Expressing imaging algorithms using a C++ based image algebra programming environment

Davender Nath Gupta

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
EXPRESSING IMAGING ALGORITHMS
USING A C++ BASED
IMAGE ALGEBRA PROGRAMMING ENVIRONMENT

by

Davender Nath Gupta

Captain, Canadian Forces
B.Eng. Royal Military College of Canada
(1984)

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in the Center for
Imaging Science in the College of
Graphic Arts and Photography of the
Rochester Institute of Technology

18 April 1990

Signature of the Author Davender Nath Gupta

Accepted by Mendi Vaez-Pravani
Coordinator, M.S. Degree Program
The M.S. Degree Thesis of Davender Nath Gupta has been examined and approved by the thesis committee as satisfactory for the thesis requirement for the Master of Science degree.

Dr. Edward R. Dougherty, Thesis Advisor

Dr. Peter G. Anderson

Dr. Harvey Rhody

10 May 1990
Date
THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF GRAPHIC ARTS AND PHOTOGRAPHY

Title of Thesis: EXPRESSING IMAGING ALGORITHMS USING A C++ BASED IMAGE ALGEBRA PROGRAMMING ENVIRONMENT

I, Davender Nath Gupta, hereby grant permission to the Wallace Memorial Library of R.I.T. to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: 10 MAY 80
EXPRESSING IMAGING ALGORITHMS USING A C++ BASED IMAGE ALGEBRA PROGRAMMING ENVIRONMENT

by

Davender Nath Gupta

Submitted to the Center for Imaging Science in partial fulfillment of the requirements for the degree of Master of Science at the Rochester Institute of Technology

ABSTRACT

Homogeneous Image Algebra is used as the basis for an image processing environment that uses the image, instead of the pixel, as the fundamental unit being manipulated. The object-oriented language C++ is used to implement the environment. Examples of applications are that include filtering, region growing, skeletonization, Fourier Transforms and Hadamard transforms are presented. This thesis concludes that C++ does have several features which are of great benefit in implementing image processing algorithms.
ACKNOWLEDGEMENTS

This work has been made possible through the support of the Canadian Forces.

I wish to thank Dr. Edward Dougherty, who first suggested my thesis topic. Work on this problem allowed me to become comfortable in Pascal, C, and C++, as well as with UNIX, thus keeping my computer skills up to date (and giving me two excuses to go to LA!).

I also wish to acknowledge the valuable assistance of the department of Computer Science and Information Technology at RIT, and specifically Dr. Peter Anderson, who provided the facilities to implement this work, and gave important insight and suggestions.
DEDICATION

To the many good friends I have made during my time here.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>vii</td>
<td>TABLE OF CONTENTS</td>
</tr>
<tr>
<td>1-1</td>
<td>CHAPTER 1 - INTRODUCTION</td>
</tr>
<tr>
<td>1</td>
<td>AIM OF THESIS</td>
</tr>
<tr>
<td>1</td>
<td>Aim of thesis</td>
</tr>
<tr>
<td>1</td>
<td>PROBLEM STATEMENT</td>
</tr>
<tr>
<td>1</td>
<td>Background</td>
</tr>
<tr>
<td>1</td>
<td>Problem</td>
</tr>
<tr>
<td>2-1</td>
<td>CHAPTER 2 - PREVIOUS WORK</td>
</tr>
<tr>
<td>1</td>
<td>GENERAL</td>
</tr>
<tr>
<td>1</td>
<td>Chapter Objective</td>
</tr>
<tr>
<td>1</td>
<td>IMAGE ALGEBRA</td>
</tr>
<tr>
<td>1</td>
<td>Context</td>
</tr>
<tr>
<td>1</td>
<td>Origin of image algebra</td>
</tr>
<tr>
<td>2</td>
<td>COMPUTER IMPLEMENTATIONS OF IMAGE ALGEBRA</td>
</tr>
<tr>
<td>2</td>
<td>General</td>
</tr>
<tr>
<td>2</td>
<td>AFATL Image Algebra</td>
</tr>
<tr>
<td>3</td>
<td>C-3PL</td>
</tr>
<tr>
<td>4</td>
<td>SLIP</td>
</tr>
<tr>
<td>4</td>
<td>Other implementations</td>
</tr>
<tr>
<td>5</td>
<td>CONCLUSION</td>
</tr>
<tr>
<td>3-1</td>
<td>CHAPTER 3 - GENERAL DESIGN</td>
</tr>
<tr>
<td>1</td>
<td>GENERAL</td>
</tr>
<tr>
<td>1</td>
<td>Chapter Objective</td>
</tr>
<tr>
<td>1</td>
<td>USERS</td>
</tr>
<tr>
<td>1</td>
<td>Intended audience and uses</td>
</tr>
<tr>
<td>1</td>
<td>Level of user sophistication</td>
</tr>
<tr>
<td>1</td>
<td>Hardware and Software</td>
</tr>
<tr>
<td>2</td>
<td>DESIGN QUALITIES</td>
</tr>
<tr>
<td>2</td>
<td>CHOICE OF MATHEMATICAL BASIS</td>
</tr>
<tr>
<td>3</td>
<td>Conclusion</td>
</tr>
<tr>
<td>4</td>
<td>USER INTERFACE</td>
</tr>
<tr>
<td>4</td>
<td>General</td>
</tr>
<tr>
<td>4</td>
<td>Block diagrams</td>
</tr>
<tr>
<td>5</td>
<td>Alphanumeric expressions</td>
</tr>
<tr>
<td>6</td>
<td>IMPLEMENTATION STYLES</td>
</tr>
<tr>
<td>6</td>
<td>Programming Styles</td>
</tr>
<tr>
<td>6</td>
<td>Procedural approach</td>
</tr>
<tr>
<td>6</td>
<td>Object-oriented approach</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
</tr>
<tr>
<td>6</td>
<td>LIBRARY VS. LANGUAGE</td>
</tr>
<tr>
<td>6</td>
<td>Desired characteristics of final system</td>
</tr>
<tr>
<td>6</td>
<td>Implementation approach</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
</tr>
<tr>
<td>9</td>
<td>SELECTION OF A HOST LANGUAGE</td>
</tr>
<tr>
<td>9</td>
<td>General</td>
</tr>
<tr>
<td>9</td>
<td>Pascal</td>
</tr>
<tr>
<td>11</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>12</td>
<td>ANSI C</td>
</tr>
<tr>
<td>14</td>
<td>C++</td>
</tr>
<tr>
<td>16</td>
<td>Conclusion</td>
</tr>
<tr>
<td>16</td>
<td>GENERAL DESIGN</td>
</tr>
<tr>
<td>4-1</td>
<td>CHAPTER 4 - IMPLEMENTATION</td>
</tr>
<tr>
<td>1</td>
<td>GENERAL</td>
</tr>
<tr>
<td>1</td>
<td>Chapter Objective</td>
</tr>
<tr>
<td>1</td>
<td>oopsSlip User Manual</td>
</tr>
</tbody>
</table>

vii
Operations and Inducement ......................................................... 1
Basis ......................................................................................... 2
FINITE HOMOGENEOUS IMAGE ALGEBRA .................................... 2
   General .................................................................................. 2
   Bound Matrix ......................................................................... 2
   Star ....................................................................................... 3
   Basis operations ................................................................. 3
   Extended operators ............................................................. 5
   Structural operators ............................................................ 7
   Minkowski operators ............................................................ 7
   Behavior of star ..................................................................... 8
CONCLUSION .............................................................................. 8

APPENDIX B - CLASS DECLARATION LISTINGS .................................. B-1
APPENDIX C - IMAGE FUNCTION DICTIONARY .................................. C-1
REFERENCES ............................................................................. Biblio-1
AIM OF THESIS

Aim of thesis. The aim of this thesis is to implement an image processing system that uses the Image as the primary unit of manipulation.

PROBLEM STATEMENT

Background. Image processing algorithms for general-purpose computers are usually designed to manipulate picture elements, or pixels. The need to implement algorithms this way tends, in many cases, to obscure the clarity of the underlying algorithm.

Several mathematical techniques, called image algebras, have been proposed to completely describe image interactions. The noteworthy approaches use the concept of the Image as the fundamental unit of manipulation. The definition of Image may vary between algebras, but is generally meant to encompass the total number of picture elements, or pixels, as a matrix or as some other form.
**Problem.** Few practical implementations have been made of image algebras. A means of implementing an image algebra expression using a dedicated language or environment is needed. Possible uses of such a system would be in research, teaching and prototyping.
CHAPTER 2 - PREVIOUS WORK

GENERAL

Chapter Objective. The aim of this chapter is to describe the work in this field to date.

IMAGE ALGEBRA

Context. Much development in image processing algorithms to date has been done empirically, based on statistical science or simply intuitive judgement. Several current problems in imaging point to the need for a standardized, coherent and complete tool to demonstrate the correctness of algorithms. An algebraic mathematical approach to image processing offers many of these qualities.

The goal of an image algebra is to express all image operations hierarchically from a small collection of proven, well behaved primitive basis operations. Not only does such an algebra provide a sound mathematical framework, but it also lays the foundation for languages and systems whose ability to express image processing algorithms can be thoroughly demonstrated, and that produce correct and robust results.

Origin of image algebra. There exists a number of different image algebras, most based on an approach to set mathematics pioneered by H. Minkowski [Min03], and refined by Georges Matheron [Mat75] and J.P. Serra [Ser81] for texture analysis of geological formations.

The work of Matheron and Serra has been directly applied in an area of image analysis called morphological image processing. The morphological approach to image processing
involves analyzing images with respect to their shapes, not only in their domain, or extent, but also in their grey level (considered a third dimension of the domain). Because of its origins, it benefits from a stricter mathematical formulation than other image processing methodologies, and is being used to overcome the limitations of traditional linear convolution-type operations.

The mathematics of Matheron and Serra were applied in a systematic way to images by two primary groups: Dougherty and Giardina ([DoG85], [DoG87b], [DoG88a]), and Ritter and his team ([RSG85], [RDW87] and [RSW87]). The heterogeneous (many-typed) image algebras developed by these two independent groups require several sorts of data types to fully describe image algorithms.

Further development of the Dougherty/Giardina image algebra resulted in a homogeneous or single-sorted image algebra ([Dou89a] and [Dou89b]). The advantage of this single data type algebra is that it can express any other type of image algebra, or any kind of image processing operation. Appendix A provides a general overview of Homogeneous Image Algebra.

COMPUTER IMPLEMENTATIONS OF IMAGE ALGEBRA

General. A concise algorithm formulation method such as image algebra provides a convenient method for defining a computing environment where it can be used. Three principal groups have been working to implement variations of Matheron and Serra's work: Ritter and his team at the University of Florida [RDW87], Lougheed and Sternberg at the Environmental Research Institute of Michigan [Ste83] and Dougherty at the Morphological Imaging Lab of the Rochester Institute of Technology ([DoS88] and [GDA90]).
**AFATL Image Algebra.** Under the sponsorship of the U.S. Air Force Armament Technical Laboratory (AFATL) and the Defense Advanced Research Projects Agency (DARPA), a research and development team at the University of Florida, Gainesville, headed by Gerhard Ritter, has developed a heterogeneous (many-sorted) image algebra based on images and templates [RSG85]. Templates are a means of relating the domain of the input to the domain of the output. A FORTRAN based language preprocessor enables a user to prototype image processing algorithms developed using AFATL Image Algebra and implement them on a computer.

Although their language is relatively complete, the template operations add a measure of complexity to the system. In the beginning, it was designed to be used on a specialized array processor, although later versions run on a general-purpose computer.

**C-3PL.** Research groups at the University of Michigan and the Environmental Research Institute of Michigan (ERIM), headed by Stanley Sternberg and Robert Lougheed, have developed a parallel pipeline image processor, called a cytocomputer, and a high-level language specifically created for image processing, called C-3PL [Ste83].

The cytocomputer applies sequences of neighborhood transformations to digitized images. Each neighborhood transformation is made in an individual processing element known as a processing stage. The computer is a serial pipeline of stages, where each stage performs a transformation on the entire image.

The image processing language used by the cytocomputer is directly based on the mathematical morphology of Minkowski, Matheron and Serra.
This system was one of the first to deal with images as a whole (as opposed to the pixel level). The image being processed is called the active image, which is modified by probing or combination with other images called structuring elements. The architecture provides for manipulation of both binary images, using silhouette transformations, and grey-scale images, using umbra transformations.

The current version of the cytocomputer is the Cyto-HSS (Cytocomputer-High Speed System) [Lou89].

The advantages of the cytocomputer are that morphological operations are executed very fast. However, since the architecture is unique, the image processing language cannot be used on other systems. It is also intended primarily for morphological image processing.

**SLIP.** Homogeneous Image Algebra was extended into Finite Homogeneous Image Algebra (FHIA) by Dougherty and Giardina [DoG87b], which provided a basis for computer implementation.

FHIA is based on the concept of the *bound matrix* of real values on a two-dimensional Cartesian grid $\mathbb{Z} \times \mathbb{Z}$, $\mathbb{Z}$ being the set of integers. The matrix has dimensions $m \times n$ rows by $n$ columns, and its position in the grid is given by the coordinates row $T$ and column $R$ of the top left-hand element in the matrix. All the values of elements outside the bound matrix are called * (star), or undefined. Undefined values are also allowed inside the bound matrix. A significant feature of FHIA is that any interaction between finite images (bound matrices) can be defined from a set, or basis, of six fundamental operators.  

