Other Details

In this section, we provide some tips and directions on how to return error codes, handle source and destination images of differing sizes, fetch graphical user input of positional information, and write output to HAPPI's information window.

Suppose that data-dependent error conditions may arise in an algorithm. Rather than returning a meaningless output image to the user, we would like to inform him/her of the nature of the error, so that more suitable data for the algorithm may be chosen. Error codes may be returned by all of HAPPI's image processing routines. By convention, HAPPI's error codes are all returned as negative integers. Thus, to return an error code from an image processing routine, the programmer would declare the routine itself as having the return type int (instead of *void*), and would return a negative integer on encountering the error condition in his/her code. The first 21 negative integers are individually defined as specific types of errors; the definitions of these error types may be found in the file "errors.h". Thus, for example, if a divide-by-zero error occurred, we could indicate this by returning the value defined for divide-by-zero errors, -1, or we could optionally use the preprocessor macro defined in errors.h for this error and write *return(DIVIDE_BY_ZERO);* in our code at the appropriate spot, letting

the preprocessor take care of the text substitution. Since all of the levels of the Image Processing Manager return integer values, the error code will be propagated back up the hierarchy of function calls in the Image Processing Manager until its value is examined and the error is handled. Errors are usually handled by the menu managers under the Image Processing menu manager (recall that it is these managers that call *IP_manager()*); typically, the menu manager will check to see if the returned error code is negative, and if it is, makes a call to a generic error display routine called *system_error()*, passing it the error code. The routine *system_error()* uses the error code to look up a string defined for the particular error code in the file "errors.c" and displays an error message to the user. Another error display routine, *display_error()*, displays a simple text message, and is for use in handling error conditions which do not have an error code defined in the file errors.c. Both of the error display routines, *system_error()* and *display_error()*, are found in the file errors.c. Other circumstances under which the programmer will likely want to use returned error codes are in memory allocation and matrix inversion; attempting to use unsuccessfully allocated memory will, at least, give meaningless results, and at most, will crash the program. The matrix and vector allocation routines described in Section 4.6 return a null pointer if the requested amount of memory cannot be allocated.

Implementation of some image processing algorithms may involve different sized source (input) and destination (output) images. For example, the radix-2 FFT in HAPPI accepts input images of arbitrary dimensions up to 512 by 512 pixels, zero-padding the input image to integer powers of 2

(independently in each spatial dimension), and outputs the magnitude of the frequency-domain image using the zero-padded dimensions. Since the destination image is created within the *CallIP()* routine of the Image Processing Manager with the same dimensions as the source image, routines that require different dimensions for source and destination images must destroy the destination image created by *CallIP()* and create their own destination image(s) with the required dimensions. An example of the code necessary to do this is shown in Figure 4.7. This code should be included, when necessary, in the processing routine itself as part of the "Your code

```
1       char   imgname[15];
2       get_name(d_image,imgname);
3       dispose_of_image(&d_image);
4       d_image = large_scale_image();
5       make_not_current(d_image);
6       put_name(d_image,imgname);
7       create_matrix(newheight,newwidth,d_image);
8       d_array = get_large_matrix(d_image);
```

Figure 4.7. Code fragment to destroy original destination image and create new one

goes here" section of the code template of Figure 4.5. The code fragment is discussed here line-by-line. Line 1 is not an executable statement, but rather a declaration of the string *imgname*, and is included for clarity. Line 2 fetches the name of the destination image from *d_image* and places it in *imgname*. Line 3 destroys the destination image that was created by *CallIP()* before program control was passed to the processing routine. Line 4 creates a new large scale destination image; the image pointer *d_image* points to the new image. Line 5 sets the destination image structure member *current* to a nonzero integer, to indicate that the new destination image is not current,

that is, it is not saved on disk. Line 6 places the name of the destination image taken from the old destination image and places it in the new destination image. Line 7 allocates the memory for the image pixel data in the new destination image using the new dimensions, *newheight* and *newwidth*. The new image dimensions are completely up to the programmer; typically, they are calculated from the input image dimensions. For example, the Fast Fourier transform routine uses the input image dimensions to calculate the destination image dimensions as the smallest integer power of two greater than or equal to the source image dimensions. Line 8 assigns the matrix pointer to the destination image pixel data to the pointer *d_array*, allowing the programmer to reference the pixel in the i-th row and j-th column of the destination image as *d_array[i][j]*.

For some processing applications, it is useful to allow the user to, while viewing an image, use the mouse to specify a particular point or region of the image as input to a processing routine. An example would be allowing the user to graphically "draw" the boundary of an image region which contains pure noise and no signal; the image data in this region may be used by the processing routine to calculate noise process statistics for use in filtering noise from the image. Such cases may be handled by using what will be called HAPPI's "rubberband utility functions", so named because they allow the user to draw a line or box on the computer display which is dynamically sized according to mouse input from the user. There are several rubberband utility functions defined in the file "Image_X1.c." Two of the most commonly used rubberband routines in HAPPI are *RubberBand()* and *RubberBandLine()*. The function *RubberBand()* is used to define a

rectangular region in an image. Upon invocation, the function is passed pointers to variables for the image buffer index, row and column position of the initial corner of rubberband box, and height and width of the box. The function monitors mouse input until it detects that the user has depressed and released the left mouse button, whereupon control returns to the calling routine, and the variables whose addresses (i.e., pointers) were passed to the function contain the desired information. The function *RubberBandLine()* is similar, but passes back, again by using pointers, the coordinates of the endpoints of the line specified by the user. Upon return from one of the rubberband functions, the programmer usually needs to search the image buffer to find the image pointer corresponding to the image index returned by the function. The reader is referred to the code for extracting arbitrary image cross-sections and subimages in the function *Img_Analysis_Manager()*, found in the file Managers.c, for examples of how to use the rubberband utility functions and of how to search the image buffer.

Text may be written to HAPPI's information window using the function *WriteInfoWindow()*, and individual lines in the information window may be cleared using the function *ClearInfoLine()*. These functions are both defined in the file "Info_X1.c". The programmer may use the information window to communicate a variety of information to the user, including: detailed prompts for input, displaying the current processing status of compute-intensive routines with long execution times, and issuing error messages for incorrect algorithm input parameters. Examples of the use of these routines may be found throughout HAPPI's code, especially in the menu managers for the

various image processing classes, found in the file Managers.c, and in the image processing routines themselves, which are found in the file IProutines.c.

4.9 User Interface Window Types and Management Tools

As fetching of input parameters is almost always done outside of HAPPI's image processing algorithms, a separate parameter fetching routine must be written for each image processing routine. The parameter fetching routine uses several types of windows to present a graphical interface to the user. This section discusses the types of windows typically used by the parameter fetching routines, and the utilities in HAPPI that create, alter, and destroy these windows.

The first thing done by the parameter fetching routine is to display the current default values (stored in the global parameter block) of the algorithm input parameters. These values are displayed in what is called a "menu window" in HAPPI. Menu windows display a list of strings, and have an identifiable space, or "sub-window", (with its own border) for each string in the list. HAPPI's main menu and submenus are all drawn using menu windows. Two routines which create and destroy menu windows are called *CreateAndDisplayMenu()* and *RemoveMenu()*, respectively; these routines are defined in the file "Menu_X1.c".

The parameter fetching routine uses what is called a "value window" in HAPPI to read user input of integer and floating-point input parameters. A value window is divided into seven sub-windows: a title sub-window

displaying the name of the parameter to be altered, a sub-window displaying the current value of an input parameter, a sub-window displaying the word "OK", and four sub-windows containing graphical "arrows". If the user clicks the left mouse button on one of the arrows, the parameter value currently displayed will be incremented or decremented by either a large or small amount, depending on which arrow the user clicks the mouse. The size of the large and small parameter increments and decrements is determined by the programmer. When the user is satisfied with the value of the input parameter, he/she clicks the left mouse button on "OK", the routine which created the value window destroys the window and exits, and the parameter entered by the user is placed in the global parameter block IPparam. Two routines used to create value windows for fetching integer and floating-point input parameters are *GetValueFromWindow()* and *GetFloatValueFromWindow()*, respectively. These routines are defined in the file "Value_X1.c".

As mentioned previously in Section 4.8, the programmer may use the Information Window at the bottom of HAPPI's screen to display prompts and useful information to the user. The information window may be accessed by the programmer from anywhere in HAPPI's code (using the functions *WriteInfoWindow()* and *ClearInfoLine()*). Thus, the programmer may also make use of the information window within the parameter fetching routine to, for example, notify the user of input parameter constraint violations.

For image processing algorithms that employ an entire two-dimensional array as an input parameter (e.g., convolution kernels, matched filter templates, and morphological structuring elements), HAPPI has

available to the programmer what are called "mask value windows". These windows allow the user to individually specify the elements of a two dimensional array for use in such algorithms. The routine *DisplayMaskMatrixMenu()*, defined in the file "Menu_X1.c", creates and displays a mask value window of programmer-defined size. Three higher-level routines, *user_binmask()*, *user_cmask()*, and *user_grmask()*, make use of *DisplayMaskMatrixMenu()* to fetch masks of different data types from the user. These three routines are defined in the file "IPparams.c".

One of the most important routines in HAPPI is the function *ActionMonitor()*, which is defined in the file "User_X1.c". *ActionMonitor()* is used extensively throughout HAPPI to monitor the user's mouse and keyboard activity. *ActionMonitor()* is called with three pointer arguments which point to *"index"*, *"action"* and *"value"* integer variables. The function does not return until the user enters a mouse button click or presses a key on the keyboard. Upon return from *ActionMonitor()*, the *index* variable contains the index of the window where the mouse cursor was located when the input was entered, the *action* variable contains the type of action detected (mouse button click or keyboard input), and the *value* variable contains the value of the input (which mouse button was pressed or which key on the keyboard was pressed). Documentation for this routine may be found in the "User" section of the HAPPI Technical Manual, Volume 4. Examples of the use of *ActionMonitor()* will be given in example code in later sections.

4.10 Writing the Parameter Fetching Routine

This section will discuss the issues involved in writing a parameter fetching routine for HAPPI, and will examine example code line-by-line. All of HAPPI's parameter fetching routines are similar in structure; the programmer may usually (and is, in fact, encouraged to) simply copy and modify an existing routine, or the code template to be discussed below, to cut down on the necessary typing.

All parameter fetching routines perform the following tasks: read the current default input parameters from the global parameter block, create and display two menu windows showing the names and current default values, respectively, of the parameters, enter a *while* loop in which *ActionMonitor()* is called to retrieve user mouse and/or keyboard input and in which the user's input is processed using a switch construct whose various cases each handle the modification of a single algorithm input parameter, remove windows created by the parameter fetching routine when the user is done modifying parameters, and exit.

Figure 4.8 is a general code template for parameter fetching routines for HAPPI. A variation of this template, ready for editing, may be found in the file "paramtempl.c".

```
1       #undef          NUMPARAMS
2       #define         NUMPARAMS  (an integer goes here)
3       void            routine_Param()
        {
4               char            value[NUMPARAMS + 1][15],
5                               *param_values[NUMPARAMS + 3];
6               int             m_width,
7                               m_height,
```

Figure 4.8. General code template for parameter fetching routine

```
8                       m_index1,
9                       m_index2,
10                      a_index,
11                      a_action,
12                      a_value,
13                      done;
14          static char *param_names[] = {
                        "Routine Name",
                        "1st Parameter Name",
                        "2nd Parameter Name",
                        :
                        :
                        "n-th Parameter Name",
                        ""};
15          CreateAndDisplayMenu(param_names,(NUMPARAMS + 1),500,500,
                &m_width,&m_height,&m_index1,'v');
16          param_values[0] = "Parameters";
            /* Code to initialize other elements of param_values[] */
17          param_values[NUMPARAMS + 1] = "OK";
18          param_values[NUMPARAMS + 2] = "";
20          CreateAndDisplayMenu(param_values,(NUMPARAMS + 2),500,500,
                &m_width,&m_height,&m_index2,'v');
21          done = 0;
22          while(done == 0) {
23                  ActionMonitor(&a_index,&a_action,&a_value);
24                  if(((a_index == m_index1)||(a_index == m_index2))&&
                                            (a_action == 1))
25                          switch(a_value) {
26                          case 1:
                            /* Code to modify member of global parameter block */
                            /* This part is up to the programmer's requirements */
27                          RemoveMenu(m_index2);
                            /* Code to modify param_values goes here          */
28                          CreateAndDisplayMenu(param_values,(NUMPARAMS + 2)
                                500,500,&m_width,&m_height,&m_index2,'v');
29                          break;
                            /* Code to handle other cases                     */
30                          case NUMPARAMS:
                            /* Code to handle case for last parameter         */
31                          case (NUMPARAMS + 1):
32                                  done = 1;
33                                  break;
34                          default: break;
                            }
            }
            RemoveMenu(m_index1);
            RemoveMenu(m_index2);
    }
```

Figure 4.8. (cont'd)

We now discuss the code template of Figure 4.8 line by line. Line 1 nullifies any previous definition of the string *NUMPARAMS;* this allows us to redefine NUMPARAMS in line 2. In line 2, the programmer should define the string NUMPARAMS to be the (integer) number of algorithm parameters to be manipulated by the parameter fetching routine. Line 3 is the routine's function header; all of HAPPI's parameter fetching routines are declared as type *void*, and are not passed any arguments. The programmer should give the routine an appropriate name in line 3; all of HAPPI's parameter fetching routine names are postfixed with "_Param" by convention. Lines 4 through 13 declare variables needed for every parameter fetching routine, and will not need to be altered by the programmer. The declaration of the array of string pointers *param_names[]* in line 14 must be modified for each parameter fetching routine. The first string, "Routine Name", should be changed to the name of the image processing routine for which the parameter fetching routine is being written. The remaining strings should be descriptive but concise names for the image processing algorithm parameters. The last string in the declaration of *param_names[]* should be a null string as shown; this is necessary because of an idiosyncrasy of the routine HAPPI uses to create and display menu windows. The number of string pointers in the array *param_names[]* should thus be two more than the number of algorithm parameters due to the string pointer for the routine name at the beginning of the array and the null string pointer at the end of the array. Line 15 creates and displays a menu window near the middle of the computer display using the strings pointed to by the array *param_names[];* this line does not need to be altered by the programmer. (The programmer is

referred to the HAPPI Technical Manual, Volume 4, "Menu" section for documentation on the function call of line 15.) Line 16 begins a section of code which builds the elements of the array of string pointers *param_values[]*. The first element of this array is assigned a pointer to the string "Parameters" in all of HAPPI's parameter fetching routines by convention. Lines 17 and 18 assign pointers to the strings "OK" and "" (a null string) to the second-to-last and last elements of *param_values[]*; this is done in all parameter fetching routines. Between lines 16 and 17, the programmer will need to insert code to initialize the remaining elements of the array *param_values[]* using the values currently in the global parameter block; this code will be specific to each parameter fetching routine. Many image processing algorithm parameters will be floating-point or integer numbers. The following two lines of example code will print a float-valued parameter from the global parameter block into a string and assign a pointer to the string to one of the elements of the array *param_values[]*:

```
sprintf(value[1],"%f",IPparam.snratio);
param_values[1] = value[1];
```

The first line prints the string representation of the floating-point parameter *IPparam.snratio* from the global parameter block into row 1 of the character array *value[][]*. The second line assigns a pointer to row 1 in *value[][]* to element 1 of the array of string pointers *param_values[]*. The code to handle an integer parameter would be similar; we would simply replace the floating-point ("%f") conversion specification in the *sprintf()* call with an integer ("%d") conversion specification. Using an incorrect

conversion specification in the *sprintf()* call is a common source of errors in the displayed parameter values, especially when existing code is being copied and modified. Some image processing algorithm input parameters may be one of a finite and small set of choices. For example, for an algorithm that processes an image by operating on individual rows or columns of the image (rather than on, say, small two-dimensional neighborhoods of the image), the programmer should give the user a choice of whether to process the image along the rows or the columns. In such cases, the convention adopted in HAPPI has been to represent such choices in the global parameter block using a *char*- or *short*-valued structure member. For example, the character-valued structure member *fit_type* in the global parameter block takes on the value 'c' if the image processing routines that use the *fit_type* parameter are to process along the columns of an image; if processing is to be done along the rows, *fit_type* is set to the value 'r'. The example code of Figure 4.9 illustrates how the programmer might assign values to array elements of *param_values[]* based on the value of a character-type member of the global

```
if(IPparam.fit_type  ==  'c')
   param_values[2] = "Column";
else if(IPparam.fit_type  ==  'r')
   param_values[2] = "Row";
else    /* Parameter fit_type has an invalid value; this should not happen */
   {Code to handle this anomalous situation}
```

Figure 4.9. Assigning value to parameter value array from small set of choices

parameter block (code to work with short integer-type members would be very similar). Note that if there were more than two possible values of a non-numerical algorithm parameter, the above code fragment could be extended

in a straightforward manner to handle each possible value. This could be done by adding more *if* statements to the list in the above code fragment or by using a *switch* construct. The programmer is referred to the parameter fetching routines in the file IPparams.c for further examples of how the array of parameter value strings is built. The function call in line 20 creates and displays a menu window containing the (character representation of the) values currently in the global parameter block. This menu window is placed next to the first menu window created in line 15, with the parameter values in the second menu window next to their names in the first menu window. Line 20 does not need to be altered by the programmer. Line 21 initializes the variable *done*; the *while* statement of line 22 uses the value of *done* to determine when to exit the *while loop* which begins on line 22 and ends with the right curly brace following line 34. Line 23 calls *ActionMonitor()*, which was discussed previously in Section 4.9; control does not return to the parameter fetching routine until the user enters some type of mouse or keyboard input. Line 24 tests the values returned (via pointers) from *ActionMonitor()* to determine if the user has clicked the mouse on either the menu window containing the parameter names or the menu window containing the parameter values. If the user has not entered valid input (a mouse click on either of these windows), the code following the *if* statement of line 24 is not executed, the value of *done* remains unchanged, and the *while* loop again executes *ActionMonitor()*, waiting for the user to enter valid input. If the user has entered valid input, then line 25 is executed, switching on the the variable *a_value*. When the user enters valid input, *a_value* will contain the number of the sub-window of the menu window on which the

mouse was clicked. Thus, if the user clicks the mouse on the uppermost sub-window on the menu window displaying the current parameter values, *a_value* will equal zero. As the zeroth sub-window of the parameter name and parameter value menu windows contain the name of the image processing algorithm and the word "Parameters", respectively, we do not wish to perform any action if the user clicks on either of these sub-windows. Hence, the list of case labels beginning in line 26 starts with *case 1*; the code following *case 1* will be executed if the user clicks on either the name or the value of the first parameter. The code for each case in the switch construct will be specific to the algorithm parameter handled by that case. The following line of code is an example of how the value of a float-valued member of the global parameter block is altered:

```
IPparam.snratio = GetFloatValueFromWindow("Signal/Noise",500,300,
                  IPparam.snratio,0.0,10000.0,0.1,1.0);
```

The return type of *GetFloatValueFromWindow()* is *float*. This line of code places the returned floating-point value in the member *snratio* of the global parameter block IPparam. As mentioned previously in Section 4.9, the function *GetFloatValueFromWindow()* creates and displays a "value window" which allows the user to alter the value of an algorithm parameter using mouse clicks. The argument list to this function will need to be altered by the programmer. Argument number zero (in C, function arguments are numbered starting from zero) is a string which is placed in the title sub-window of the value window; often, it is useful to indicate in this string any constraints on the algorithm parameter being altered. For example, if the

algorithm parameter must be greater than zero, the programmer might pass the string "Signal/Noise (>0)" to *GetFloatValueFromWindow()* as argument zero. The next two arguments control the position on the computer display where the value window will appear; these need not be altered by the programmer in most cases. Argument number 3 should be the member of the global parameter block which is to be altered. The function *GetFloatValueFromWindow()*. uses this argument to read the *current* value of the global parameter block member at the time *GetFloatValueFromWindow()* is called. (The global parameter block member is altered only after a new value is returned by *GetFloatValueFromWindow()*.) Arguments number 4 and 5 are the lower and upper limits, respectively, on the value to be returned. The function *GetFloatValueFromWindow()* enforces these limits by forcing the returned value to be the lower (upper) limit value if the user attempts to enter a value below (above) the lower (upper) limit value. Arguments number 6 and 7 are the small and large increments by which the value displayed in the value window is changed when the user clicks on the small and large arrows, respectively, in the value window. The value window created by the above call to *GetFloatValueFromWindow()* will thus not allow the user to enter values below zero or above ten thousand, and will increment (decrement) the value displayed in the value window by 0.1 when the user clicks on the small up (down) arrow, and will increment (decrement) the value displayed in the value window by 1.0 when the user clicks on the large up (down) arrow. The related function *GetValueFromWindow()* creates and displays a value window which returns an integer. Note that the arguments to this function for lower and upper parameter limits and small and large

parameter increments should naturally be *integers*; errors could result if floating-point numbers were passed. If the algorithm parameter to be altered is not a number, but rather a character or short integer representing one of a small set of choices, the code to alter the parameter may simply cycle the parameter through its possible values as the user repeatedly clicks the mouse on the parameter. Example code to cycle through the possible values for a global parameter block member called *dummyvalue* whose possible values are 'a', 'b', and 'c' might look like the following:

```
if(IPparam.dummyvalue == 'a')
        IPparam.dummyvalue = 'b';
else if(IPparam.dummyvalue == 'b')
        IPparam.dummyvalue = 'c';
else if(IPparam.dummyvalue == 'c')
        IPparam.dummyvalue = 'a';
else    /* IPparam.dummyvalue has incorrect value; this should not happen. */
```

Alternatively, we could do the same thing with a switch construct:

```
switch(IPparam.dummyvalue)
{
case 'a' :      IPparam.dummyvalue = 'b';
                break;
case 'b' :      IPparam.dummyvalue = 'c';
                break;
case 'c' :      IPparam.dummyvalue = 'a';
                break;
default :       /* IPparam.dummyvalue has an incorrect value      */
                /* This should not happen.                        */
                break;
}
```

The programmer is again referred to the various parameter fetching routines in the file IPparams.c for further examples of code to modify members of the global parameter block. Line 27 of the code template of Figure 4.8 removes

the menu window containing the parameter values. The code following line 27 will be specific to the particular parameter being modified, and will be identical to that used to assign the initial values to the elements of *param_values[]* in the code between lines 16 and 17. Line 28 is identical to line 20, and simply redraws the menu window displaying the newly modified array *param_values[]*. Line 29 is an all-important *break* statement, which passes program control to the end of the switch construct; without it, program control simply passes to the code following the next case label, a situation we wish to avoid. Lines 28 and 29 do not need to be altered by the programmer. Following line 29, the programmer should insert as many case labels as necessary to handle alteration of all the of algorithm parameters. Line 30 begins the code to handle the last parameter. All code past line 31, inclusive, is common to all parameter fetching routines and does not need to be altered by the programmer. The programmer should use the information window where necessary to inform the user of any special parameter constraint violations (besides the minimum & maximum value constraints enforced by *GetValueFromWindow()* and *GetFloatValueFromWindow()*). Policy for handling such violations is up to the programmer. Example code to force an integer parameter to be odd is shown in Figure 4.10. Line 1 modifies the parameter block according to the user's input. The user may enter any integer between the upper and lower limits, inclusive, passed to *GetValueFromWindow()*. Line 2 tests the integer returned by *GetValueFromWindow()* and placed in IPparam.mask_size to see if it is even. If the integer in IPparam.mask_size is even, lines 3 and 4 are executed, informing the user of the parameter constraint violation and decrementing the value of IPparam.mask_size,

respectively. The arguments to *WriteInfoWindow()* are as follows: Argument zero is the window index of the information window; this is a global variable, declared in Globals.h, and does not need to be changed by the programmer. Argument number 1 is the string to be written in the information window.

```
1       IPparam.mask_size = GetValueFromWindow("Mask Size (odd)",500,300,
                            IPparam.mask_size,3,511,1,10);
2       if((IPparam.mask_size % 2) == 0)     /* If entered mask size was even      */
                {
3               WriteInfoWindow(instr_window_index,"Mask size must be odd;
                        decrementing",'c',2);
4               IPparam.mask_size -= 1;
                }
5       RemoveMenu(m_index2);
6       sprintf(value[1],"%d",IPparam.mask_size);
7       param_values[1] = values[1];
8       CreateAndDisplayMenu(param_values,(NUMPARAMS + 2),500,500,&m_width,
                        &m_height,&m_index2,'v');
9       break;
```

Figure 4.10. Example code for enforcing constraints on parameters

Argument number 2 specifies whether the string is to be written right-justified, left-justified, or centered in the information window by the values 'r','l', and 'c', respectively. Argument number 3 specifies the line number (line 1, 2, or 3) in the information window to which the string passed as the argument number 1 will be written. Lines 5 through 9 make up the remainder of the code that would be included is a block of code to modify a member of the global parameter block.

The parameter fetching routine should be placed in one of the files IPtest.c, IPtest2.c, or IPtest3.c along with the new image processing algorithm

it has been written for. With the algorithm and parameter fetching routine written according to the code templates presented in this document, the programmer need only make modifications to four of HAPPI's source code files and recompile the source code to integrate his/her routine into HAPPI. The next section details these final steps.

4.11 Putting it All Together

This section details the final steps necessary to integrate an image processing routine into HAPPI. Here, we describe changes the programmer will need to make to HAPPI's source code files to cause HAPPI to display an image processing menu selection for the new routine, prompt the user for the input image(s) when the new routine is selected, and execute the parameter fetching routine and algorithm calls via the Image Processing Manager. If the programmer is not only adding a new routine but creating a new class of routines, then the changes necessary for each file will be more extensive than if a new routine is being added to an existing class of routines. For each source code file discussed in this section, we will first discuss the changes necessary to add a new routine to HAPPI under an existing class, then address changes necessary to add a new class of routines to HAPPI .

4.11.1 Editing Menus.h

The first file the programmer needs to edit is Menus.h. This file contains the declarations and initializations for all of HAPPI's static menus, as

well as preprocessor macros defining attributes of the static menus. If the programmer is adding an algorithm under an existing class, the following changes should be made to this file:

1) Update the preprocessor macro which defines the size of the menu for the image processing class under which the new algorithm is being added;
2) Add a string containing the name of the new routine to the static declaration and initialization of the menu item text for the image processing class.

The preprocessor macros defining the menu sizes for each image processing class are of the form:

#define *classname*MENUSIZE *size*

where *classname* is the name of an image processing class (in all capital letters by convention) and *size* is the (integer) number of items (including the menu title and the "Exit" menu item) in the menu for that class. The programmer would thus increment *size* by the number of new routines being added to the menu for class *classname*. The declaration and initialization of menu item text for the trend removal class or processing routines is shown in the code listing of Figure 4.11, which is excerpted from Menus.h. Beginning in line 4, *TRENDREMOVALMENU[]* is declared and initialized as a static array of pointers to the strings in lines 5 through 10. To add a new image processing routine to the trend removal menu shown above, the programmer would thus

change the menu size from "5" to "6" in line 1, and would insert a string containing the title of the new routine into the list beginning in line 5. The routine which creates and displays menu windows places the string pointed to by the first element of *TRENDREMOVALMENU[]* in the first subwindow of the

```
1       #define TRENDREMOVALMENUSIZE   5
2       #define TRENDREMOVALMENUX      ((Main_Menu_Width/7)+2)
3       #define TRENDREMOVALMENUY      (5*(Main_Menu_Height/2) + MAINMENUY)
4       static char *TRENDREMOVALMENU[] =
                        {
5                       "Trend Removal",
6                       "Row or Column Fit",
7                       "Surface Fit",
8                       "Widowed RC Fit",
9                       "Exit",
10                      ""
                        };
```

Figure 4.11. Example of menu text item declaration

menu window, the second string in the second subwindow, and so forth. The position of the new routine in the list is up to the programmer; however, the new routine should be positioned so as to "make sense" to the user. If the new routine is related to other existing routines, it should be grouped with them rather than being simply placed at the bottom of the list. The programmer should make note of where the new routine is inserted in the list, as subsequent modifications to other files will depend on this.

If the programmer is creating a new image processing class, the changes that must be made to Menus.h are as follows:

3) Update the preprocessor macro which defines the menu size for the Image Processing menu;
4) Add a string containing the name of the new class to the static declaration and initialization of the Image Processing menu, IMG_PROCESS_MENU;
5) Declare and initialize a new menu for the new class and define menu attributes for the new menu.

Steps 3 and 4 above are similar to steps 1 and 2; the programmer simply edits a different preprocessor control line and initialization list. Step 5 may be done using the menus for other classes as examples; all menu declarations are similar in form. The new menu should have a size, x location, and y location attribute defined with preprocessor control lines, and should have a descriptive, easily remembered name. The x and y location attributes of the trend removal menu are defined in lines 2 and 3, respectively, of Figure 4.11. These attributes are used to determine where the menu will be drawn on the screen, and are up to the programmer. If the new menu is to be displayed directly to the right of the Image Processing submenu, then the x location attribute should be set to that of the trend removal menu example of Figure 4.11, namely ((Main_Menu_Width)/7 + 2). (Note: the variable Main_Menu_Width is a global variable declared in the file Globals.h.) By convention, the menus for each image processing class are drawn with their menu titles directly to the right of the name of the class in the Image Processing menu. Thus, if we let *num* be the (integer) position of an image processing class in the Image Processing menu, then the y location attribute should be set to (((*num* + 2)*(Main_Menu_Height)/2) + MAINMENUY) to

adhere to this convention. (Note that the top, or *zeroth*, position in any menu window is occupied by the menu's title.) The static declaration and initialization of the menu text items for the new class will be similar to the ones for the existing classes. The first, or *zeroth*, string in the initialization is always the menu title; this is followed by as many strings as necessary to denote the routines to be accessed via the menu, with these strings followed by an "Exit" string and a null string.

4.11.2 Editing Globals.h

The next file to edit is Globals.h. This file includes declarations of global variables that are used throughout HAPPI, data type definitions for the image buffer and the global parameter block, and the enumeration lists for the *class* and *subclass* variables passed to *IP_manager()*. If the programmer is adding an algorithm under an existing class, the following changes should be made to this file:

1) Add any new parameters needed by the new processing routine to the data type definition of the global parameter block;
2) Add an enumerator for the new routine to the enumeration list of the IP_SUBCLASS defined type under the applicable class of processing routines.

Many algorithm parameters function similarly in HAPPI's various image processing routines; for example, all convolution-based routines take a

convolution mask size as a parameter. The practice adopted in HAPPI has been to share a single member of the global parameter block between processing routines that make similar use of that member. Hence, the *mask_size* member of IPparam is used by all of HAPPI's convolution-based processing routines. Before the programmer adds a new member to the global parameter block data type definition, he/she should first check to see if appropriate members have already been defined. The data type definition of the global parameter block begins with the line "struct param {" in Globals.h. Following this line are the declarations for the individual structure members. The programmer is referred to the calls to the individual image processing routines (located in the file IPmanager.c to be discussed in greater detail below) to see which members of IPparam are used by a particular processing routine, and which members are shared between processing routines. If it is deemed necessary to create a new member in the global parameter block, the programmer simply adds a declaration of the appropriate type for the new member to the structure declaration.

To find the enumeration list for the IP_SUBCLASS defined data type, the programmer should search Globals.h for the string "IP_SUBCLASS". Integer values are explicitly assigned to the enumerators in this list according to the following scheme: Within each image processing class, the enumerators for the subclasses of that class are assigned values according to the menu position of the routine corresponding to the enumerator. The scheme is illustrated for the "trend removal" class of processing routines as follows: The menu for the trend removal class contains (at this writing) three routines: "Row or Column Fit", "Surface Fit", and "Windowed RC Fit", which appear in that order. The

enumerators declared for these routines in Globals.h are "rcfits", "surrje", and "wls2dsur", respectively (Note: The names of these enumerators were derived from the original names of the processing routines. It is not necessary for them to resemble the menu text, as they are not seen by the user.). The integer assignments for the enumerators in the trend removal class are thus rcfits=1, surrje=2, and wls2dsur=3. The enumerators for the subclasses of each processing class are assigned values in a like manner. Hence, the enumerator corresponding to the first processing routine in the menu for any given class is assigned a value of 1, the enumerator for the second processing routine in the menu for any given class is assigned a value of 2,and so forth. When the programmer modifies the enumerator list for the IP_SUBCLASS defined type, he/she should thus recall the menu position of the new routine being added (this position is established when the file Menus.h is edited), and modify the assigned values in the enumerator list as appropriate; if the menu text for the new routine was inserted before the end of the menu when Menus.h was edited, then the enumerator for the new routine should be inserted at the corresponding point in the enumerator list within the group of enumerators for the applicable processing class. The new enumerator should be assigned the value that was previously assigned to the enumerator it is displacing, and the assigned values for all subsequent enumerators within the applicable processing class should all be incremented to reflect that they have been "bumped down" one position on the menu.

If the programmer is creating a new image processing class, the changes that must be made to Globals.h are as follows:

3) Add any necessary declarations to the global parameter block, as in step 1 above;
4) Declare an external integer to hold the menu index of the menu for the new processing class;
5) Add an enumerator for the new processing class to the enumeration list of the IP_CLASS defined type.
6) Insert a group of enumerators for the new class into the enumerator list of the IP_SUBCLASS defined data type, following the previously discussed scheme for explicitly assigning values to the enumerators.

A global variable for each of HAPPI's static menus is declared in Globals.h with a declaration of the form:

extern int *menuname*_Menu;

where *menuname* is a descriptive name for the menu, with a capital first letter by convention. The programmer should thus add to Globals.h a declaration of the above form with *menuname* being descriptive of the new class of processing routines. This new external integer will be needed in a subsequent step when a new menu manager is created to handle the new processing class.

To find the enumeration list for the IP_CLASS defined data type, the programmer should search Globals.h for the string "IP_CLASS". Integer values are explicitly assigned to the enumerators in this list according to a scheme similar to that for the IP_SUBCLASS defined data type: The values are assigned to the enumerators according to the menu position of the

corresponding processing class in the Image Processing menu. Thus, (at this writing), the enumerator *noise_filter*, which corresponds to the first processing class in the Image Processing menu, "Noise Filters", is assigned a value of 1. When the programmer modifies the enumerator list for the IP_CLASS defined type, he/she should thus recall the menu position of the new class being added (this position is established when the file Menus.h is edited), and modify the assigned values in the enumerator list as appropriate.

4.11.3 Editing IPmanager.c

The next file to edit is IPmanager.c. This file contains the functions *IP_manager()*, *GetParams()*, *CallIP()*, and the parameter fetching and image processing support routines which are called by *GetParams()* and *CallIP()*, respectively. Recall that the parameter fetching support routines call the actual parameter fetching routines, and the image processing support routines call the actual image processing routines (the reader is referred to Figure 4.3 in Section 4.3 for an illustration of the flow of control between these functions). If the programmer is adding a processing routine under an existing class, the following changes should be made to this file:

1) Edit the parameter fetching support routine for the applicable processing class, adding to the support routine's switch construct a case label and accompanying code to call the parameter fetching routine for the new algorithm;

2) Edit the image processing support routine for the applicable processing class, adding to the support routine's switch construct a case label and accompanying code to call the new image processing routine.

As may be recalled from the discussion of Section 4.3, *IP_manager()* is passed a class and subclass, which together determine the image processing routine to be executed. *IP_manager()* then call *GetParams()* and *CallIP()* in turn, passing the class and subclass on to both of these routines. *GetParams()* and *CallIP()* both "switch" on the image processing class passed to them and call a support routine for that particular class, passing the subclass to the support routine. The support routines in turn "switch" on the subclass passed to them and call the specific parameter fetching or image processing algorithm determined by the subclass. The parameter fetching support routines have names prefixed with "P_", and the image processing support routines have names prefixed with "C_", by convention. Hence the parameter fetching and image processing support routines for the trend removal class of image processing routines are called *"P_Trend()"* and *"C_Trend()"*, respectively. The entire source code of *P_Trend()* is listed in Figure 4.12. (Note: The string *BAD_IMG_PROCESS_SUBCLA* is defined with a preprocessor macro in the file "errors.h" as the error code to be returned if the subclass passed to a support routine is not defined in the enumerator list for the defined type IP_SUBCLASS).

Thus, if he/she were adding a new processing routine under the trend removal class of processing routines, the programmer would simply add a case label and accompanying code for the new routine to the switch construct

beginning in line 5 of the listing of Figure 4.12. The expression following the word "case" in the case label should be equal to the enumerator for the subclass of the new routine (this is the enumerator added to the enumerator list for the IP_SUBCLASS defined type when the file Globals.h is edited). The

```
1       int     P_Trend(subclass)
2       IP_SUBCLASS subclass;
                {
3               int     error;
4               error = 1;
5               switch(subclass)
                        {
6                       case rcfits :           Rcfit_Param();
7                                               break;
8                       case surrje:            Surf_Param();
9                                               break;
10                      case wls2dsur :         Wls2dsur_Param();
11                                              break;
12                      default :               error = BAD_IMG_PROCESS_SUBCLA;
13                                              break;
                        }
14              return  error;
                }
```

Figure 4.12. Source code for support routine *P_Trend()*

statements following the new case label are simply a call to the parameter fetching routine and a break statement. If we have written our parameter fetching routine to return an error code, we should assign the returned value to the variable *error* (declared in line 4 of the listing in Figure 4.12). The error code will then be passed back to the manager routine which called *IP_manager()* and handled with a generic error display function. Thus, the new lines of code added to the switch construct would be of the form:

```
case newroutine :      error = Newroutine_Param();
                       break;
```

The modifications to the image processing support routine are similar to those for the parameter fetching support routine. A case label, with accompanying code to call the image processing routine, is added to the switch construct found in the image processing support routine. Each image processing support routine contains a single switch construct with case labels identical to those in the switch construct of the corresponding parameter fetching support routines. The new case label is thus identical to the one added to the switch construct in the parameter fetching support routine. The block of code following the case label consists (for the vast majority of processing routines) of the call to the processing routine, a line of code used to build an entry in the output image's history structure, and a break statement. If the processing routine returns any error codes, the returned value should be assigned to the variable *error*. (All of the support routines use this variable name for returned error codes by convention.) The argument list in the call to the processing routine consists (with the exception of the argument(s) for the destination image pointer(s)) of the appropriate members of the global parameter block IPparam. The destination image pointer(s) passed to the processing routine is (are) the temporary image pointer(s) *temp_img* (and *temp_img2*, if it is needed). The character array *history*, declared in each of the image processing support routines, is used to build a string describing the processing that has just been performed on an image. This string is saved in the history structure of the destination image upon successful completion of a processing routine. Example code to be added to the switch construct of the

image processing support routine might look that in Figure 4.13. In line 1, we have the case label and function call to the new routine, with the argument list as described above and the return value assigned to the variable *error*.

```
1       case newroutine :       error = Newroutine(IPparam.s_image1,temp_img,
                                        IPparam.int_element1,IPparam.float_element1,
                                        IPparam.int_element2);
2                               sprintf(history,"newroutine(image_var,image_var,
                                        %d,%f,%d);",IPparam.int_element1,
                                        IPparam.float_element1,IPparam.int_element2);
3                               break;
```

Figure 4.13. Example code to add to image processing support routine

The *sprintf()* call in line 2 bears some further explanation. Argument number 1 to *sprintf()* is the "format string". The format string begins with a descriptive string for the new routine (this may be identical to the new routine's name, but does not have to be), followed by a left parenthesis, followed by a comma-separated list whose members are determined by the argument list for the new routine as follows: For every argument to the new routine of type IMAGE, the element of the comma-separated list is just the string "image_var" (note that the "image_var" strings themselves are *not* enclosed in double quotes in the *sprintf()* call). For every non-image argument to the new routine, the corresponding element of the comma-separated list is a conversion specification of the appropriate type (%f for float-valued arguments, %d for integer arguments, and so forth). The comma-separated list is terminated with a right parenthesis and a semicolon. The remaining arguments to the *sprintf()* call are just the appropriate members of

the global parameter block, and are identical to the corresponding arguments in the call to the processing routine in line 1. The *sprintf()* call of line 2 is truly necessary only if the programmer is incorporating the new processing routine into HAPPI's built-in macro language. It will not cause problems if the programmer does not add the new routine to the macro language, but HAPPI's "convert history to macro" function is the only function that requires the use of the *sprintf()* call of line 2. The rigid form of the format string in the *sprintf()* call is necessary for HAPPI's macro functions to properly interpret each entry in an image's history structure when converting the history structure to a macro. Instructions on how to add a new routine to the macro language in addition to adding it to the interactive user interface are beyond the scope of this document. However, once a new routine has been integrated into HAPPI, it may be added to the macro language by editing only one file, "macro_calls.c". Brief instructions on what changes need to be made are found in comments in this file; the changes involve mostly copying and modifying existing code, and the skilled programmer should be able to incorporate new processing routines into the macro language easily.

Occasionally, certain processing routines may require additional "set-up" or "clean-up" code beyond the three lines given in Figure 4.13. The programmer is referred to the support routines *C_Trend()* and *C_Flaw()* in IPmanager.c for examples of such situations and how they are handled. In particular, these routines contain examples of the creation of a second destination image, *temp_img2*, for processing routines which produce two output images. (Recall from Section 4.3 that *temp_img* is created within

CallIP(), and *temp_img2* is created within the image processing support routines only where needed.)

If the programmer is creating a new image processing class, the changes that must be made to IPmanager.c are as follows:

3) Create a new parameter fetching support routine for the new class; this may be most easily done by copying and modifying an existing parameter fetching support routine;
4) Edit the function *GetParams()*, adding to this function's switch construct a case label with accompanying code to call the parameter fetching support routine for the new processing class;
5) Create a new image processing support routine for the new class, again by copying and modifying an existing image processing support routine.
6) Edit the function *CallIP()*, adding to this function's switch construct a case label with accompanying code to call the image processing support routine for the new processing class.

As both the parameter fetching and image processing support routines are all very simple and similar, performing steps 3 and 5 above is very straightforward. In creating the support routines, the programmer should refer to Globals.h to assure that the case labels used in the support routines' switch constructs exactly match the enumerators of the enumerator list for the IP_SUBCLASS defined type. Both of the new support routines should declare and return the *error* variable, and the image processing support routine should declare the *history* variable and include calls to the functions

copy_history() and *append_history()* (the calls to these functions may be copied from existing support routines without modification).

The functions *GetParams()* and *CallIP()* both contain a single switch construct, and the modifications to these files in steps 4 and 6 above are also very straightforward. The case label added to each function's switch construct must exactly match the enumerator added for the new processing class in the enumerator list for the IP_CLASS defined type. For *GetParams()*, the code following the new case label is just a call to the new parameter fetching support routine of the form:

```
error = P_Newclass(subclass);
```

and a break statement. Note that the subclass passed to *GetParams()* is passed on to the parameter fetching support routine. For *CallIP()*, the code following the case label is just a call to the new image processing support routine of the form:

```
error = C_Newclass(subclass);
```

and a break statement. Note that the subclass passed to *CallIP()* is passed on to the image processing support routine.

4.11.4 Editing Managers.c

The final file to edit is Managers.c. This file contains the Image Processing Menu Manager and all of its subordinate menu managers, as well as the menu managers for HAPPI's "Images", "Macros", "Special Functions", "Quit", and "Buffer" main menu items. If the programmer is adding an algorithm under an existing class, the following changes should be made to this file:

1) Edit the function *Init_IPparam()*, inserting code to initialize any new members that were added to the global parameter block IPparam when Globals.h was edited;
2) To the menu manager function which handles the applicable class of processing routines (the names of these managers are listed below), add a case label and accompanying code to the menu manager's switch construct (this is most easily done by copying and modifying code associated with the case label for an existing routine);
3) Within the manager modified in step 2, find the preprocessor control line defining the word EXITVALUE as an integer value, and increment the integer in this definition by the number of processing routines being added.

The function *Init_IPparam()* is called only once, during initialization, and assigns default values to every member of the global parameter block. If a member of the global parameter block is not explicitly assigned a value at

initialization, the member will contain random memory garbage, and when the user calls a processing routine which uses that member, that random garbage will be displayed in the value window created in the parameter fetching routine. At that point, the user may change the value of the member, in the usual way, to an appropriate value. However, if he/she does not change the value of the member, the garbage value stored in that member will be passed to the processing routine selected by the user. Depending on the extent to which the selected image processing routine checks its input parameters, passing garbage to a processing routine could result in unpredictable output. The purposes of step 1 above are thus to help prevent "incorrect " input parameters from being passed to processing routines, and to assure that "correct" and representative default values for all algorithm input parameters are always presented to the user for every processing algorithm. The code added to *Init_IPparam()* will be of the form:

```
IPparam.mynewparameter = mydefaultvalue;
```

where *mynewparameter* is the new member added to IPparam by the programmer, and *mydefaultvalue* is the value the programmer has chosen, based on experience with his/her processing routine, as a representative default value for the parameter. Note that if no new members have been added to IPparam, this step is not necessary.

To perform step 2 above, the programmer needs to know the names of the Image Processing Menu Manager's subordinate menu managers for the various image processing classes; these are (at this writing):

Noise_Filters_Manager()
Morphology_Manager()
Trend_Removal_Manager()
Edge_Detection_Manager()
Convolution_Manager2()
Contrast_Enhancement_Manager()
Flaw_Detection_Manager()
Img_Measurement_Manager()
Math_Manager()

Additionally, there are two subordinate menu manager "skeletons" in Managers.c; these functions currently do nothing but return to their calling routine, *Img_Process_Manager()*, once the user selects the "Exit" item on their menus. The menu manager skeletons are called *New1_Manager()* and *New2_Manager()*, respectively. The code for *New1_Manager()* and *New2_Manager()* is not compiled unless the preprocessor control line:

```
#define          EXPAND
```

is included at the top of Managers.c.

As discussed in Section 4.3, *Img_Process_Manager()* (the Image Processing *Menu* Manager) and all of its subordinate menu managers may be regarded as a large nested switch construct. The integer expressions that this switch construct "switch" on are supplied by the function *ActionMonitor()*, discussed in Section 4.9. Upon return from *ActionMonitor()*, the integer-valued variable *value*, whose address is passed to *ActionMonitor()*, contains the number of the subwindow of the menu window in which the user has clicked the left mouse button. (Recall that the uppermost subwindow in a menu window is numbered zero, not one.) Within *Img_Process_Manager()*, the *value*

variable used by *ActionMonitor()* is declared as type IP_CLASS, and the case labels used in the function's switch construct are just the enumerators for the IP_CLASS defined type. Thus, in the execution of *Img_Process_Manager()*'s switch construct, the integer returned by *ActionMonitor()* in *value* is compared to the enumerators in the enumerator list for the IP_CLASS defined type. Since the integer values explicitly assigned to these enumerators in Globals.h are equal to the menu positions of the corresponding image processing classes in the Image Processing menu, the code associated with the case label matching the selected image processing class is always executed. A similar scheme is used within *Img_Process_Manager()*'s subordinate menu managers. The *value* variable used by *ActionMonitor()* in these menu managers is declared as type IP_SUBCLASS, and the case labels for the switch construct in each menu manager are just the group of enumerators from the IP_SUBCLASS enumerator list for the particular class of routines served by that menu manager.

Thus, the code modifications of step 2 above proceed as follows: To the switch construct of the appropriate menu manager, add a case label, with the expression following the word "case" in the case label equal to the enumerator for the new routine (this is the enumerator added to the enumerator list for the IP_SUBCLASS defined type when Globals.h is edited). Then, copy the code associated with an existing case label and place it directly after the new case label, and modify the copied code to suit the new routine. The code copied in this step, for processing routines which take only one input image, will look like the example code of Figure 4.14. The code of Figure 4.14 is explained line by line as follows: Line 1 simply highlights the menu item selected by the

user. The programmer need not change line 1, as the variable *value*, returned by *ActionMonitor()* and passed to *HighLightItem()*, determines which item is highlighted. The function *get_image_from_user()* called in line 2 has a

```
case mynewsubclass :
1       HighLightItem(menu_index,value);
2       source = get_image_from_user("Select source image for
                              MyNewRoutine","Explanatory Comments",
                              menu_index,EXITVALUE,prev_menu,exit_item);
3       if(source != NULL)
                {
4               IPparam.s_image1 = source;
5               subclass = mynewsubclass;
6               IPerror = IP_manager(class,subclass);
7               if(IPerror <= 0)
8                       system_error("IP manager: Processing Class:
                                      MyNewRoutine",IPerror);
                else
9                       add_image_to_buffer(IPparam.d_image1);
                }
10      break;
```

Figure 4.14. Example code associated with case label for new processing routine

return type of IMAGE; this function writes the two strings passed to it as arguments number 0 and 1 to the information window to prompt the user for an input image. The two strings in the call to *get_image_from_user()* should be changed to appropriate prompts by the programmer. The remaining arguments to this function should be left unchanged. The variable *source* (and *source2*, if it is needed) is declared in each subordinate menu manager as an IMAGE variable. Thus, the call to *get_image_from_user()* returns an image pointer and assigns its value to the variable *source*. If the user does not click the left mouse button on an image window, *get_image_from_user()* returns a

null pointer. Line 3 thus checks to see if the user has in fact clicked the left mouse button on an image window, and if he/she has done so, passes control on to line 4; otherwise, control is passed to line 10. In line 4, the image pointer in *source* is assigned to the image pointer *s_image1* in IPparam. The programmer need not modify lines 3 or 4. Line 5 assigns the enumerator for the appropriate subclass to the variable *subclass* (*subclass* is declared as an IP_SUBCLASS-type variable in all of the subordinate menu manager routines); the programmer should edit line 5 to assign the enumerator for his/her new routine to *subclass*. Note that the value assigned to *subclass* is thus identical to the expression following the word "case" in the case label preceding line 1. Line 6 makes a call to the Image Processing manager, assigned the returned error code to the integer variable *IPerror*. The variable *class* passed to *IP_manager()* in line 6 is declared as type IP_CLASS in all of the subordinate menu managers, and is assigned the enumerator for the appropriate class at the beginning of each subordinate menu manager. Line 7 checks the returned error code from *IP_manager()*; control passes to line 8 if an error occurred, and to line 9 otherwise. Line 8 calls a generic error-handling routine which displays the string passed as the argument number 0 to *system_error()* in an "acknowledgement window". Argument number 1 to *system_error()* is the generic error code that was returned by *IP_manager()*; this code is used to look up an error message which is also displayed in the acknowledgement window. The programmer thus need only modify the string passed as argument number 0 to *system_error()* to an appropriate message. If no error occurs during image processing and line 9 is executed, the function *add_image_to_buffer()* is called, adding the processing routine's destination

image to the global image buffer and also displaying it on the screen. Line 10 is the all-important break statement that passes control to the end of the switch construct. The programmer need not modify lines 9 or 10. If a processing routine takes two input images the code of Figure 4.14 needs to be modified somewhat. The modifications involve adding another call to *get_image_from_user()* and another test of the returned image pointer similar to the one in line 3. The programmer is referred to the code in *Math_Manager()* for examples of how to deal with two input images.

Step 3 above is fairly straightforward. In each subordinate menu manager, the word EXITVALUE is first undefined, then redefined with a preprocessor control line as the menu position of the menu's "Exit" item. Thus, if a menu for a particular processing class has three processing routines, the routines themselves occupy menu positions 1, 2 and 3, while the menu title occupies menu position zero, and the "Exit" item occupies menu position 4. For such a menu manager we would see the preprocessor control lines:

```
1                    #undef EXITVALUE
2                    #define EXITVALUE 4
```

at the beginning of the manager's code. If we are adding new routines to this menu, we are "bumping down" the "Exit" item, so we need to adjust the value of EXITVALUE to reflect it new menu position by replacing the '4' in line 2 above with the appropriate value.

If the programmer is creating a new image processing class, the changes that must be made to Managers.c are as follows:

4) To the function *Init_Manager()*, add a call to the function *CreateStaticMenu()* to create a static menu for the new processing class;
5) Create a new subordinate menu manager (and submanagers if necessary) for the new image processing class (this may be most easily done by copying and modifying an existing menu manager);
6) Modify *Img_Process_Manager()*, adding to its switch construct a case label and accompanying code for the new menu manager;

Recall that when HAPPI is started, all static data structures, including static menu windows, whose contents will not change the entire time the program is running, are initialized. These and other initialization functions are controlled by the function *Init_Manager()*. Within *Init_Manager()*, the programmer will find a separate call to the function *CreateStaticMenu()* for every class of image processing routines. The programmer should copy and modify one of these calls to create a static menu for the new processing class. The new call to *CreateStaticMenu()* should be placed after all the other calls to this function in *Init_Manager()*. The function header of *CreateStaticMenu()* is shown in Figure 4.15.

```
CreateStaticMenu(menu,no,x,y,width_menu,height_menu,menu_index,direction)
char   **menu;                    /* input - menu item string          */
char   direction;                 /* input - direction of arrangement  */
int    no;                        /* input - number of menu items      */
int    x,y;                       /* input - position at which to draw menu */
int    *width_menu, *height_menu; /* returned - menu dimensions        */
int    *menu_index;               /* returned - menu window ID         */
```

Figure 4.15. Function header of *CreateStaticMenu()*

The programmer should pass arguments to *CreateStaticMenu()* as follows: The *menu* argument should be the name of the array of character pointers for the menu item text declared and initialized in the file Menus.h. The *no, x, and y* arguments should be passed as the menu size, the menu x location, and menu y location, respectively, for the appropriate class defined with preprocessor macros in Menus.h. Thus, for the call which creates the static menu for the morphology class of processing routines, the *menu, no, x,* and *y* arguments are passed as MORPHOLOGYMENU, MORPHOLOGYMENUSIZE, MORPHOLOGYMENUX, and MORPHOLOGYMENUY, respectively. Within *Init_Manager()*, the variables *width* and *height* are declared, and their addresses are passed in all calls to *CreateStaticMenu()* as the *width_menu* and *height_menu* arguments, respectively. Although the values returned in these variables are not used within *Init_Manager()*, the addresses of the variables still need to be passed to *CreateStaticMenu()* so that the argument list is syntactically correct. The *menu_index* argument should be passed as the address of the global variable (declared in Globals.h) for index of the menu for the new class. Thus, for the call which creates the static menu for the morphology class of processing routines, the *menu_index* argument is passed as *&Morph_Menu*, the address of the global integer variable Morph_Menu declared in Globals.h. This argument is used by *CreateStaticMenu()* to assign a unique integer to the menu index variable for each static menu. The *direction* argument determines whether the menu items will be drawn on top of or next to each other. By convention, this argument is passed as *'v'* (indicating a

"vertical", or vertically stacked, menu) in all the calls to *CreateStaticMenu()* for the image processing menus.

The menu manager "skeletons" *New1_Manager()* and *New2_Manager()* in Managers.c may be used as starting points for creating new subordinate menu managers. These functions constitute a bare minimum of code to implement a menu manager routine within HAPPI. An abbreviated version of the function *New1_Manager()* is listed in Figure 4.16.

The particular code added to the skeleton of *New1_Manager()* to create a subordinate menu manager for a new class of processing routines will of course depend on the nature of the new class. In creating the menu manager for a new class, the programmer should think about which existing class of processing routines is most like the new class, and copy and modify code from the menu manager for that class. The code of *New1_Manager()* shown in Figure 4.16 constitutes the bare minimum code necessary to draw a menu of processing routines for an image processing class, fetch and process user input, and call the new processing routine via *IP_manager()*.

We now discuss the code of Figure 4.16 line-by-line, noting which lines need to be changed to create a menu manager for a new processing class. The programmer should change the function name in line 1 to a descriptive name for the new processing class. Lines 2 and 3 are the same for all menu managers, and do not require modification. The arguments *prev_menu* and *exit_item* are used to pass information about the "parent" menu manager (*Img_Process_Manager()* in this case) to each subordinate menu manager; this information makes it possible to allow the user to exit a subordinate menu

```
1       New1_Manager(prev_menu,exit_item)
2       int     prev_menu;
3       int     exit_item;
        {
4       int                     index;
5       int                     action;
6       IP_SUBCLASS             value;
7       int                     menu_index;
8       int                     done = 0;
9       int                     IPerror;
10      IMAGE                   source;
11      IP_CLASS                class = new1_proc;
12      IP_SUBCLASS             subclass;
13      #undef          EXITVALUE
14      #define         EXITVALUE   4
15      DisplayStaticMenu(New1_Menu,NEW1PROCESSMENUX,NEW1PROCESSMENUY);
16      menu_index = New1_Menu;
17      while(! done)
                {
18              ActionMonitor(&index,&action,&value);
19              if((index == prev_menu)&&(value == exit_item))
20                      done = 1;
21              else if((index == menu_index) && (action == 1))
                        {
22                      switch(value)
                                {
                                /* case 0: do nothing  */
23                              case new1_1: {same or similar code as in Figure 4.14}
24                              case new1_2: {same or similar code as in Figure 4.14}
25                              case new1_3: {same or similar code as in Figure 4.14}
26                              case EXITVALUE: {
27                                      HighLightItem(menu_index,EXITVALUE);
28                                      done = 1;
29                                      break;
                                        }
                                }
30                              UnHighLightItem(menu_index,value);
                        }
                }
31              RemoveStaticMenu(menu_index);
        }
```

Figure 4.16. Abbreviated code for code skeleton *New1_Manager()*

manager by selecting the exit item of the parent menu. Lines 4 through 10 are also common to all of *Img_Process_Manager()*'s subordinate menu managers, and do not require modification. The *index*, *action*, and *value* variables declared in lines 4-6 are for use by *ActionMonitor()*. The *menu_index* variable simply holds the index (a unique identifying number) of the menu window which is drawn by the subordinate menu manager. The *done* variable is used as the condition of the menu manager's controlling *while* loop, and is set to a value of 1, aborting the loop, only when the user selects the menu's "Exit" item or the parent menu's "Exit" item. The *IPerror* variable is used to hold the returned error code from IP_manager. The *source* variable is an image pointer variable (data type IMAGE), and holds the address of an image selected by the user. For some managers, it is necessary to declare additional image pointer variables to hold the addresses of other input or output images used or created by processing routines. The programmer is referred to the code of *Img_Analysis_Manager()* and *Math_Manager()* for examples of how multiple input and/or output images are handled. Line 11 both declares and initializes the *class* variable; the programmer should change the initialization value in this line to be the enumerator for the new processing class. Line 12 simply declares the *subclass* variable which, along with the *class* variable is passed to IPmanager; the programmer need not change line 12. In lines 13 and 14, the word EXITVALUE is undefined and then redefined to a value equal to the menu position of the "Exit" item for the menu drawn by this menu manager. The programmer should change line 14 to define EXITVALUE to the appropriate value. Line 15 displays the static menu for the new processing class. The programmer should modify the argument list of the call to

DisplayStaticMenu() in line 15 as follows: Argument number 0 should be the menu index of the static menu for the new processing class. (Recall that the menu indices for the static menus are global variables declared in Globals.h and assigned values at initialization by the function *CreateStaticMenu()*.) Arguments number 1 and 2 should be the x and y pixel locations, respectively, where the menu for the new processing class will be drawn on the screen. These locations are defined with preprocessor control lines in Menus.h. Line 16 assigns the value of the menu index of the static menu for the new processing class to the variable *menu_index*; this variable is used in several lines of code in the remainder of the menu manager, hence by assigning the appropriate value to *menu_index* in line 16, we avoid having to change the many lines of code in which it appears. The programmer should change line 16 to assign the appropriate menu index to the *menu_index* variable; this will be identical to the menu index passed as argument number 0 to *DisplayStaticMenu()* in line 15. Lines 17 through 22 set up the controlling loop construct of the menu manager, fetch mouse input via ActionMonitor(), and test the returned input. These lines do not require modification. Lines 23 through 25 are the case labels corresponding the various routines in the new processing class. (Note that although consecutive case labels have consecutive line numbers in this abbreviated code example, the code following each case label in the actual menu manager occupy several lines.) The programmer should replace the words "new1_1", "new1_2", and "new1_3" in the case labels with the appropriate enumerators for the individual processing routines in the new processing class. Although there are three case labels in this code example, the programmer should add or delete case labels as appropriate to the

number of processing routines in the new class. The remainder of the code, from lines 26 through 31, is common to all menu managers and does not require any modification. The code following each of the case labels in the new menu manager is, as noted in Figure 4.16, the same as, or similar to, the example code of the previous Figure 4.14. The programmer will need to make the same kinds of modifications to this code as were discussed in connection with Figure 4.14. The programmer is encouraged to examine all of Img_Process_Manager()'s subordinate menu managers to see how they serve routines with different input & output requirements.

The function *Img_Process_Manager()* contains a single switch construct, with case labels for each of its subordinate menu manager. The programmer should add a new case label to this construct for the menu manager serving the new processing class, with the word following "case" equal to the enumerator for the new processing class. The programmer should then copy the code following one of the other case labels and modify it to call the menu manager for the new processing class. The code following the case labels of *Img_Process_Manager()*'s switch construct is of the form shown in Figure 4.17, where *MenuManagerName()* is the name of the appropriate subordinate menu manager. The programmer needs to modify only line 3 of the above code fragment to call the menu manager for the new class after copying the four lines of code from an existing case label. The programmer should also modify the preprocessor control line within *Img_Process_Manager()* that defines the word EXITVALUE; the value defined

for EXITVALUE should be incremented by the number of new processing classes being added to the image processing menu, so that it properly reflects the menu position of the "Exit" item on that menu.

```
1       HighLightItem(menu_index,value);
2       ClearInfoLine(instr_window_index,0);
3       MenuManagerName(menu_index,EXITVALUE);
4       break;
```

Figure 4.17. Example code for case labels in *Img_Process_Manager()*

The above completes the code modifications necessary to integrate a new processing algorithm or group of algorithms into HAPPI. The next step is to compile, and debug, if necessary, the modified code. Compilation of HAPPI is usually accomplished using the UNIX *make* utility. The details of *make* are beyond the scope of this document; we give here only a brief description of its operation. In a large program such as HAPPI, the source code is distributed over many files, and changes in one file may necessitate the recompilation of several other files. These file dependencies are explicitly declared in what is called a "make file" (the name of HAPPI's make file is "makefile"). The *make* utility reads the make file, checks the file dependencies, checks the date and time of last modification of all appropriate files, then selectively recompiles all files which have been modified since the last compilation and all of their dependent files. Thus, to recompile HAPPI, the programmer need only type "make" at the UNIX prompt while in the directory containing HAPPI's source

code and make file; *make* does the rest of the work. The programmer is referred to the Stellix Programmer's Guide or [UNIX references] for further information on *make*.

Once the programmer gets HAPPI to compile without error, he/she should test it thoroughly. The parameter fetching routine should be thoroughly exercised by checking to see that the default values for all input parameters are correct, and attempting to change all of the input parameters. The processing routine itself should be tested by using an input image for which the output may be easily predicted. For example, in testing HAPPI's two-dimensional FFT routine, a 2-dimensional rectangular pulse was used as the input image, with the resulting two-dimensional sinc function indicating the correct operation of the routine. Once the programmer is confident the new parameter fetching and image processing routines are working correctly, the parameter fetching routine should be moved to the file IPparams.c and the processing routine should be moved to the file IProutines.c, and HAPPI should be once again recompiled. Since IPparams.c and IProutines.c typically do not need to be recompiled often, this step helps keep the frequently recompiled files IPtest.c, IPtest2.c, and IPtest3.c relatively small, so that compile time is minimized.

The programmer is encouraged to the use the symbolic code debugger *dbx* in the event that his/her code does not run properly. Details of *dbx* are beyond the scope of this document; however a summary of some basic *dbx* commands is given here. The debugger is invoked by typing "dbx *filename*" at the UNIX prompt where *filename* is the name of the command used to invoke HAPPI; this command is "happi" on many systems where the program is

installed, but the version of the program maintained in the directory /home/catd/src on the Image Processing Lab's Stellar GS1025 computer is currently called "snappy". For *dbx* to do symbolic debugging, the source code of the program being debugging needs to have been compiled with the "-g" compiler option; this is taken care of by HAPPI's make file. Two useful *dbx* commands are *where* and *print*. The *where* command gives the *dbx* user a stack trace; this shows where in the hierarchy of function calls the error which caused the program to crash occurred. The *print* command simply prints the value of a variable at the time the program crashed. On-line help may be accessed from within dbx by typing "help" at the *dbx* prompt.

4.12 Common Programming Errors

In this section, we briefly describe some of the more common programming errors observed when new code has been added to HAPPI. While our coverage of programming errors cannot be exhaustive, it is hoped that this section will help the programmer avoid some of the errors committed by the original programmers of HAPPI.

A common cause of fatal errors (those that result in the program crashing) in HAPPI is attempting to access an array element that does not exist. HAPPI uses many dynamically allocated arrays, and the programmer may occasionally lose track of which arrays are currently defined and/or the current dimensions of those arrays. Messages such as "Segmentation fault" and "Bus error" issued by the operating system at abnormal termination of HAPPI frequently indicate such an error. The code causing the problem can

often be found using *dbx*. Another, related error which may result in the program crashing and the same error messages as above being issued is the failure to check for null pointers returned by memory allocation functions. The matrix allocation routines discussed in Section 4.6 return pointers to the amount of memory requested by the programmer only when it is possible for the system to allocate that memory. If the system cannot allocate the amount of memory requested, the routines return a null pointer, and it is up to the programmer to check for this.

The programmer is also responsible for freeing dynamically allocated memory once it is no longer being used. Failure to do so can result in a condition known as memory fragmentation, wherein new memory allocation requests cannot be satisfied due to the needed memory being tied up by data structures that have not been deallocated after they are no longer in use. The programmer should make a habit of typing in the appropriate memory deallocation routines at the end of his/her routine immediately after using an allocation routine.

If the default values of a processing routine's input parameters are not written correctly in the menu window created by the parameter fetching routine, the programmer should check the calls to *sprintf()* in the parameter fetching routine. This type of error is often caused by an incorrect conversion specification being used in the *sprintf()* call.

One potentially elusive source of error is the accidental use of function or variable names that are either already defined elsewhere in the code or are defined by UNIX. Becoming familiar with the file Globals.h will help the programmer to avoid some of these problems. If it is suspected that a variable

or function name is already defined, the programmer should search HAPPI's source code for the name using the UNIX *grep* command. If it is suspected that a function name is identical to a system call name defined by UNIX, the programmer should check to see if there is a UNIX manual page for the name using the UNIX *man* command.

Another elusive error is the accidental placement of a semicolon directly after the closing parenthesis of the conditional expression of a *for*, *while* or *do* loop. In C, executable statements are terminated with semicolons, so the programmer is used to placing a semicolon at the end of almost every line of code. If a semicolon is placed directly after the expression of a loop construct, however, the compiler will interpret this as a null, or "do nothing" statement, to be executed as many times as indicated by the loop construct's conditional expression, and it will appear as if the loop is not being executed at all. As image processing routines characteristically make extensive use of looping, this error occurs more often than might be expected.

4.13 Adding New Convolution Kernels to HAPPI

New convolution kernels may be added to HAPPI without writing a single line of source code. Under the "Convolution" menu item on HAPPI's Image Processing menu are (at this writing) two submenu items, "Template List", and "User-defined". The sub-submenu under the "Template List" item is built when HAPPI is invoked by the reading the data file "templates.happi" in HAPPI's source code directory. This file contains an arbitrary number of convolution kernel specifications of the form shown in Figure 4.18. Note that

this file is *not* a C source code file, and the lines of the file are not terminated with semicolons. All italicized fields in the above figure are to be defined by the user editing the templates.happi file. The *mynewkernel* field should be set

```
title = mynewkernel
size = sizeofnewkernel
row_1_data
row_2_data
:
:
row_n_data
hot_row = row
hot_col = col
denom = denominator
```

Figure 4.18. Convolution kernel format for kernels read from templates.happi file

to the text the programmer wants to appear in the Template List menu. The *sizeofnewkernel* field should be set to the (integer) length of one side of the new kernel. Note that only square convolution kernels may be defined in this file. The *row_1_data* through *row_n_data* fields should be set to the convolution kernel weights for each row of the kernel, respectively. The individual weights must be separated by spaces on each line, and only integer data are allowed in these fields. The *row* and *col* fields should be set to the row and column, respectively, of the convolution kernel to which the convolution sum will be accumulated. Usually, these are set to the center row and column. The *denominator* field is set to a floating-point number by which the convolution sum will be divided before its value is written to the destination image of the convolution routine. Typically, the value used is equal to the sum

of the convolution kernel weights. As an example, the kernel for a 3x3 uniform-weight lowpass filter is shown in Figure 4.19.

```
title = low_pass_1
size = 3
1 1 1
1 1 1
1 1 1
hot_row = 1
hot_col = 1
denom = 9.0
```

Figure 4.19. Convolution kernel for 3x3 lowpass filter

CHAPTER 5: DIGITAL X-RAY IMAGE FORMATION

5.1 Introduction

In this chapter, we provide as background the basic relations of X-ray image formation and discuss in general terms the formation of digital X-ray images as they are presented to HAPPI for processing. The remainder of this thesis will explore how several of HAPPI's image processing routines affect the size of idealized image features.

5.2 X-ray Radiography

Much of the discussion in this section paraphrases parts of the chapter on radiological methods in the text by Halmshaw (1987). The reader is referred to this text and its references for further details. X-rays are a form of electromagnetic radiation, of the same physical nature as visible light, with wavelengths of about 10 nm to 10^{-4}nm. The wavelength of X-rays allows them to penetrate all materials with partial absorption during transmission. X-rays travel in straight lines outward from a source, and for all practical purposes cannot be focused. Thus, in a typical radiography setup, a conical beam of X-rays emanates from the X-ray source. A radiograph is produced by placing an X-ray source and a piece of photographic film on opposite sides of the specimen to be examined, and exposing the film to the radiation transmitted through the specimen for a long enough period of time to sensitize the silver halide crystals in the film. The necessary exposure time will depend on the

intensity of the X-ray source, the sensitivity of the film, and the X-ray absorption properties of the specimen. The basic law of X-ray absorption is:

$$I_x = I_0 \exp(-\mu x) \tag{5.1}$$

where x is the thickness of the material, I_0 is the incident intensity of radiation, I_x is the transmitted intensity, and μ is a constant, known as the linear absorption coefficient, whose value depends on the material and the X-ray wavelength. Some proportion of incident X-rays will be re-emitted within the specimen as scattered radiation, and can, under some conditions, travel in a different direction to the primary beam. Equation 5.1 is strictly only valid for monoenergetic radiation and narrow-beam conditions under which the amount of scattered radiation reaching the detector is negligible, but is often applied in other than these conditions to determine an "effective" value of μ for practical applications.

The response of radiographic film to incident radiation is measured in terms of *optical density* D, which is defined as:

$$D = \log_{10}(I_0/I_t) \tag{5.2}$$

where I_0 is the intensity of light incident on one side of the film and I_t is the intensity of light transmitted through the film. Optical density of film is typically plotted as a function of the logarithm of the *exposure* E, which is

defined as the product of the incident X-ray intensity I and the exposure time t:

$$E = It \tag{5.3}$$

The typical "D vs. $\log_{10}E$" characteristic curve of a given photographic film will look like Figure 5.1.