---

$^1$Further discussion of Finite Homogeneous Image Algebra is given in Appendix A.
The original work to implement FHIA was performed by Paramjit Sehdev at the Stevens Institute of Technology [DoS88], and was called Structured Language for Image Processing (SLIP). It consisted of a library of Pascal and assembly language subroutines that executed on a DEC VAX computer. The objective of the project was to provide a portable means for researchers to experiment with FHIA.

I have worked on and improved this implementation ([DGS89] and [GDA90]), proving the concept of a portable structured imaging environment.

Other implementations. The use of object-oriented languages for image processing and image algebra implementation is beginning to be examined. Peter Marineau and Michael M. Skolnick at the Rensselaer Polytechnic Institute [MSk89] are defining an object-oriented imaging language for mathematical morphology. A commercial parallel processing computer, the AIS-5000 by Applied Intelligence Systems, is also programmed in Objective-C [ScW88]. These approaches do not appear to be based on a specific algebraic means of expressing algorithms.

Michael A. Jenkins and Janice I. Glasgow of Queen's University at Kingston, Ontario have taken a different approach using array theory, the "study of nested, rectangularly-arranged collections of data objects" ([JeG89]) as the basis for their Nested Interactive Array Language (Nial). Nial can be used in several different programming styles (procedural, object-oriented, etc.) [JGM86].
CONCLUSION

Image processing systems based on a complete mathematical formulation have been designed and built (Ritter, Lougheed and Dougherty). Object oriented techniques are being explored for their use in image processing.

To date, there appears to have been no attempts at building an image-as-object oriented image processing system based on a global mathematical foundation such as an image algebra.
Chapter Objective. The objective of this chapter is to describe the requirements, constraints and design choices in implementing an image-as-object image processing system.

Intended audience and uses. The intended users of this system are the students and research staff of the Rochester Institute of Technology. The system will be used for demonstration, prototyping of image processing applications (especially morphological processing), and teaching.

Level of user sophistication. The computer experience of users varies from novice to expert. Many fall into the novice category, and are not familiar with computer internals or data structures. The user will be familiar with at least one of the “fundamental” computer languages (FORTRAN, Pascal, C, etc.). The user will also be familiar with the basic principles of image processing.

Hardware and Software. The equipment available includes Sun, VAX/VMS, UNIX/Ultrix, IBM-PC and Macintosh systems. Languages available include FORTRAN, BASIC, Pascal, C and C++. The final software should be able to interface as much as possible with these systems.
DESIGN QUALITIES

The system must provide an image processing environment that is portable, simple and efficient.

Portability can be achieved by ensuring that the system can function without special hardware or software. This requirement allows the system to be used on any general-purpose computer.

The system should be simple to use and maintain. There is no provision for user training, other than the documentation provided. Simplicity encourages use, and reduces potential sources of errors.

Although efficiency is compromised to some degree by the portability requirement, the system should provide for as timely a response as possible.

CHOICE OF MATHEMATICAL BASIS

Several authors ([CDM88], [JeG89]) have noted the requirement for an image processing language or system to be built on a sound mathematical basis. Using mathematical theory as a starting point for system design ensures a consistent and complete set of operators, provides a mechanism for correctness proofs and eliminates redundant and contradictory constructions, among other advantages.
The primary requirement of a mathematical basis for an image processing system is simplicity. A simple means of expressing interactions between images translates into simpler algorithms and implementations.

To keep the system simple, the number of objects manipulated must be kept to a minimum. Since Finite Homogeneous Image Algebra (FHIA) has only one data type, the image, and the interactions between images are defined in a simple manner, it is a good choice as the basis for an image processing system.

A possible disadvantage of using FHIA is that it contains some fundamental differences compared to more traditional methods. In FHIA, the domains of the image are important, as well as the concept of undefined element, or ∗ (star). These two features cause results generated by an FHIA-based system and a traditional system to differ, although the FHIA result is rigorous mathematically. Traditional algorithms often take "shortcuts" or simplifications even though the results may suffer (for example, division by zero: FHIA returns a ∗, but most return a 0).

The advantages of using FHIA is that a logical means of defining image interactions is developed (for example, what does the addition of two images mean?). This powerful tool in turn leads to a logical definition of higher-level image manipulations (extended operators, filters). This advantage of the use of a defined mathematical model for image interactions such as FHIA outweighs the apparent disadvantage.

Other images algebras such as AFATL Image Algebra, with its images and templates, do not have as straightforward an approach to defining image interactions, as does FHIA. This makes the expression of algorithms more complicated.
**Conclusion.** Finite Homogeneous Image Algebra will be used as the mathematical foundation for this implementation.

**USER INTERFACE**

**General.** Image processing algorithms can be specified in many ways. Two primary means of expression have been developed for describing Image Algebra image manipulations: block diagrams and mathematical expressions.

**Block diagrams.** Block diagrams, suggested by Dougherty and Giardina [DoG87a], are a graphical means of programming in which each function is represented by a "black box" with a specified number of inputs, outputs and a defined action. Figure 3-1 gives an example of a block diagram.

![Block diagram example](image)

Fig 3-1 Example of block diagram (dilation)
The block diagram conventions are similar to that of a general data-flow diagram. The main difference is in the means of displaying how an action is implemented over the domain and range of an image.

The **domain** box is connected to all the **tran** boxes, meaning that the elements of the domain structure are fed one at a time to the input of the **tran** procedure. The highest **tran** box receives the first element, the second gets the second element, and so on (represented by the three vertical dots) until the last element. The action of the **range** box can be similarly interpreted. The results from the **tran** and **add** functions are combined together using the **extmax** function.

Each of the functions are "defined" by lower levels of block diagrams, until, at the lowest level, only the FHIA basis operations (image add, image multiply, etc) are used.

Using the block diagram, the independent and parallel nature of the loop contents stands out, which is the major advantage of this visual tool. Other advantages of block diagrams are its visual representation of data flow and the abstraction of the implementation from the design.

A graphical user interface is required for the use of block diagrams as a programming method. Sophisticated hardware and software would be required for direct entry of a diagram. In addition, the user may not design the most efficient method of implementing an algorithm. A visual language editor and compiler/optimizer is therefore required for this approach.

Although block diagrams have been used to some degree of success in illustrating complex imaging algorithms, a formally defined syntax has not been developed, rendering
computer implementation difficult. For example, looping and control mechanisms have not been consistently defined.

**Alphanumeric expressions.** Since FHIA consists of mathematical expressions concerning images, such as \( C = A + B \), where \( A \), \( B \) and \( C \) are images, it can translate well into an alphanumeric means of programming. Such an approach could be simpler to implement than using block diagrams, because special user interface hardware would not be required.

Since the syntax of describing image interactions for an alphanumerically-based system would follow more directly from the image algebra, this approach would require creation of an image algebra library using a common programming language, or of a special-purpose programming language.

**Conclusion.** An alphanumeric user interface will be used to implement image algebra. Block diagrams can however be used as a supplementary documentation tool.

**IMPLEMENTATION STYLES**

**Programming Styles.** There are two main styles in which this system can be implemented: the object-oriented approach and the procedural (structured) approach. The choice of style will influence the architecture of the software system.

**Procedural approach.** The procedural approach is the traditional "structured" programming style. General-purpose languages such as FORTRAN, Pascal and C support this style, which is characterized by the dividing of algorithms into procedures and functions, with data fed into a procedure and a result returned. Data hiding, and the grouping of a number of
procedures that operate on data with that data (modularity) is possible to a certain degree using procedural languages.

The advantage of this approach is that it is the one most widely used at the moment, and the one most familiar to the potential users of the system. Traditional general-purpose computer architectures and operating systems are also "tuned" to this style of programming [Str88].

**Object-oriented approach.** "Object-oriented" refers to environments and programming techniques that support data abstraction, data hiding, procedural modularity and concurrence [Str88a].

Pure object-oriented systems consist of groupings of "virtual devices", called objects, that can "communicate" with each other. Objects encapsulate conventional data structures and the routines needed to manipulate the data. Objects are organized into classes, or data types of a similar function. Classes can further be defined into subclasses. The "actualization" of an object is called an instance.

Each object is self-contained, and cannot directly access the data of other objects. This restricted access is one of the major advantages of the object-oriented paradigm, in that the implementation details of the object are hidden, and the permitted actions with the data are well-defined and strictly controlled. In this system, defining images as objects would enforce the FHIA concept as the image as the fundamental unit of manipulation, and prevent the user from dealing directly with pixels.

The concept of "message passing", used in such object-oriented languages as Smalltalk or Lisp variations, does not quite apply here, especially since most image operations are
Mathematical in nature, and several involve two images (such as filtering, dilation) to produce a third image as result.

The most severe disadvantage of this approach is the intended user's non-familiarity with object-oriented paradigms (at this time). Since the target users have different computing backgrounds and skills (with some having no formal introduction to programming, structured or otherwise), use of this approach could compromise one of the basic design considerations, simplicity.

Another drawback is the difficulty of implementing an object-oriented approach with a portable language. The closest to a "portable" language is C++ [Str88b]. It can allow routines written in ANSI C, since the latter is a subset of C++. Specialist object-oriented programming languages such as Smalltalk would compromise portability even further.

Many object-oriented tools and techniques available at the moment are better suited to applications such as model building, computer-based simulations and expert system knowledge representation [Kos88]. The image processing applications targeted by this system are more of a mathematical nature, and lend themselves more efficiently to traditional data-only structures. However, the possibilities of treating images as objects does make for simpler program design.

**Conclusion.** The ideal implementation methodology would be a combination of both the object-oriented approach, for its data hiding and data-based modularity, and the procedural approach, because of its familiarity to the user. It is important that the user be able to choose which approach to take, for maximum flexibility according to the situation.
LIBRARY VS. LANGUAGE

Desired characteristics of final system. To implement Finite Homogeneous Image Algebra, the final system needs the following characteristics:

a. Structures: Since the image is a structure of data (the matrix, and its location in space, at a minimum), a mechanism for referring to the whole grouping of data with one reference is required. This simplifies the code, increases readability and forces the user to refer to the data at a higher level;

b. Procedural Modularity: A means of grouping procedures into modules, either on the basis of their functions ("structured programming") or their relationship to the data ("object-oriented programming") is needed to organize the large number of functions in the library into a manageable system;

c. Equational expression: Since image algebra is mathematical, functions should be implemented as equations (such as \( C = A + B \); where \( A, B \) and \( C \) are images).

Implementation approach. There are two possibilities to implement FHIA: complete design of a new, FHIA-based language; and building of a library and tools using an existing language.

The first alternative would require a full syntactical analysis, as well as construction of a parser and compiler. This can be quite complicated, expensive, and may not meet the desired system qualities, especially simplicity to the user, since they would have to learn another language. Portability can be another obstacle to the successful use of a new language. The advantage of this approach is that the language could incorporate the "image processor's way of thinking", as some imaging constructs ("repeat over entire image domain") translate awkwardly into existing programming languages.
The second alternative of providing a library of constructs based on an existing language could be simpler for the user to learn and cheaper to implement. Some languages provide tools to implement most or all of the required characteristics for the system. The disadvantage is that not all tools may be available or can not be constructed.

**Conclusion.** The second approach, that of building an imaging library, is the one selected for this study, mainly because it is the simpler approach.

**SELECTION OF A HOST LANGUAGE**

**General.** Four computer languages were examined for their suitability to host a FHIA library: Pascal, FORTRAN, C and C++. This section describes the advantages and disadvantages of each.

**Pascal.** The first implementation of FHIA, called SLIP (Structured Language for Image Processing) was done in Pascal [DoS88]. This implementation did not use structures, and therefore the matrix, its dimensions and its location were passed with every procedure call, resulting in statements such as:

```
Add ( AR, AT, MA, NA, A, 
     BR, BT, MB, NB, B, 
     BR, BT, MB, NB, B );
```

The problem with this approach is that the algorithm flow can be obscured by having so many arguments "cluttering up" each procedure call. The second incarnation of SLIP [DGu89] used structures to refer to images, but declared all images as the same fixed size.
Use of the record type definition allows the bound matrix and \( \{T, R, M, N\} \) values to be passed around as one unit. Pascal also allows variable matrix sizes, by using pointers and the new and dispose procedures to allocate and deallocate memory.

The disadvantage of Pascal is the calling convention. A statement such as

\[
\text{Add}(A, B, B);
\]

where \( A \) and \( B \) are images, raises the question, which image is input, and which is output? In this case, a consistent procedure design could designate the last argument to always be the output, but without such a policy (or correct documentation), errors can be made.

If one wants to use variable-sized matrices for maximum use of memory, the onus is on the user to designate that variable names refer to pointers to image structures. The above call would then look like:

```pascal
var
    A, B: "Image;
begin
    (...) 
    Add(A, B, B);
    (...) 
end.
```

Failure to include the pointer symbol `^` in the declaration would result in an error. The syntax could be improved by defining \texttt{Add} as a function returning a pointer. The call then becomes:

\[
B = \text{Add}(A, B);
\]

which is much closer to the desired format.

The main disadvantages of Pascal are memory management and portability. The latter problem refers to the fact that no standard means of grouping procedures into modules has been defined. This is illustrated by the experience of porting code written in Macintosh Turbo Pascal.
(version 1.00A) to VAX Pascal. At each transfer (required to use image display routines on the VAX), the module headers had to be rewritten, from the Turbo to the VAX style:

Turbo Pascal module header:  unit MODULENAME(moduleid)
VAX Pascal module header:  module MODULENAME

In addition, VAX Pascal procedures that can be called from outside the module must have the designation [global].