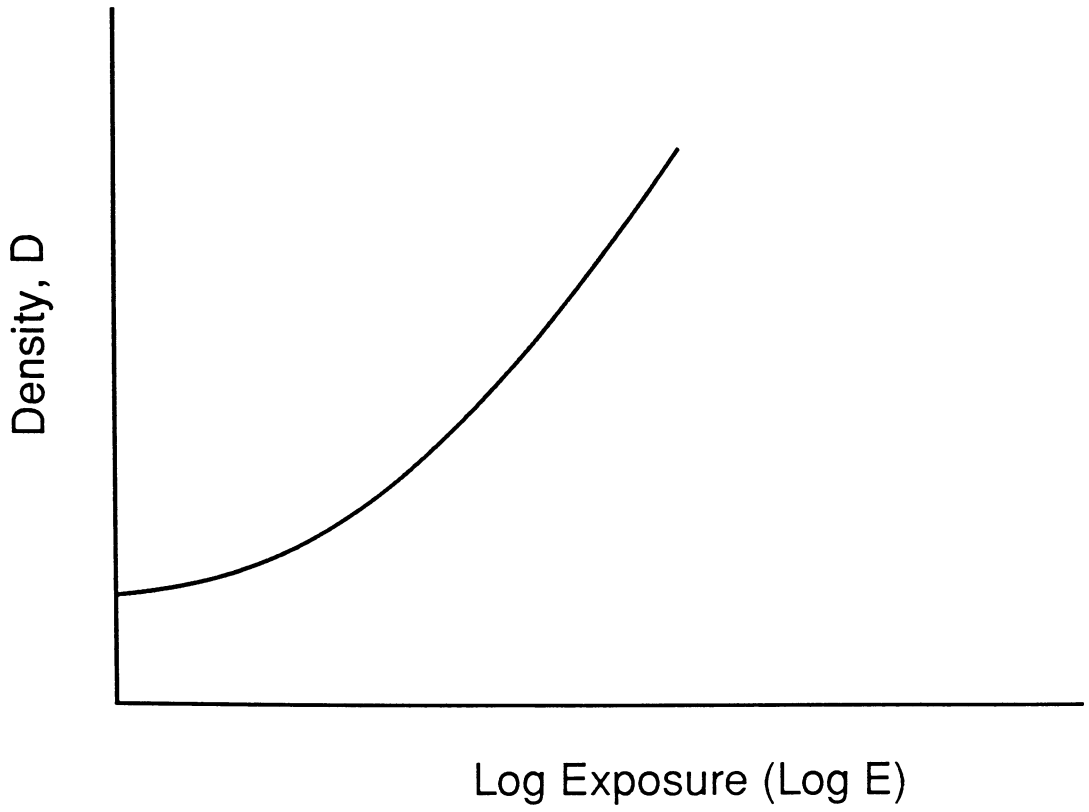


Figure 5.1. Typical density vs. log(exposure) curve

It may be seen that there is a portion of the film's characteristic curve for which density D is approximately linear in $\log_{10}E$. Strictly speaking, the slope of the curve is slightly greater at higher exposures. However, the resulting higher optical densities require much higher-powered light sources for proper viewing. There is thus a tradeoff between achievable contrast (which translates to specimen thickness sensitivity) and viewability (Halmshaw 1973). Workers in the X-ray Image Processing Group at ISU have found that for purposes of digital analysis, higher optical densities, with their larger dynamic range, are more desirable than lower optical densities. The video cameras and scanners typically used in digitizing radiographs usually have adjustable sensitivity that allows them to compensate for low light levels. When a radiograph is produced using exposures which keep the film density in the linear region of the characteristic curve (as is common practice), there will be a relatively simple and direct relationship between film density and material thickness. Provided that the assumptions behind Equation 5.1 hold, material thickness could theoretically be calculated using density measurements from the radiograph, film characteristic curve data, and Equation 5.1. However, in practice, it is suggested that such calculations be bypassed in favor of an empirical density-thickness calibration using a radiographic tool known as a step wedge (Halmshaw, 1979). A step wedge is simply a block of the same material as that being inspected (so as to have the same absorption coefficient μ), which has several graduated "steps" of increasing thickness machined into it. The wedge is placed by, and radiographed with, the specimen so that film density measurements from the specimen may be directly compared to density measurements for a known

material thickness. Such an empirical calibration automatically accounts for departures from the film's ideal characteristic curve and for errors due to Equation 5.1 not being exact. In this thesis we will assume that suitable calibrations can be done which will, for purposes of doing image processing, transform film density data into material thickness data.

On a radiograph, the scattering of X-rays within the specimen is manifest in the blurring of sharp edges, termed *unsharpness*, and in the reduction of contrast. If we consider a one-dimensional slice of an X-ray image (i.e., a film density function of a single spatial variable), the effect of scattering may be modeled using a *line spread function*, or LSF, which is the integration over one dimension of the two-dimensional point spread function, or PSF. This line spread function is convolved with the 1-d slice representing the ideal film density response in the absence of scattering to arrive at a slice which accounts for scattering. The scattering unsharpness line spread function has been found to be (Fishman, *et al.*, 1981, Notea, 1983):

$$LSF_S(x) = (a/2)\exp(-a|x|) \tag{5.4}$$

where the variable x represents distance along the 1-d slice, and a is a characteristic parameter whose value is determined by the radiographic system *and the material being examined.* The parameter a has units of inverse distance.

Another source of blurring in the radiograph is the finite spatial extent of the X-ray source. X-rays are emitted from every point of the source, and hence, any single point in the specimen is imaged on the film by X-rays from

the many spatially distinct points of the source. Blur due to the finite spatial extent of the X-ray source is termed *geometric unsharpness*, and may be modeled using a LSF of the form (Notea, 1983):

$$LSF_g(x) = (1/U_g)\{u(x + U_g/2) - u(x - U_g/2)\} \tag{5.5}$$

where U_g is the geometric unsharpness parameter, which has units of length, and $u(x)$ is a unit step function. Thus, $LSF_g(x)$ has a rectangular distribution; it assigns equal weight to all values of the ideal film density for x in the range $[-U_g/2, U_g/2]$.

5.3 Typical Apparatus for Digital Processing of X-ray Images

The apparatus used by the X-ray Image Processing Group in the Electrical and Computer Engineering Department at Iowa State University for processing NDE X-ray images is shown schematically in Figure 5.2. This setup is typical of those used for a variety of industrial applications. The radiograph is illuminated from below by the lightbox, and the transmitted light is converted to an analog electronic signal by the video camera. The video signal from the camera is then digitized by the frame grabber and moved to the memory and/or hard disk of the computer in which the frame grabber is installed. The resulting digital image is then transmitted across an Ethernet network, and processed and displayed on a powerful central host computer.

Each of the components in the path from the original specimen to the digital representation of the radiograph within the host computer is a source

of noise. The quantum nature of electromagnetic radiation results in photon counting noise from the X-ray source. Porosity, graininess, or other texture of the specimen may show up in the radiograph. If this texture is considered

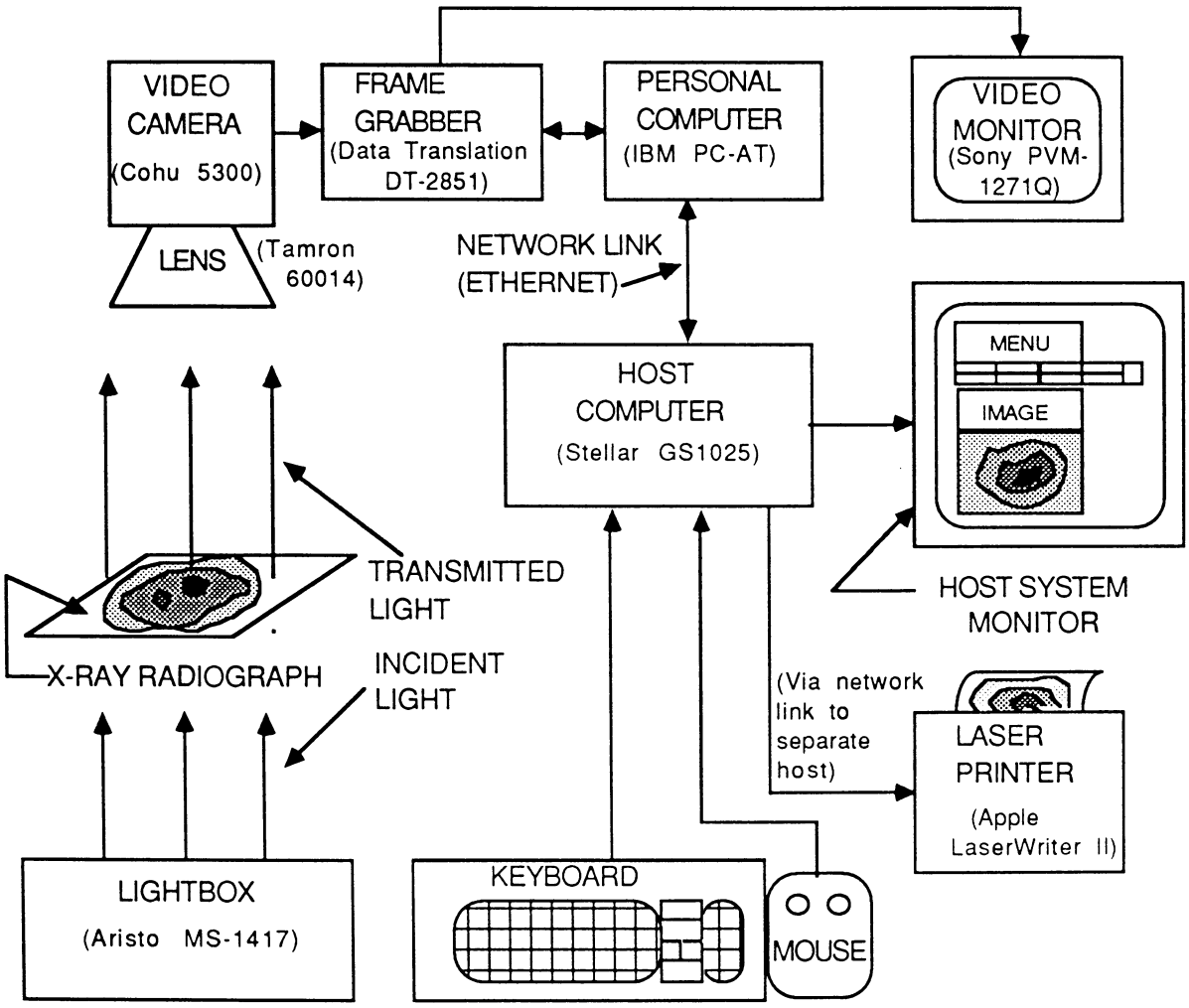


Figure 5.2. Image processing group's digital image processing setup

normal (i.e., not indicative of a flaw), but makes it more difficult to accurately detect and measure true flaws in the specimen, it may be considered to be noise. The radiographic film will introduce film grain noise, due to the finite

size of the silver halide crystals in the film emulsion. The lightbox used to illuminate the radiograph will have some flicker and field non-uniformity. The image sensing element and electronics in the video camera will produce noise. The frame grabber electronics will also produce noise, and as with all digitized signals, the image that is finally processed by the host computer will contain quantization noise.

The Central Limit Theorem of statistics states that the probability distribution of the sum of independent random variables, in the limit as the number of random variables in the sum goes to infinity, is gaussian, regardless of the distributions of the individual random variables in the sum. This theorem is behind the assumption of gaussian noise made in many analyses in the study of random phenomena. Rather than attempting to model all of the independent noise sources in the lab setup of Figure 5.2, we will instead in this thesis appeal to the Central Limit Theorem. For purposes of investigating how HAPPI's various processing routines affect the size of image features, we will use idealized image features bathed in additive white gaussian noise.

To inquire into the plausibility of using gaussian noise in our test images, we extracted portions of a digitized image of a real radiograph of a flat metal plate. The extracted image regions were selected so as to have - as well as could be determined by eye - stationary mean and variance. Histograms of these image regions were computed, and are shown in Figure 5.3. With a little imagination, the reader may see that the histograms of Figure 5.3 appear to have an approximately gaussian shape. While the gaussian noise in our test images may not always accurately model the complex real-world noise

processes present in a digitized radiograph, the measurements made on the test images and presented herein are, at the least, a reference point for more detailed future work.

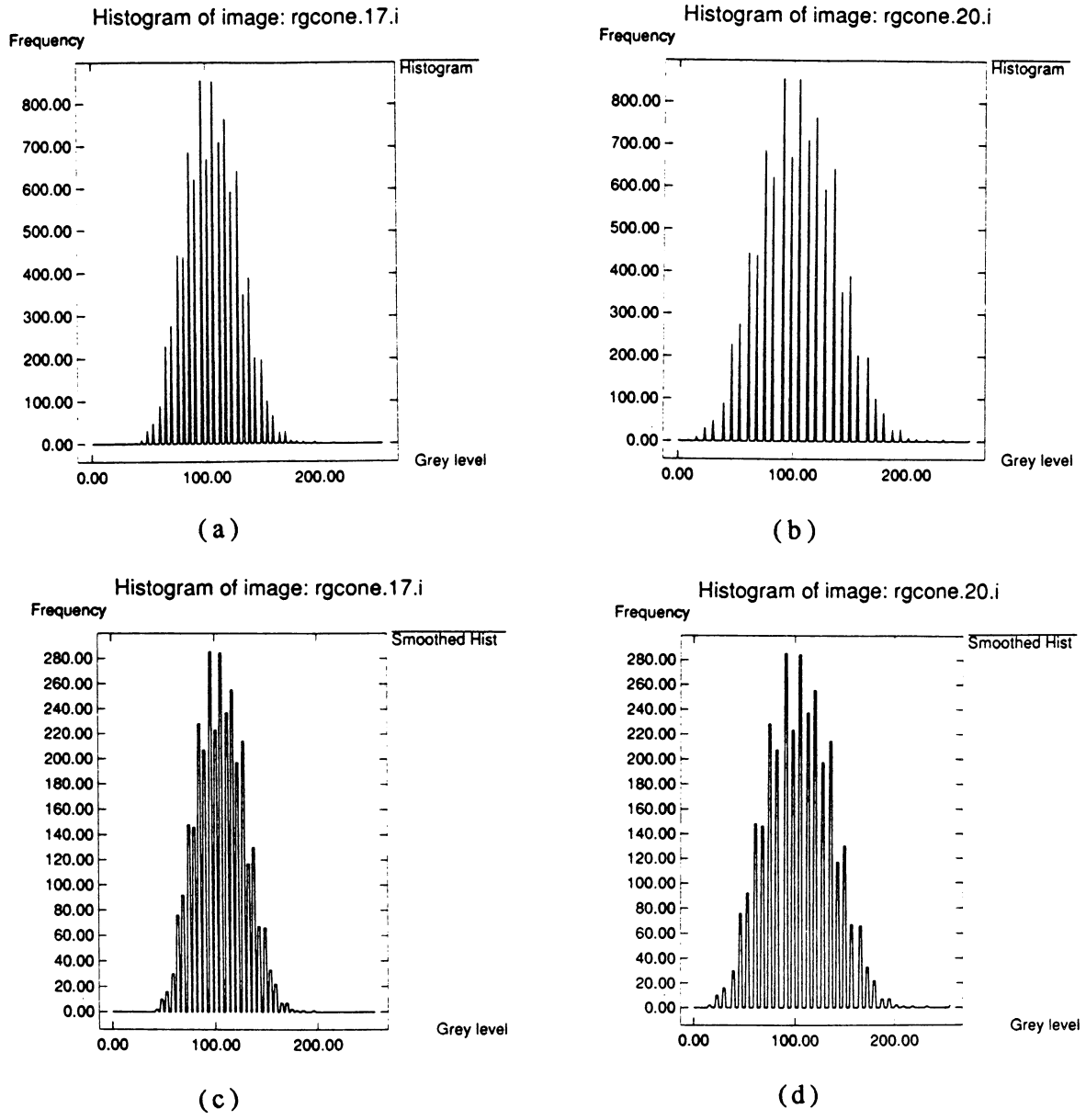


Figure 5.3. Histograms of two separate regions of an image with locally stationary mean and variance: *(a)* Histogram of first region; *(b)* Histogram of second region; *(c)* and *(d)* Three-point smoothed versions of *(a)* and *(b)*, respectively

CHAPTER 6: FEATURE SIZE MEASUREMENT

6.1 Introduction

In this chapter, we discuss the issues involved in determining the size of an image feature in a digitized X-ray radiograph. The methods used in this study, and the rationale behind them, are presented. The study in this thesis is concerned with *presenting data* on the influence of several of HAPPI's image processing routines on image feature size. Thus, the size measurement methods used are intended to be reasonable, and not necessarily optimal. It is hoped that, in the interest of quantitative NDE, future studies might explain the observations presented here with a theoretical formulation.

6.2 Feature Size Measurement and Edge Detection

A radiograph is, ideally (i.e., not accounting for noise and unsharpness), a two-dimensional projection of the three-dimensional distribution of the X-ray absorption coefficient in a specimen. As the X-rays travel in a straight path from the source through the specimen to the film, they are attenuated by an amount that depends on the absorption coefficient of the specimen material and the distance traveled through the material, as per Equation 5.1. We may speak of the *through-thickness* of a specimen (or of a flaw within the specimen) along a particular X-ray's path as the distance between the X-ray's entry and exit points on the specimen (or flaw). In many NDE situations, one is concerned with locating and determining the size of

voids or cracks in a material. Such types of flaws are usually filled with a gas, often air, and have an absorption coefficient orders of magnitude lower than that of the specimen material. Thus, an X-ray passing through such a flaw in a specimen of a given through-thickness will be attenuated less than one passing through the same specimen with no such flaw present, with the difference in attenuation depending upon the through-thickness of the flaw and Equation 5.1.

The types of measurements we will be concerned with in this thesis will be those of dimensions in the "film plane", that is, distances between points on a radiograph. These distances represent *projections* of dimensions of actual physical dimensions in a specimen. To keep the study basic, the image features studied are idealized models of a square flat-bottom hole in a flat plate, radiographed with a parallel-beam X-ray source directed perpendicular to the bottom of the hole. In terms of an idealized image (in which noise and unsharpness are not modeled), this translates to a uniform image background of one single grey level (modeling the flat plate), with a square foreground of a lower uniform grey level (modeling the hole). The actual images used had a foreground of higher intensity than the background, however, this does not introduce any inconsistency, as the edge location measurements taken would be the same had the foreground had the lower intensity. The square foreground region was placed at the exact center of all test images. The test image dimensions were 511x511 pixels, and the square image feature dimensions were 101x101 pixels. Figure 6.1 is an example of one of the test images used (note that any streaking in this particular image is due to the printer used and is not present in the actual image):

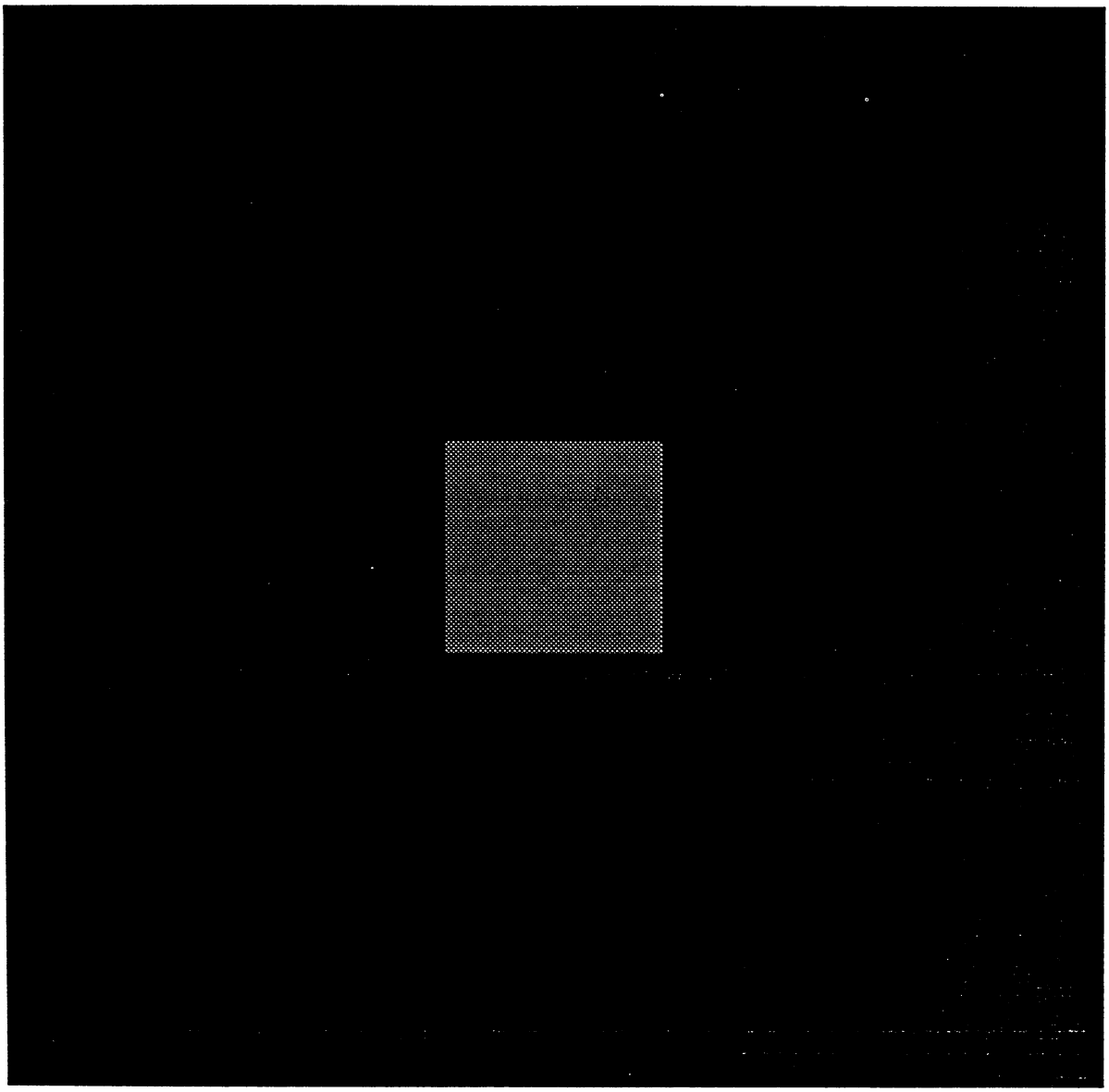


Figure 6.1. Image modeling ideal square flat-bottom hole

Determination of image feature size implies determination of the feature's edge locations. The size of the feature in a particular direction in the film plane is then just the distance between edge locations along that direction. Thus, our problem is one of edge detection and registration. In the absence of noise, the edges of a feature may be located precisely with respect to an edge definition. This definition may be given as, for example, the locus of points of a particular intensity somewhere between the peak intensity of the feature and the background intensity. Fishman *et al.* (1981) use such an edge definition to determine edge locations for a few ideal flaw geometries when scattering unsharpness is modeled using Equation 5.4. Their methods do not address edge location in the presence of noise, and depend upon knowing radiographic system parameters (namely the constant a in Equation 5.4).

In the presence of noise, edge locations must be *estimated* using as much of the relevant available data, which implies smoothing of the data in the neighborhood of the edge. Smoothing implies a tradeoff between detection and localization (i.e., accurate registration of location) of edges, as has been noted by Canny (1986) and Bergholm (1987). Canny illustrates that there is a natural sort of "uncertainty principle" in the edge detection problem. Maximizing accurate edge detection (i.e., having a high detection rate of true edges while having a low false-alarm rate) in the presence of noise amounts to maximizing the signal-to-noise ratio (SNR) in the vicinity of the edge, which is done by some sort of smoothing. However, too much smoothing can smear, or displace, an edge, resulting in inaccuracies in size measurements.

In analyses of one-dimensional step edge profiles bathed in additive white gaussian noise, Canny, Bergholm, and others have found that when the

definition of edge location is taken as the maximum of the gradient magnitude of the edge profile, the gaussian function is the most reasonable smoothing function for estimating edge locations. Here, "most reasonable" is meant in the sense of simultaneously maximizing SNR and edge localization while suppressing spurious response and being computationally efficient. In his analysis of one-dimensional edge profiles, Canny assumes that two-dimensional edges have locally constant cross-section; this assumption is true of smooth edge contours and of ridges, but not of corners. By "locally constant cross-section", it is meant that in a neighborhood about the edge, image intensity is constant along lines perpendicular to the edge direction. Figure 6.2 illustrates a region of an edge with a locally constant cross-section:

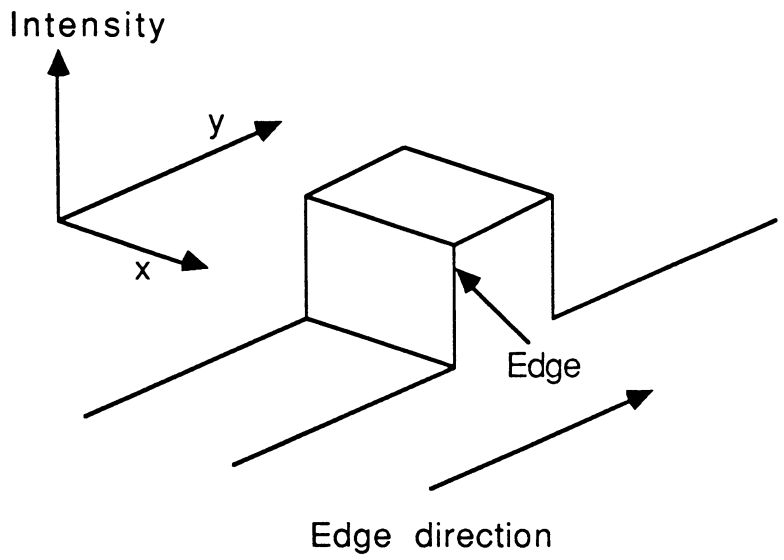


Figure 6.2. Edge with locally constant cross-section

The idea of smoothing or averaging an image in the direction perpendicular to the edge direction is important to convolution-based edge detectors and to the measurements presented in this thesis. When the underlying edge signal in a noisy image has locally constant cross-section, and the image noise process is stationary, we may diminish the noise variance (thereby improving the SNR and thus improving edge detection) *without smearing the edge* by forming an average of the image pixels along the direction perpendicular to the edge direction. The attainable improvement in edge detection and localization performance will depend upon the noise amplitude and the length of the locally constant cross-section: the longer the locally constant cross-section, the more we can expect to reduce the noise variance, and thus to increase edge detection performance.

Canny's edge detection operators are designed such that they smooth an image in the direction perpendicular to the edge direction. These operators are a set of convolution masks which have a nearly rectangular profile in one direction, and which in the perpendicular direction have a profile that is the derivative of the gaussian function. Note that convolving with the derivative of the gaussian is equivalent to convolving with a gaussian and then taking the derivative, due to the following property of the convolution integral (see Haykin, 1983, p. 39 for a proof):

$$\text{If } f(x) = g(x)*h(x), \quad \text{then}$$

$$df(x)/dx = (dg(x)/dx)*h(x) = g(x)*(dh(x)/dx) \tag{6.1}$$

where * denotes convolution and $df(x)/dx$ is the derivative of $f(x)$. Thus, Canny's edge detection operators effectively take the derivative of the gaussian-smoothed image in one direction while simultaneously forming an essentially unweighted average in the perpendicular direction. Figure 6.3 depicts orthogonal profiles and a grey scale display of Canny's edge detection operators.

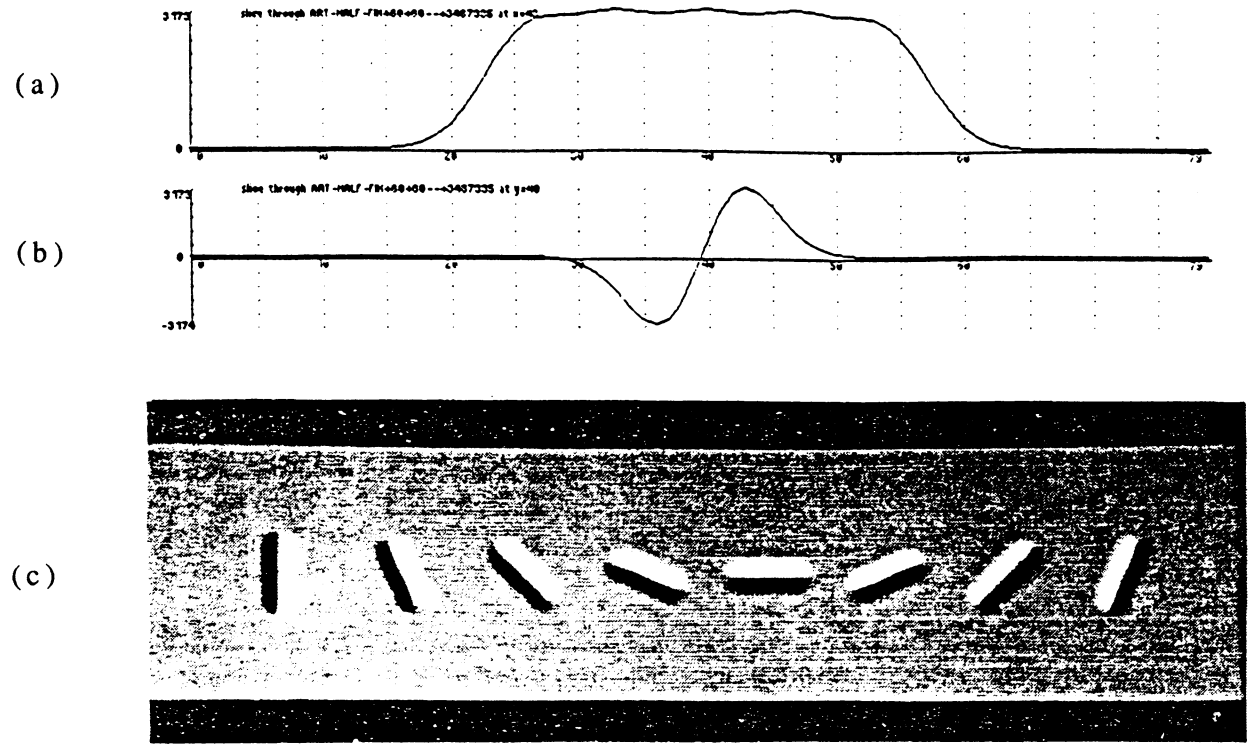


Figure 6.3. Canny's edge detection operators *(a)* Profile perpendicular to edge direction; *(b)* Profile parallel to edge direction; *(c)* grey scale display of several masks

We note here that traditional edge detection operators, such as the Sobel masks, also do some averaging (albeit somewhat weighted) perpendicular to the edge direction. The Sobel mask for vertical edges is as shown in Figure 6.4. When this mask is convolved with a vertical edge, the contribution to the convolution sum from the mask weights of 1, 2, and 1 in the bottom three pixels of the mask constitutes a weighted average of image pixels in the horizontal direction. Similarly, the contribution from the mask weights in the upper three mask pixels is also a weighted average of image pixels in the horizontal direction. For a perfectly vertical edge, this averaging has the effect of smoothing noise without smearing the edge.

- 1	- 2	- 1
0	0	0
1	2	1

Figure 6.4. Sobel mask for vertical edges

6.3 Measurement Methods

In this study, we have chosen to measure edge locations in a one-dimensional sense. For our test images (which are noisy versions of the square flat-bottom hole of Figure 6.1), the direction of the left and right edges is perfectly horizontal, so we have measured edge locations in each (horizontal) image row independently, using only a single row of data for each measurement. The edge location and feature size measured from a single row may thus be regarded as random variables with a different realization for each image row. From our set of row-by-row measurements, we calculate a mean and variance of edge location and image feature size. In what follows, mean and variance of edge location are denoted by μ_e and σ^2_e, respectively, while mean and variance of feature size are denoted by μ_f and σ^2_f, respectively. Standard deviation of edge location and feature size are thus denoted by σ_e and σ_f, respectively.

For a test image composed of the flat-bottom hole of Figure 6.1 bathed in a stationary noise field, each image row will have the same SNR. Thus, the statistics for edge location and feature size calculated from our row-by-row measurements will give some sense of how precisely we can locate an edge with a given SNR using a single one-dimensional slice through the edge in the edge direction. This information in turn gives us a sense of the achievable increase - through averaging perpendicular to the edge direction - in edge location performance for a given image feature having locally constant cross-section. To visualize the above ideas, consider the image of a crack-like flaw

illustrated in Figure 6.5 and the task of estimating the crack width in the presence of noise. If we take a single 1-d slice across (i.e., normal to the direction of crack propagation) the crack in the region of constant cross-section and use it to estimate the crack edge locations and crack width by some edge detection scheme, our estimates will be realizations of random variables with certain means and variances (namely μ_e and σ^2_e for edge location and μ_f and σ^2_f for feature size). Since the crack has locally constant cross-section along some part of its length, we can reduce image noise variance by averaging in the direction perpendicular to the crack edges (i.e., along the length of the crack) over the region of constant cross-section. If we then use

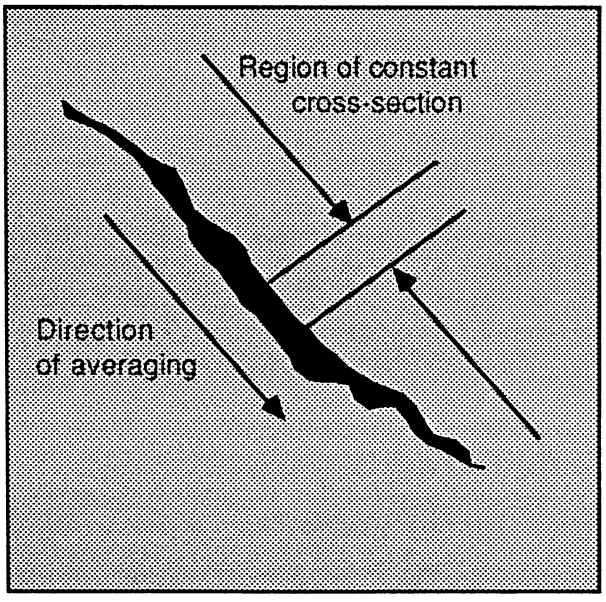


Figure 6.5. Crack-like image feature with region of constant dimension

the same 1-d edge detection scheme on such an "averaged slice," our edge location and feature size estimates will again be realizations of random variables, but with improved mean and variance. (An "improved mean" is one which is closer to the actual value being estimated, and an "improved variance" is simply a smaller variance.)

For this study, two methods of locating edges in one-dimensional slices of a noisy image were tried. Both methods used as a first step the smoothing of the image slice by a (one-dimensional) gaussian function. The first method then searched each smoothed slice for the location of pixels with grey level halfway between the grey level of the image feature and that of the image background; the second method searched each smoothed slice for the location of the gradient maximum. We will henceforth refer to the first method as the "half power point method" and the second method as the "gradient maximum" method. Implementation details of the two methods are discussed later in the chapter.

The gaussian blur function used in both methods of edge location has one adjustable parameter, namely the standard deviation, σ, of the gaussian. We will refer to this σ as the "blur parameter", and denote it as σ_b. This parameter controls the tradeoff between detection and localization, and its optimal value for a given image will depend on the image's SNR. It was thus necessary to determine what value of σ_b to use for an image of a given SNR. As mentioned at the beginning of the chapter, we were not primarily concerned with finding absolute optimal measurement methods; rather, we only required that our measurements be reasonable and consistent. Thus, a detailed analysis

to find precise values of the optimal blur parameter for a given SNR was not conducted. Rather, reasonable values of blur parameter as a function of SNR were experimentally determined. Our investigation of the influence of HAPPI's processing routines on image feature size then was conducted by using the *same value of blur parameter* for measuring edge location and feature size in the pre-processed and post-processed images. In this way, we were applying the *same* edge location operator to a given pair of pre-processed and post-processed images in order to quantify the effects of the processing routines themselves. The scheme for measuring the effects of processing on feature size is illustrated in Figure 6.6.

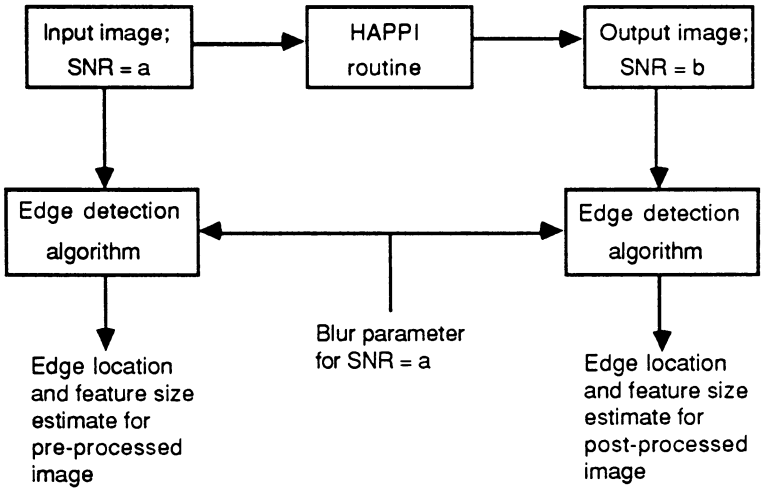


Figure 6.6. Scheme for measuring effects of processing on feature size

6.3.1 The Program `mrowblur`

A program, `mrowblur` (for "multiple-sigma row blur"), was written to determine a reasonable value of blur parameter at a given SNR. Inputs to the program were as follows: a (noisy) test image; a starting and ending row in the image; an initial, increment, and final value for the blurring parameter σ_b; the blur function length in terms of σ_b (i.e., number of sigmas); a flag selecting either the gradient maximum method or the half power method of edge detection; and, if the half power method was selected, the image grey levels of the foreground (feature) and background. The program's operation was as follows: The range of test image rows specified by the inputs was blurred with a 1-d gaussian having the user-specified initial value of σ_b, and edge locations in each gaussian-blurred row were determined by the user-specified method. The amplitude of the 1-d gaussian was adjusted for each value of σ_b so that the total area under the gaussian was identically equal to one.

To find edge locations using the gradient method, the program computed the *central difference* of each gaussian-blurred row. The central difference $y[n]$ of a 1-d sequence $x[n]$ is defined as:

$$y[n] = x[n+1] - x[n-1] \tag{6.2}$$

For a data set $x[n]$ defined on the interval n=0 to n=N, $y[0]$ and $y[N]$ cannot be calculated from the above equation, and must be otherwise defined or left undefined. In our case, the edges of image features were nowhere near the

border of the image, and so the first and last elements of the central difference formed from each gaussian-blurred image row were left undefined. The central difference of each gaussian-blurred image row was then searched for two maxima, and these maxima were declared as the edge locations for that row. In searching for the two maxima, advantage was taken of the a priori knowledge that the image feature was brighter than the background and centered in the image. To help the reader visualize the following discussion, Figure 6.7 shows an unblurred image row (a), the same row after gaussian blurring (b), and the central difference of the gaussian-blurred row (c).

Let us denote the sequence containing the central difference of a gaussian-blurred image row as $y[n]$, with n ranging from 0 at the leftmost pixel in the the row to N at the rightmost pixel. To find the left edge location in a particular row, the program searched the sequence $y[n]$ starting at its middle data point ($y[N/2]$ for even N or $y[(N-1)/2]$ for odd N) and scanned to the left, searching for the *most positive* value of $y[n]$. To find the right edge location, the program searched $y[n]$ from the middle data point and scanned to the right, searching for the *most negative* value of $y[n]$.

To find edge locations using the half power method, a simple thresholding scheme was used, and the program again took advantage of the a priori knowledge that the image feature was brighter than the background and centered in the image. Let us denote the sequence containing a gaussian-blurred image row as $x[n]$, with n ranging from 0 at the leftmost pixel in the the row to N at the rightmost pixel. To find the left edge location in a particular row, the program searched the sequence $x[n]$ starting at its middle

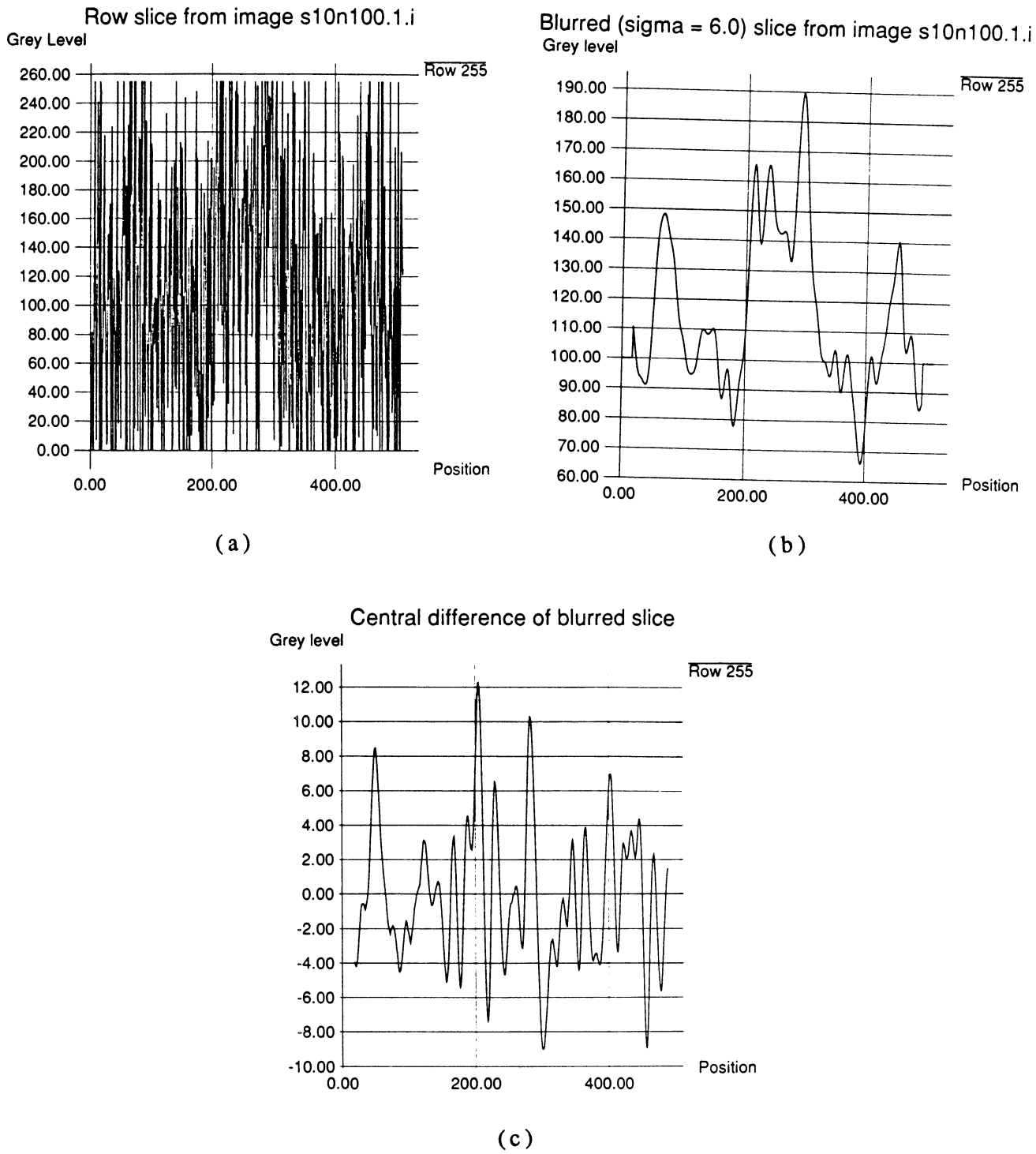


Figure 6.7. Image row at three steps of gradient maximum method of edge detection *(a)* Unblurred image row; *(b)* Gaussian-blurred version of (a); *(c)* Central difference of (b)

data point (x[N/2] for even N or x[(N-1)/2] for odd N) and scanned to the left, searching for the first value of x[n] to fall below the grey level halfway between the feature and background grey levels (we will call this grey level the "half power grey level"). To find the right edge location, the program searched x[n] from the middle data point and scanned to the right, again searching for the first value of x[n] to fall below the half power grey level.

The above implementation of the half power method was found not to have very good edge location performance for images with low SNR. The reason for this poor performance was evident from examining gaussian-blurred slices from low-SNR images and histograms of edge locations found using the method. In the histograms of edge locations, it was seen that several edges were declared very close to the center of the image feature (recall that the search algorithm starts looking for the edge locations at the center of the image feature). In the gaussian-blurred slices, it was seen that the noise amplitude was large enough to make the image slice dip below the half power grey level at many points inside the image feature. From these observations it was concluded that the simple thresholding scheme used in the half power edge location method made the method too sensitive to noise. To improve performance, a modified thresholding scheme, based on one used by Canny, was implemented. This scheme uses two threshold values and a sort of hysteresis to lower the sensitivity of edge location to noise. The sequence x[n] is again scanned to the left and to the right starting from the middle data point. However, the scan algorithm initially searches for the first value of x[n] to fall below a threshold *lower* than the half power grey level, then reverses scan direction and looks for the first value of x[n] to go back above

the half power level. (The second, lower threshold in this scheme was set to 0.25 times the differences between the (higher) feature grey level and the (lower) background grey level.) The thresholding-with-hysteresis scheme brought about some improvement in the edge location performance of the half power method, but at low SNR values, the method still did not compare very favorably with the gradient maximum method.

Upon finding edge locations for each image row using the user-specified edge detection method, the mean and variance of the locations of the left and right edges were calculated and stored. The program then incremented the value of σ_b and repeated the entire process until the final value of σ_b was reached. The stored values of edge location mean and variance for each σ_b value were then written out to a file in a format suitable for making a range bar plot of $(\mu_e \pm 1\sigma_e)$ vs. σ_b. The program "xgraph" was then used to generate these plots.

A sequence of test images with decreasing SNR was created using the Add White Noise function in HAPPI. The definition of SNR used for these images was the *ratio of the signal strength to rms noise level;* other definitions of SNR, such as the square of this ratio, and the logarithm of the square of this ratio are also used in the literature. By "signal strength," we mean the difference between the foreground (i.e., the signal) and the background grey levels. The original noiseless image from which the sequence was created had a background grey level of 100 and a foreground grey level of 150, yielding a signal strength of 50 grey levels. These foreground and background values were picked so as to "center" the image feature within the 0-255 grey level dynamic intensity range of HAPPI's image format, and so that noise could be

added to the original image at rms levels high enough to achieve an SNR as low as 0.25 without the noise significantly saturating the 0-255 grey level dynamic range.

6.3.2 Determination of Critical Values of Blur Parameter

The program `mrowblur` was run on each test image in the sequence using the gradient maximum method, and the results were graphed in a sequence of range bar plots. For the most part, the graphs appeared well-behaved. At any given SNR, edge location variance σ^2_e decreased, and mean edge location μ_e moved closer to true edge location, as the blur parameter σ_b increased. However, some anomalies showed up in one or two plots, and were found to be due to one or two outliers in the histograms of edge locations. The program `mrowblur` was thus modified to discard these outliers in calculating edge location mean and variance. The modified version of `mrowblur` discarded edge locations outside the range $\pm 2\sigma_e$ in the histogram of edge locations and recalculated the values of μ_e and σ_e. This version of `mrowblur` was then run again on the sequence of test images, and another sequence of range bar plots was made. This sequence is shown in Figure 6.8. From the sequence of plots in Figure 6.8, we may observe the following:

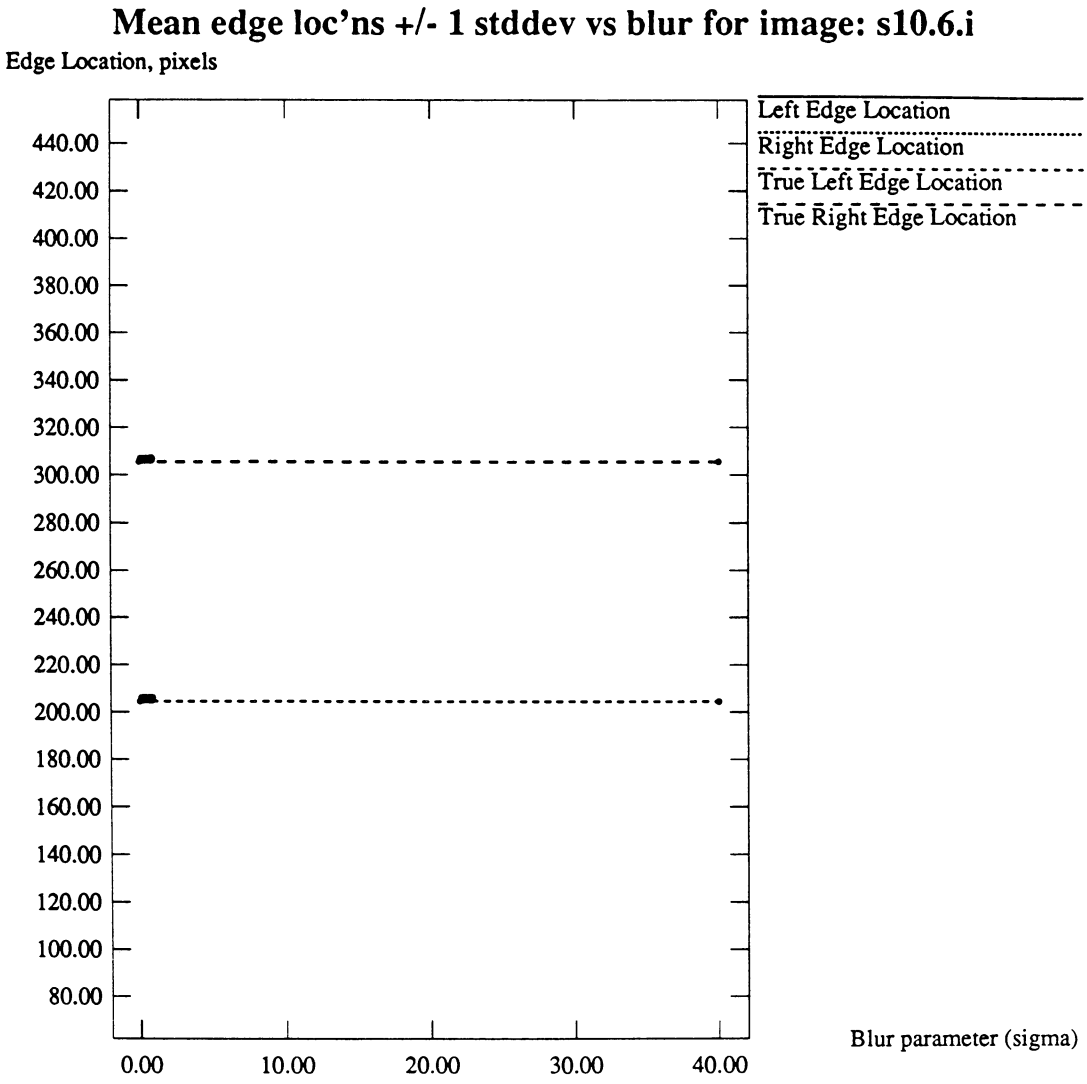
1) At a given SNR, edge location standard deviation σ_e and bias (i.e., magnitude of the difference between mean edge location μ_e and true edge location) decrease rapidly with increasing σ_b at low values of σ_b, but slow or stop decreasing with σ_b once a *critical value* of σ_b is reached. The critical value

of σ_b is thus the smallest (and therefore most computationally efficient) value of blur parameter that will yield the best attainable edge location estimate with the present edge location scheme.

2) The critical value of σ_b on each range bar plot was higher for images with lower SNR, and lower for images with higher SNR. For images with SNR above say 10, the critical value of σ_b was so low that for all practical purposes of edge location, it could be considered to be zero (i.e., no blurring is required for edge detection in an essentially clean image).

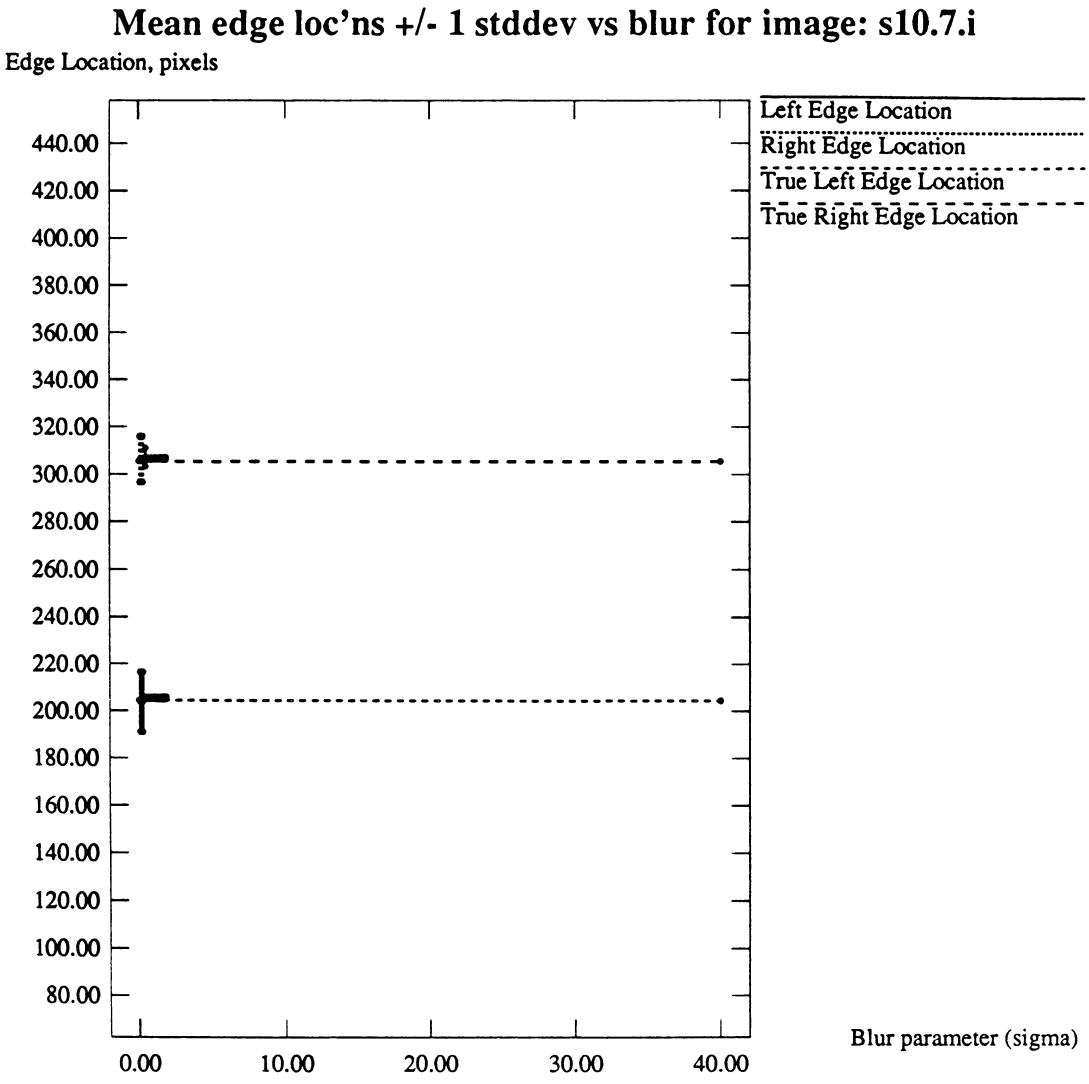
3) The terminal value reached by σ_e at the critical value of σ_b was larger for smaller SNR's, and smaller for larger SNR's.

The critical values of σ_b at each SNR were picked off of the plots of Figure 6.8 by eye, and plotted as a function of SNR as shown in Figure 6.9. The critical σ_b values were selected from each plot as the value of σ_b at which the edge location bias and standard deviation for both the left and right edge ceased to decrease with increasing σ_b. For parts (f), (g), and (h) of Figure 6.8, edge location bias and standard deviation fluctuate slightly about their terminal values as σ_b continues to increase beyond its critical value, making the selection of the critical σ_b values somewhat subjective. In selecting critical σ_b values from these three plots, we took into account the notion that the critical σ_b value should be a strictly decreasing function of SNR, and were



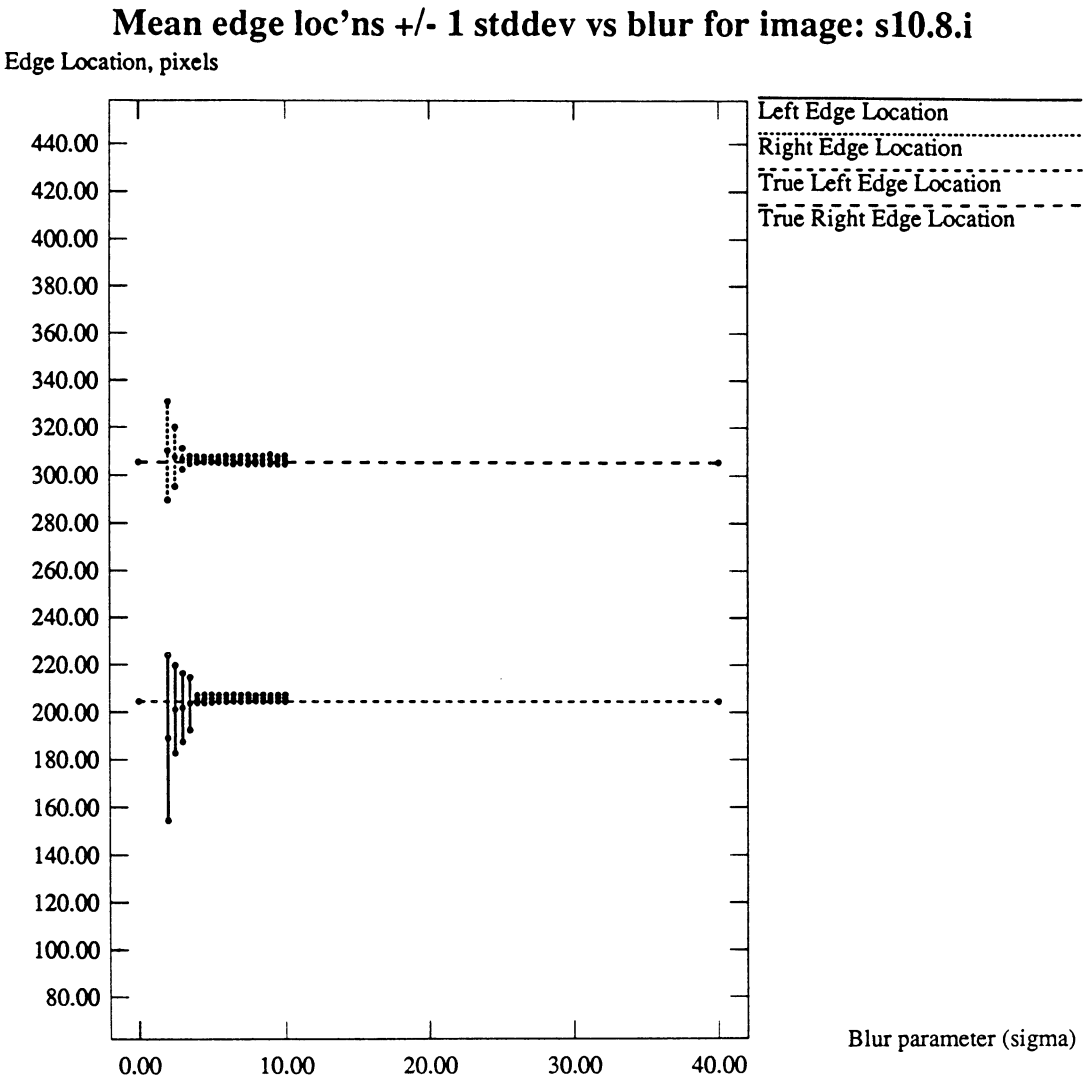
(a)

Figure 6.8. Sequence of range bar plots of mean edge location ± one standard deviation vs. blur parameter σ_b, calculated using gradient maximum method for various values of SNR *(a)* SNR = 10.0



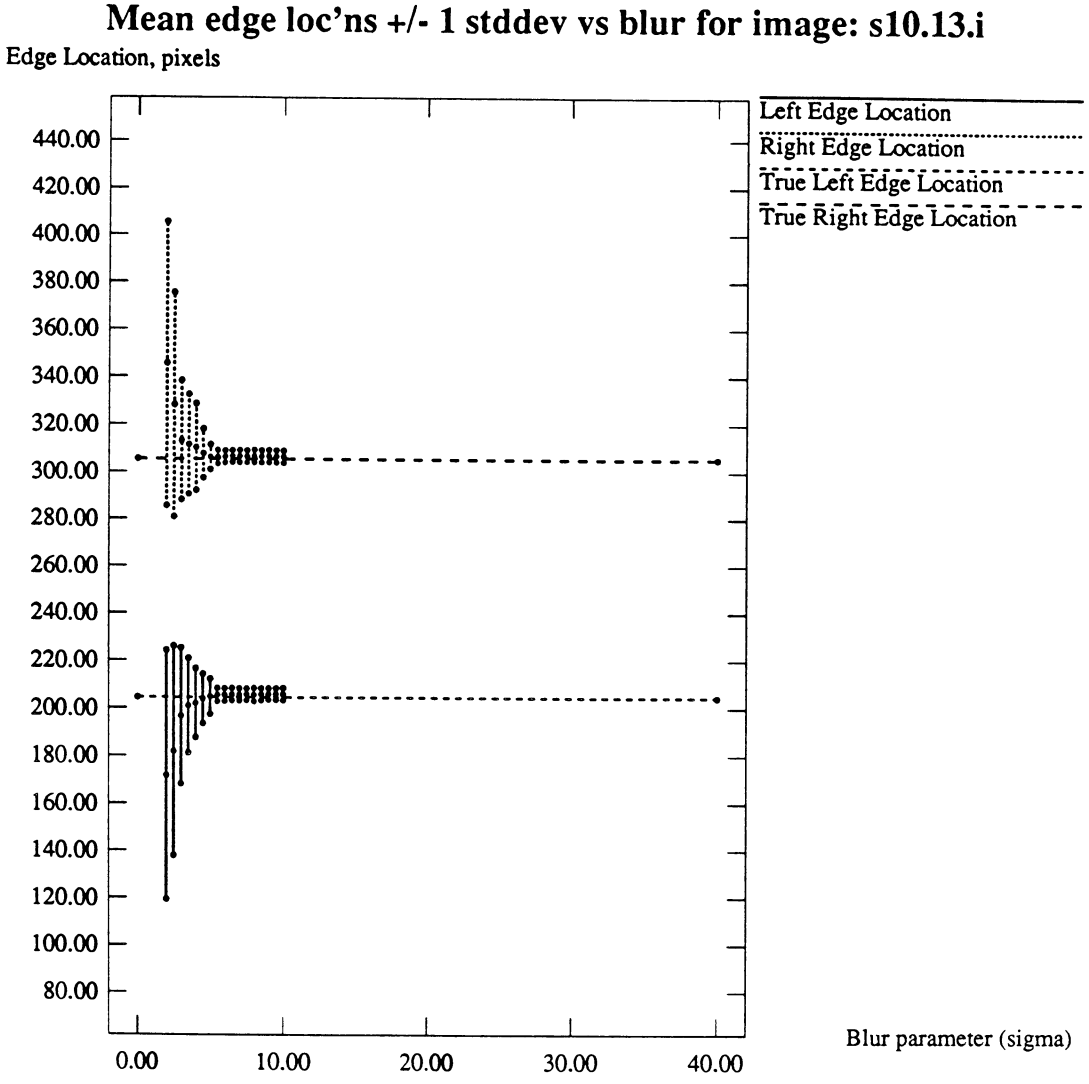
(b)

Figure 6.8. (cont'd) *(b)* SNR = 5.0



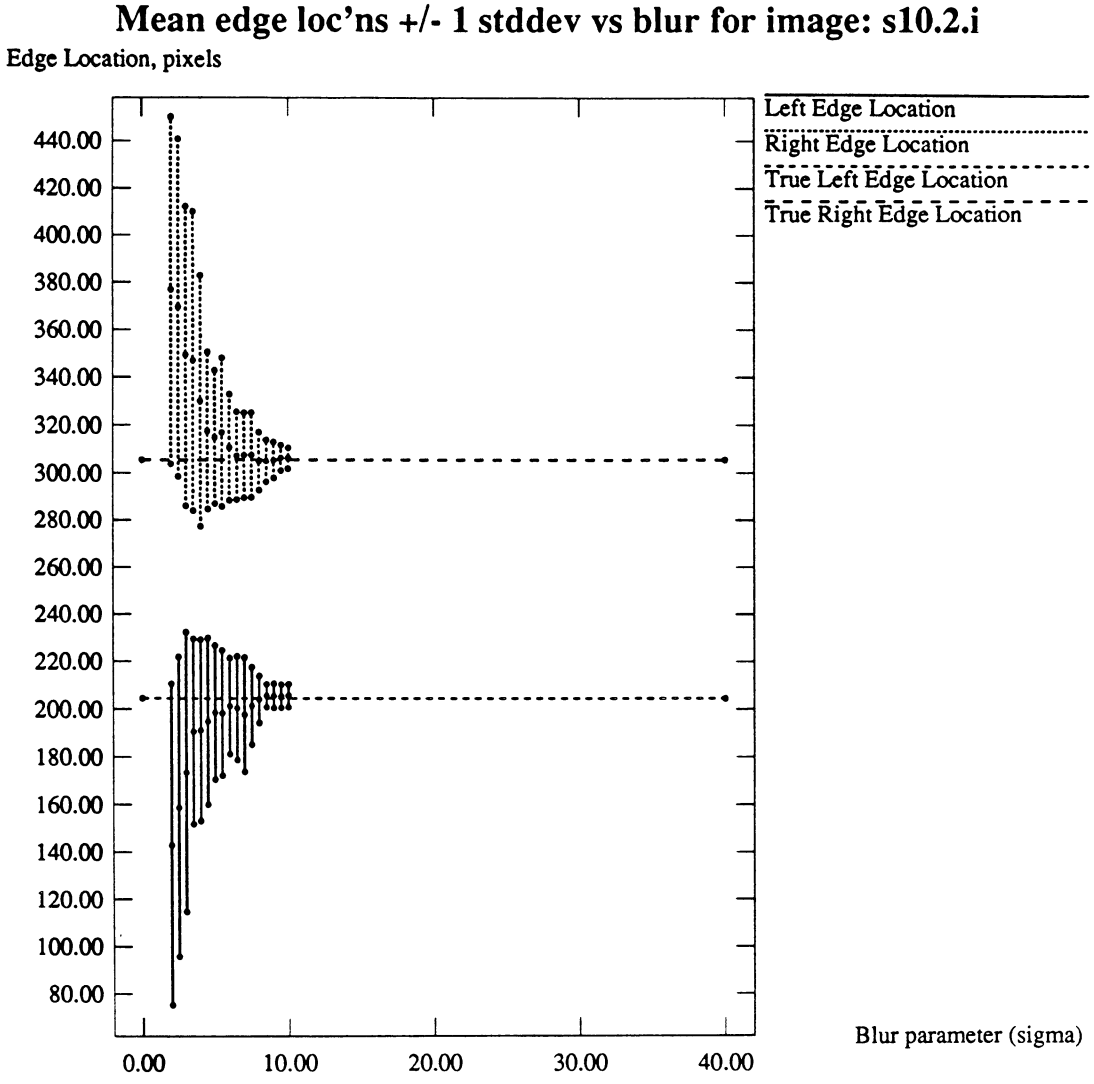
(c)

Figure 6.8. (cont'd) *(c)* SNR = 2.0



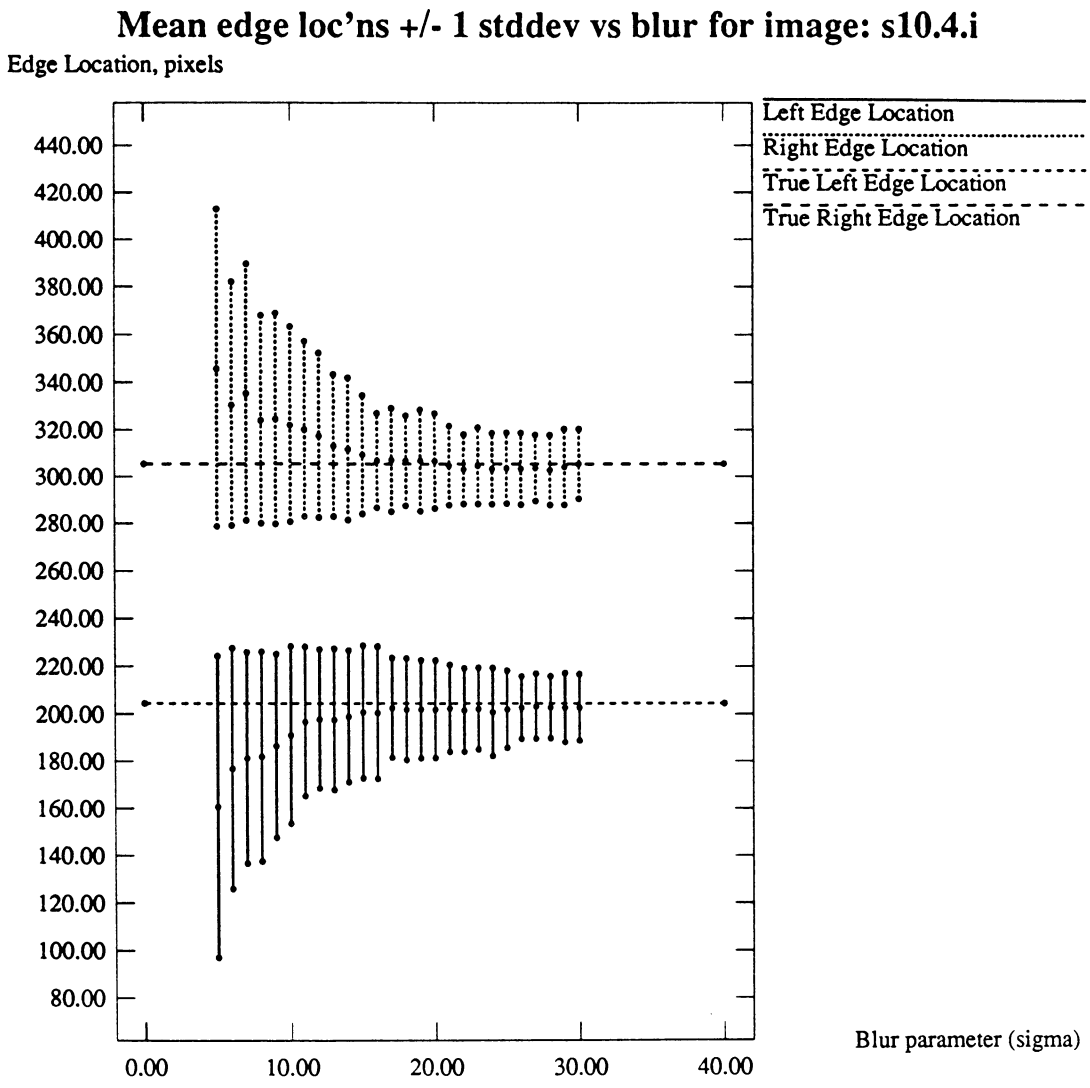
(d)

Figure 6.8. (cont'd) *(d)* SNR = 1.5



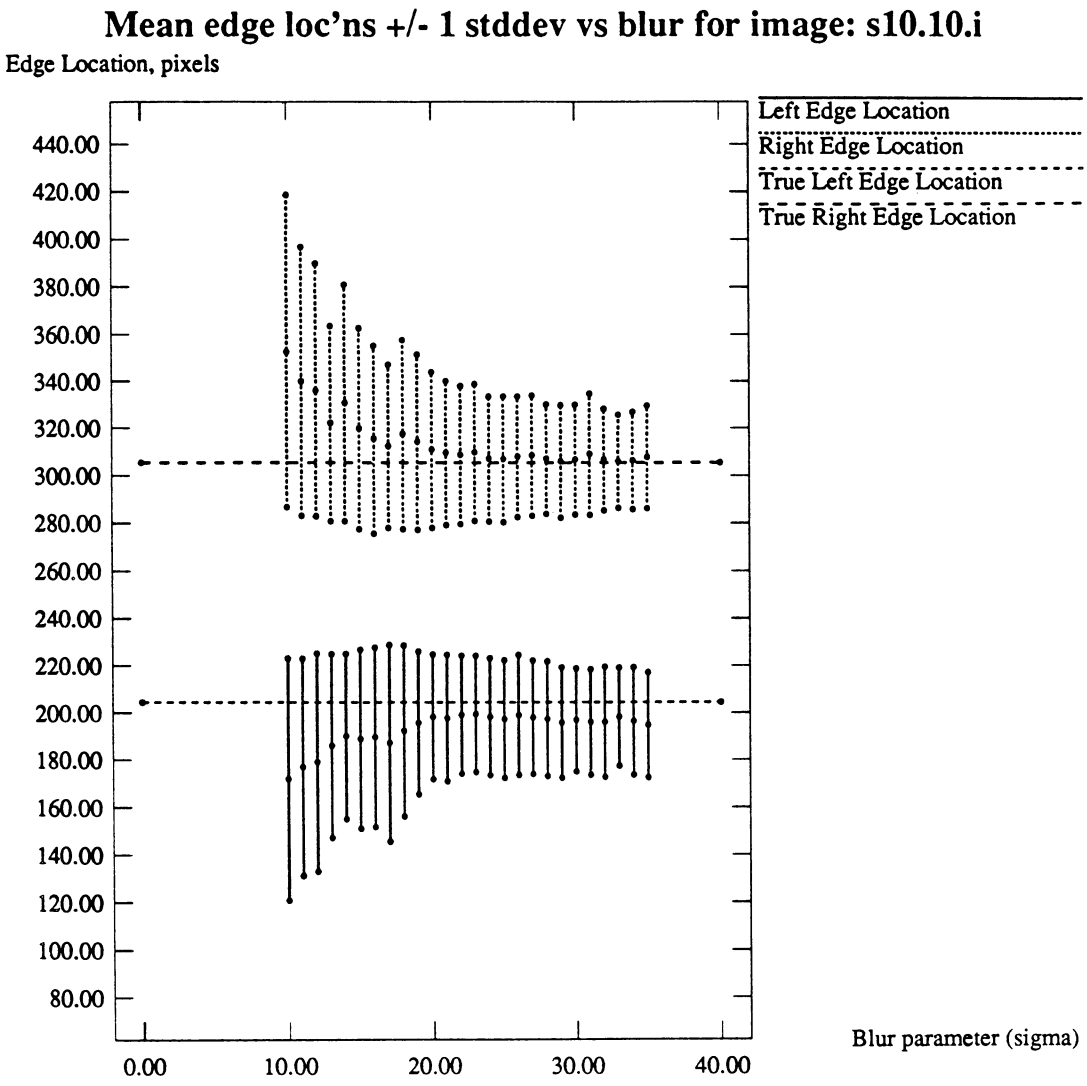
(e)

Figure 6.8. (cont'd) *(e)* SNR = 1.0



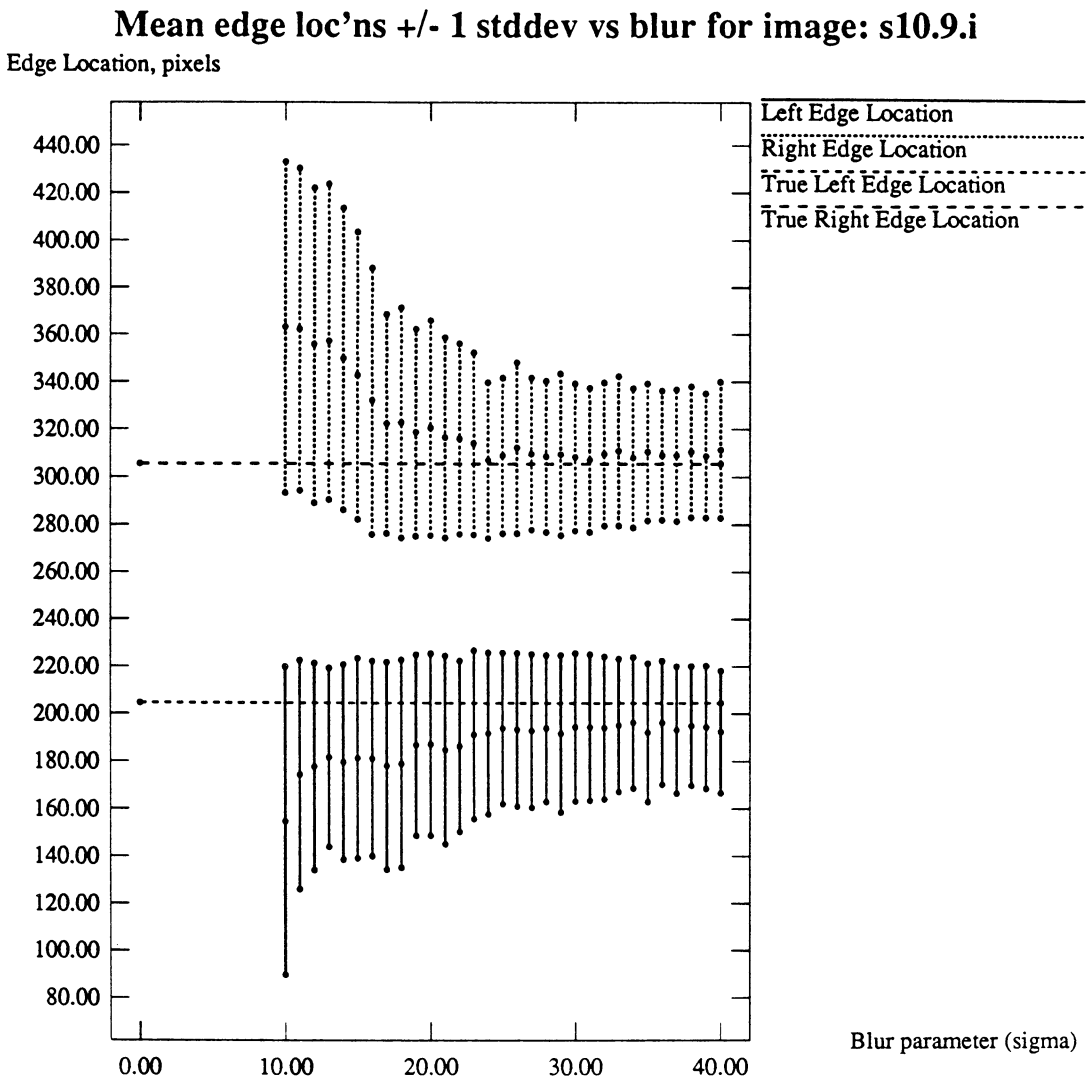
(f)

Figure 6.8. (cont'd) *(f)* SNR = 0.5



(g)

Figure 6.8. (cont'd) *(g)* SNR = 0.33



(h)

Figure 6.8. (cont'd) *(h)* SNR = 0.25

somewhat conservative, that is, we selected critical σ_b values that were well into the region of the plots where edge location bias and standard deviation had stabilized to within the bounds of the fluctuations about their terminal values.