The memory management issue is another main problem when using Pascal. The onus is on the user to ensure images are allocated with new before use, and deallocated with dispose when finished. Failure to do so makes the available memory fill up quickly, and causes the procedure to crash. By not using pointers, and declaring arrays to be of a fixed maximal size, memory allocation and deallocation is more automated (although it is a waste of memory to use a 64x64 element array to contain a 3x3 matrix).

FORTRAN. Standard FORTRAN-77 has two major drawbacks: lack of a structure declaration mechanism, and a lack of pointers.

Because there is no structure declaration mechanism, reference to a procedure must be in the form:

```plaintext
CALL ADD( AR, AT, MA, NA, A,
           BR, BT, MB, NB, B,
           BR, BT, MB, NB, B);
```

Functions cannot be used to return images in FORTRAN, because the return type of a function must be one of the predefined data types. Therefore, an “equational” expression cannot be used when dealing with images.
VAX FORTRAN does contain a structure mechanism (STRUCTURE) which greatly simplifies implementation. For example, the above code fragment would look like:

```
STRUCTURE /IMAGE/
  UNION
    MAP
      INTEGER*2 R   ! Upper left-hand corner coord (column)
      INTEGER*2 T   ! Upper left-hand corner coord (row)
      INTEGER*2 M   ! Number of rows in image
      INTEGER*2 N   ! Number of columns in image
      INTEGER*2 PIXEL(1:128,1:128) ! Image elements
    END MAP
  END UNION
END STRUCTURE

(....)
IMAGE A, B
(....)

CALL ADD(A, B, B)
(....)
```

The STRUCTURE statement is not yet part of the standard FORTRAN.

The other disadvantage of standard FORTRAN is the lack of a pointer mechanism. This means that images must be declared to be the maximum allowable size, before compile time, resulting in either a waste of space, or a program abort due to a lack of space.

**ANSI C.** ANSI C is a standard definition of C that allows structure declarations, pointers, and a standard means of grouping procedures into modules. A major advantage of this version of C is its portability.

Structures are declared by using the `struct` specifier. The resulting structures can be then referred to by pointers.
Pointers can be used to provide arrays whose size is not known at compile time. The malloc and free functions are called to allocate and deallocate memory, however this creates the same problem as mentioned in Pascal above: the onus is again on the user to decide when an image is needed. Failure to properly initialize and deallocate an image could have disastrous consequences. Another problem with allocation and deallocation is that these extraneous statements “clutter up” the implementation.

Another problem with pointers, also present in Pascal, is how to address elements pointed to by an address which is itself an element of a structure. For example, a C implementation of an image structure is:

```c
typedef int pix;  /* PIXEL TYPE */
typedef struct image /* IMAGE STRUCTURE */
{
    int R;          /* col no of top left-hand corner*/
    int T;          /* row number of top lh corner */
    int M;          /* number of rows */
    int N;          /* number of columns */
    pix *pixel;     /* pixel values */
} image;
```

To find out the size of an image, the following expression is required:

```c

{  
    image* A;
    int nbr_of_rows, nbr_of_cols;

    nbr_of_rows = A->M;
    nbr_of_cols = A->N;
(...)
}
```

Accessing pixels directly is more complicated:

```c

pixell_1 = *(A->pixel + (rownum) + (colnum) );
```
The subscript operator [] cannot be used because the element array was not declared with a row length dimension: the compiler does not know how long a row is. Macros or functions can be designed to make access to image elements more natural, especially to novice programmers.

The use of header files and function prototypes provide a convenient means of grouping procedures into modules. Functions can also be implemented to return pointers to images, so "equational" procedure formats can be used. The application examples show several cases of this type of expression.

One problem with functions remains: they are still an unnatural way of formulating an arithmetical expression. If A and B are integers, is it not preferable to write "B = A+B;" instead of "B=add(A,B);"?

A similar argument can be made for images. Defining operators for image algebra can be done on a UNIX system, by defining a syntax, and writing a preprocessor using utilities such as lex and yacc, but the implementation can be quite complicated.

Another problem with function calls (also present in Pascal and FORTRAN), is that "natural" function names such as open, close, abs, and max cannot be redefined when applied to images. This increases the probability of user errors.

**C++.** C++ is a superset of the C language that supports controlled access to structures ("classes"). Another feature of C++ that applies to this implementation is the use of constructors and destructors.

A class is a data structure that contains within it the relevant functions. In the case of a class library, the image classes contain the structure mentioned above in C, as well as the
prototype declarations for all functions that can access the class. This meets the modularity requirement for a host language, in that procedures are organized as a function of the data (class) on which they operate.

A feature of classes is the ability to control which parts of the class can be accessible from the "outside". In a class library, the data elements are kept "private", and are not directly accessible from instances other than itself or those declared "friend". Functions can be made to insulate the user from the data internals (setImageLoc, showImageLoc). This ability to force the user to consider the data structure as a whole is another of the desired attributes of a host language.

With classes, operators (+, *, =) can be "overloaded", or redefined, to have meaning when applied to new classes. This capability allows the statement:

\[ B = A + B; \]

to mean "add image B to image A and return the result in B". This ability to use equational expressions is another desired attribute of a host language. Functions such as open and max can also be overloaded to have their image algebra meaning when applied to images, and still keep their original definition when applied to file streams (open) or numbers (max).

A major advantage of C++, when compared to C, Pascal and FORTRAN, is the capability to define constructors and destructors to allocate memory automatically when a variable is declared, and to mark the allocated memory for disposal when the variable is no longer needed. This frees the user from most of the responsibility for memory management (although the user must be careful not to refer to the variable outside the scope in which it was declared, since it no longer exists). It also has the important secondary effect of removing statements from the implementation that do not have a direct relationship with the algorithm.
At the present time, the major disadvantage of C++ is its portability. There are two slightly incompatible versions that exist of C++: version 1.2 ([Str86]) and version 2.0 ([Gut89], [Lip89]). Upward migration of the class library from a version 1.2 compiler to a version 2.0 compiler should not have an adverse effect on the library, although this has not been tested because a version 2.0 compiler was not available.

In addition, only a certain number of operators can be overloaded in C++. This is a problem because, for example, the often used operator `extadd`, used as:

\[ B = \text{extadd}(A, B); \]

could be more clearly expressed as:

\[ B = A <+> B \]

**Conclusion.** C++ appears to be the best choice to meet the requirements of hosting a FHIA library, having a well-controlled structure mechanism, a means to organize modules and a means to express image algebra equations in their natural mathematical form. Its resemblance to C will make its use somewhat easier.

**GENERAL DESIGN**

This system will use an alphanumeric user interface, based on Finite Homogeneous Image Algebra, to implement a class library written in C++.

The general design of the system will consist of:

a. a **class library** written in C++, defining the image and the functions needed to operate on it; and

b. a **preprocessor**, to interpret special operators not possible in C++.
The C++ class library will be implemented in a manner that, when appropriate, can allow both an object-oriented function call style:

```cpp
    a.extend(b);
```

and a more traditional function call style:

```cpp
    extend(a, b);
```

This will allow more flexibility from the user's point of view.
CHAPTER 4 - IMPLEMENTATION

GENERAL

Chapter Objective. The elements needed to implement Finite Homogeneous Image Algebra have been defined in a class library called the Object Oriented Programming System Standard Library for Image Processing (oopsSlip). The objective of this chapter is to describe the features and implementation of the class library.

oopsSlip User Manual. This chapter and Appendix C (Class Function Dictionary) are intended to be used together and constitute the user manual for the system.

Examples of the use of the class library are given in Chapter 5.

Notation Conventions. To aid in explaining the implementation, the following notation conventions are used:

- **Courier font** denotes predefined keywords, identifiers, or other computer-related objects, used directly ("literally") in the system;
- **IMAGE** or **IMAGE** is a generic image type, any of the predefined image classes;
- **PIX** or **PIX** is a generic pixel type, corresponding to any of the legal element classes
FEATURES

General. The system consists of the oopsSlip class library and a preprocessor.

oopsSlip Class Library. The oopsSlip class library is a set of three groups of classes:

a. type classes (Coord, Byte), which are derived from the C++ fundamental types;

b. image classes (ByteImage, IntImage, RealImage, ComplexImage), which are images that have all the same functions and methods, but are composed of one of the fundamental or derived types; and

c. image structure classes (Domain, ByteRange, IntRange, RealRange and ComplexRange), which are classes derived from the image classes.

Preprocessor. Six additional operators are added to the normal overloadable C++ operator set. This is to accommodate the extended operators (extadd, extmult, extmax, and extmin) and basic morphological operators (dilate and erode). A preprocessor decodes these special symbols into function calls, then compiles and links the user program.

DERIVED TYPE CLASSES

General. C++ contains fundamental data types such as int, float, and double. To accommodate the image processing algorithms in the oopsSlip class library, two other type classes are defined: Byte and Coord.

Byte. The Byte is an integer type that takes up one byte of space. It can have values from 0 (zero) to 255. It is equivalent to using the type char for integer operations, and is implemented by a macro substitution. This type is defined for ease of use only.
**Coord.** This is used to denote the coordinates of an object, and is a pair of int values.

The convention for referring to a coordinate is (row, column). The only operators defined for Coord are <, == and !=. The method to refer to coordinate components is (here is of type Coord):

- rowComponent = here.row();
- columnComponent = here.col();

or
- rowComponent = row(here);
- columnComponent = col(here);

The brackets in the expressions are important because they refer to member functions of the class Coord. This will be further explained in the section on Function Calling Syntax.

**IMAGE CLASSES**

**Abstract data type.** The "generic" image class is an implementation of the concept of bound matrix, an M row by N column matrix, with the top left-hand corner element at row T and column R in a universal Cartesian grid, as illustrated in figure 4-1. The location of an element is referred to as a pixel, and its value, the grey value.

The term image header refers to the set of image coordinates and dimensions [T, R, M, N].
Frames of reference. The Cartesian grid is used to position images, regions within an image, and image elements in a universal frame of reference, called the grid coordinates. Grid coordinates are used because relative position of image regions can be important in FHIA. A frame of reference local to the image, image coordinates, locates elements by their (row, column) coordinates relative to the top left hand element in the image matrix (being element (0,0)). This reference is used for manipulations within an image.

Element values. Elements of the bound matrix are either defined or undefined. Defined elements can have any value according to the data type being used (double, int, Byte). Undefined values can be anywhere in the image, and can represent either noise, corrupted data, or simply "filler" to make an image that does not have a rectangular domain (like an

Fig 4-1 Coordinates of image overlaid on Cartesian grid
image from a periscope, which can be round) fit the rectangular bound image matrix data type. The undefined element value is given by the constant STAR.

**Image types.** The elements of the bound matrix can be composed of one of the fundamental or derived types. The designations and corresponding types are:

- **IntImage** corresponds to int;
- **RealImage** corresponds to double;
- **ByteImage** corresponds to Byte;
- **ComplexImage** corresponds to complex.

All image types have the same functions and operator definitions.

**Null images.** The null image is an image with 0 (zero) rows and 0 (zero) columns. It can be located anywhere in the grid. It is denoted generically by nullIMAGE, which corresponds to the following values for the following types:

- **nullIntImage** is used with an IntImage;
- **nullRealImage** is used with a RealImage;
- **nullByteImage** is used with a ByteImage;
- **nullComplexImage** is used with a ComplexImage.

**IMAGE CLASS OPERATORS**

**Overloaded C++ operators.** Several C++ operators are given specific meanings with respect to images. There are two types of operators: binary (operating on two images) and unary (operating only on one image).
**Unary operators.** Expressions with unary operators group right-to-left:

\[
\text{unary-expression:} \\
\text{unary-operator expression}
\]

\[
\text{unary-operator: one of} \\
! \\
\]

**180 degree rotation (\(\cdot\)).** The result of the unary - operator on an image is that the image is rotated 180 degrees about the origin (0,0) of the global Cartesian grid.

**Complement (\(!\)).** The result of the unary ! operator on an image is that every defined pixel becomes undefined (= \text{STAR}), and every undefined pixel is given the value 1. All elements outside the resulting image are considered to have value 1.

**Binary operators.** The binary operators, in decreasing precedence, are:

* multiply two images or an image and a scalar
/ divide two images or an image by a scalar (division by zero gives \text{STAR})
% zero divide 2 images or an image by a scalar (division by zero gives zero)
+
+ add two images or an image and a scalar
 subtract two images or a scalar from an image
< is one image smaller than the other
> is one image larger than the other
== is one image equal to the other
! = is one image not equal to the other
<= is one image smaller or equal than the other
>= is one image larger or equal than the other

**Multiplicative Operators.** The multiplicative operators *, / and % group left-to-right:

\[
\text{multiplicative-expression:} \\
\text{expression * expression} \\
\text{expression / expression} \\
\text{expression % expression}
\]
**Multiplication (×).** The × (image multiply) operator performs a multiplication between corresponding defined elements of the intersecting parts of the operand images. The domain of the result is the intersection of the two images. If the images do not intersect, the result is a null image. If one or the other or both of the corresponding elements is undefined, the corresponding element in the result is undefined. Multiplication of an image by a scalar is defined as the multiplication of each defined element of the operand image by that scalar. Multiplication is commutative.