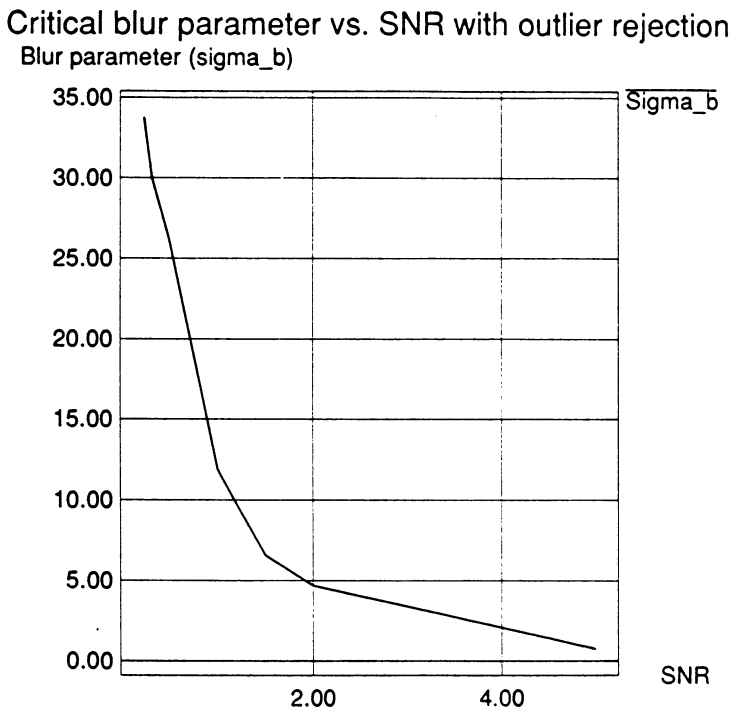


Figure 6.9. Critical value of blur parameter vs. image SNR

With these critical values of σ_b in hand, we were now ready to implement the scheme depicted in Figure 6.6 to measure the effect of HAPPI's processing routines on image feature size. The same sequence of test images

used as input to the program `mrowblur` was processed by several of HAPPI's noise filters and by a few other of HAPPI's routines. Another program, `rowblur`, was used to find edge location mean and variance in the pre-processed and post-processed images. The operation of, and inputs to, `rowblur` were similar to those of `mrowblur`, with the differences being that `rowblur` only took a single value of blur parameter, and gave as output not only edge location mean and variance, but feature size mean and variance as well. The outlier rejection scheme discussed above in connection with `mrowblur` was also implemented in `rowblur`. Both the gradient maximum and half power edge detection methods were implemented in `rowblur`. However, as mentioned earlier in this chapter, the half power method was found to have relatively poor edge location performance at low SNR values, and so only the gradient maximum method was used in the measurements presented in the remainder of this thesis. Table 6.1 lists the values of blur parameter fed to the program `rowblur` for each value of the SNR of the *pre-processed* image.

Table 6.1. Critical values of blur parameter as function of SNR

Test image SNR	Critical value of σ_b
10.0	0.1
5.0	0.8
2.0	4.6
1.5	6.4
1.0	12.6
0.5	24.9
0.33	30.7
0.25	33.8

Note that the post-processed images in general will have a different (and, we hope, higher) SNR than the pre-processed images, but we use the *same* value of blur parameter in the `rowblur` program for measuring the pre-processed and post-processed images. In this way, we are applying the same operator to both images to obtain an estimate only of the effect of the processing routine alone on image feature size. In the next chapter, we present some background on the processing routines used and the results of our measurements.

CHAPTER 7: MEASUREMENT RESULTS

7.1 Introduction

In this chapter, we present feature size measurements on unprocessed noisy test images and on processed test images which have been filtered with a variety of HAPPI's processing routines. Specifically, the routines tested were the adaptive smoothing and modified adaptive smoothing filters, the Kalman filter, the median and weighted median filter, the root filter, and the sigma filter, all found under HAPPI's "Noise Filters" menu. Also tested were the histogram equalization and expand grey level (linear contrast stretch) routines, both from HAPPI's "Contrast Enhancement" menu, and a simple uniform-weight lowpass filter from HAPPI's "Convolution" menu. The effect of the scattering line spread function on mean and variance of edge location is briefly examined. Finally, selected feature size measurement results are compared with size measurements attainable using the traditional Sobel edge detection operator for doing edge location by visual inspection.

7.2 Effect of Processing Routines on Feature Size

We have chosen to display the effects of processing on image feature size in the following figures by plotting the mean feature size and the ± 1 standard deviation range of feature size (i.e., $\mu_f \pm 1\sigma_f$) vs. the SNR of the pre-processed image. (Note: the symbols μ_f and σ_f were defined in Section 6.3 of the previous chapter.) The input images to each processing routine were the

same sequence of test images discussed in Subsection 6.3.1, and thus, for each of the graphs of ($\mu_f \pm 1\sigma_f$) vs. SNR in this chapter, the data for the pre-processed images is identical. Two images from this sequence, with SNR's of 1.0 and 0.25, respectively, are shown in Figure 7.1 and Figure 7.2.

Ideally, the output image from one of HAPPI's processing routines should have a higher SNR (thus yielding better edge location and size measurement performance) than the input image. However, regardless of the SNR of the output image, we have plotted the value of ($\mu_f \pm 1\sigma_f$) for each output image *at the SNR of the corresponding input image*. In this way, we are able to visualize *the improvement (or degradation) of feature size estimate* wrought by a given processing routine on a particular image with a particular SNR.

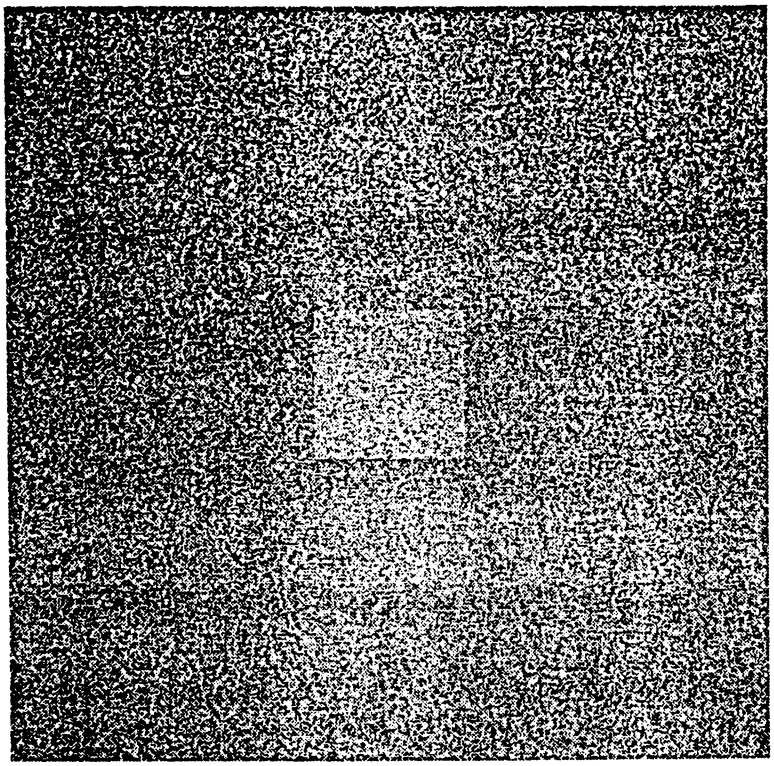


Figure 7.1. Test image with SNR = 1.0

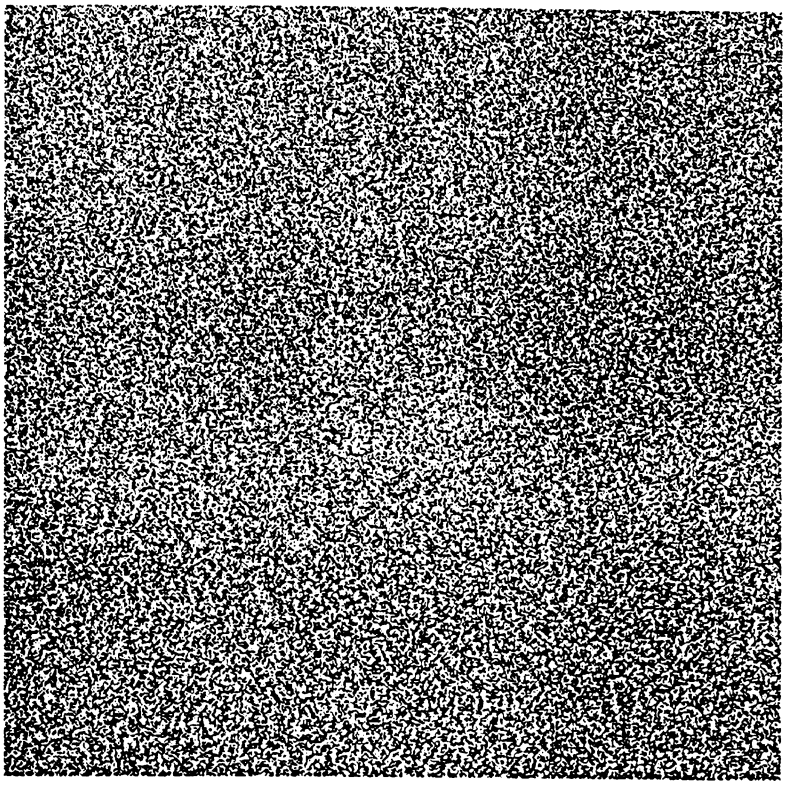


Figure 7.2. Test image with SNR = 0.25

The reader may wish to refer to Figure 7.3 in the next subsection to visualize the graph features in the following discussion. In the graphs of ($\mu_f \pm 1\sigma_f$) (i.e., measured size) vs. SNR presented in this chapter, there are three line styles, one each for the measured size in the pre-processed image, the measured size in the post-processed image, and the actual image feature size (which was 101 pixels for all test images). The measured size in the pre-processed images is plotted with a solid line, while the measured size in the post-processed images is plotted with a dotted line, and the actual size is plotted with a dashed line. Three curves appear on the graph in the solid line style of feature size data for the pre-processed images. The top curve represents the

mean feature size plus one standard deviation (i.e., $\mu_f + 1\sigma_f$), while the middle curve represents mean feature size, and the bottom curve represents mean feature size minus one standard deviation (i.e., $\mu_f - 1\sigma_f$). Another such set of three curves in the dotted line style of feature size data for the post-processed images also appears on the graph. Finally, in each graph, a single straight line in the dashed line style, representing actual feature size (which has no random variation), is plotted for reference. In the next several subsections, we briefly discuss the theory behind each of the processing routines tested, state the parameter values used in each routine, present measurement results, and give some discussion of salient points.

7.2.1 Adaptive Smoothing Filter

Happi contains two adaptive smoothing filter routines, the "Adaptive Smoothing Filter" and "Modified Adaptive Smoothing Filter", found under the Noise Filters menu. These filters are based on the paper by Kuan *et al.* (1985) and are similar in operation. The assumed image degradation model used in developing the filter is of the form (Zheng and Basart, 1988):

$$y(i,j) = x(i,j) + u(i,j) \tag{7.1}$$

where $y(i,j)$ is the observed, degraded image, $x(i,j)$ is the original image before degradation, and $u(i,j)$ is the signal-dependent degradation term. The degradation term $u(i,j)$ is given by:

$$u(i,j) = f(x(i,j))n(i,j) \tag{7.2}$$

where $f(x(i,j))$ models the signal dependency, and $n(i,j)$ is iid(0,1) random noise. The filter produces an estimate of each pixel value of the form:

$$\hat{x}(i,j) = \bar{x}(i,j) + k(i,j)(y(i,j) - \bar{x}(i,j)) \tag{7.3}$$

where $\hat{x}(i,j)$ is the estimate of the original, undegraded image at location (i,j), $\bar{x}(i,j)$ is the local mean, and $k(i,j)$ is a local calibration factor, given by:

$$k(i,j) = (1 - V_{u(i,j)}/V_{y(i,j)}) \tag{7.4}$$

where $Vu(i,j)$ is the local variance of the signal-dependent noise, and $Vy(i,j)$ is the local variance of the observation. Since $n(i,j)$ is zero-mean, the covariance between $x(i,j)$ and $u(i,j)$ is zero (Zheng and Basart, 1988), and $Vy(i,j)=Vx(i,j) + Vu(i,j)$, so that $k(i,j)$ may be written as:

$$k(i,j) = V_{x(i,j)}/(V_{x(i,j)} + V_{u(i,j)}) \tag{7.5}$$

Note from Equation 7.5 that if the local SNR is much greater than 1, then $k(i,j)$ is approximately equal to 1, and the estimate $\hat{x}(i,j)$ in Equation 7.3 is equal to the observation $y(i,j)$, while if the local SNR is much less than 1, then $k(i,j)$ is very small and the estimate $\hat{x}(i,j)$ in Equation 7.3 is approximately equal to the local mean $\bar{x}(i,j)$.

Both the adaptive smoothing filter and the modified adaptive smoothing filter take two parameters: an *increment* and a *window size*. The modified adaptive smoothing filter additionally requires the user to specify an area of the input image from which to calculate a value of noise variance which is used globally throughout the image, whereas the adaptive smoothing filter requires no such input, and automatically calculates local noise variance from the first difference of the input image. The *window size* parameter simply specifies the length of one side of the square window used to calculate signal and noise variance. The *increment* parameter specifies the number of pixels by which the window used to calculate signal and noise variance is moved for each such calculation. For both adaptive smoothing filters, an increment of 1 and a window size of 7 were used. The window size is constrained by HAPPI to be an odd number. A 7x7 window yields a sample size of 49 from which to calculate signal and noise variances; a minimum sample size of 30 to 50 is generally considered necessary for meaningful calculations of statistics, and so a window size of 7 is the smallest odd value that yields a "good" sample size. A larger window size was avoided to keep execution time down.

Figure 7.3 is a plot of measured feature size vs. SNR for the adaptive smoothing filter. Let us denote the value of μ_f and σ_f in the pre-processed images by $\mu_{f(pre)}$ and $\sigma_{f(pre)}$, respectively, and similarly denote the value of μ_f and σ_f in the post-processed images by $\mu_{f(post)}$ and $\sigma_{f(post)}$, respectively. The following observations may be made about Figure 7.3:

1) The value of $\sigma_{f(pre)}$ becomes very large at low values of SNR. Recall, however, that the statistics we are calculating for feature size are from a

population of feature sizes measured from *individual noisy image rows*. It is to be expected that when edge location and feature size are estimated from a single image row with a low SNR, the measurement will likely be

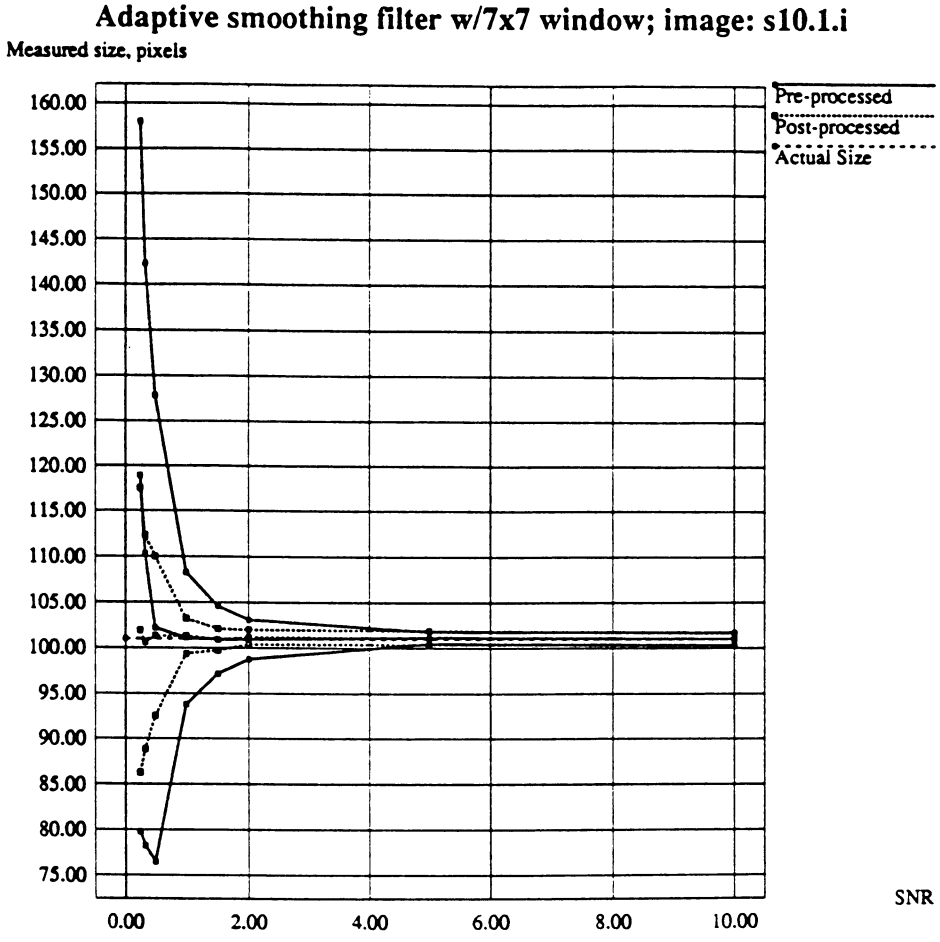


Figure 7.3. Measured size vs. SNR for adaptive smoothing filter

considerably in error for any given row. Only when the measurements are averaged do they begin to reasonably approximate the true feature size.

2) The value of $\mu_{f(pre)}$ stays quite close to the actual feature size until the SNR drops below about 0.5; as SNR decreases further, $\mu_{f(pre)}$ begins to increase rapidly.

3) The value of $\sigma_{f(post)}$ is generally smaller than that in the pre-processed images, and is especially so as SNR decreases. For SNR larger than about 5, the filter does not appear to improve the feature size estimate.

4) The value of $\mu_{f(post)}$ stays much closer to the actual feature size than $\mu_{f(pre)}$ at low SNR. However, there are small fluctuations in $\mu_{f(post)}$ about actual feature size as SNR decreases.

Figure 7.4 is a plot of measured size vs. SNR for the modified adaptive smoothing filter. It may be seen that the above observations about Figure 7.3 apply to Figure 7.4 as well. In fact, the behavior of the graphs of measured size vs. SNR for pre-processed and post-processed images is qualitatively very similar for all of the noise filters used in this study; the differences between these filters in terms of influence on measured feature size are mostly quantitative. Figure 7.5 compares measured size vs. SNR for the adaptive smoothing filter and modified adaptive smoothing filter.

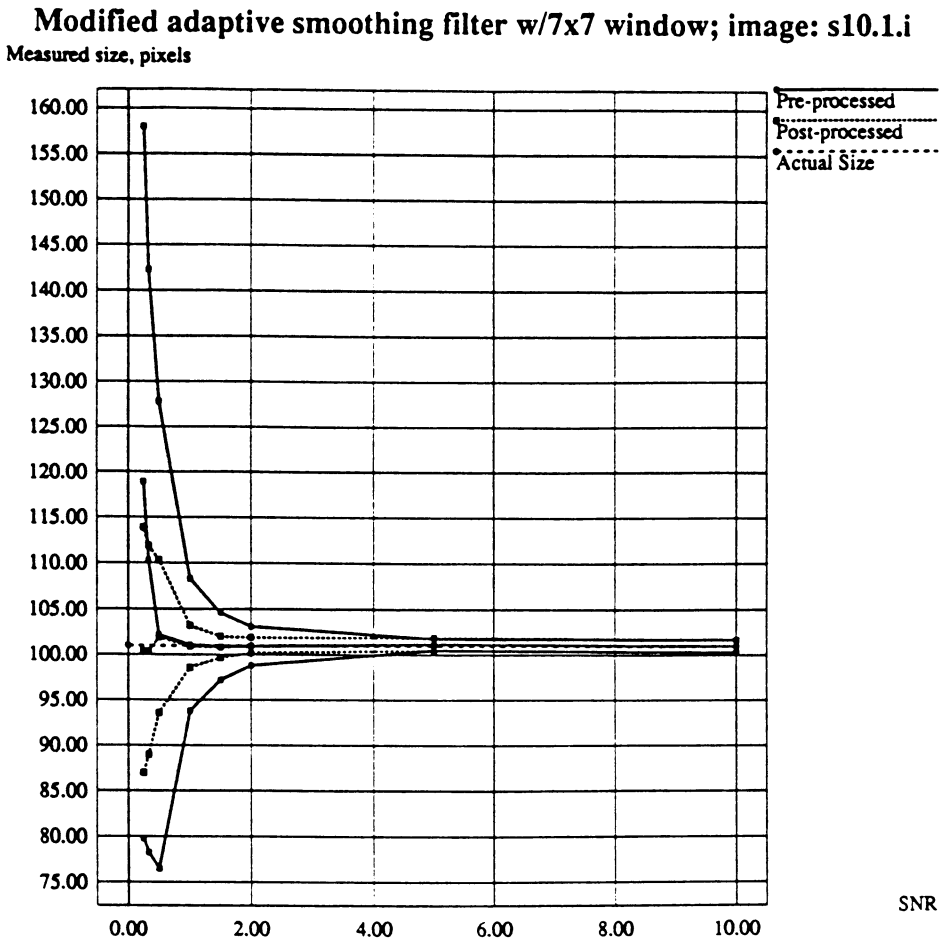


Figure 7.4. Measured size vs. SNR for the modified adaptive smoothing filter

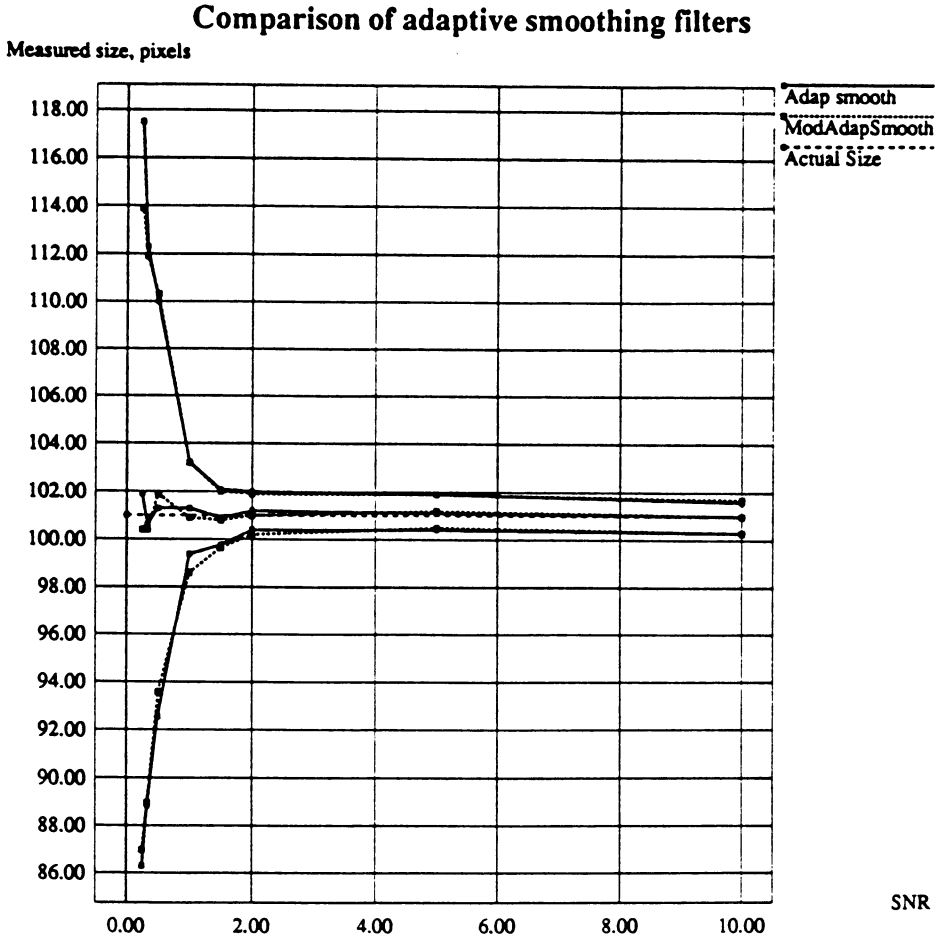


Figure 7.5. Measured size vs. SNR for adaptive and modified adaptive smoothing filters

7.2.2 Kalman Filter

The Kalman filter is an optimal linear filter for recursively separating two random processes which may have overlapping spectral density functions. In the Kalman filter formulation, one random process usually represents the signal of interest while the other represents unwanted measurement error. The random processes are formulated in a state vector representation and the mean square error is minimized, resulting in a recursive algorithm which constitutes the Kalman filter. The reader is referred to Brown (1983) for a well-written introduction to Kalman filtering theory. The Kalman filtering routine presently in HAPPI was translated from the one used in Safae-nili's (1989) thesis; Safae-nili's Kalman filter routine was based on the work of Biemond (1983, 1986). In this filter, the random process representing the signal is modeled as an AR(1) process in both the horizontal and vertical direction. The filter models not only degradation from noise but also blur, which may be caused by the imaging system used to create the image. The imaging system blur is modeled as a 2-d circularly symmetric gaussian function. Both pre- and post-blur noise are modeled. The input parameters for HAPPI's Kalman filtering routine and their meaning are listed in Table 7.1 (note: PSF stands for Point Spread Function). For the experiments in this study, the parameter values used were as listed in Table 7.2. The parameter PZero was set equal to the value of the noise variance in each test image. The parameters RhoH and RhoV were set to the relatively high value of 0.9 to reflect the nature of the test image feature; since the feature was of a

Table 7.1. Parameters of HAPPI's Kalman filter

Parameter	Meaning
RhoH	Horizontal correlation coefficient of signal process
RhoV	Vertical correlation coefficient of signal process
SigmaU	Standard deviation of pre-blur noise
SigmaW	Standard deviation of post-blur noise
PZero	Initial error estimate (see Brown (1983) for theory)
Beamsize	Length of one side of imaging system's gaussian blur PSF
Beamvariance	Variance of imaging system's gaussian blur PSF

Table 7.2. Kalman filter parameter values used

Parameter	Value
RhoH	0.9
RhoV	0.9
SigmaU	0.001
SigmaW	1.0
Beamsize	3
Beamvariance	0.1

single, constant grey level, its pixels are very highly correlated. In this Kalman filter routine, it turns out that if the ratio of SigmaU to SigmaW is very small, the filter will primarily smooth noise, while if this ratio is large, the filter will primarily do deblurring of the gaussian imaging system PSF. Because our test images are not intended to model imaging system blur, we chose values of SigmaU and SigmaW which made the ratio SigmaU/SigmaW small, so that the filter would only filter noise. The values of Beamsize and Beamvariance were also chosen to be the smallest values allowed by the current implementation of the routine so as to deemphasize the deblurring action of the filter. The current implementation of HAPPI's Kalman filter only accepts square images whose dimensions are integer powers of two. Figure 7.6

is a graph of measured size vs. SNR for the Kalman filter. Note that the graph has the same general characteristics as those of the previous subsection. However, the Kalman filter, as run with the above parameter values, did not perform quantitatively as well - with respect to measured feature size - as the adaptive smoothing filter of the previous subsection. In a subsequent subsection, the performance of all of the noise filters is compared.

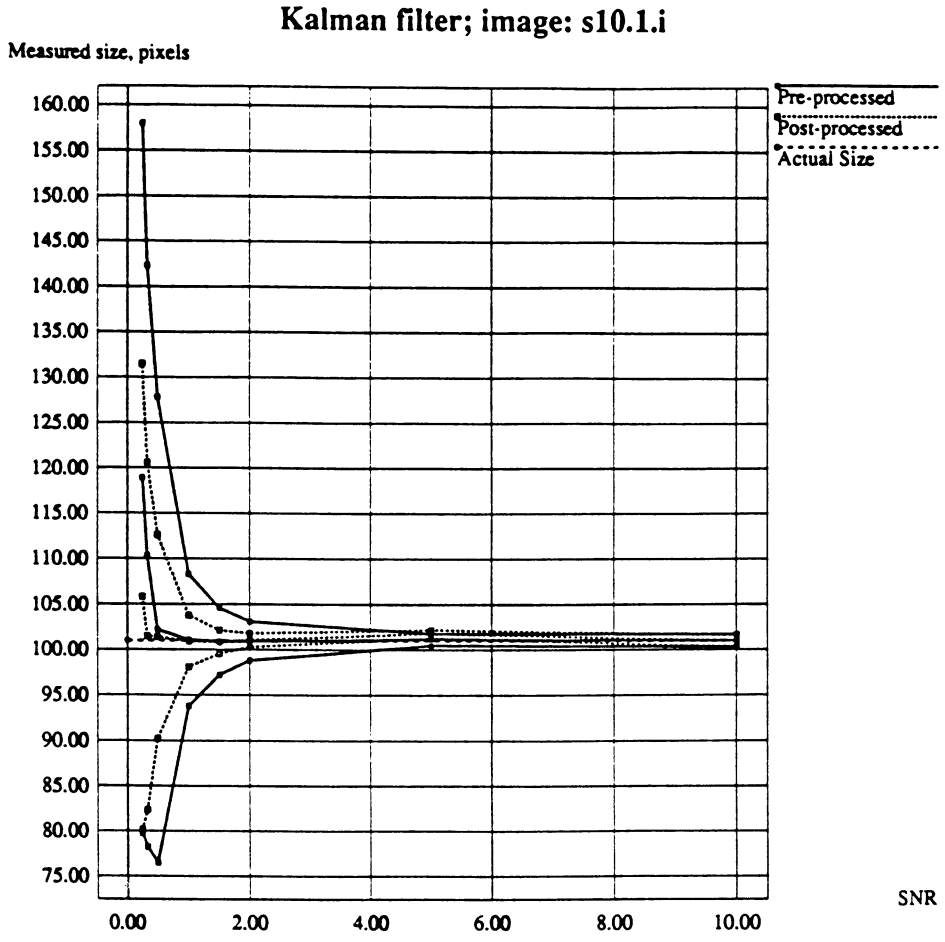


Figure 7.6. Measured size vs. SNR for the Kalman filter

7.2.3 Median Filter

The median filter is a nonlinear filter which is good for suppressing impulsive noise while preserving edges. This filter simply moves a window through the image and replaces the pixel at the center of the window with the median of all the pixels in the window. HAPPI contains two median filtering routines. The first routine, called simply the "median filter" performs just the algorithm described above, and is implemented using the fast algorithm of Ahmad (1987). This routine takes as its only parameter the size of the window in which the median is computed. The second routine is called the "weighted median filter," and takes a *center pixel weight* parameter as well as a window size parameter. To compute the median of a set of numbers, the set is ranked in ascending or descending order, and, for a set with an odd number of elements, the middle value in the ranking is identified as the median; for an even number of elements, the median is computed as the average of the two middle values in the ranking. In the weighted median filter, the pixel at the center of the filter window is replicated in the ranking used to find the median. The value of the *center pixel weight* parameter is the number of times the center pixel is replicated. The weighted median filter thus has increased likelihood that the center pixel in the filter window will be selected in the computation of the median. For consistency of window size with other routines and for purposes of having a "good" sample size, a 7x7 window was used in both the median and weighted median filters in our tests. The center pixel weight in the weighted median filter was set to 10, which is about 20% of the sample size

of 49 obtained with a 7x7 window. Also, a 15x15 window was tried with the median filter. Figure 7.7 is a graph of measured size vs. SNR for the median filter using a 7x7 window; Figure 7.8 shows measured size vs. SNR for the median filter using a 15x15 window.