**Division (/).** The / (image divide) operator performs a division of the first operand image by the second on the intersecting domain of the images. If one or the other or both of the corresponding elements on the intersection are undefined, the result is undefined. If the corresponding element in the divisor is 0 (zero) then the result is undefined. If the images do not intersect, the result is a null image. Division of an image by a scalar is defined as the division of each defined element of the image by that scalar. Division by 0 (zero) sets the pixel to STAR.

**Zero-division (÷).** The ÷ (image zero-divide) operator performs a "zero-division", in that if the corresponding element in the divisor is a 0 (zero), the resulting pixel is set to 0 (zero). If the images do not intersect, the result is a null image. Zero-division of an image by a scalar is defined in an analogous manner to division.

**Additive Operators.** The additive operators + and - group left-to-right:

\[
\text{additive-expression:} \\
\text{expression} + \text{expression} \\
\text{expression} - \text{expression}
\]

**Addition (+).** The domain of the result of the + (image add) operator on images is the intersection of the domains of each of the operand images. The resulting grey values are the sum of the corresponding elements of each operand, if they are both defined, and STAR if one,
or both are undefined. If one or both of the operands is a null image, the result is a null image.

Addition of an image to a scalar is defined as the addition of the scalar to every defined element of the image. Addition is commutative.

**Subtraction (−).** Image Subtraction is defined in an manner analogous to addition.

**Relational Operators.** The relational operators group in the same manner as their C++ counterparts:

```
relational-expression:
  image-expression < image-expression
  image-expression > image-expression
  image-expression <= image-expression
  image-expression >= image-expression
```

The operators < (smaller), > (larger), <= (smaller or the same), >= (larger or the same) relate the extent of the domain (or size) of the images. The extent of the domain of an image, or its **cardinality**, is the number of defined elements in the image. The relational operators yield the constant **FALSE** (=0) if the specified relation is false, and **TRUE** (=1) if it is true. A null image has a cardinality of 0.

**Equality Operators.** The equality operators test whether an image is the same as another:

```
equality-expression:
  image-expression == image-expression
  image-expression != image-expression
```

The == (equal to) operator returns **TRUE** (=1) if the image header and image matrices are identical. The != (not equal) operator is the converse of the == operator.
Additional Image Class Operators (Extended Operators). Six additional (non C++) operators are provided in order to implement the extended operators and the morphological functions dilate and erode. In order of decreasing precedence, they are:

[+]  dilate
[-]  erode
<@>  extmax (extended maximum)
<^>  extmin (extended minimum)
<*>  extmult (extended multiplication)
<++> extadd (extended addition)

These operators have the highest precedence, so that they are evaluated first.

Evaluation order is from left to right. The syntax is:

```
extended-arithmetic-expression:
   expression <+> expression
   expression <*> expression
   expression <^> expression
   expression <+> expression
```

The action of the extended arithmetic operators is analogous to the normal arithmetic operators on the intersection of the operand images. This intermediate result is extended by unioning the remaining domain of the second operand to the result, then unioning the remaining domain of the first operand to the result. If the domains do not intersect, the result is the union of the domains. If one or the other image is null then the result is the domain of the non-null image. If both operands are null, the result is a null image. Extended operations between images and scalars are not defined. Extended operators are not commutative.

Since these are not valid C++ operators, any program containing these symbols must first be passed through a preprocessor. This step consists of recognizing the symbols and replacing them with equivalent function calls.
The advantage of using these symbols instead of a function call is that these are common image operators, and are used at least as often as the image arithmetic operators such as +, or *. It can also make an equational expression more readable:

\[
\begin{align*}
    h &= a \cdot (f \leftrightarrow g); \\
    h &= \text{erode}(a, \text{extadd}(f,g));
\end{align*}
\]

are equivalent statements. The image arithmetic is highlighted when operators are used.

**IMAGE CLASS FUNCTIONS**

**Organization.** The power of the oopsSlip library resides in the set of more than 100 image processing functions available in the Image classes. These routines can be subdivided into groups according to their functions.

The following discussion is an overview of the principal image functions. A detailed list of the function prototypes is given in the header file listings of Appendix B, and each is described in the Function Dictionary of the user manual at Appendix C.

**Constructors and destructors.** The image classes have constructors and destructors that allocate and deallocate space for the image. The constructor is called automatically in C++ when a declaration is made:

- `IntImage` \( \text{im1} \);
- `RealImage` \( \text{im2}(0,1,2,3,4.1) \);
- `ComplexImage` \( \text{im3}(10,10,2,2) \);
The constructor statement consists of the *typename*, *variable name*, and a group of optional *initialization arguments*, in parenthesis. The order of the initialization arguments and default values if not specified are:

- **T**: row number of top left hand corner of image (default 0);
- **R**: column number of top left hand corner of image (default 0);
- **M**: number of rows in the image (default 0) - row numbers go from 0 to M-1;
- **N**: number of columns in the image (default 0) - columns go from 0 to N-1;

The constructor function is also called when assigning one image to another during declaration:

```c
IntImage im4 = im1;
```

Other types of initialization arguments, using a second image (here called `other`), are:

```c
RealImage name(other, 0);
```

sets name's header the same as other, but with all *defined* elements = 0

```c
RealImage name(otherLoc, M, N, 0);
```

sets name to same location as other (the coordinate value `otherLoc` contains the location of other), but with dimensions M by N, and all elements = 0.

Variables can be declared at any point in a program, but they must be declared before they are used, and are automatically deallocated when the variable passes out of scope.

Destructors are automatically called by the compiler, so they are not to be called by the user.

**Self management functions.** Access to elements of the image (*T, R, M, N, element values and locations*) is performed through management functions:
area shows the total number of defined and undefined elements
card shows the number of defined elements
elem shows the element value at a certain location
setElem sets the element value at a certain location to a certain value
setAll sets all the elements in the image to a certain value
loc shows the image location \((T, R)\)
setImageLoc sets the image location \((T, R)\)
numCols shows the number of columns \((N)\)
numRows shows the number of rows \((M)\)
nextElemLoc shows the location of the next defined element in the image
nextElemVal shows the value of the next defined element in the image

Other management functions are:

- print: print the image as a matrix to the standard output
- writeImage: writes the image to a file stream
- readImage: reads the image from a file stream

**Basis functions.** Six operators are defined as the FHIA basis: image addition (+), image multiplication (*), image division (/), extended maximum (extmax), ninety degree rotation (ninety) and basis translation (tran). Appendix A describes the formal definition of each of these functions.

**Inquiry functions.** These operators return a boolean value TRUE or FALSE about certain aspects of the image:

- isCompImage checks if the image is a complemented image
- isNullImage checks if the image is null or empty
- isDomainEqual compares the domains of two images
- isHeaderEqual compares the headers \((T, R, M, N)\) of two images
- isCardEqual compares the cardinality of two images
**Arithmetic functions.** The arithmetic operators +, -, *, /, and % are defined to operate between two images, or an image and a scalar. If a scalar is one of the arguments, it is transformed to an image of equal domain as the other argument, but with all image elements equal to the scalar. In this way, the result of an image operated on by a scalar results in an image of same domain as the input image.

Other arithmetic functions that operate between two images:

- **max** find the maximum value between corresponding elements
- **min** find the minimum value between corresponding elements

There are also arithmetic functions that operate only on one image:

- **sub** additive inverse (negate all defined elements)
- **abs** absolute value all defined elements
- **comp** complement the image
- **sqr** square all defined elements
- **sqrt** find the square root of all defined elements
- **recip** reciprocal of all defined elements
- **sin, cos, tan** trigonometric functions of all defined elements

**Structural functions.** These functions perform manipulations on the domain of the image:

- **flip** flip the image over the origin
- **window** restrict the image to a certain window
- **transpose** transpose the rows and columns
- **minbound** find the minimal bound matrix
- **intersect** find the intersecting domain between two images
- **ninety2** rotate image 180 degrees about the origin
- **ninety3** rotate image 270 degrees about the origin

**Extended functions.** A set of extended functions are provided:

- **extend** image extension
- **extadd** extended addition
extmult  extended multiplication
extmin  extended minimum
extmult  extended maximum

**Morphological functions.** The morphological functions `dilate`, `erode`, `open`, `close`, and `minksub` (Minkowski subtraction) are implemented. Binary operations can also be performed on grey scale images by using `b_dilate`, `b_erode`, `b_open`, and `b_close`.

**Statistical functions.** Statistics on the image can be obtained with the following functions:

- `avg`  find the average pixel value
- `med`  find the median pixel value
- `max`  find the maximal pixel value
- `min`  find the minimum pixel value
- `sum`  find the sum of the pixel values
- `ssq`  calculate the sum of squares

These routines return a value typed according to the type of the input image (for example, `max` on an `IntImage` returns an `int`).

**Thresholding functions.** A set of thresholding functions provides flexibility in algorithm design: `thresh`, `threshAbove`, `threshBelow`, `threshBelowEqual`, `threshBetween`, `threshDefined`.

**Higher-level functions.** There are many other routines that perform higher level image manipulations, such as the Fourier transform, `filter`, and `hist`. The implementation of some higher level functions is discussed in Chapter 5.
**FUNCTION CALLING SYNTAX**

**Member function invocation.** The object-oriented paradigm assumes that objects act on messages. This mechanism is simulated in C++ by invoking member functions (functions grouped with a given object) using the syntax

```
instanceName.functionName(arguments);
```

where:
- `instanceName` is the name of an instance of the object
- `functionName` is the name of a member function of that instance
- `arguments` are the arguments passed to the function

For example, the class `RealImage` contains the member functions `ninety()`, `thresh()` and `extend()`. If the instance `iml` of `RealImage` is created, then these member functions are called using the syntax:

```
RealImage      iml;
RealImage      im2;
double         tVal;
(...)

iml.ninety();  /* "iml, rotate by ninety degrees" */
iml.thresh(tVal);  /* "iml, threshold yourself at tVal" */
iml.extend(im2);   /* "iml, extend yourself by im2" */
```

This form of function invocation can also be used inside expressions, such as:

```
for (int i = 0; i < iml.area(); i++) {};
```

which cycles through all the elements, defined and undefined, of image `iml`. 
Functions can be "chained", for example the function ninety2() is equivalent to chaining the ninety() function:

```cpp
iml.ninety2(); // "iml, rotate by 180 degrees" */
iml.ninety().ninety(); /* "iml, rotate by 90 degrees, then..."*/
/* ...rotate by 90 degrees again" */
```

The arguments in the function invocation can be any data type or another object (such as another image).

Friend function invocation. The more familiar C style of function invocation is simulated by defining friend functions, where the argument objects have access to each other's member functions. Using this syntax, the above examples would read:

```cpp
RealImage im1;
RealImage im2;
double tVal;
(...)
ninety(im1);  /* "rotate im1 by ninety degrees" */
thresh(im1, tVal);  /* "threshold im1 at tVal" */
im1 = extend(im1, im2); /* "extend im1 by im2" */
for (int i = 0; i < area(im1); i++) {;}
```

IMAGE STRUCTURE CLASSES

Domain class. The Domain class contains an ordered list of coordinates of the defined elements in an image. The image traversal order is from left to right (increasing grid column number), top to bottom (decreasing grid row number).

The utility of the Domain class is superseded by the nextElemLoc() and showElemLoc() and other functions, so there is usually no requirement to extract the Domain.
separately. A comparison of the use of this class is illustrated in the implementation of grey-scale
dilation, given in Chapter 5 (Application Examples). This example shows that extracting the
Domain in its entirety before use wastes memory, processing time, clutters the algorithm and
adds another data type to the system that does not necessarily add to the functionality. The
equivalent functions adhere better to the object-oriented paradigm that the image contains all "it
needs to know about itself". It is also the reason for selecting FHIA as a theoretical basis: only one
data type is necessary to perform all image computations.

The Domain class is included in this implementation as for compatibility with image
processing algorithms described in [DoG87a] and other sources.

The only routines available for the Domain class is:

```
showDomElem               show a particular Domain element
```

The domain is extracted using the `extractDomain()` function of the image, and its size
is given by finding the number of defined elements (card) of the image.

Range classes. The Range classes are an ordered list of the defined elements of the
image. Order of image traversal is the same as for Domain, so the i-th element of a Range class
corresponds to the i-th element of the Domain class extracted from the same image.

There is a Range class corresponding to each type of Image:

```
IntRange corresponds to IntImage;
RealRange corresponds to RealImage;
ByteRange corresponds to ByteImage;
ComplexRange corresponds to ComplexImage.
```
The utility of the Range classes has also been superseded by the implementation of the Image class, for the same reasons as the Domain class above. The implementation of dilation in chapter 5 illustrates this. Note that neither the Domain or Range classes were used in any implementation in the oopsSlip class library.

The only operator defined for the Range classes is:

\[
\text{showRngElem} \quad \text{show a particular Range element}
\]

The domain is extracted using the extractRange() function of the image, and its size is given by finding the number of defined elements (card) of the image.

FOR_DOMAIN_OF CONSTRUCT

**Purpose.** The purpose of the for_domain_of() construct is to implement the image processing paradigm of "processing over the domain of an image". For example, in convolution, one image is translated over another by doing a tran of the origin of the first image over every defined pixel of the second. At each tran, an operation is performed.

**Syntax.** The construct syntax resembles the for() do/ endfor of FORTRAN:

\[
\text{for\_domain\_of(img) \{ }
\quad \text{statement}
\} \text{ end\_for(img);}
\]

The identifier img is the name of the image over which the translations take place.
The construct is implemented as the macro:

\[
\text{Coord here} = \text{img.nextElemLoc()}; \\
\text{for (int i=0; i<\text{img.card(); i++}, here=\text{img.nextElemLoc(here)} )}
\]

Inside the for_domain_of() loop, there are two variables that are automatically defined:

Here is the grid coordinates of the defined element presently being operated on;
\(i\) is a counter from 0 to the number of defined elements.