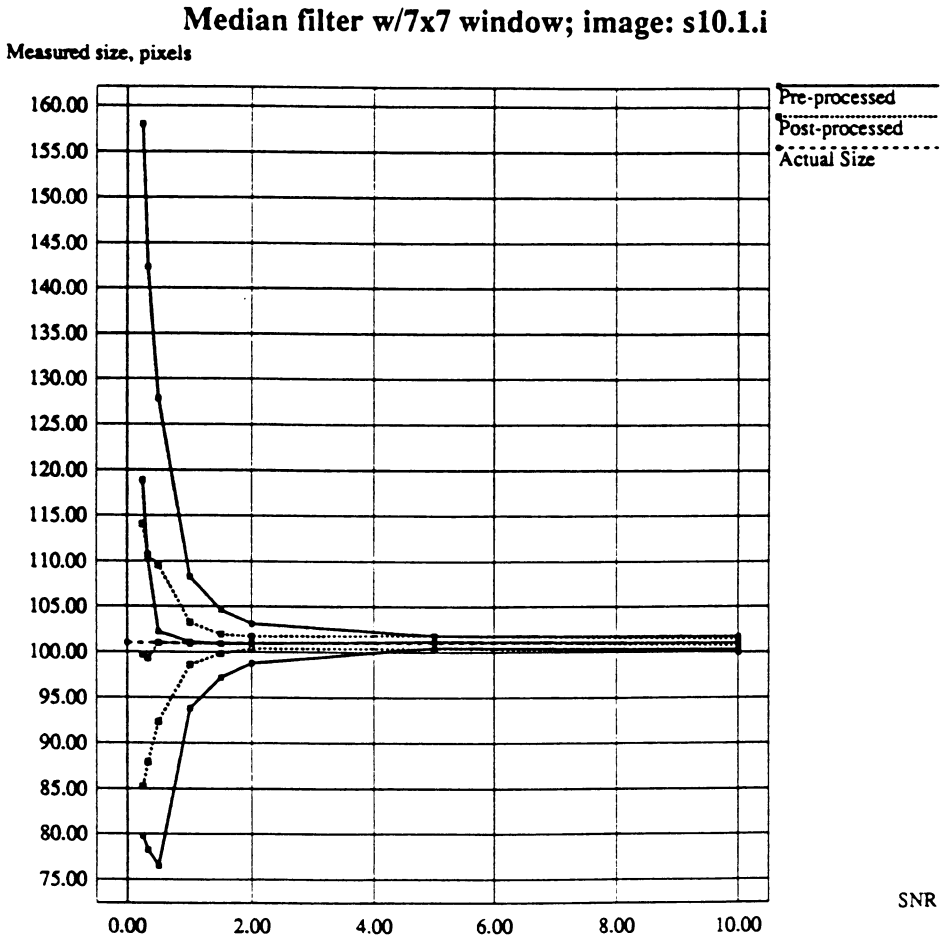
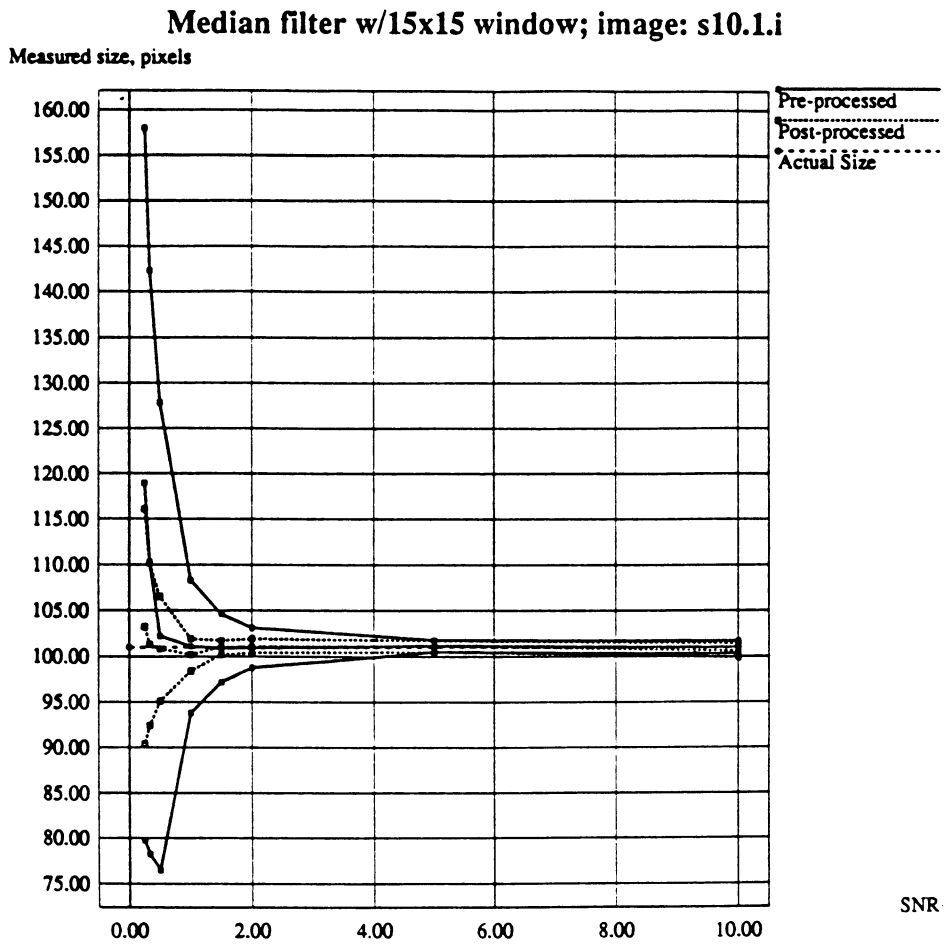


Figure 7.7. Measured size vs. SNR for median filter with 7x7 window



7.8. Measured size vs. SNR for median filter with 15x15 window

Note that the median filter's performance with a 15x15 window is slightly better than its performance with a 7x7 window. Figure 7.9 shows measured size vs. SNR for the weighted median filter using a 7x7 window and a center pixel weight of 10. The performance of the weighted median filter is

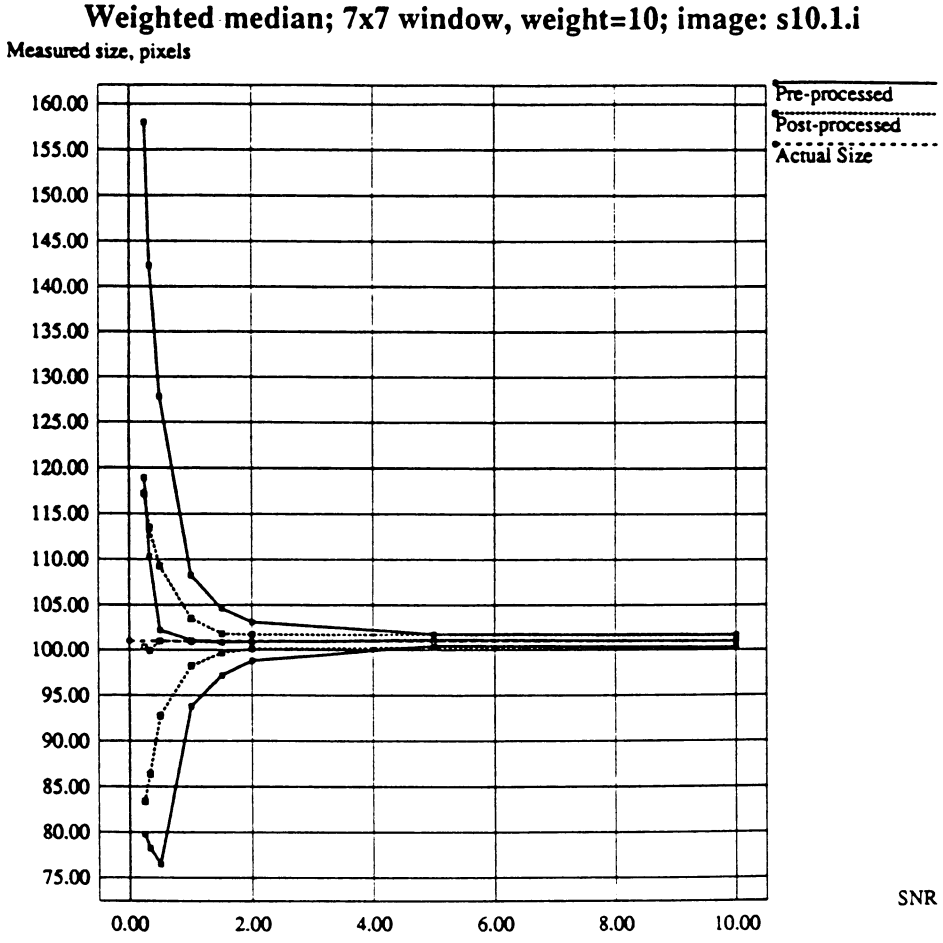
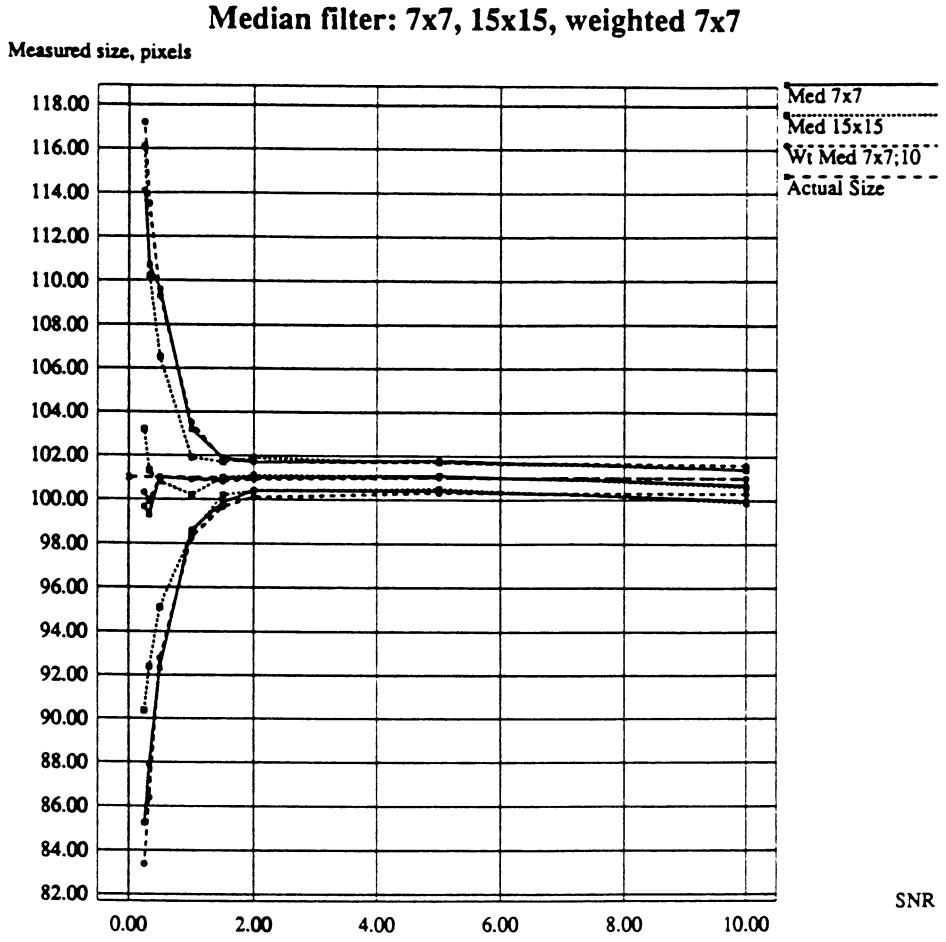


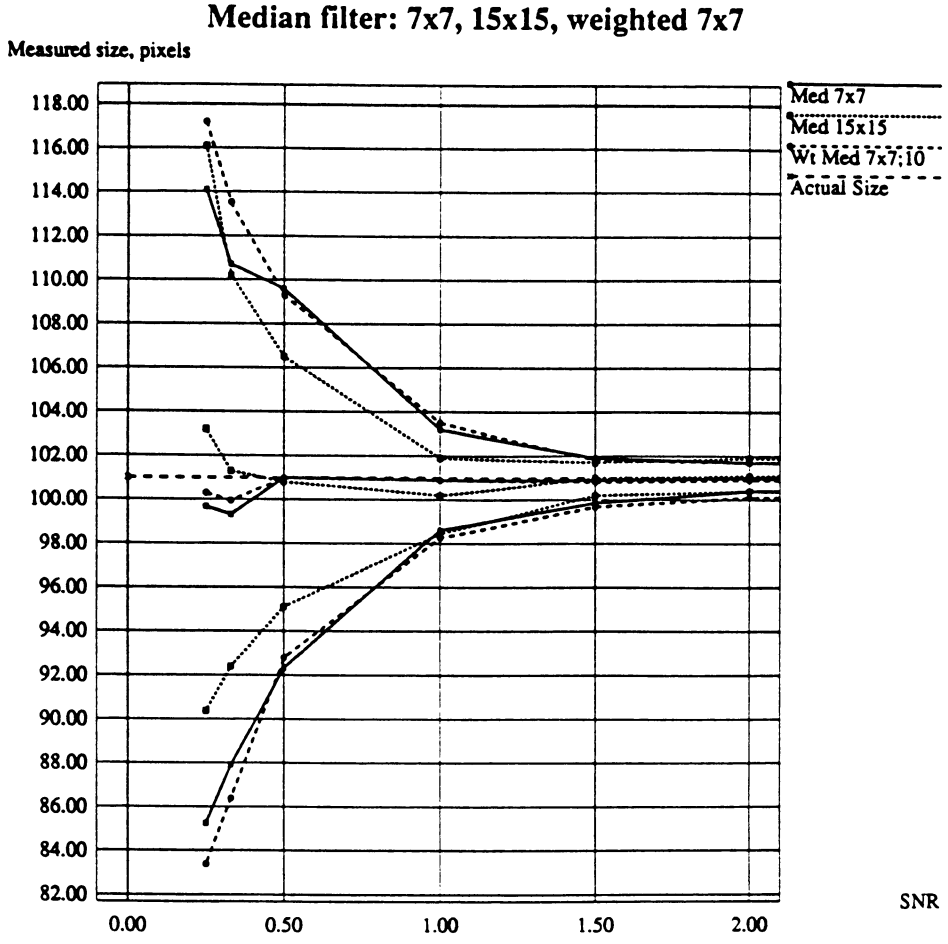
Figure 7.9. Measured size vs. SNR for weighted median filter

slightly worse than that of the regular median filter with the same window size. The performance of the median and weighted median filters from Figures 7.7, 7.8, and 7.9 is compared in Figure 7.10 (note that only measured size of post-processed images is plotted in this graph).



(a)

Figure 7.10. Comparison of measured size performance of median filter with two window sizes and weighted median filter *(a)* Plot of full SNR range tested



(b)

Figure 7.10. (cont'd) *(b)* Enlargement of portion of (a) from SNR=0 to SNR=

7.2.4 Root Filter

The root filter is a nonlinear filter defined by the equation (Jain, 1989, p. 291):

$$U(\omega_1,\omega_2) = |V|^{\alpha}\exp\{j\theta_v\} \tag{7.6}$$

where $v(x_1,x_2)$ is the input image, $u(x_1,x_2)$ is the output image, $V(\omega_1,\omega_2)$ and $U(\omega_1,\omega_2)$ are the Fourier transforms of $v(x_1,x_2)$ and $u(x_1,x_2)$, respectively, and j is the imaginary operator. This filter operates by taking the Fourier transform of the input image, forming the magnitude and phase of the resulting frequency-domain data, and raising the magnitude to the power α while leaving the phase unchanged. The transformed frequency-domain data is then inverse Fourier transformed to yield the spatial-domain output image $u(x_1,x_2)$. Using a value of α less than one makes the root filter behave like a high-pass filter, while a value of α greater than one results in a low-pass filter effect. It was found that values of α greater than about 3.5 introduced artifacts in the root filtered image and greatly distorted the image feature. A value of 2.5 was used for α in all processing done with the root filter. Figure 7.11 shows measured size vs. SNR for the root filter. It may be seen from Figure 7.11 that the mean feature size of the post-processed image $\mu_{f(post)}$ has a slight dip in it at SNR=0.33. The graphs for other processing routines have similarly nonmonotonic behavior of $\mu_{f(post)}$ at low SNR, as may be seen, for example, from Figure 7.10(b), but this behavior seems to be more pronounced for the root filter.

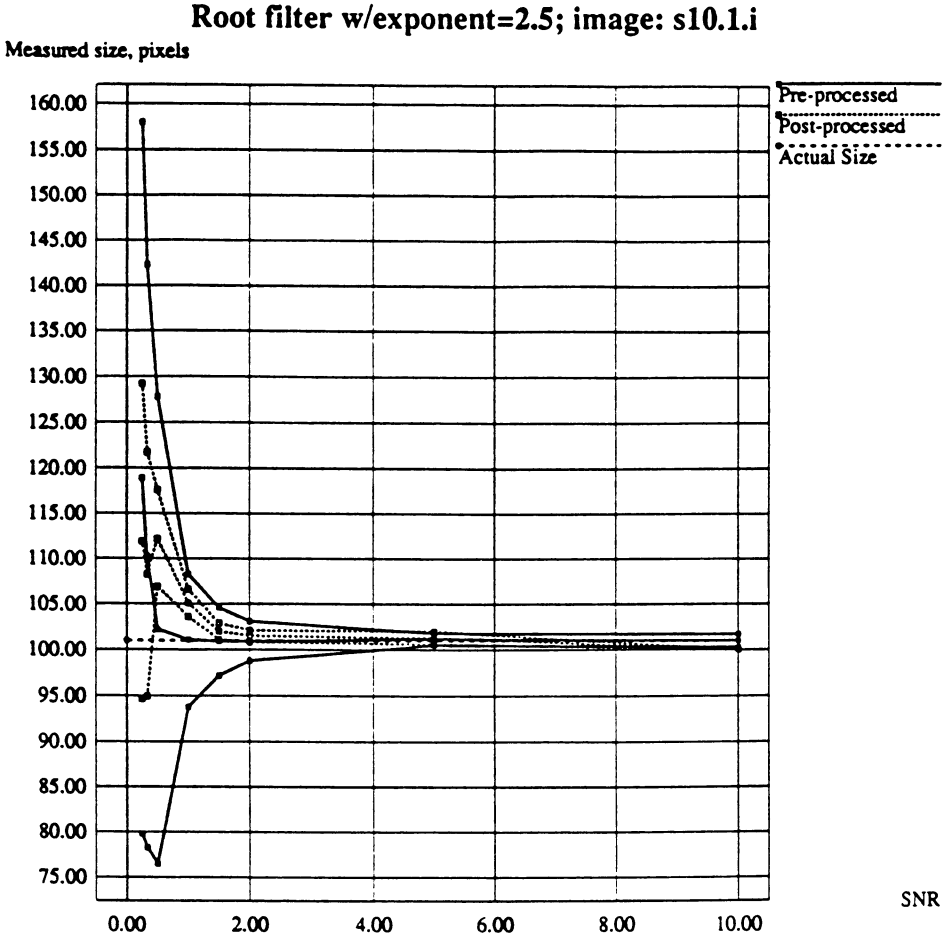


Figure 7.11. Measured size vs. SNR for root filter

7.2.5 Sigma Filter

HAPPI's sigma filter is based on the paper by Lee (1984). The sigma filter determines the rms value, σ, of the pixel intensities within a moving window, and averages all pixels whose intensities fall within $\pm 2\sigma$ of the intensity of the center pixel in the window. The average thus computed is assigned to the center pixel in the window. To calculate the rms value of pixel intensities in the window, the filter first forms the first difference of the input image. As the window moves through the input image, a corresponding window of the same size is moved through the first difference image, and the rms value σ is calculated as the standard deviation of the first difference image's pixels in the window. The filter may be applied repeatedly to an image. The sigma filter takes two parameters, the *number of passes*, and the *window size*. The *number of passes* parameter determines how many times the filter is applied to the image. The *window size* parameter is simply the length of one side of the square window in which the value of σ is computed and pixels in the $\pm 2\sigma$ range are averaged. For all images processed with the sigma filter, a window size of 7x7 was used, and the number of passes was set to 1. Figure 7.12 shows measure size vs. SNR for the sigma filter. It may be seen from Figure 7.12 that the graph behaves qualitatively like the other graphs for the previously discussed noise filters.

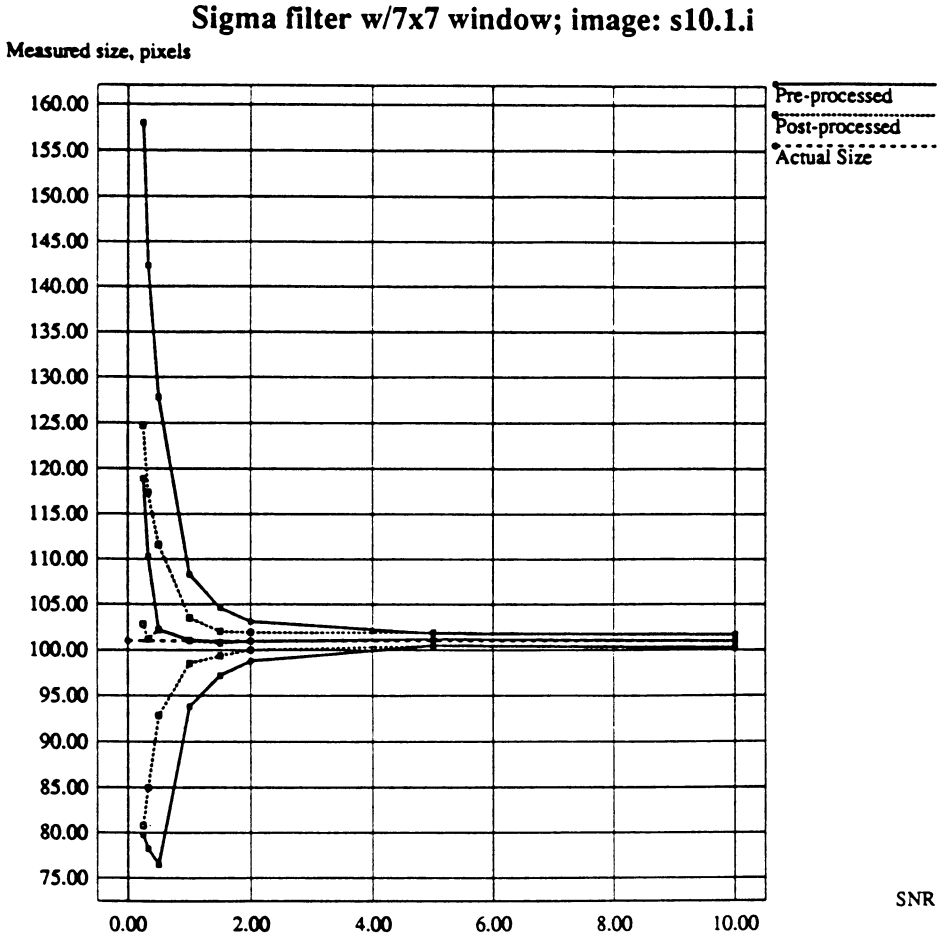


Figure 7.12. Measured size vs. SNR for sigma filter

7.2.6 Lowpass Filter

A uniform-weight lowpass filter was applied to the sequence of noisy test images. This filter simply moves a window through the image and replaces the pixel at the center of the window with an unweighted average of all the pixels in the window. A 7x7 window was used for our tests, for consistency with the window size used in the other filters. Figure 7.13 shows measured size vs. SNR for the lowpass filter.

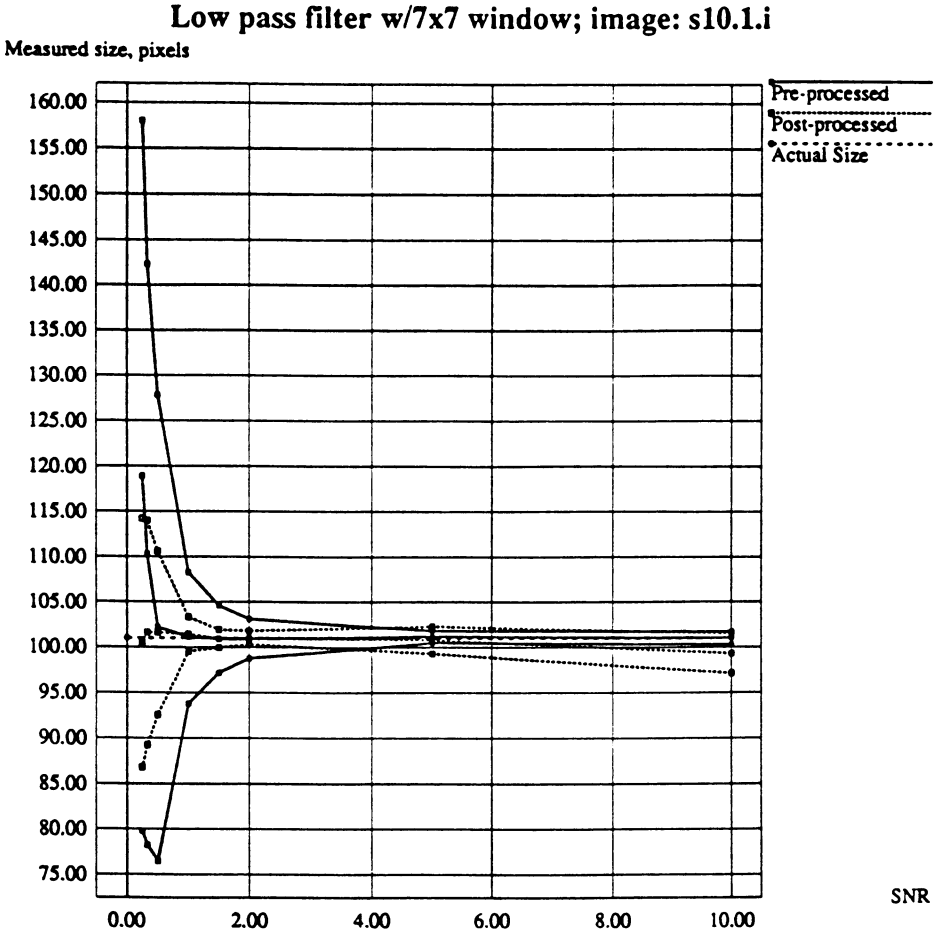


Figure 7.13. Measured size vs. SNR for lowpass filter with 7x7 window

The graph of Figure 7.13 behaves much the same as those for the noise filters of the previous subsections at values of SNR below about 4.0. Note, however, that at higher SNR values, the feature size estimate for the post-processed images is slightly worse than that for the pre-processed images. This behavior is likely attributable to the relatively low values of blur parameter used in the program `rowblur` for images with SNR's of 5 and 10 (refer back to Table 6.1). Recall that these values were obtained *from the sequence of pre-processed images* under the tacit assumption that they would be more than adequate for post-processed images.

Unlike the noise filters of the previous subsections, the lowpass filter is not designed to keep edges sharp in an image. When the lowpass filter is convolved with the 2-d pulse of Figure 6.1, the output image is a 2-d trapezoidal pulse. A 1-d slice through the trapezoidal pulse will then be a 1-d trapezoidal pulse, which will have a finite, *constant* slope in an interval about an edge location, as shown in Figure 7.14.

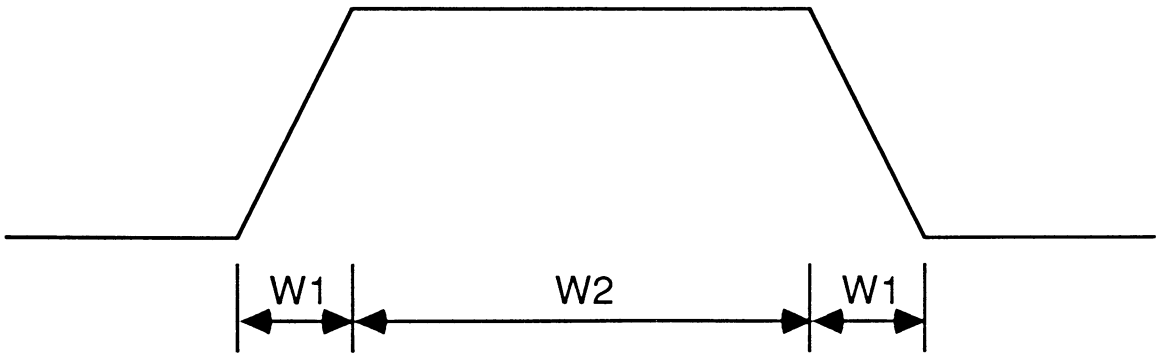


Figure 7.14. One-dimensional trapezoidal pulse obtained by taking 1-d slice from convolution of square-edge pulse with uniform-weight lowpass filter

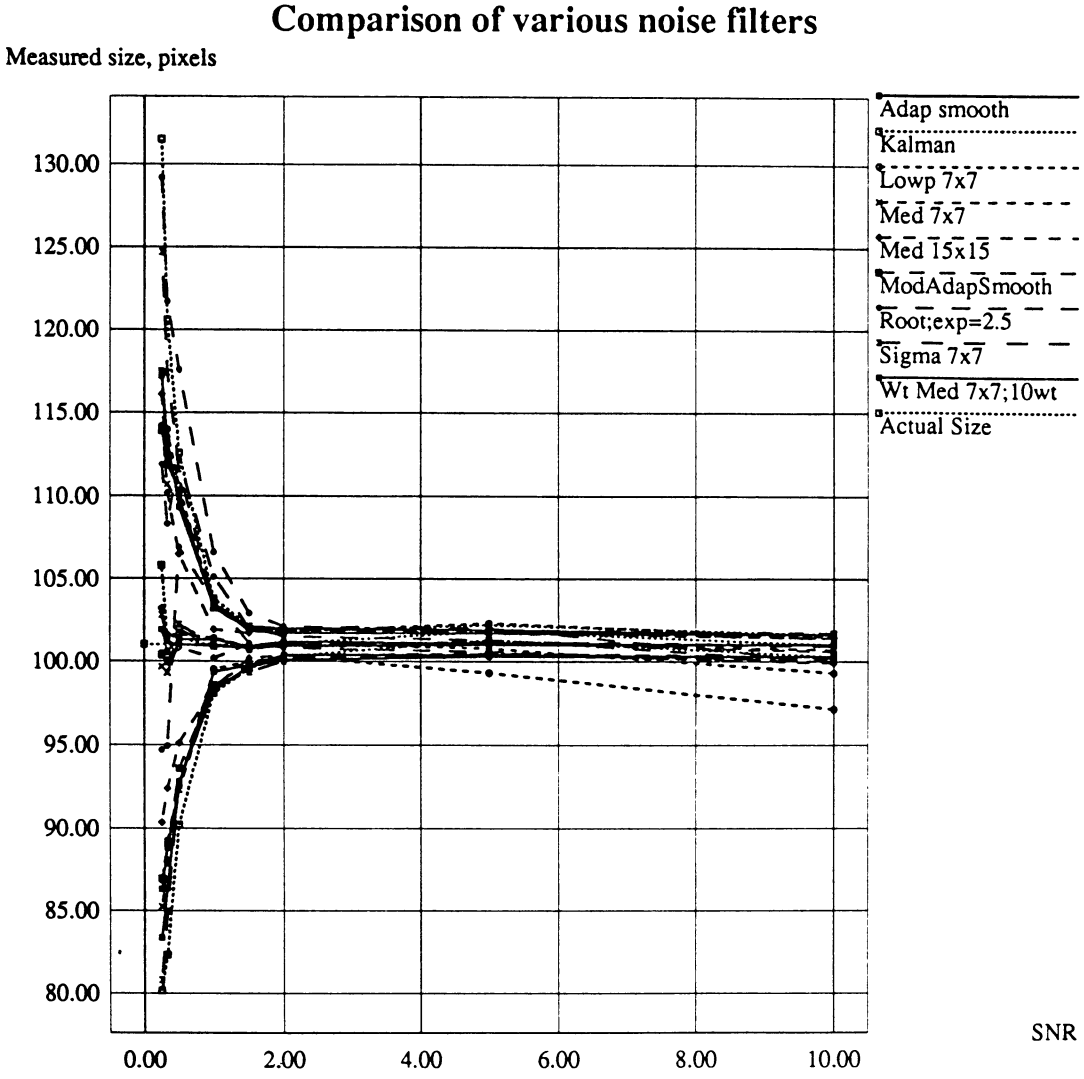
Under the "gradient maximum" definition of edge location then, the edge location is ambiguous for such a trapezoidal pulse. Convolution of the trapezoidal pulse with a gaussian smoothing function, *provided the width of the smoothing function is not too small compared to the dimension W1 in Figure 7.14 but is smaller than half the dimension W2 in Figure 7.14*, will yield a pulse with smoothed edges of non-constant slope, for which the edge locations will no longer be ambiguous under the gradient maximum edge definition. The post-processed images from the lowpass filter were run through `rowblur` with slightly higher values of blur parameter than the nominal values of Table 6.1, and the feature size estimate for the post-processed images was seen to improve to be at least as good as that for the pre-processed images at high SNR values.

7.2.7 Comparison of Noise Filters and Overall Characteristics

In Figure 7.15, we compare the performance of all of the filters in Subsections 7.2.1 through 7.2.6. Since the feature size data for pre-processed images were the same for each filter, only feature size of post-processed images is plotted here for purposes of comparison between filters. It is clear that there is a strong dependence of $\sigma_{f(post)}$ on SNR, with $\sigma_{f(post)}$ dramatically decreasing as SNR increases over the range 0.25 to about 2.0. For an SNR (of pre-processed images) above about 2.0, measured feature size in post-processed images is about the same for all of the filters (except for the anomalous behavior of the lowpass filter seen in Figure 7.13 at high SNR values), and has

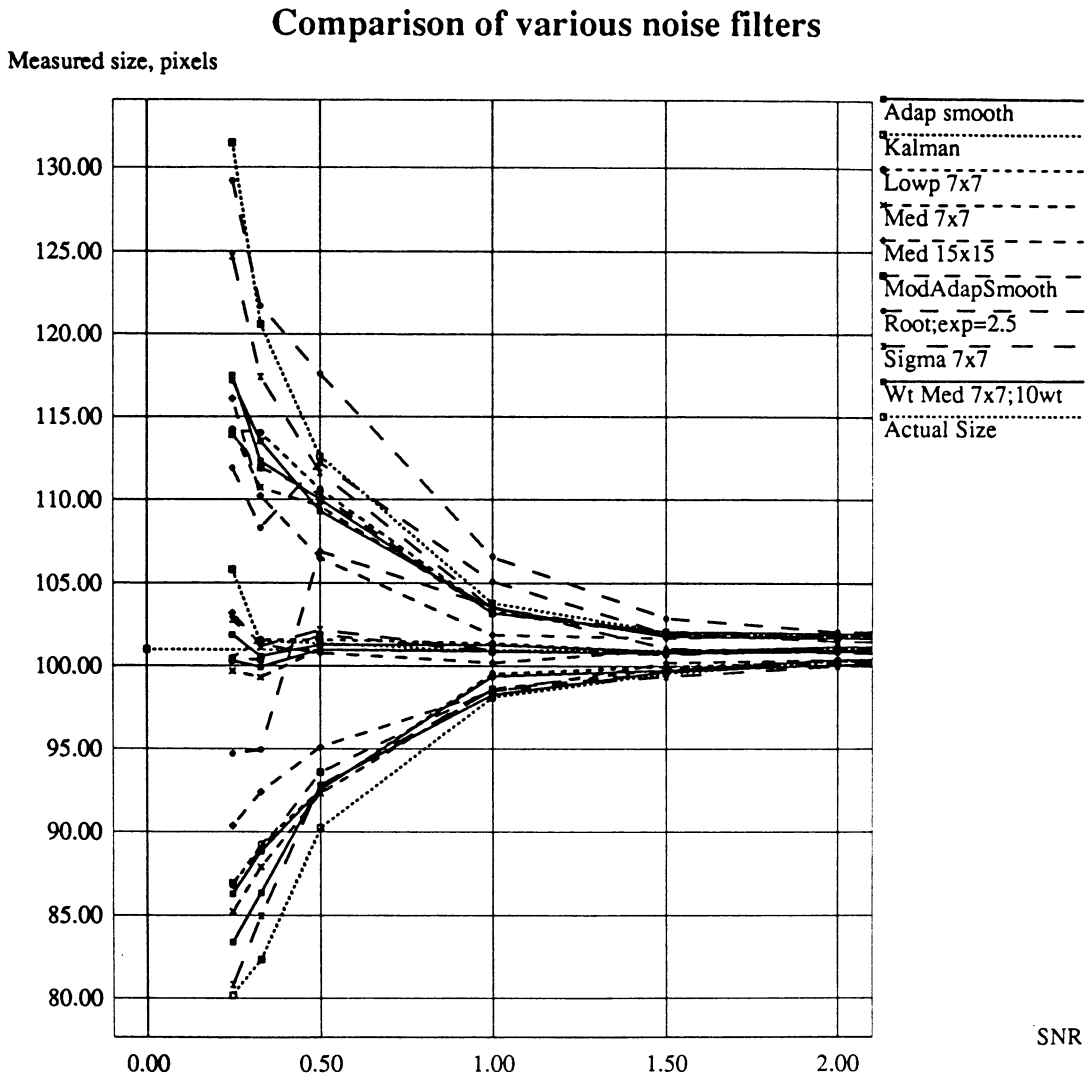
a mean very close to the actual feature size of 101 pixels and a standard deviation of about one pixel. Below SNR=2.0, the various filters, with the exception of the root filter, manage to keep the mean feature size close to the actual size, but with small fluctuations about the mean. The main difference between most of the noise filters then seems to be in their effect on the *standard deviation* of the feature size estimate, $\sigma_{f(post)}$. We recognize that lower values of $\sigma_{f(post)}$ are associated with higher values of SNR, so the data plotted in Figure 7.15 are also an indirect measure of how the various noise filters raise SNR. In all the graphs of measured size vs. SNR in Subsections 7.2.1 through 7.2.5, the value of $\sigma_{f(post)}$ is less than that of $\sigma_{f(pre)}$ at all values of SNR, by anywhere from a few percent at high SNR's to 300% low SNR's. This is intuitively satisfying, as it indicates that all of these noise filters serve to raise the SNR of the input image, yielding an improved feature size estimate.

The reader may have noted the anomalous behavior of the root filter in Figure 7.11. In Figure 7.16, we plot the same information as in Figure 7.15 but with the data set for the root filter removed. From Figure 7.16 we may clearly see that the Kalman filter has the worst feature size performance (i.e. highest standard deviation of feature size), followed by the sigma filter. We also note that the Kalman filter has the worst feature size bias (i.e. magnitude difference between estimated mean feature size and actual feature size), at about 5 pixels, at the lowest SNR value tested. It is of interest that the output images from the Kalman filter appeared to have more sharply defined edges than those from the median filter, which looked mottled at low SNR values. In spite of a visual appearance of better output image quality from the Kalman filter, the programs used in this study to measure edge location and feature



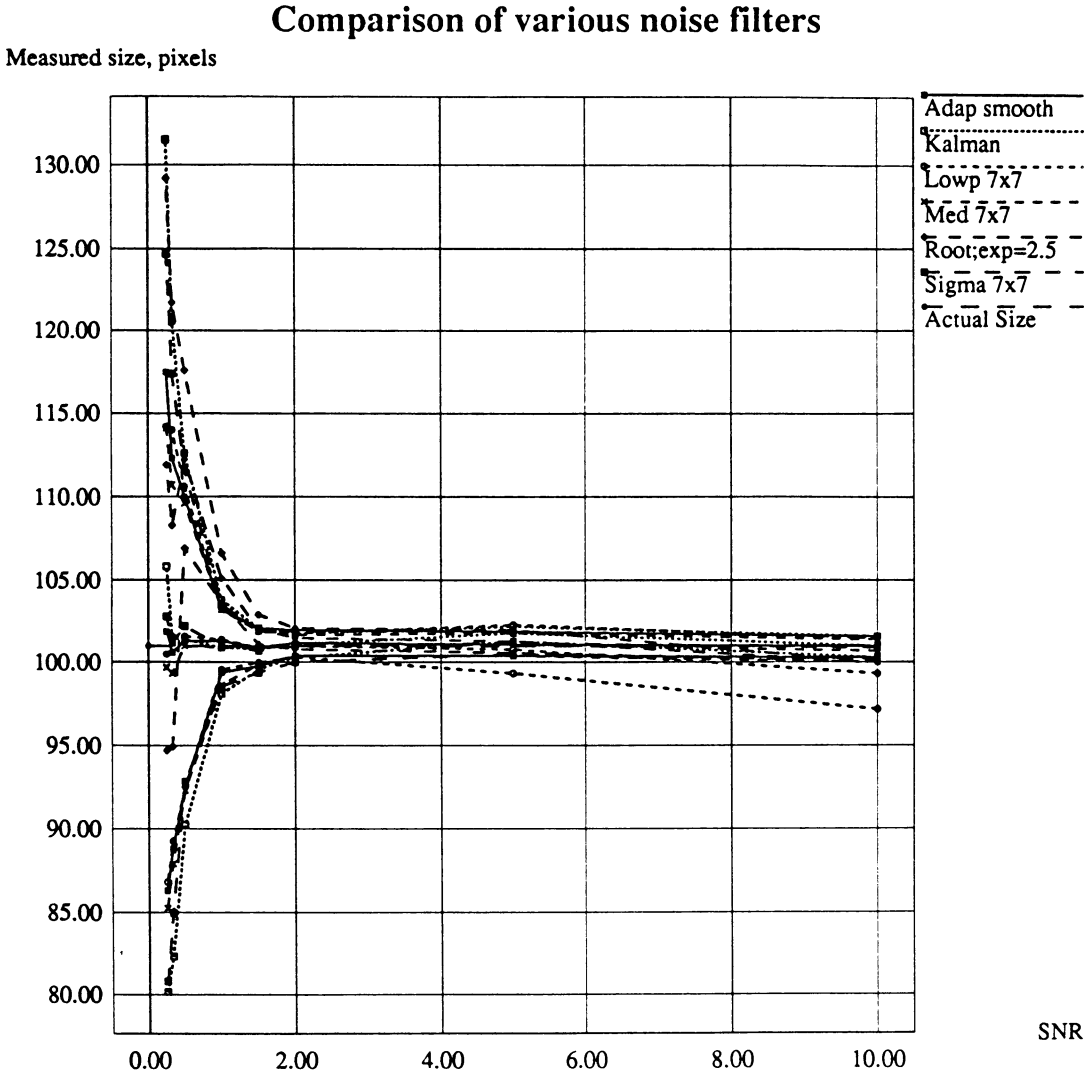
(a)

Figure 7.15. Comparison of various noise filters *(a)* All filters; full range of SNR plotted



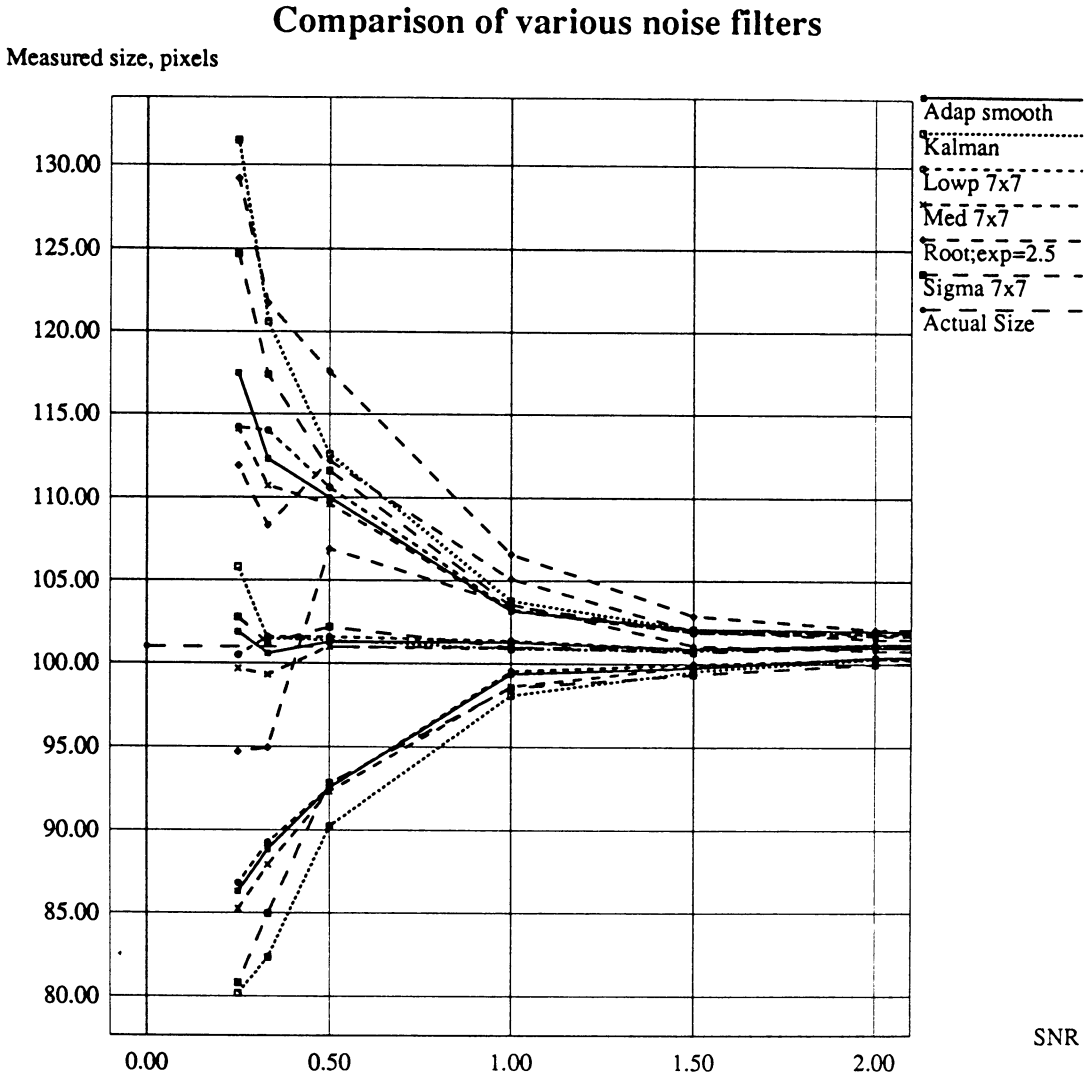
(b)

Figure 7.15. (cont'd) *(b)* Selected filters; full range of SNR plotted



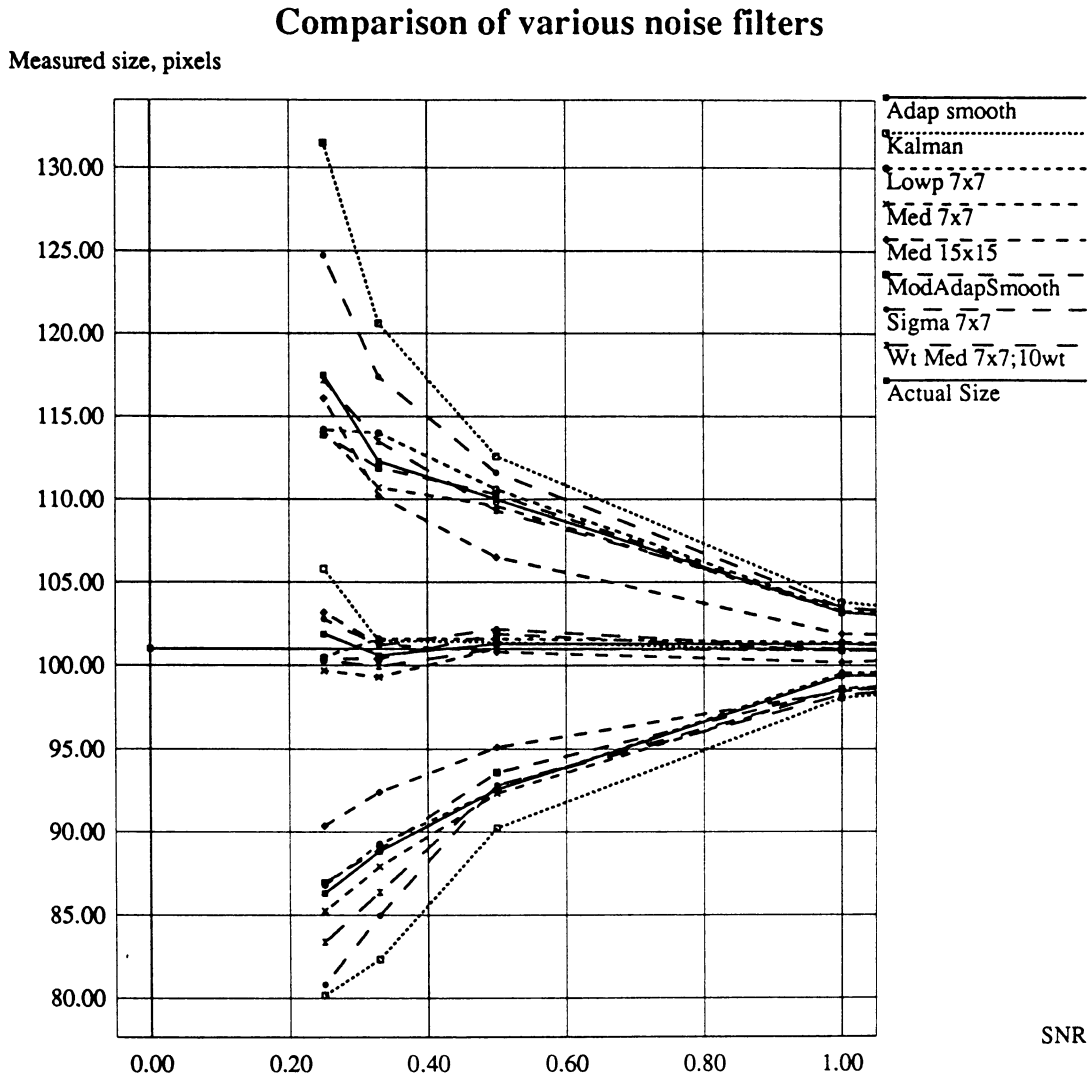
(c)

Figure 7.15. (cont'd) *(c)* All filters; range from SNR=0.0 to SNR=2.0 plotted



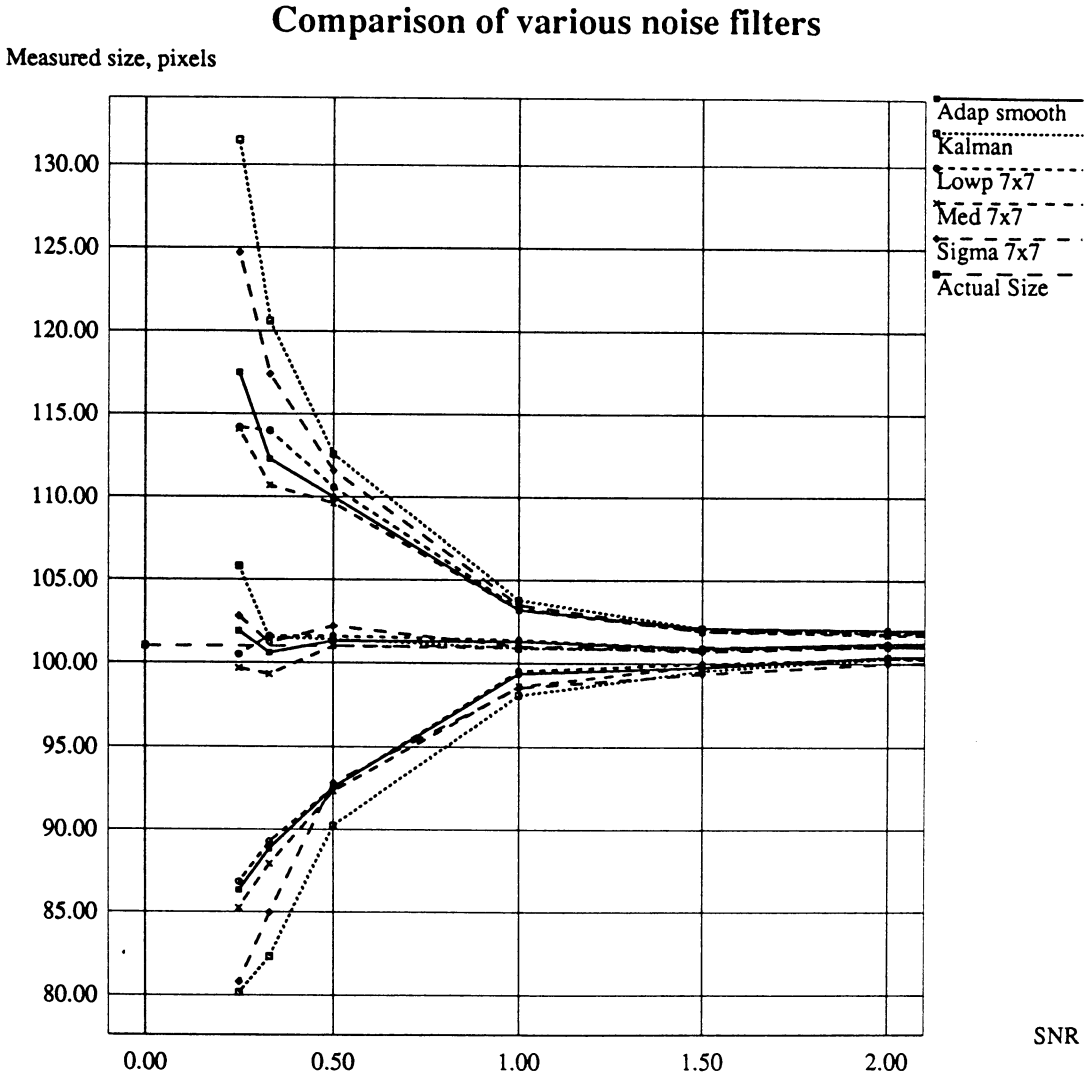
(d)

Figure 7.15. (cont'd) *(d)* Selected filters; range from SNR=0.0 to SNR=2.0 plotted



(a)

Figure 7.16. Comparison of noise filters without root filter *(a)* All filters except root; range from SNR=0.0 to SNR=1.0 plotted



(b)

Figure 7.16. (cont'd) *(b)* Selected filters without root filter; range fror SNR=0.0 to SNR=2.0 plotted

size in an objective, repeatable, and statistically meaningful way found the Kalman filter to have significantly poorer performance than other faster and less sophisticated algorithms. The adaptive smoothing filter, lowpass filter, and median filter with 7x7 window all have a standard deviation of feature size lower than those for the Kalman and sigma filters, and among themselves are fairly close in feature size performance. The median filter with 15x15 window has the very lowest standard deviation of feature size of all the filters.

The measurements presented graphically in Section 7.2 above are tabulated in Table 7.3 for reference. The numbers without parentheses are the mean feature size, and the numbers in parens are the corresponding standard deviation of feature size. The numbers across the top of the table are the SNR values of the pre-processed images, and the first row of mean and standard deviation data is for the pre-processed images. In the far right column, the execution time of each routine on the Stellar GS1025, as carefully timed with a stopwatch, is listed.

We see from the table that at the lowest SNR level used in this study (SNR=0.25), the standard deviation of post-processed feature size for the Kalman filter was about 78% higher than that for the median filter with a 7x7 window, and almost 100% higher than that for the median filter with a 15x15 window. We also note that the execution time of the Kalman filter is about 36 times that of the median filter with a 7x7 window. Clearly, at least within the domain of the idealized images studied here, the median filter gives the best feature size estimate while simultaneously having the fastest execution time. The other two filters with feature size estimation performance close to that of the median filter, namely the adaptive smoothing filter and the lowpass filter,

Table 7.3. Mean and standard deviation of feature size for pre- and post-processed images

Routine	SNR								Proc. time,
	0.25	0.33	0.5	1.0	1.5	2.0	5.0	10.0	mm:ss
Pre-processed images	118.9 (39.10)	110.3 (32.02)	102.2 (25.68)	101.1 (7.272)	100.9 (3.693)	101.0 (2.145)	101.2 (0.712)	101.0 (0.678)	N/A
Adaptive Smoothing Filter	101.9 (15.60)	100.6 (11.74)	101.3 (8.714)	101.3 (1.914)	100.9 (1.174)	101.2 (0.802)	101.2 (0.673)	101.0 (0.679)	04:17
Kalman Filter	105.9 (25.67)	101.5 (19.13)	101.4 (11.16)	100.9 (2.854)	100.8 (1.273)	101.1 (0.757)	101.8 (0.434)	101.0 (0.630)	19:07
Median Filter 7x7	99.67 (14.43)	99.31 (11.39)	101.0 (8.662)	100.9 (2.262)	100.9 (1.045)	101.0 (0.656)	101.1 (0.707)	100.7 (0.693)	00:32
Median Filter 15x15	103.2 (12.85)	101.3 (8.918)	100.8 (5.962)	100.2 (1.712)	101.0 (0.731)	101.1 (0.749)	101.1 (0.612)	100.6 (0.733)	00:54
Mod. Adapt. Smoothing Filter	100.4 (13.47)	100.4 (11.47)	101.9 (8.337)	100.9 (2.282)	100.8 (1.158)	101.0 (0.877)	101.2 (0.703)	101.0 (0.689)	03:17
Root Filter	111.9 (17.23)	108.3 (13.36)	112.3 (5.390)	105.1 (1.503)	102.0 (0.928)	101.5 (0.614)	101.3 (0.660)	100.0 (0.00)	02:50
Sigma Filter	102.8 (21.94)	101.2 (16.19)	102.2 (9.373)	101.0 (2.502)	100.7 (1.306)	101.0 (0.956)	101.2 (0.721)	100.9 (0.710)	07:06
Weighted Median Filter	100.3 (16.93)	99.94 (13.57)	101.0 (8.250)	100.9 (2.643)	100.8 (1.083)	100.9 (0.802)	101.0 (0.689)	101.0 (0.660)	10:10
Lowpass Filter 7x7	100.5 (13.70)	101.7 (12.39)	101.6 (9.04)	101.4 (1.861)	100.9 (0.943)	101.0 (0.748)	100.8 (1.493)	99.31 (2.158)	01:35

have execution times respectively 800% and 300% slower than that of the median filter.

7.2.8 Histogram Equalization

As its name implies, the histogram equalization algorithm attempts to equalize (i.e., make uniform) the histogram of the input image. This algorithm is a contrast enhancement tool, and not a noise filter. Unlike the filters of the previous subsections, histogram equalization is a *point transformation* - each pixel is transformed by the same function, and the values of its neighboring pixels have no influence on its transformed value. The transformation function for histogram equalization is simply the cumulative distribution function for the image. The reader is referred to Gonzalez and Wintz (1987, p. 146) and Jain (1989, p. 241) for further details. HAPPI's histogram equalization routine takes no parameters. Figure 7.17 shows measured size vs. SNR for HAPPI's histogram equalization routine. It may be seen from Figure 7.17 that at SNR's below about 1.0, histogram equalization has little effect on the feature size estimate. However, at higher SNR's, the feature size estimate for post-processed images is far worse than that for pre-processed images. This is due to the fact that for test images with high SNR, histogram equalization greatly *increased* the noise variance without increasing the signal strength by nearly as much. At high SNR values, the histograms of the pre-processed test images were strongly bimodal, with the histogram data tightly clustered about the two modes, and with one mode strongly dominating. When such a histogram is equalized, the histogram data representing the noise tends to get spread out by

a much larger factor than the data representing the clean signal; the result is a dramatic decrease in SNR, which in turn degrades feature size estimate. Figure 7.17 shows the histograms of a test image with an SNR of 10 and the histogram-equalized version of the test image. From Figure 7.16 we can conclude that for purposes of edge detection and feature size measurement in noisy images, histogram equalization does not effect any improvement.

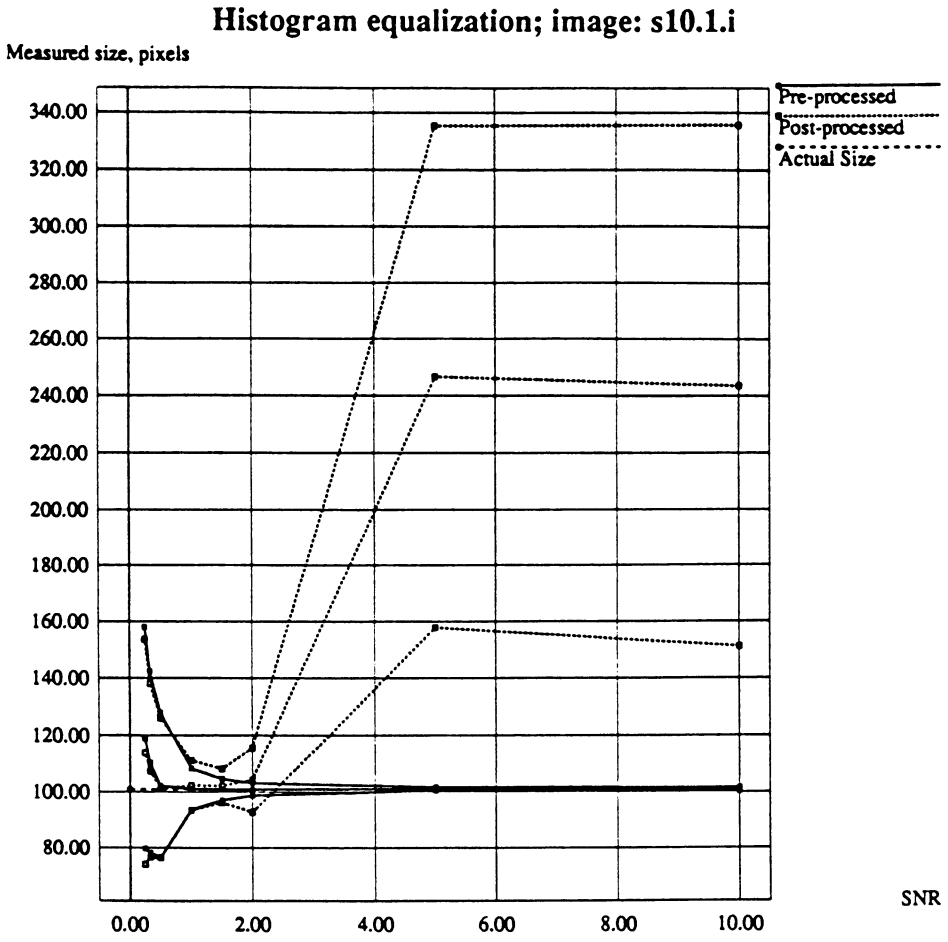
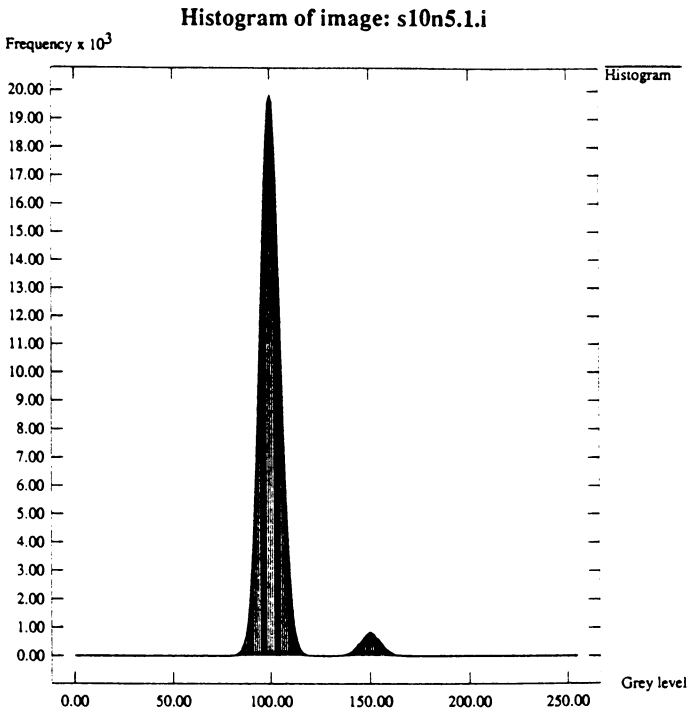


Figure 7.17. Measured size vs. SNR for histogram equalization

(a)



(b)

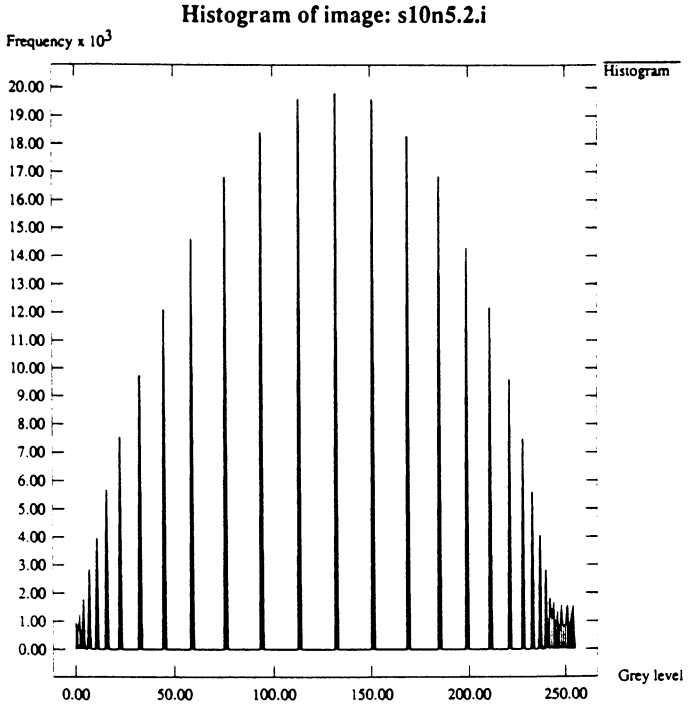


Figure 7.17. Histogram equalization results *(a)* Histogram of original test image with SNR = 10.0; *(b)* Histogram of histogram-equalized image from (a)

7.2.9 Contrast Stretching

HAPPI's contrast stretching routine, "Expand Grey Level," found under the Contrast Enhancement menu, does a simple remapping of individual pixel grey levels. Like histogram equalization, it is a point transformation. The routine takes two parameters, an upper and a lower grey level. All image data below the lower grey level is mapped in the output image to grey level zero; image data above the upper level is similarly mapped to grey level 255, the maximum grey level in HAPPI's image format. All image data falling between the upper and lower grey levels is remapped to the 0-255 grey level range in a linear fashion. The transformation may be written as:

$$y(i,j) = (x(i,j)-\alpha)*255/(\beta - \alpha); \qquad \alpha < x(i,j) < \beta \tag{7.7.1}$$

$$y(i,j) = \alpha; \qquad x(i,j) < \alpha \tag{7.7.2}$$

$$y(i,j) = \beta; \qquad x(i,j) > \beta \tag{7.7.3}$$

where $y(i,j)$ is the output image, $x(i,j)$ is the input image, and β and α are the upper and lower grey levels, respectively. In our tests, we used an upper grey level of 150 and a lower grey level of 100, equal to the grey levels of the foreground and background, respectively, in our original noiseless test image. Figure 7.18 shows measured size vs. SNR for the contrast stretch routine. From the figure, we may see that contrast stretching has almost no effect on the feature size estimate of the post-processed image. Table 7.4 lists feature size mean and standard deviation data for histogram equalization and contrast stretching in the same format as Table 7.3.

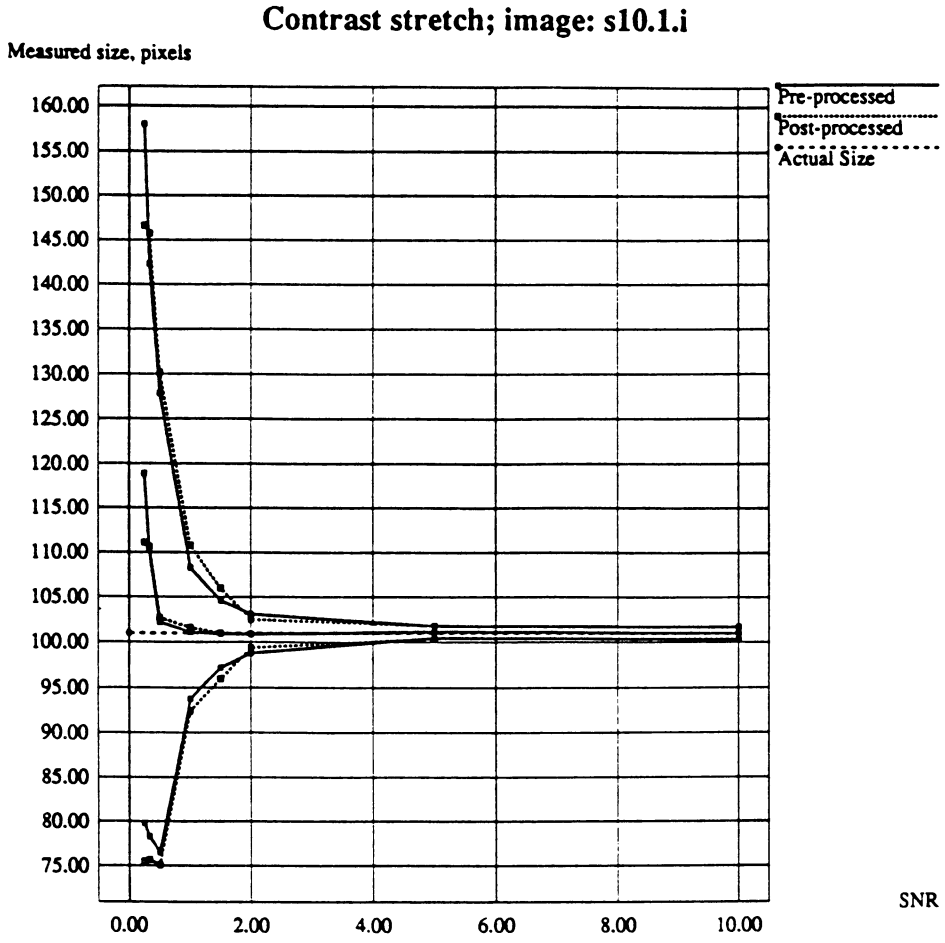


Figure 7.18. Measured size vs. SNR for contrast stretch routine

Table 7.4. Mean and standard deviation of feature size for histogram equalization and contrast stretch

Routine	SNR								Proc. time, mm:ss
	0.25	0.33	0.5	1.0	1.5	2.0	5.0	10.0	
Pre-processed images	118.9 (39.10)	110.3 (32.02)	102.2 (25.68)	101.1 (7.272)	100.9 (3.693)	101.0 (2.145)	101.2 (0.712)	101.0 (0.678)	N/A
Histogram Equalization	113.9 (39.81)	107.3 (30.76)	101.3 (24.88)	102.4 (8.805)	102.2 (5.969)	104.3 (11.42)	246.6 (88.75)	243.4 (92.33)	00:08
Contrast Stretch	111.1 (35.55)	110.7 (35.02)	102.7 (27.58)	101.6 (9.206)	101.0 (5.006)	100.9 (1.547)	101.1 (0.718)	101.0 (0.661)	00:09

7.3 Effects of Scattering LSF

In an actual radiograph, we would not see perfect step edges like those in our test images, even if the physical specimen from which the radiograph was made had such a perfect edge. Even perfect step edges in the specimen will be imaged on a radiograph with a certain amount of blur due to different physical phenomena, such as scattering, geometric unsharpness, film unsharpness, etc. Thus, while the measurements presented so far give a sense of how well we can measure feature size and of how HAPPI's processing routines change the size of an *idealized image feature*, they are somewhat removed from the results we would get with images of real radiographs. To briefly investigate the effects of non-ideal edge profiles on edge location and feature size measurement performance, we convolved a test image similar to that of Figure 6.1 with a two-dimensional version of the line spread function

of Equation 5.4, using various values of the scattering unsharpness parameter *a* of that equation. The test image used in this case consisted of a uniform background of grey level 100 with a uniform-width, uniform-intensity vertical stripe of grey level 150 centered in the image, as shown in Figure 7.19.