The scope of these variables is local to the loop, so the constructs can be nested (this is the reason for the \"\{ \" before the declaration of here.

The construct must be terminated with either the end_for() statement (img is optional), or a second \"\}\".

For example, to set all the defined elements of an image to their negative value (the implementation of inv):

\[
\text{for_domain_of(*this) } \{ \\
\quad \text{setElem(here, \cdot elem(here));} \\
\} \text{ end_for();}
\]

Further examples of the construct are shown in Chapter 5.
ERROR HANDLING

Preconditions and Postconditions. Most routines in the oopsSlip class library are defined with a set of conditions that the data must meet before the action is taken (preconditions), and guarantee that the data is in a given state after execution (postconditions). These conditions are listed in the Function Dictionary.

It is the responsibility of the calling routine to ensure that the data meets the listed preconditions. Failure of the data to meet these conditions can cause erroneous output conditions (that may or may not be caught by the receiving function), or simply halt execution.

Error flag errno. Procedure errors are signalled by setting the global variable errno to a certain value. The value errno = 0 means that the execution was successful. The function checkErrno() translates the value of errno into its equivalent message. The list of values that errno can take is given in the header file error.h. The calling routine must check errno for possible error conditions.

SOURCE FILE ORGANIZATION

General. The source files are organized on the basis of their operand data types and their function.

Header Files. The header file oopsSlip.h must be included in every program using the oopsSlip library. It contains certain global declarations, and in turn includes the following header files:
error.h           Error routines header file
Byte.h            Byte class header file
Coord.h           Coord class header file
Domain.h          Domain class header file
Mathrout.h         Overloaded math routines header file

iImage.h, rImage.h, bImage.h, cImage.h
    Class declaration files for the IntImage, RealImage,
    ByteImage, and ComplexImage classes;

iRange.h, rRange.h, bRange.h, cRange.h
    Class declaration files for the IntRange, RealRange,
    ByteRange, and ComplexRange classes;

. **Source files.** The source files are organized by function and data type.

Image functions:

- Arith.C          Arithmetic functions
- Extend.C         Extended operators
- Fileops.C        File and input/output operators
- Filter.C         Filter and convolution
- Image.C          Constructors and destructors
- Matrix.C         Matrix operators
- Morph.C          Morphological operators
- Operator.C       Operator definitions
- Stats.C          Statistical functions
- Support.C        Support and miscellaneous functions
- Trans.C          Type translation operators

Domain and Range functions:

- Domain.C
- Range.C

These source files are expanded during library installation to implementations for each of
the classes. The file names are prefaced with i, r, b, or c, in the same manner as the image
header files, to reflect their data type.
PROGRAM DEVELOPMENT CYCLE

General. The following description of the program development cycle refers to scripts and preprocessors that are defined for the UNIX system. The class library has not been tried on other systems, due to the lack of a C++ compiler.

Library Installation. The library is installed by running the installSlip script. This first expands the source files as necessary to one version per image type (a set of function definitions for each of the IntImage, RealImage and other types). Then a make script is run to compile and link the files into the oopsSlip.a object library.

Using C++. To write procedures in C++, both the <stream.h> and "oopsSlip.h" header files must be included.

Program Compilation. Programs are compiled by using the slip script. This first runs a preprocessor to convert any special operators to function calls, then passes the resulting files to the CC (C++) compiler. The oopsSlip.a object library is linked in.

Restrictions and Caveats. There are no known restrictions on C++ when using the oopsSlip library.

A major source of confusion (from personal experience) during the compilation phase is that the error messages generated by the CC compiler can be vague or misleading. Often, the problem is simply a missing semi-colon or bracket. The preprocessor does not cause any lines to change numbers, so line numbers returned in the CC error messages can be used when debugging an application program.
CHAPTER 5 - APPLICATION EXAMPLES

GENERAL

Chapter Objective. The objective of this chapter is to demonstrate the use of the oopsSlip class library by implementing a number of image processing applications. These applications were used to verify and validate the system.

Applications. The applications presented in this chapter are:

a. Dilation: an example of the use of the nextElemLoc() function, and how to avoid the use of the Domain and Range classes;
b. Region Growing: topological operation testing the class library, using ByteImage.
c. Skeletonization: a morphological algorithm;
d. Fourier Transform: a mathematical algorithm using ComplexImage;
e. Hadamard Transform: demonstration of matrix operations, using RealImage; and
f. Filtering: a morphological approach, using iteration over a domain and range with an IntImage.

Algorithm Illustration. The algorithms are illustrated using the block diagram conventions of [DoG87a].

GREY SCALE DILATION

Algorithm Description. Dilation is a fundamental morphological operation which consists of expanding the objects in an image by a structuring element. A structuring element
(also called a template, or probe) is used in morphology to determine the properties of an image. It can be considered an image in its own right, that can be operated on itself.

The algorithm, illustrated in Fig 5-1, takes as input an image im1 and a structuring element se. The dilation is the union of each translate and offset of the image by the structuring element [DoG88c]. The expression "translating the image by an element of the domain of the structuring element" means that for each defined element of se, the origin of the image im1 is moved to the grid location of the element. The image is then offset by the value of the element of se, in other words, the se element value is added to each defined element of im1. This resulting translated and offset image is then unioned (using the extended maximum) with the other translates/offsets, making up the final image.

![Diagram](image)

**Fig 5-1 Example of block diagram (dilation)**

**Implementation.** This algorithm can be implemented directly from the block diagram. A first attempt at a C++ implementation is shown in Fig 5-2.
#include <stream.h>
#include "oopsSlip.h"

IntImage dilate(IntImage& iml, IntImage& se)
{
    int i;            /* counter */
    Domain seDom;     /* domain of structuring element */
    IntRange seRng;   /* range of structuring element */
    IntImage imout;   /* output image, initialized to null */

    /* get domain and range of structuring element */
    seDom = se.extractDomain();
    seRng = se.extractRange();

    /* do translates and offsets and union (extmax) the whole */
    for (i = 0; i < se.card(); i++) {
        IntImage temp = iml;        /* get copy of iml for moving */
        temp = temp.tran(seDom.showDomElem(i)) + seRng.showRngElem(i);
        imout = imout <@> temp;
    }

    /* all done */
    return imout;
}

Fig 5-2 Implementation of Dilate using Domain and Range

There are three major features which distinguish this C++ code from C:

a. the member function calling syntax ident.functionname: this is a means of indicating that an object will perform an action;

b. identifier declarations in the body of a procedure: in C++, identifiers can be declared at any point (although I prefer to group them at the top of the scope in which they are to be used, which creates a type of "data dictionary" for that scope); and

c. the use of the + operator to add a scalar (the value returned by seRng.showRngElem(i)), to an image.

Otherwise, the constructs are similar to that of C. In implementing the oopsSlip library, several module implementations which were previously written in the C language were simply optimized to take advantage of the features of C++. 

5-3
The overloaded operator + in this implementation is interpreted as "offset", because the operands it is given. The additional operator <@> is used here, for extmax.

This first attempt to implement dilate used the Domain and IntRange data types: the information in these data types is already contained in the input image iml, so their use gives no real advantage. The Domain and Range constructs can be avoided, thereby reducing the number of data types (and storage) required, as shown in Fig 5-3.

```c
#include <stream.h>
#include "oopsSlip.h"

IntImage dilate(IntImage& iml, IntImage& se) {
    int i;                /* counter */
    Coord sePixLoc=se.nextElemLoc(); /* grid location of se elem used */
    IntImage imout;        /* output image, init to null */

    /* do translates and offsets and union (extmax) the whole */
    for (i = 0; i < se.card(); i++, sePixLoc=se.nextElemLoc(sePixLoc)) {
        IntImage temp = iml;
        temp = temp.tran(sePixLoc) + se.elem(sePixLoc);
        imout = imout <@> temp;
    }

    /* all done */
    return imout;
}
```

**Fig 5-3** Implementation of grey-scale dilation without using Domain and Range

In this case, the domain and range of the image is being extracted as the algorithm progresses, using the nextElemLoc() function to find the location of the next defined element (domain), and elem() function to extract its value (range).
If the `nextElemLoc()` function is called without an argument, it returns the location of the first defined (non-STAR) element in the image, starting from the top left-hand corner. The result is a `Coord` value, `sePixLoc` in this case, containing the coordinates of the element in the universal image grid. This coordinate can be used as input to a subsequent call to `nextElemLoc()`, which then returns the next location, traversing in a left to right, top to bottom manner. If there are no more defined elements, it returns back to the top, which is why a control counter limited to the number of elements in the image (card) is required.

This second implementation is closer to the ideal of "only one data type", in that only the Image is being manipulated.

The implementation can be further simplified by the use of the `for_domain_of()` construct, as shown in Fig 5-4.

```c
#include <stream.h>
#include "oopsSlip.h"

IntImage dilate(IntImage& iml, IntImage& se)
{
    IntImage imout; /* output image */

    /* do translates and offsets and union (extmax) the whole */
    for_domain_of(se) {
        IntImage temp = iml;
        temp = temp.tran(here) + se.elem(here);
        imout = imout (@> temp;
    } end_for();

    /* all done */
    return imout;
}
```

Fig 5-4 Implementation of dilation using `for_domain_of()`
The advantage of this implementation is that it clearly shows that an action is being done over the domain of an image (in this case, \textit{se}). The \texttt{for} loop used previously, although equivalent, does not express this paradigm as effectively. If it was not for the need to reset the image \texttt{temp} for each iteration, this implementation would follow directly from the block diagram.

**REGION GROWING**

**Purpose.** The purpose of the region growing procedure is to start with a given seed region \( f_0 \) (usually a single element) and find the region of connected elements containing \( f_0 \), in the image \( f \), and whose grey values vary from the average value of the seed by less than a uniformity parameter \( t \). [DGS89] describes an algorithm using dilations to accomplish this technique.

**Algorithm.** The algorithm is iterating, starting with a binary (1/*) seed \( f_k \). The seed is dilated by a 3x3 constant mask of 1's, resulting in a larger image that has all elements, belonging to \( f \), that are in the domain of \( f_k \) or are a neighbor of \( f_k \). The domain of the original seed is then removed from the dilated version, leaving only the boundary of \( f_k \), with boundary elements = 1 and all other elements = * (undefined). This is the region to be tested, \( R_k \). Multiplying \( R_k \) with the original image retrieves the grey levels of the test elements.

The average grey level of \( R_k \) is then computed, and compared to the actual values. Only the elements whose grey value varies from the average by less than the uniformity criterion are allowed to remain. A binary image of the elements of \( R_k \) whose grey values meet the uniformity criterion is computed.
The chosen pixels are then appended to $f_k$. If there are no new pixels, the procedure halts, otherwise the new (larger) $f_{k+1}$ is used as input to the next iteration.

The algorithm is illustrated in block diagram form in Fig 5-5.

![Block diagram for Grow](image)

**Implementation.** The oopsSlip implementation of grow is given in Fig 5-6.

What makes this implementation different from C is the use of the overloaded $\ast$, for both an image multiply and scalar image multiplication. Previous implementations of FHIA required different names for each of these operations (hence the use of $\text{mult}$ for image multiply, and $\text{scalar}$ for scalar image multiplication in the block diagram).

The algorithm makes use of many of the math functions in the library. Although they can be invoked by either the member function syntax ($\text{.}$) or the friend syntax, it is sometimes preferable to use the friend syntax when using several operators on the same image (this is a matter of personal preference).
```cpp
#include <stream.h>
#include "oopsSlip.H"

RealImage grow(RealImage& iml, RealImage& seed, int uniformityParam)
{
    RealImage dilationMask(-1, 1, 3, 3, 0);  /* 3x3 dilation mask of 1's */
    RealImage lastSeed(0, 0, 0, 0, 0);       /* remember last result */

    /* repeat cycle until the region (seed) does not grow any more */
    do {
        RealImage boundary;       /* boundary of seed */
        RealImage newPixels;      /* new elements to add to region */
        double avgGreyLevel=0;    /* average grey level of region */

        /* locate boundary of dilated region and mark it with 1's */
        RealImage dilatedSeed = dilate(seed, dilationMask);
        boundary = dilatedSeed <+> sub(seed);
        boundary= recip(boundary).threshDefined();

        /* boundary now contains the outline of the seed...
         * ...get the average value of the elements of the seed */
        RealImage copyiml=iml;
        avgGreyLevel = avg(window(copyiml, seed));

        /* determine which elem of boundary should be joined to image */
        RealImage bdyGreyValues = iml*boundary;
        boundary = -avgGreyLevel * boundary;
        newPixels = bdyGreyValues + boundary;
        newPixels=recip(threshBelowEqual(abs(newPixels),uniformityParam));

        /* newPixels now contains any pixels to be joined to the region */
        lastSeed = seed;
        seed = lastSeed <@> newPixels;
    } while (card(lastSeed) != card(seed));

    return seed;
}
```

Fig 5-6  oopsSlip implementation of Grow
SKELETONIZATION

**Aim.** Skeletonizing is used to characterize or compress an image object by reducing it to its skeleton. The skeleton SK(X) of a connected image object X is defined as the set of the centers of the maximal disks that can fit inside X. A disk B is maximal if it is the maximum size disk that can fit in X. The size of the maximal disk will vary, depending on the area of B being examined. S_r(X), r>0, denotes the rth skeleton subset, i.e. the set of the centers of the maximal disks whose radius is equal to r.