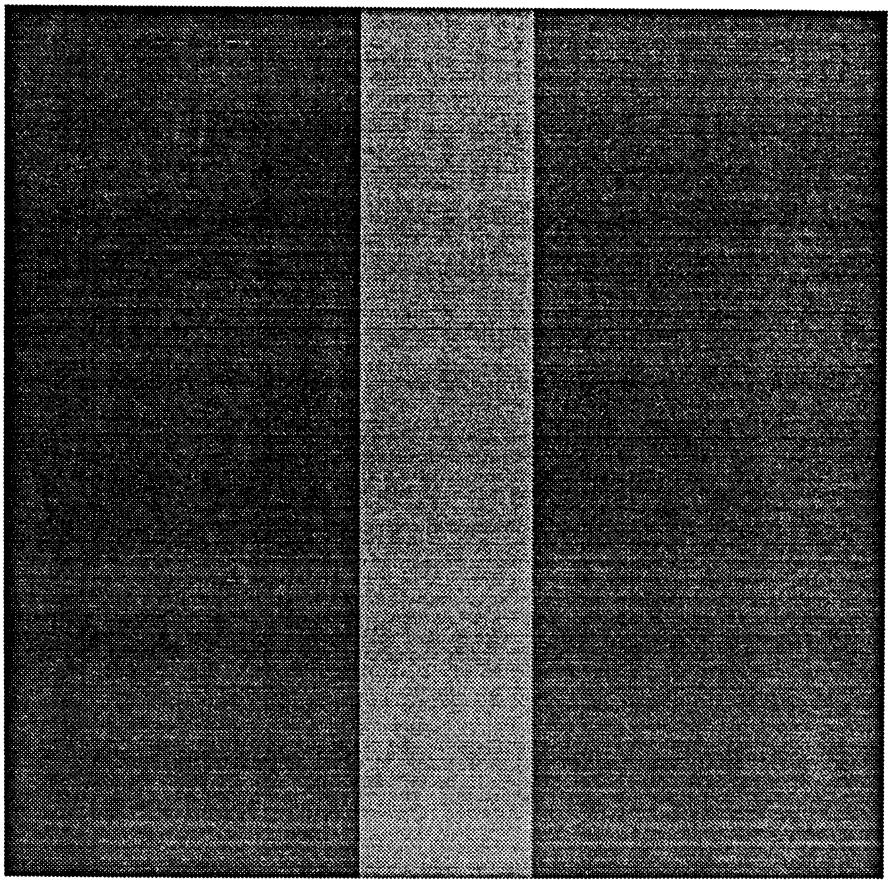
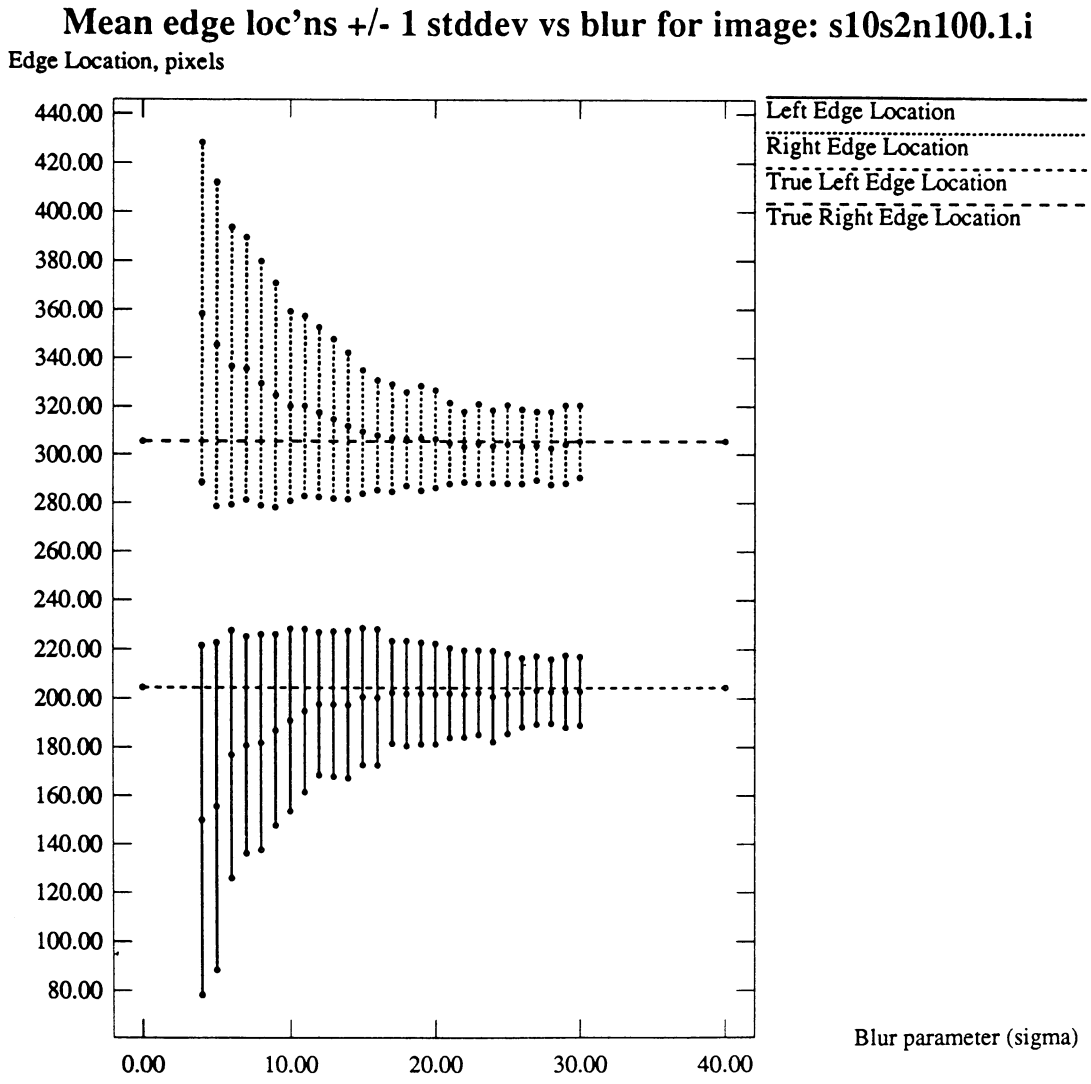


Figure 7.19. Test image used to test effects of scatter unsharpness

This test image could be considered a primitive model of a slot of uniform cross-section in a flat plate. We used this test image instead of the image of Figure 6.1 so that we could apply large 2-d scattering unsharpness blur functions to it and still calculate valid edge location and feature size statistics from row ensemble averages. Had we convolved the test image of Figure 6.1 with a large 2-d version of the LSF of Equation 5.4, we would have smeared the perfect step edges of that image in *both* spatial directions, and would not have had a large sample size of 1-d image slices with identical statistical properties from which to compute the mean and standard deviation of edge location and feature size.

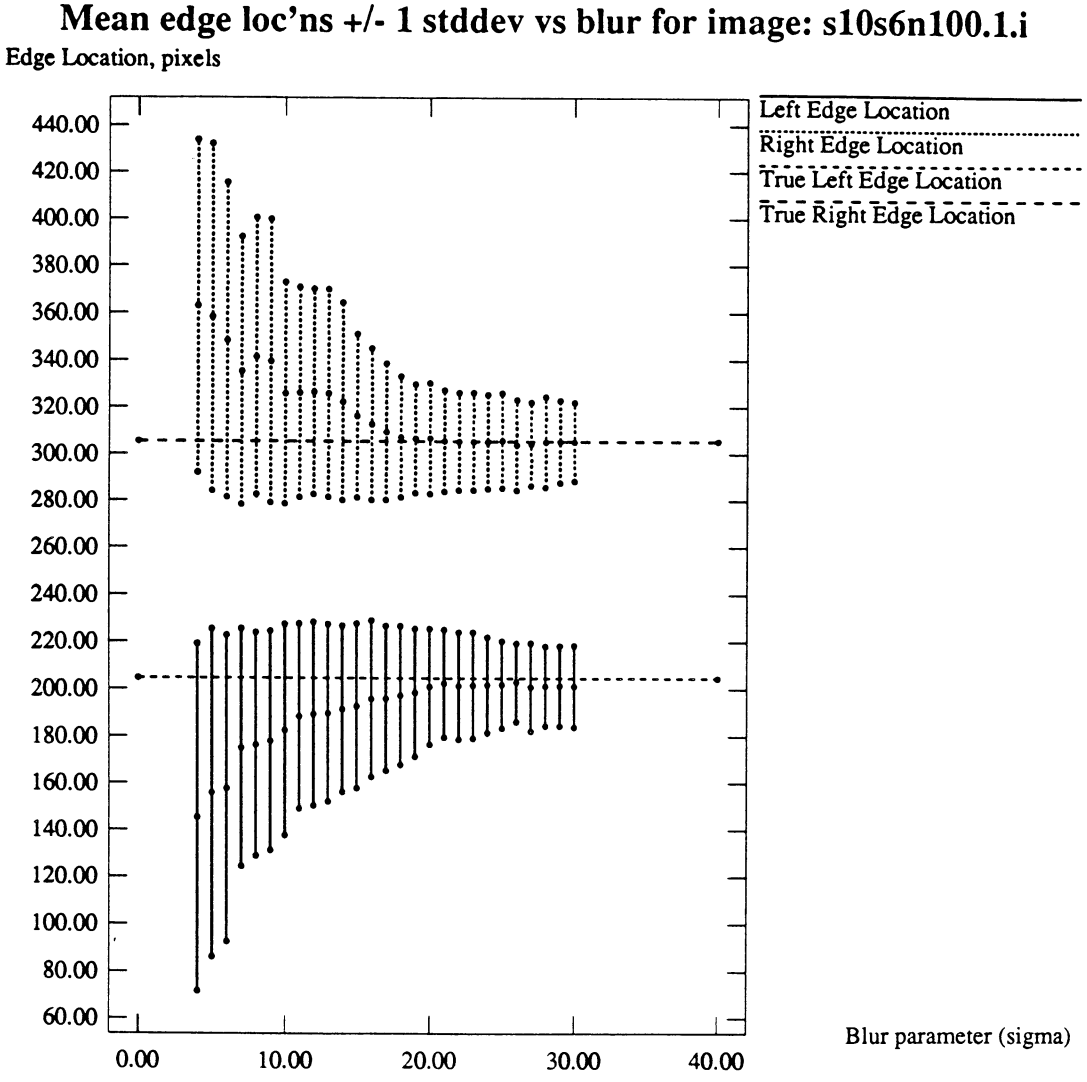
Noise was added to the blurred images in various amounts, and the resulting blurred, noisy images were run through the program `mrowblur` to generate plots of edge location mean and variance vs. gaussian blur parameter, σ_b. The gradient maximum method was again used exclusively when running `mrowblur`. A few selected range bar plots from mrowblur are shown in Figure 7.20. We may observe the following from the plots of Figure 7.20:

1) As the scattering unsharpness parameter a of Equation 5.4 decreases, the critical value of the gaussian blur parameter σ_b increases. (Refer to Section 6.3.2 for a discussion of how critical σ_b values were determined from range bar plots of edge location vs. σ_b.)



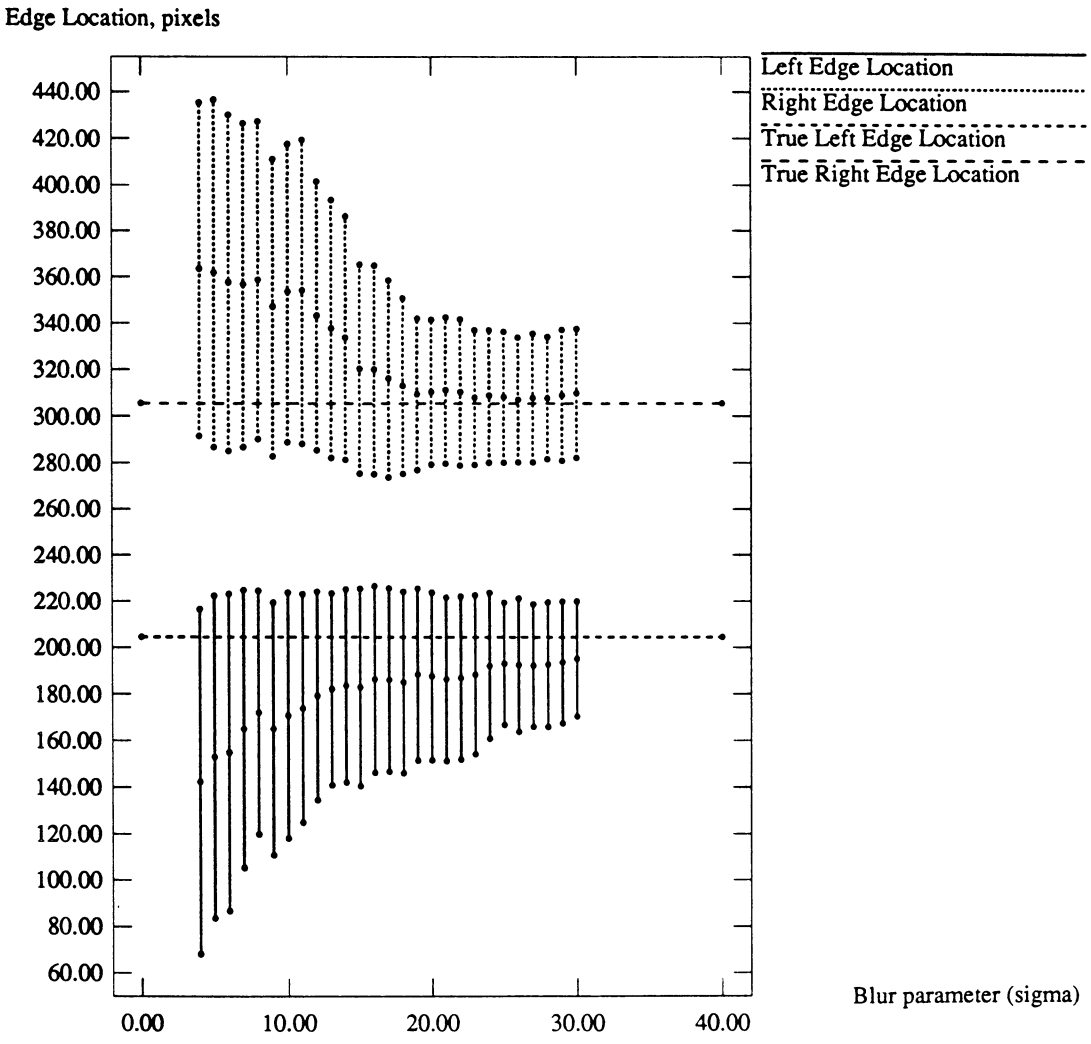
(a)

Figure 7.20. Range bar plots of mean edge location ± one standard deviation vs. blur parameter for test images with scatter blur and noise *(a)* SNR=0.5; scatter blur parameter a=0.4



(b)

Figure 7.20. (cont'd) *(b)* SNR=0.5; scatter blur parameter a=0.1



(c)

e 7.20. (cont'd) *(c)* SNR=0.5; scatter blur parameter a=0.04

2) As the parameter a decreases, the terminal value of edge location standard deviation σ_e (i.e. the almost constant value attained by σ_e as σ_b increases beyond its critical value) increases as well.

The above two observations may be explained by considering the effect of the parameter a on the size of the scattering unsharpness blurring function; small values of a yield spatially large blurring functions, and vice versa. A small value of a will thus result in more smearing of sharp edges. When edges are detected using a gradient scheme such as the one we have used in this study, a slowly rising edge will naturally be harder to locate precisely in a noise field of a given strength than a sharply rising edge in the same noise field.

To investigate the relationship between SNR and feature size estimation performance in an image with blurred step edges, we blurred the test image of Figure 7.19 with a 2-d version of the LSF of Equation 5.4, using an unsharpness parameter of 0.2. Various amounts of noise were added to the blurred test image to produce a sequence of test images of decreasing SNR similar to that described in Section 6.3.1. The program `mrowblur` was run on this sequence of test images to produce a sequence of range bar plots of edge location mean and standard deviation vs. gaussian blur parameter σ_b. Critical σ_b values were determined from the sequence of range bar plots in the same way as described in Section 6.3.2.

The critical σ_b values for the blurred test image are shown in Table 7.5. Note that at high SNR's the critical σ_b values are somewhat higher for the blurred test image than those for the test image with perfect step edges (Cf.

Table 6.1). The higher σ_b values seen with the blurred image is most likely due to the fact that to a *gradient* operator, a *slowly rising* edge bathed in noise of a given strength has a lower signal-to-noise ratio than a step edge bathed in noise of the same strength. Thus, it was neccessary to use slightly larger σ_b values with the blurred test image to reduce the heightened effect of noise on edge location estimate. At lower SNR's, the critical σ_b values are slightly lower than the corresponding values in Table 6.1, but the percentage difference of σ_b between the two tables at low SNR is small.

Table 7.5. Critical values of blur parameter as function of SNR for blurred test image

Test image SNR	Critical value of σ_b
10.0	1.0
5.0	2.0
2.0	5.0
1.5	7.0
1.0	10.0
0.5	22.0
0.33	28.0
0.25	33.0

To investigate how HAPPI's noise filters might fare in improving feature size estimate in an image with blurred step edges, we ran the adaptive smoothing filter on the sequence of noisy blurred test images created from the test image of Figure 7.19. The critical σ_b values of Table 7.5 were fed as input to the `rowblur` program to determine mean and standard deviation of feature size, and the pre-processed and post-processed size estimates were plotted as a

function of SNR as shown in Figure 7.21. Note that at high SNR's the standard deviation of feature size for the blurred pre-processed image is much larger (at about 5 pixels) than that for the pre-processed images in Section 7.2, due to the gradient operator having to search for a weaker (and thus harder-to-locate) signal.

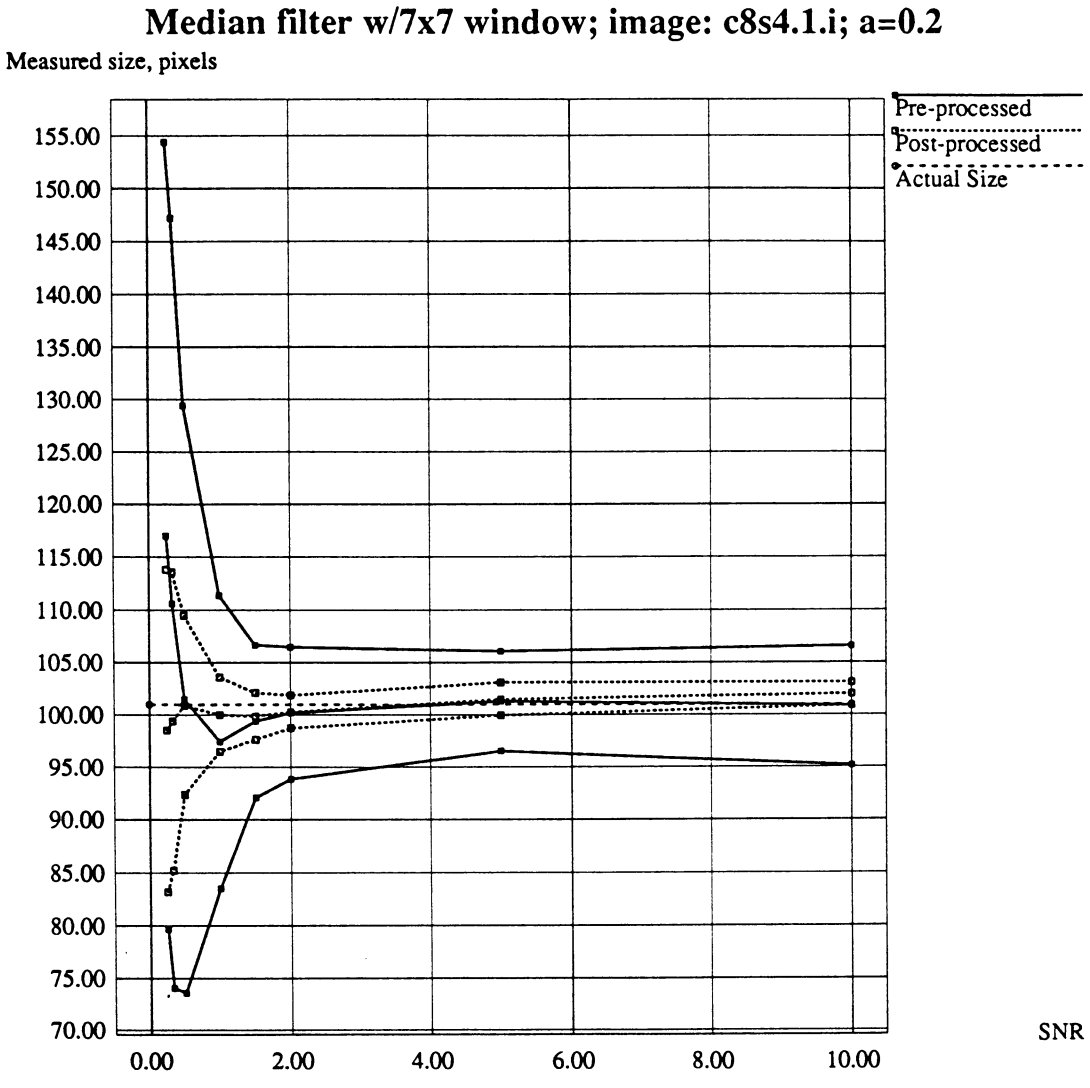


Figure 7.21. Measured size vs. SNR for adaptive smoothing filter and blurred test image

7.4 Comparison of Edge Location/Feature Sizing Methods with Sobel Operator

To help put the edge location and feature size measurements of Section 7.2 in perspective, we also measured edge locations in pre-processed and post-processed images by a simple "seat-of-the-pants" method that might be used by an image processing practitioner. Our method was to simply apply HAPPI's Sobel edge detection routine to the same pre-processed and post-processed images studied in Section 7.2, and use HAPPI's "Pixel Analyzer" utility to find the coordinates of edge points in the resulting Sobel-processed images. (The Pixel Analyzer is described in Chapter 3, Subsection 3.3.3.) In Table 7.6, we list feature size measurements as could best be determined by eye using HAPPI's Pixel Analyzer on Sobel-processed images; the table is similar in format to Table 7.3, but does not list processing time for running the Sobel routine on each post-processed image, as the routine's processing time - approximately 50 seconds on the Stellar GS1025 - was identical for each 511x511 test image. The numbers without parentheses in each table cell are the estimated feature sizes (in pixels). The numbers in parentheses in each table cell are not *computed* standard deviations of feature size as in Table 7.3, but are rather a subjective visual estimate (made using the Pixel Analyzer) of the width, in pixels, of the edge response in the Sobel-processed image.

It was seen that at high SNR's, a user could quite reliably determine feature size to within two or three pixels using the Sobel routine and the Pixel Analyzer. However, at low SNR's (below about SNR=1), the output of the Sobel routine was so noisy as to make the image feature undetectable, and thus unmeasureable, by eye. Poor performance of the Sobel routine at low SNR is to

be expected, as the Sobel masks are only 3 pixels by 3 pixels square, and are thus not capable of doing very much noise smoothing.

Table 7.6. Visually estimated feature size and edge response width using Sobel routine and Pixel Analyzer

Routine	SNR							
	0.25	0.33	0.5	1.0	1.5	2.0	5.0	10.0
Pre-processed images	-	-	-	-	101 (2)	101 (2)	101 (2)	101 (2)
Adaptive Smoothing Filter	-	-	-	100 (2)	100 (2)	101 (2)	101 (2)	101 (2)
Kalman Filter	-	-	-	101 (3)	101 (3)	101 (3)	101 (3)	101 (3)
Median Filter 7x7	-	-	-	102 (15)	101 (12)	101 (11)	101 (2)	101 (2)
Median Filter 15x15	-	-	-	102 (14)	101 (12)	101 (12)	101 (2)	101 (2)
Mod. Adapt. Smoothing Filter	-	-	-	100 (2)	100 (2)	101 (2)	101 (2)	101 (2)
Root Filter	-	-	-	101 (2)	101 (2)	101 (2)	101 (2)	101 (2)
Sigma Filter	-	-	-	101 (4)	101 (3)	101 (2)	101 (2)	101 (2)
Weighted Median Filter	-	-	-	101 (3)	101 (3)	101 (3)	101 (2)	101 (2)

CHAPTER 8: CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

8.1 Summary

In this thesis, we have described and evaluated HAPPI, an integrated NDE X-ray image processing software package to which the author was a contributor. Contemporary issues in image processing were discussed to lend some perspective to HAPPI's design and features, and the design objectives were set forth. Our evaluation of the finished product concentrated on ways in which it could be improved. A detailed, step-by-step procedure for extending the image processing functionality of HAPPI was given. This procedure includes an overview of such things as program control flow and data objects, and provides documentation that was previously missing or incomplete. The procedure will allow any competent C programmer to add processing routines to the program, and is intended to encourage the continued maintenance and development of the program.

In the latter part of the thesis, we have investigated the influence of HAPPI's image processing routines on image feature size. Methods of edge detection and feature size calculation were presented. In the presence of noise, such calculations will be random variables. Our measurement methods have been implemented so as to quantify the randomness of the measurements. The measurement methods were applied to idealized images of simple specimen geometries to get a sense of the limits of measurement accuracy under ideal circumstances. Test images processed with a variety of HAPPI's routines were measured before and after processing, with the results

providing some quantification of how various processing routines affect feature size estimates. The measurements presented show that, in our test cases at least, HAPPI's noise filters improved the feature size estimate. Other processing routines that were not intended as noise filters were tested and found not to improve, and in fact to worsen in some cases, the feature size estimate.

In general, for noisy unprocessed images and noisy images processed with HAPPI's noise filtering routines, the standard deviation of our feature size estimate was large (e.g., 60% of feature size for an unprocessed noisy image and 20% to 30% for processed images) at low SNR (e.g., SNR=0.25) values, and decreased dramatically as SNR increased to about 2.0. For SNR values in the range 0.25 to 2.0, HAPPI's noise filters improved the standard deviation of feature size estimate by anywhere from few percent at high SNR's to about 300% at low SNR's. Above an SNR of 2.0, the standard deviation of feature size estimate was essentially unchanged by processing. It was seen that among the noise filters, the median filter provided the best feature size estimate at low SNR's while also having the fastest execution time. The Kalman filter gave the worst feature size estimate while also having the longest execution time.

The effect of blurred edges on the feature size estimate was briefly investigated. It was seen that more slowly rising edges require a larger blur parameter to be used in our edge detection scheme. Also, slowly rising edges bathed in noise of a given strength had higher edge location variance than sharply rising edges bathed in the same noise field. Finally, it was seen that for an image with blurred edges, noise filtering can significantly improve the standard deviation of feature size estimate at SNR values above SNR=2.

Results from our edge location and feature size measurement scheme were compared with those attainable from a simple and subjective approach using the Sobel operator. For SNR's above about 1.0, the two methods yielded similar results. Our scheme showed its utility at SNR values below 1.0; the Sobel operator could not produce any edge location information at such low SNR values.

8.2 Suggestions for Further Work

As suggestions for improvement to HAPPI have already been made in Chapter 3, we will here discuss only suggestions for further work in quantifying the effect of processing routines. There are many parameters (e.g., contrast, noise levels, noise distributions, feature edge profile) which may be varied in test images for feature size measurement, and the various processing routine parameters may be varied as well. There are also many more methods of finding edges than are discussed in this thesis. There is thus a large multidimensional space to explore in investigating the topic of "influence of processing algorithms on feature size," and the work presented here could be extended in many directions.

One possible topic of investigation is the optimization of filter parameters for purposes of edge detection and feature size estimation. Optimal parameters for a given filter will depend on such image characteristics as SNR, contrast, feature edge profile, and feature size. We saw in Chapter 6 that in general, the lower the SNR in an image, the larger the blurring function needed to achieve a given edge location performance. However, in practice we

cannot increase the size of the blurring function without bound. If the feature size is very small compared to the size of blurring function used, then the feature will be smeared and peak signal strength will be diminished. Blurring function size is also limited in practice by the size of the image being blurred. There is thus an interaction between minimum detectable feature size, SNR, image size, and the processing routines, which would be a useful topic for investigation.

As mentioned in Chapter 6, it is hoped that the data presented in this thesis might prove useful in developing analytical expressions or empirical models for change in image feature size with processing. Such developments would be useful, for example, to an NDE inspector concerned with accuracy of flaw size measurements. The expressions or models could be used to judge the size accuracy of alternative processing schemes against other concerns such as detection accuracy, robustness with respect to variation in attributes of input images, complexity of user interaction, and computational cost-efficiency.

BIBLIOGRAPHY

Ahmad, M. O., and D. Sundararajan. "A Fast Algorithm for Two-Dimensional Median Filtering." *IEEE Transactions on Circuits and Systems* Vol. 34(11) (1987): 1364-1374.

Bergholm, F. "Edge Focusing." *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol. 9(6) (1987): 726-741.

Biemond, J. "A Fast Kalman Filter for Images Degraded by Both Blur and Noise." *IEEE Transactions on Acoustics, Speech, and Signal Processing* Vol. 31(5) (1983): 1248-1256.

Biemond, J. "Stochastic Linear Image Restoration." In *Advances in Computer Vision and Image Processing.* Vol. 2, *Image Enhancement and Restoration.* ed. T.S. Huang. Greenwich, Conn.: Jai Press 1986: 213-273.

Blagden, D., and J. Scanlan. "Rapid Prototyping for Imaging and Video Design." *Advanced Imaging* (Nov. 1990): 43-45.

Brown, R. A., J. Xu, and J. P. Basart. "Software Design and Features for an NDE Image Processing Workstation." In *Review of Progress in Quantitative Nondestructive Evaluation* Vol. 9A, ed. D. O. Thompson and D. E. Chimenti. New York: Plenum Publishing Corporation, 1990: 1073-1078.

Brown, R. G. *Introduction to Random Signal Analysis and Kalman Filtering.* New York: John Wiley and Sons, 1983.

Canny, J. "A Computational Approach to Edge Detection." *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol. 8(6) (1986): 679-698.

Christian, K. *The C and UNIX Dictionary.* New York: John Wiley and Sons, 1988.

Clark, A. F. "The International Image Processing Standard - The Story so Far." In *Proceedings of the 'Image Processing and its Applications Conference.'* Savoy Place, London, UK: Institution of Electronic Engineers, 1992: 575-578.

Doering, E. R. "Detection of anomalies in digital images using pixel classification." M.S. thesis, Iowa State University, 1987.

Doering, E. R. "Monthly Report." See *Iowa State University*

Eizember, C. A., and W. A. Graeme. "Nondestructive Testing Meets Image Processing." *Advanced Imaging* (Apr. 1990): 41-44.

Fishman, A., S. Wajnberg, A. Notea, and Y. Segal. "Extraction of Dimensions from Radiographs." *Materials Evaluation* Vol. 39 (1981): 744-747.

Frei, W. "Digital Image Processing Software." In *Digital Image Processing.* Proceedings of SPIE, Vol 528. ed. A.G. Tescher. Los Angeles, Calif.: (Jan. 22-23, 1985): 88-94.

Gonzalez, R. C., and P. A. Wintz. *Digital Image Processing.* 2d ed. Reading, Mass: Addison-Wesley, 1987.

Halmshaw, R. "Film Density and Contrast in Radiography." *British Journal of NDT* (Nov. 1973): 187-188.

Halmshaw, R. "Defect Size Measurement by Radiography." *British Journal of NDT* (Sep. 1979): 245-248.

Halmshaw, R. *Non-Destructive Testing.* London: Edward Arnold, 1987.

Hancock, L., and M. Krieger. *The C Primer.* New York: McGraw-Hill, 1986.

Haykin, S. *Communication Systems.* 2d ed. New York: John Wiley and Sons, 1983.

Iowa State University, Center for Advanced Technology Development. "X-Ray Radiograph Image Analysis Program: Final Report." U.S. Department of Commerce Grant ITA 87-02, Ames, Ia.: Institute for Physical Research and Technology, 1990.

Iowa State University, Electrical Engineering and Computer Engineering Department, NDE Image Processing Group. "Monthly Report." Ames, Ia.: Iowa State University, May 1990.

Iowa State University, Electrical Engineering and Computer Engineering Department, X-ray Image Processing Group. *HAPPI Technical Manual, Vols. 1-4.* Ames, Ia.: Iowa State University, 1990.

Iowa State University, Electrical Engineering and Computer Engineering Department, X-ray Image Processing Group. *HAPPI User's Manual.* Ames, Ia.: Iowa State University, 1990.

Jain, A. K. *Fundamentals of Digital Image Processing.* Englewood Cliffs, N.J.: Prentice-Hall, 1989.

Kelley, A., and I. Pohl. *A Book on C.* Menlo Park, Calif.: Benjamin/Cummings, 1984.

Kernighan, B. W., and D. M. Ritchie. *The C Programming Language.* 2d ed. Englewood Cliffs, N.J.: Prentice Hall, 1988.

Kuan, D. T., A. A. Sawchuck, T. C. Strand, and P. Chavel "Adaptive Noise Smoothing Filter for Images with Signal-Dependent Noise." *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol. 7(2) (1985): 165-177.

Lee, J. S. "The Sigma Filter and Its Application to Speckle Smoothing of Synthetic Aperture Radar Images." In *Statistics: Textbooks and Monographs*. Vol. 53, *Statistical Signal Processing*. ed. E.J. Wegman and J.G. Smith. New York: Marcel Dekker, 1984: 445-459.

Mazor, B., ed. "Scientific/Industrial Imaging: An Industry/Technology Roundtable." *Advanced Imaging* (Aug 1990): 56-75.

Morgan, R., and H. McGilton. *Introducing UNIX System V*. New York: McGraw-Hill, 1987.

Notea, A. "Evaluating Radiographic Systems Using the Resolving Power Function." *NDT International* (Oct, 1983): 263-270.

Nye, A., and T. O'Reilly. *X Toolkit Intrinsics Programming Manual for X Version 11*. Sebastopol, Calif.: O'Reilly and Associates, 1990.

O'Reilly, T. "The Toolkits (and Politics) of X Windows." *UnixWorld* (Feb. 1989): 66-72.

Paragon Imaging Inc. "Visualization Workbench" Lowell, Mass.: Paragon Imaging Inc., 1989. product literature.

Pfeiffer, D. "The Case for the Embedded Imaging Approach." *Advanced Imaging* (Oct. 1990): 36-40.

Pratt, W. K. *Digital Image Processing*. 2d ed. New York: John Wiley and Sons, 1991.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1988.

Preston Jr., K. "Benchmarks for Image Processing." *Advanced Imaging* (May 1990): 30-38.

Safae-Nili, A. "A study of the effect of the media turbulence and data processing in Fourier synthesis imaging." M.S. thesis, Iowa State University, 1989.

Scheifler, R. W., J. Gettys, and R. Newman. *X Window System: C Library and Protocol Reference*. Bedford, Mass.: Digital Press, 1988.

Schwarz, R. "The Electronic Imaging Industry: Where We Stand." *Advanced Imaging* (Aug. 1990): 17-19,76.

Segal, Y. and F. Trichter. "Limitations in Gap Width Measurements by X-ray Radiography." *NDT International* (Feb. 1988): 11-16.

Sheppard, L. M. "Detecting Material Defects in Real Time." *Metal Progess* (Nov. 1987): 53-60.

Stellar Computer Inc. *Stellar Graphics Supercomputer Model GS1000 System Overview*. Document MD-0001 Release 2.0, Newton, Mass.: Stellar Computer Inc. 1987.

Stellar Computer Inc. *Stellar C Language User's Guide*. Document MD-3201(a) Release 2.0, Newton, Mass.: Stellar Computer Inc. 1988.

Stellar Computer Inc. *Stellix Operating System Programmer's Guide*. Document MD-1004 Release 2.0, Newton, Mass.: Stellar Computer Inc. 1988.

Stephenson, T. "About PIK: The Programmer's Imaging Kernel Standard." *Advanced Imaging* (Aug. 1990): 20-22,77.

Wallace, G. "The JPEG Still Picture Compression Standard." *Communications of the ACM* Vol. 34(4) (Apr. 1991): 30-44.

X-ray Image Processing Group. *HAPPI Technical Manual.* See *Iowa State University*

X-ray Image Processing Group. *HAPPI User's Manual.* See *Iowa State University*

Yencharis, L. "Imaging Markets: 1989 - 1994." *Advanced Imaging* (Aug. 1990): 10-16.

Yenarchis, L. "Imaging Chip Overview '91: Into the RISC-y '90's." *Advanced Imaging* (Oct. 1990): 23-26,77.

Zheng, Y., and J. P. Basart. "Image Analysis, Feature Extraction, and Various Applied Enhancement Methods for NDE X-ray Images." In *Review of Progress in Quantitative Nondestructive Evaluation* Vol. 7A, ed. D. O. Thompson and D. E. Chimenti. New York: Plenum Publishing Corporation, 1988: 795-803.