**Algorithm description.** Maragos and Schafer [MaS86] describe an algorithm, improved by Dougherty [DoG87], to obtain the skeleton of a binary (1/*) image object using erosions and openings. This algorithm is illustrated in Fig 5-7.

The input image is eroded by a disk of radius n, to determine where in the image disks of radius n fit. Square disks are used, for example (bold numbers designate the origin of the disk):

- a disk of radius 2: 
  
  \[
  \begin{array}{c}
  1 \\
  1
  \end{array}
  \]

- a disk of radius 3: 
  
  \[
  \begin{array}{ccc}
  1 & 1 & 1 \\
  1 & 1 & 1 \\
  1 & 1 & 1
  \end{array}
  \]

- a disk of radius 4: 
  
  \[
  \begin{array}{ccc}
  1 & 1 & 1 \\
  1 & 1 & 1 \\
  1 & 1 & 1 \\
  1 & 1 & 1
  \end{array}
  \]

and so on.

Opening the erosion with a disk of radius 2 indicates the areas where smaller disks can fit. The result of the opening is then removed from the result of the erosion - this gives the rth
skeleton subset, which is unioned to the output. This process is repeated with disks of ever-increasing size, until the erosion gives a null set, meaning that no bigger disks can fit in the image.

**Fig 5-7** Block diagram of skeleton

**oopsSlip Implementation.** Figure 5-8 shows the implementation of the algorithm using C++ and the oopsSlip class library.

Since the algorithm is made for binary images, the operators in the implementation are binary (using `b_open` and `b_erode`). Scope is very important when declaring an image variable: since `open1` and `output` are used inside and outside the `do` loop (the cardinality of `open1` is used to control the `do-while` loop), they must be declared outside the loop. The images `diski` and `erosl` disappear outside the loop, which is the desired effect, since they are not needed elsewhere.
The additional operators $\langle\oplus\rangle$ (extended maximum) and $\langle\oplus\rangle$ (extended addition) are used to clarify the implementation.

```c
#include <stream.h>
#include "oopsSlip.h"

ByteImage skel(ByteImage& im)
{
    int i = 1;           /* erosion disk size (radius) */
    ByteImage disk2(0,1,2,2,1); /* opening disk of size 2 */
    ByteImage open1;     /* result of opening */
    ByteImage output;    /* output image */

do {
    ByteImage diski(ceil((i-1)/2), ceil(i/2), i, i, 1);
    ByteImage eros1 = b_erode(im,diski);
    open1 = b_open(eros1, disk2);
    output = output $\oplus$ recip(eros1 $\oplus$ open1.sub());
    i++;
} while (open1.card());

    return output;
}
```

Figure 5-8 oopsSlip implementation of skel

**C Implementation.** Figure 5-9 gives the C implementation (not oopsSlip) of the algorithm. The ialib.h header file refers to another implementation of image algebra functions, in C. The slip.h header file contains a similar image structure declaration to the C++ version.

Note that all images are declared as pointers, and that the routines *initialize* and *free_image* are required to allocate and release the image storage spaces. This adds lines of code that are not directly related to the algorithm. Forgetting to release an image by omitting a call to *free_image* could prove disastrous, large images quickly filling up available memory.
```c
#include <stdio.h>
#include <math.h>
#include "slip.h" /* image structure definitions */
#include "ialib.h" /* function prototypes */

image *skel(im)  
  image *im;  /* input bilevel image */  
  {  
    image *diski;  /* eroding disk for ith iteration */  
    image *disk2;  /* 2x2 opening disk */  
    image *eros1;  /* result of erosion */  
    image *open1;  /* result of opening eros1 by disk2 */  
    image *output;  /* total skeleton */  
    int i = 1;  /* iteration number */  
    int opensize; /* card of open1 */  
    disk2 = initialize(0,1,2,2,1);  /* get disk d2 */  
    output = initialize(im->r, im->t, 1,1,star);  
    
    do {  
      diski = initialize(ceil((i-1)/2),ceil(i/2),i,i,1);  
      eros1 = b_erode(im,diski);  
      open1 = b_open(eros1,disk2);  
      opensize = card(open1);  
      open1 = comp(open1);  
      output = extmax(output,im_min(eros1,open1));  
      i++;  
      free_image(eros1);  
      free_image(open1);  
      free_image(diski);  
    } while (opensize);  
    return output;  
  }

Figure 5-9  C implementation of skeleton

In the C version illustrated above, no provision was made for different types of images, all images were stored as double.

Note that the C program implemented in figure 5-9 was not done using the oopsSlip class library. When using a C style with oopsSlip, initializing and freeing image spaces is not required, and images are not normally defined as pointers (unless one requires pointers to objects).
FOURIER TRANSFORM

Generalized Picture Transform. It can be shown [GoW87] that any linear discrete transform can be expressed as the product of the image F, and transformation matrices P and Q. If P is a m row by n column matrix, then P is a m\times m matrix, and Q is a n\times n matrix. The transformed image \( \Psi(F) \) is then given by:

\[
\Psi(F) = PFQ
\]

In image algebra, it is assumed that the image is translated to the grid origin to perform the matrix operations, since all pure matrix operations are performed at the origin ([DoG85]), and the element grid coordinates (and not their image coordinates) are used. The resulting image is translated back to the original location.

Discrete Fourier Transform. The discrete Fourier transform matrices are of the complex type. The elements of the transform matrix P are given by:

\[
P(k, j) = \frac{1}{m} e^{i[2\pi/m]kj} = \frac{1}{m} \cos \left( \frac{2\pi}{m} k j \right) + \frac{i}{m} \sin \left( \frac{2\pi}{m} k j \right)
\]

where \( m-1 \leq k \leq 0 \) and \( 0 \leq j \leq m-1 \)

Notice the positive exponent for the element definition: this is because since the transform matrix P is at the origin (grid coordinate (0,0)), the row components \( k \) of the element grid coordinates are all negative, and the column coordinates \( j \) are positive, making the \( kj \) product negative, and therefore the exponent, in reality, negative (as it should be). This is more fully explained in [DoG85].
The elements of the postmultiplier matrix $Q$ are defined in a similar fashion to those of $P$, except $n$ is used instead of $m$.

The definitions of $P$ and $Q$ so far still require referencing individual elements. An alternative means of generating the matrices is possible, by referring to image algebra. The function $\cos$ can be represented algebraically with the power series:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{2k!}$$

This approach can be translated into image algebra by letting $X$ be an image, and the power function $X^k$ meaning to perform an image multiply of $X$ on itself $k$ times (and $X^0$ to be an image with same domain as $X$ but with the defined elements = 1), and the power series truncated at value $N$ ($N$ determines the precision of the series approximation):

$$\cos(X,N) = \sum_{k=0}^{N/2} (-1)^k \frac{X^{2k}}{2k!} \quad \text{for } N \text{ even}$$

$$\cos(X,N) = \sum_{k=0}^{(N-1)/2} (-1)^k \frac{X^{2k}}{2k!} \quad \text{for } N \text{ odd}$$

The sine of an image, and any function that can be represented by a power series, can be computed in a similar manner. $N$ is assumed to be either chosen by the user or determined automatically.

The product $kj$ in the earlier definition of the transform matrix can be built by building an $m$ by $m$ matrix (for the $P$ case), or an $n$ by $n$ matrix (for $Q$), denoted $Z_{mm}$ and $Z_{nn}$ respectively, where each element is the product of its grid row and column coordinates. The definition of $P$ and $Q$ then becomes:
\[ P = \frac{1}{m} \cos \left( \frac{2\pi}{m} Z_{mm} \right) + \frac{i}{m} \sin \left( \frac{2\pi}{m} Z_{mm} \right) \]

\[ Q = \frac{1}{n} \cos \left( \frac{2\pi}{n} Z_{nn} \right) + \frac{i}{n} \sin \left( \frac{2\pi}{n} Z_{nn} \right) \]

**Inverse Fourier Transform.** The inverse transform is computed by inverting the transformation matrices and performing the matrix multiplication:

\[ \hat{F}(\Psi) = P^{-1} \Psi Q^{-1} \]

**oopsSlip Implementation.** The implementation is shown in figures 5-10 and 5-11. The first figure presents some of the supporting functions: the generation of the P and Z matrices, and the \( \cos() \) and \( \text{pow()} \) functions. The computation of \( \sin() \) and \( Q \) is similar.

In the implementation, only images are manipulated: there is no reference to individual elements, or columns or rows, except for the definition of the Z matrix (this implementation is the simplest possible). Even a complex algorithm such as the Fourier transform still can be traced back down to the basic operators such as \( \ast \) and \( + \). This illustrates the power of using Image Algebra as the basis for an image processing system.
#include <stream.h>
#include "oopsSlip.h"

RealImage pow(RealImage& iml, int exponent)
{
    RealImage imout = iml;

    if (exponent == 0) { /* if raised to power of 0. */
        imout.threshDefined(); /* set all defined elements to 1 */
    }
    else for(int k = 0; k < exponent; k++) imout = imout * iml;
    return imout; /* all done */
}

RealImage cos(RealImage& iml, int nbrOfTerms)
{
    RealImage imout;

    for (int k=0; k < nbrOfTerms/2; k++) {
        imout = imout + ( pow(-1,k) * pow(iml,2*k) / (double)fact(2*k) );
    }
    return imout;
}

RealImage generateZ(RealImage& iml) /* generate Z matrix for image */
{
    RealImage Z(0,0,iml.numRows(),iml.numCols());

    for_domain_of(Z) {
        Z.setElem(here, (here.row() * here.column()) );
    } end_for();

    return Z;
}

ComplexImage generateP(RealImage& iml) /* generate P matrix for DFT */
{
    double m = (double)iml.numRows(); /* number of rows in input img */
    ComplexImage P(0,0,m,m,0); /* output P matrix */
    RealImage Z = generateZ(iml); /* Z matrix */

    P.setRe( (1/m) * cos((2*PI/m)*Z) );
P.setIm( (1/m) * sin((2*PI/m)*Z) );

    return P;
}

Fig 5-10 Implementation of DFT (supporting functions)
ComplexImage fourier(RealImage& iml) /* calculate DFT for image */
{
    ComplexImage imout; /* transform */
    ComplexImage P = generateP(iml); /* pre-multiplication matrix */
    ComplexImage Q = generateQ(iml); /* post-multiplication matrix */
    Coord imloc = iml.loc(); /* remember where image was */
    iml.setImageLoc(0,0); /* translate iml to grid origin */
    imout = cross(cross(P,iml), Q); /* calculate DFT = PxFxQ */
    iml.setImageLoc(iml.loc()); /* set image back to original loc */
    imout.setImageLoc(iml.loc()); /* and output to same place */
    return imout;
}

Fig 5-11 Implementation of DFT

HADAMARD TRANSFORM

Mathematical description. The Hadamard transform is another application of the
generalized discrete picture transform, not requiring complex arithmetic. If the input image is
square, and is dimensioned of a power of 2 (...64,128,256,512,... rows and columns), then the
transform matrix P can be generated quickly by a series of translates and extensions. The basic
2x2 Hadamard kernel is:

\[
H_{2 \times 2} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}
\]

Larger versions of the Hadamard matrix are generated by the property that if \( H_{JJ} \) is a
Hadamard matrix, then [Pav82]:

\[
H_{2J \times 2J} = \begin{pmatrix} H_{J \times J} & H_{J \times J} \\ H_{J \times J} & -H_{J \times J} \end{pmatrix}
\]
The Hadamard transform for an image $F$ of dimension $m$ rows by $n$ columns, is:

$$\Psi(F) = PFQ$$

where $P = \frac{1}{\sqrt{m}} H_{mm}$ and $Q = \frac{1}{\sqrt{n}} H_{nn}$

Since the inverse matrices $P^{-1}$ and $Q^{-1}$ are identical to $P$ and $Q$, the inverse transform is of the same form as the forward transform.

**Implementation.** The oopsSlip implementation of the Hadamard transform is shown in figure 5-12.
RealImage hadMatrix(int order) {
    if (isPowerOf2(order) != TRUE) return NullRealImage;

    RealImage imout(0, 0, order, order, STAR);
    RealImage last(0, 0, 1, 1, 1); /* H11 is a 1x1 matrix= 1 */

    for (int i=0; i < (int)(log(order)/log(2)); i++) {
        imout.extend(lastsetImageLoc(0, last.numCols()));
        imout.extend(lastsetImageLoc(-last.numRows(), 0));
        imout.extend(subImageLoc(-last.numRows(), last.numCols()));

        last = imout;
    }

    return imout;
}

RealImage hadamard(RealImage& iml) {
    int m = iml.numRows();
    int n = iml.numCols();

    if ((isPowerOf2(m) || isPowerof2(n)) != TRUE) return NullRealImage;

    RealImage imout = cross(cross(iml, had_matrix(m)), had_matrix(n));
    imout = (1./sqrt(m*n)) * imout; /* normalize */

    return imout;
}

Fig 5-12  oopsSlip implementation of Hadamard Transform
FILTERING

**Aim of algorithm.** The moving average filter is used for many different purposes: to smooth out irregularities, to emphasize edges or textures, to increase contrast, and so on. This implementation of a filter uses morphological techniques to create another approach to the algorithm.

**Algorithm description.** The algorithm is presented by Dougherty [Dou89a], and is illustrated in Fig 5-13.

---

**Fig 5-13** Block Diagram of FILTER

This block diagram is very similar to that of dilate, except for:

a. the input mask must be rotated 180 degrees about the origin (the - (minus) image operator);

b. instead of adding the specific element value to the translated image, a multiplication of the image by the scalar is performed;

c. an add of all the results is performed at the end of the algorithm, instead of the extmax of the dilation.
The advantage of this approach is that the number of image translations and multiplications made is dependent not on the number of elements in the image, but by the size of the mask. For a 3x3 mask, only 9 translations and multiplications are made. If these operations can be implemented in hardware, the result is a very fast algorithm.

**oopsSlip Implementation.** Figure 5-14 shows the oopsSlip implementation of the filter algorithm.

```c
#include <stream.h>
#include "oopsSlip.h"

IntImage filter(IntImage& iml, IntImage& msk)
{
    IntImage imout = iml;  /* output image */
    imout.setAll(0);       /* initialize imout to 0's */
    mask = -msk;           /* mask, rotated 180 degrees */

    /* do translates and offsets and image sum the whole */
    for_domain_of(mask) {
        IntImage temp = iml;
        temp = temp.tran(here) * mask.elem(here);
        imout = imout + temp;
    }  end_for();

    /* all done */
    return imout;
}
```

Fig 5-14  oopsSlip implementation of filter

The implementation is also similar to that of dilate, except for two additional initialization statements: the (minus) operator, and the setAll() function call.

The actual input image msk is copied to a temporary image mask, before rotation, because the unary - (minus) operator modifies the the contents of the image itself. If msk were
directly acted upon, another 180 degree rotation would be required before exiting the routine to restore \textit{mask} to its original value.

The input image \textit{imout} is initialized to the same domain as \textit{im1}, but containing all zeros. The image add operator $+$ operates only on the intersection of the images: using the default initialization of a null image (0 rows by 0 columns), which happens when an image is declared without any arguments, would result in a null sum, and eventually to a null resulting image.

The advantages of this algorithm are apparent when comparing this implementation to that of the more common one of two nested loops (one for summing over the rows, another for the columns): there is no reference to either the size of the image, or whether it is stored in \text{(row,column)} or \text{(column,row)} format, a common source of confusion for novice users. In addition, masks can be of arbitrary size and shape, as well as contain "holes" of undefined elements, without having to modify the code.

\textbf{C Implementation.} For comparison purposes, a C implementation of filter using the same algorithm, is shown in Fig 5-15. Although the \texttt{Domain} and \texttt{Range} classes are used, a similar function to \texttt{nextElemLoc()} could have been written to achieve the same effect as the \texttt{oopsSlip} implementation.

Three drawbacks of C as compared to C++ are apparent in this implementation:

\begin{itemize}
  \item \textbf{a.} the need to declare structures as pointers, then expressly allocate (\texttt{initialize}) and deallocate them;
  \item \textbf{b.} the pointer notation needed to access elements of the \texttt{Range} and \texttt{Domain} stacks; and
  \item \textbf{c.} the absence of overloaded operators (use of \texttt{add} and \texttt{scalar}).
\end{itemize}
image *filter(im,mask)
    image *im; /* input image */
    image *mask; /* filtering mask */

    { image *image2; /* Scratch copy of Image for calculations */
      image *imout; /* output image */
      int i,j; /* counters */
      domain_stack *maskdom; /* Domain of input mask */
      range_stack *maskrng; /* Range of input mask */

      if ((imout=initialize(im->r,im->t,1,1,star)) == NULL)
        return NULL; /* if no more memory, quit */

      /* Rotate mask by 180 degrees */
      mask = minus(mask);

      /* Get range and domain of mask */
      if ((maskdom = domain(mask)) == NULL) { /* if null image */
        free_image(imout);
        mask = minus(mask); /* restore mask to input state */
        return NULL;
      }
      maskrng = range(mask);

      /* Move Mask over all of Image and add results */
      for (i=0; i < maskdom->length; i++) {
        image2 = copy_image(im);
        image2 = tran(image2,(maskdom->stk+i)->x,(maskdom->stk+i)->y);
        image2 = scalar(*(maskrn->stk+i),image2);
        imout = add(imout,image2);
        free_image(image2);
      }
      mask = minus(mask);
      return imout;
    }

Fig 5-15  C implementation of Filter

Image Results.  An edge detector was implemented using the filter function, and the eight Kirsch masks. Fig 5-16 gives the edge detector block diagram.

The basic Kirsch mask K0 is is a directional gradient mask:
\[
K_0 = \begin{pmatrix}
-5 & 3 & 3 \\
-5 & 0 & 3 \\
-5 & 3 & 3 \\
\end{pmatrix}
\]

This generates positive values when traversing from a region of lower grey value to higher grey value, and negative in the reverse case. By rotating the mask \(K_0\) up to seven times 45 degrees.

Fig 5-16 Original Image
256 by 256 elements, 256 grey levels
(courtesy Gopal Sundaramoorthy)

Fig 5-17 Edge detector
about the origin, one generates the seven other compass masks K1 through K7. The directionality is useful to detect edges in all directions. The results of the operator on the original image (Fig 5-16) are given in Fig 5-18.

Fig 5-18a. Gradient with K0 mask (normalized to 256 grey levels)

Fig 5-18b. Final image.

(Processing time, 15 minutes)
CHAPTER 6 - DISCUSSION

GENERAL

**Chapter Objective.** The objective of this chapter is to discuss aspects of the oopsSlip library implementation.

MEETING THE SYSTEM REQUIREMENTS

**Design Qualities.** As discussed in chapter 3, the required design qualities of the system are that it be **portable**, **simple**, and **efficient**.

**Portability.** The oopsSlip system is portable to any system that has a C++ version 1.2 compiler, and runs under UNIX (because of the preprocessor and shell scripts). Minor modifications (especially deletion of the additional operator symbols) would remove the requirement for a UNIX environment, although this has not been tested because of the a C++ compiler is not available locally on other than a UNIX system.

Code written using the oopsSlip library, even if written in a C procedural style, will not compile using C.

No special hardware is required for use of the class library.

**Simplicity.** Use of this class library is simplified by:

a. functions apply to any image type: the name of the function does not depend on the image type, and common names and operators are used;
b. equational expressions are encouraged;
c. memory allocation and deallocation is transparent to the user;
d. detailed knowledge of C++ is not necessary to implement simple routines (since C is a valid subset of C++ and the procedural style of function calls is available).

However, some factors tend to complicate the system:

a. C++ compiler error statements are sometimes vague or misleading, although, with practice, one learns how to interpret them;
b. the large number of functions and calling syntaxes available;
c. the different philosophy of image processing algorithm design, using image algebra, can seem alien at first;
d. the differences between C++ and C.

Efficiency and Performance. Compared to dedicated image processing machines, this system is very slow. However, when compared to similar algorithms implemented in C on the same platform, performance is comparable.

One construct in particular suffers from poor performance: the for_domain_of() loop. For example, coding the threshDefined() function using the domain loop:

```c
for_domain_of(img) {
    img.setElem(here,1);
} end_for();
```

instead of the more direct:

```c
for (int i=0; i< img.area(); i++) {
    if (img.elem(i)!=STAR) img.setElem(i,1);
}
```

results in a slower execution time by an order of 10 (10 seconds using the domain loop instead of under 1 for the direct method). This is explained by the need for the domain loop to make several
function calls for each iteration of the loop, whereas the directly coded version accesses the
element through a simple address offset *(elem(i) is defined inline as a pointer offset)*. This
does not however detract from the usefulness of the construct, especially when the number of
defined elements is small (such as the dilate type of constructs, operating over small structuring
elements).

Absolute measures of performance are difficult to obtain because of the lack of a CPU
timing function. The C++ library function *clock()* simply counts the number of processor ticks
since the first call of the function: this also includes time when the function being timed is not
executing. Using the *clock()* function, filtering a 256 by 256 element *ByteImage* using a 3 by
3 mask took approximately two minutes.

**BUT IS IT OBJECT-ORIENTED?**

**Requirements of an Object-Oriented System.** Bertrand Meyer, principal
architect of the object-oriented language Eiffel, postulates a hierarchical list of seven levels of
system "object-orientedness" [Mey88]:

1. Object-based (data-based) modular structure (the most basic level);
2. Data abstraction (objects are implementations of abstract data types);
3. Automatic memory management (without programmer intervention);
4. Classes;
5. Inheritance (classes are an extension or restriction of another);
6. Polymorphism and dynamic binding (modules can refer to objects
   of more than one type, and operations have different realizations
   in different classes);
7. Multiple and repeated inheritance.
A system that meets all the levels in this hierarchical list can be called an "object-oriented system".

The oopsSlip class library does not meet all the above requirements, mainly because of the restrictions of the chosen implementation language, C++ version 1.2. C++ is described as having an object-oriented "flavor" [Cox86] because many feel that it does not provide or enforce a complete object-oriented environment.

C++ does provide the most basic level of object-orientedness since it encourages the organization of system modules on the basis of data structures (the class construct). It also provides facilities for describing abstract data types (class, struct). These two first levels of "object-orientedness" can also be achieved by a skilled programmer in almost any language, including FORTRAN ([Mey88]). C++ simply provides the tools to make the task easier.

The third level, memory management without user intervention, is in practice realized by C++, with the constructor and destructor facilities. Although these routines must be written, the invocation when instances pass in and out of scope is invisible to the user. Note that the language itself, like C, does not have facilities for true "garbage collection".

The fourth level, provision for classes (binding together of the data type with the associated modules) is implemented by the C++ class construct.

Basic inheritance is provided for in C++ version 1.2, although it has not been used in the implementation of the oopsSlip class library, for lack of a "generic class" mechanism: ideally, a generic Image class would be defined, which would be independent of the element data type (int, double, complex, etc). Then the type-specific image classes such as IntImage would simply be defined by inheritance.
This lack of genericity was overcome by the use of the "identifiers" \texttt{IMAGE} and \texttt{PIX}. All the source files that apply to more than one image type are written with the pseudo-identifiers used instead of the actual type identifier. During the system build process, the code generator expands these source files into one file for each type, using the UNIX \texttt{sed} utility, much like a macro expansion. These files are then compiled and linked.

The sixth level, polymorphism and dynamic binding, is provided for by the \texttt{friend} construct and by overloading operators. Polymorphism has not been exploited in the oopsSlip library implementation, because the type generation method mentioned above would render polymorphism too difficult. Overloading is used to a large degree in the class library, and is one of the major advantages of choosing C++ over other languages.

\textbf{Conclusion}. The implementation of the oopsSlip library allows the use of as much of a degree of object-orientedness as desired. Such flexibility is a result of the C++ language design. The member functions in the library have been implemented using mainly an object-oriented style, and such style (or even an improved style) is encouraged in further applications of the library.

The oopsSlip library is object-oriented in the generic sense in that it is centered about the abstract data type, \texttt{image}. 

\begin{center}
\end{center}
USE OF IMAGE ALGEBRA

**General.** Image Algebra has been used in this system to define the basic image operations (add, multiply, divide, etc). A knowledge of FHIA is not required to use this software, however the user must be aware of the significance of the * (star), or undefined element. In many cases, the star will not be included in images, so that their effect is unnoticed. Someone who is familiar with the action of the star can use it to advantage: for example, setting the background to * can reduce the effects of the background on the image object of interest.

The greatest advantage of using FHIA as the basis for the functions in this system is that it provided a consistent framework in which to define image interactions.
CHAPTER 7 - FUTURE DIRECTIONS

GENERAL

Chapter Objective. The objective of this chapter is to discuss directions for further investigations in this subject.

USE OF C++ VERSION 2.0

C++ version 2.0 does not appear to support the idea of "generic class". With such a facility, only one Image class would need to be defined, then the type-specific classes could be simply generated from the generic, at a compiler level. The source code would not be affected, as it is in the oopsSlip implementation. Multiple inheritance, provided in 2.0, could allow an Image class to be defined by inheriting from a Matrix class and a virtual Image class, but a code expansion at the source level, similar to what is done now, would still be necessary to provide type-specific classes.

C++ version 2.0 would probably not provide any significant advantages with this design.

OTHER OBJECT-ORIENTED LANGUAGES

Ada: Ada provides tools for defining generic classes [Mey88] that would avoid the artificial means (source code expansion) of defining a generic class (package in Ada) as performed here. The code would be written in a similar fashion as this system was implemented (using PIX for
the element data type, for example), but the compiler would make the substitution, providing a facility for error checking not present in oopsSlip.

Since the data types are known in advance of the application implementation, the artifices provided by Ada to simulate encapsulation (public and private parts) are not a restriction here. Operator overloading is also provided [Cox86], giving the same advantages as C++.

From an objective point of view, Ada, although not being a complete object-oriented language, would be better than C++ for implementation of the FHIA class library, because of its generic package capability.

**Other languages.** "Pure" object-oriented languages such as Eiffel may be applicable to this type of image processing, because they meet the requirements of an object-oriented system more fully than does C++. However, it may be difficult to obtain adequate performance because of the type of compiler, or that the language is tied to a platform that is inefficient for image processing.

**VISUAL INTERFACE**

The block diagrams shown as algorithms in chapter 5 could benefit from being translatable directly into code. Ideally, one could select from a number of "prefabricated" functions, then store the new function in the library.

The major obstacle for the success of a block diagram visual interface is the need to define a mechanism to indicate "perform over the domain" of the image, or a similar construct. The
convention used in this document would probably be too awkward for use as a programming convention.
CHAPTER 8 - CONCLUSION

A need was expressed for an image processing system that uses the image as the primary unit of data manipulation, instead of individual pixels. The system must be portable (must run on a general-purpose computer), simple and efficient. The proposed solution was the creation of an object-oriented class library, based on Finite Homogeneous Image Algebra.

This class library was implemented in C++, using the language features of overloading and class type definitions to implement the image as a bound matrix. Image types were created whose elements could have any of four different data types (int, double, complex, or Byte). Commonly used operators such as +, -, <, > and others were defined to have meaning when applied to images, or between images and scalars. Many functions were defined to manipulate images. A preprocessor was developed to implement image operators not definable in C++ (for example, [+]).

Function definitions allowed the use of either an "object-oriented", or member function invocation style (a.thresh(val);), or a more traditional function calling style (thresh(a, val);). Function names are independent of the data type of the image, and many have the same name whether applied to images or scalars (max, abs).

Applications were presented that illustrated the use of the class library in developing high level imaging functions.

The class library system achieved the goals of portability and simplicity, with the efficiency of the system being comparable to equivalent algorithms programmed in C on the same general-purpose computer. The class library, although being designed in an object-oriented style, allows
varying degrees of "object-orientedness" when being used. Finite Homogeneous Image Algebra provided a consistent basis to implement functions in a hierarchical style.
APPENDIX A - INTRODUCTION TO IMAGE ALGEBRA

GENERAL

The aim of this appendix is to give an introduction to the image algebra developed by Dougherty and Giardina. More detailed information can be found in [DoG87a], [Dou89a] and [Dou89b].

HOMOGENEOUS IMAGE ALGEBRA

Images. An image is described mathematically in Homogeneous Image Algebra (HIA) as a real-valued function defined on a subset of the Cartesian grid $\mathbb{Z} \times \mathbb{Z}$, $\mathbb{Z}$ being the set of integers. The subset where the image $f$ is defined is called the domain of the image, denoted by $D[f]$. The set of element values (and not their position on the grid) is called the range of the image.

Operations and Inducement. Operations on functions are induced by operations on their domains and ranges. The concept of inducement, taken from group theory, allows the properties of operations on the set of real numbers $\mathbb{R}$ to transfer over to operations on images.

Operations on or between images are induced in one of two ways: domain-induced, by mappings on subsets of $\mathbb{Z} \times \mathbb{Z}$, and range-induced, mappings in $\mathbb{R}$. An example of a domain-induced operation is unit translation $[\mathcal{T}]f$, which is the translation of $f$ one unit to the right (therefore, an operation on the position of the image elements or domain). Image addition $f + g$, where $f$ and $g$ are images, is a range-induced operation, because it is defined as a pointwise sum (an operation on the element values or range) on the intersection of the domains, and is
undefined (does not have a value) elsewhere. The properties of addition in \( \mathbb{R} \) such as associativity and commutativity extend to image addition by inducement [DoG87b].

**Basis.** At the core of HIA is a minimal collection of seven basis operations from which all higher level image processing routines can be formed by composition [Dou89a]. The definition of the basis is a consequence of the inducement procedure.

These basis operations have the power to express high level image processing algorithms such as gradient edge detection, region growing, convolution and dilation [DoG88b], as well as describing other image algebras [Dou89b].

**FINITE HOMOGENEOUS IMAGE ALGEBRA**

**General.** Homogeneous image algebra is called that way because it requires only one data type, the image. However, the definition of image in HIA allows infinite domains, which cannot be implemented in practice. Allowing only finite domains defines a subalgebra of HIA called finite homogeneous image algebra (FHIA), realizable using the concept of the bound matrix [DoG87a].

**Bound Matrix.** A bound matrix is an array of finite domain, as illustrated in Fig A-1, where \( a_{pq} \) is either a real number or has value "undefined", \( m \) and \( n \) are the dimensions of the smallest matrix containing the domain of all defined values, and \((t,r)\) is the location of the top left-hand corner (row,column) of the matrix in the universal \( \mathbb{Z} \times \mathbb{Z} \) grid\(^1\).

---

\(^1\) Note that the \((t,r)\) convention for image location is the opposite of that used in [DoG87] and related articles. This is to maintain consistency with the \((row, column)\) coordinate designation.
\[
\begin{pmatrix}
a_{00} & a_{01} & \cdots & a_{0(n-1)} \\
a_{10} & a_{11} & \cdots & a_{1(n-1)} \\
\vdots & \vdots & \ddots & \vdots \\
a_{(m-1)0} & a_{(m-1)1} & \cdots & a_{(m-1)(n-1)} \\
\end{pmatrix}
\]

Fig A-1 (top) Bound Matrix - (bottom) Overlay of bound matrix in the grid

**Star.** The "undefined" value, called star (*), indicates that element \(a_{pq}\) has an unknown or no grey value and is not in the image domain. The bound matrix is called such because it is surrounded (bound) by stars. The bound matrix can also contain stars (for example, after a pixel division with a zero). It is possible for an image not to have a rectangular domain, although its bound matrix will be still rectangular (with stars filling the remaining space in the matrix).

**Basis operations.** A consequence of the finite nature of the domains in FHIA is that the number of basis operations reduces to six [Dou89b]. The definitions of the basis operations are:
a. Add: \((f + g)(x) = f(x) + g(x)\) on the intersection of their domains \((x \in D[f] \cap D[g])\)

b. Multiply: \((f \cdot g)(x) = f(x)g(x)\) on the intersection of their domains \((x \in D[f] \cap D[g])\)

c. Divide: \((f / g)(x) = f(x)/g(x)\) if \(x \in D[f] \cap D[g]\) and \(g(x) \neq 0\)

d. Extended Maximum:
\[
(f \oplus g)(i,j) = \begin{cases} 
  f(i,j) & (i,j) \in D[f] - D[g] \\
  \max(f(i,j),g(i,j)) & (i,j) \in D[f] \cap D[g] \\
  g(i,j) & (i,j) \in D[g] - D[f]
\end{cases}
\]

( \((D[f] - D[g])\) means the domain of \(f\) that is not part of the domain of \(g\) )

e. 90-degree rotation: \(([N](f))(i,j) = f(-j,i)\)

f. Unit Translation: \([T](f)(i,j) = f(i-1,j)\)

The actions of the ninety degree rotation and the unit translation are illustrated in Fig A-2.

![Fig A-2 Domain basis operators](image-url)

As an example of the use of bound matrices and the basis, Fig A-3 illustrates the direct addition operation.
**Extended operators.** An important aspect of image operations is the concept of extension. As defined in the basis operation "<@>" (extended maximum), the output is the maximum of the pixel values where the domains intersect, extended by unioning this interim result with the remaining domains of the two images where they do not intersect. This type of operation can be applied to the other image arithmetic operators. For example, extended addition is defined directly as:

\[
(f _{\leftrightarrow} g)(i,j) = \begin{cases} 
    f(i,j), & (i,j) \in D[f] - D[g] \\
    f(i,j) + g(i,j), & (i,j) \in D[f] \cap D[g] \\
    g(i,j), & (i,j) \in D[g] - D[f]
\end{cases}
\]

This operator can also be expressed in terms of the basis and lower-level operators:

\[
f _{\leftrightarrow} g = (f + g) <@> [(h/h) * f] <@> [(k/k) * g]
\]

where
\[ h = E[1, D[f]] \triangleleft E[0, D[g]] \]
\[ k = E[1, D[g]] \triangleleft E[0, D[f]] \]

\( E[n, D[f]] \) denotes the creation of an image that has the same domain as \( f \) where all the defined pixels have grey value \( n \) (function derivable from the basis)

\( \triangleleft \) denotes the extended minimum operator (which is defined directly in a similar manner as extended maximum, and is also derivable from the basis)

Fig A-4 illustrates the extended addition operation.

The most common extended operators are extension, extended addition, extended multiplication and extended maximum and extended minimum.

\[ f = \begin{pmatrix} 2 & 1 & 4 \\ 3 & * & * \end{pmatrix}_{(1,0)} \quad g = \begin{pmatrix} 8 & * \\ -7 & 4 \end{pmatrix}_{(2,0)} \]

Fig A-4  Extended Addition
**Structural operators.** These operators act on the domain of the image, and do not modify the image values. The principal structural operators are: domain, range, constant (create an image of a given domain with a constant value), selection (windowing) and generalized translation.

**Minkowski operators.** Another group of operations are the Minkowski operators, used in morphological (shape-based) image processing. There are two main operators: Minkowski addition (dilation) and Minkowski subtraction.

For example, dilation of S by a “structuring element” E, denoted S[+]E, is defined by taking the domain of S, D[S], translating the structuring element E by each value (i,j) of D[S], and taking the extended maximum of all translations.

A binary image is defined as an image where all elements have value 1 or star. A binary image example of dilation (Minkowski addition) is given in Fig A-5. Notice how the operation expands (dilates) the domain of the image.
The domain of $S$, $D[S]$, gives the locations of the defined elements of $S$:

$$D[S] = \{(2,1),(2,3),(1,1),(1,2),(1,4),(0,1),(0,2),(0,3)\}$$

There will be 8 translations of $E$, once by each element of $D[S]$, for example:

$$\text{TRAN}[E;2,1] = \text{translates } E \text{ by 2 columns up and 1 row to the right} = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{bmatrix}_{(3,0)}$$

Performing the eight translations and doing an extended maximum:

$$S = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{bmatrix}_{(3,0)}$$

which is the dilation of $S$ by $E$.

Fig A-5  Binary Dilation (Minkowski Addition) (from [DoG88c])

Behavior of star. A notable feature of image algebra is the behavior of the star. Even if images are restricted to rectangular domains (which may be an undesirable restriction), inducement causes image processing operations to treat undefined values in different ways. For instance, when convolving with a mask (which is simply a small image), values outside the image domain (stars) are treated as zero; however, when dilating an image by a structuring element (also considered a type of image), values outside the image domain are treated as minus infinity [DoG88b].
CONCLUSION

Homogeneous Image Algebra, and its sub-algebra, Bound Matrix Image Algebra, possesses the capability of expressing all image processing operations in a formal mathematical manner. This formalism can be used as the basis to describe the semantics of an image processing language.
APPENDIX B - CLASS DECLARATION LISTINGS

This appendix contains the class declaration listings for the Image, Domain, Range, Byte and Coord classes. The following conventions are followed with respect to generic types (Image and Range):

- **PIX or PIX** is an identifier that is replaced during library installation by int, double, complex, or Byte.

- **IMAGE or IMAGE** is an identifier that is replaced during library installation by IntImage, RealImage, ComplexImage or ByteImage.

- **RANGE or RANGE** is an identifier that is replaced during library installation by IntRange, RealRange, ComplexRange or ByteRange.
class Coord {
    int r; /* row number */
    int c; /* column number */

public:
    Coord(int a=INT_MIN, int b=INT_MIN);

    int row();
    int col();

    friend ostream& operator<<(ostream& s, Coord& a);
    friend int operator==(Coord& a, Coord& b);
    friend int operator!=(Coord& a, Coord& b);
};
class Domain {

    Coord* domainSet; // set of domain coordinates

public:
    Domain(IntImage& im);
    Domain(RealImage& im);
    Domain(ComplexImage& im);
    Domain(ByteImage& im);
    Domain(int newSize);
    ~Domain();

    Coord showDomElem(int i=0);
};

class RANGE {

    PIX* RANGESet; // set of RANGE coordinates

public:
    RANGE(IntImage& im);
    RANGE(RealImage& im);
    RANGE(ComplexImage& im);
    RANGE(ByteImage& im);
    RANGE(int newSize);
    ~RANGE();

    Coord showRngElem(int i=0)
};
class IMAGE {

  /* image header */
  
  int R;   /* column number of top LH corner */
  int T;   /* row number of top left hand corner */
  int M;   /* number of rows in matrix */
  int N;   /* number of columns in matrix */

  /* management flags */
  
  int COMP;  /* TRUE if complement(all outside elems = 1) */

  /* image matrix */
  
  PIX* PIXPTR;  /* pointer to matrix elements */

public:

  /* assignment and initialization operators */

  IMAGE(int r=0, int t=0, int m=1, int n=1, PIX val=STAR);
  ~IMAGE();
  IMAGE(IMAGE& img);
  void operator=(IMAGE& img);

  /* image functions */

  IMAGE abs();
  friend IMAGE abs(IMAGE& img);

  int area();
  friend int area(IMAGE& img);

  double avg();
  friend double avg(IMAGE& img);

  friend IMAGE b_close(IMAGE& img, IMAGE& structuringElement);
  friend IMAGE b_dilate(IMAGE& img, IMAGE& structuringElement);
  friend IMAGE b_erode(IMAGE& img, IMAGE& structuringElement);
  friend IMAGE b_open(IMAGE& img, IMAGE& structuringElement);

  int card();
  friend int card(IMAGE& img);

  friend IMAGE close(IMAGE& img, IMAGE& structuringElement);

  IMAGE comp();
  friend IMAGE comp(IMAGE& img);

  friend IMAGE convolute (IMAGE& img, IMAGE& kernel);

  IMAGE cos(int cutoff=0);
  friend IMAGE cos(IMAGE& im, int cutoff=0);
}
IMAGE cross(IMAGES im1, IMAGES im2);
friend IMAGE dilate(IMAGES img, IMAGES structuringElement);
friend PIX dot(IMAGES img, IMAGES img_2);

PIX elem(int i, int j);
PIX elem(int i);
PIX elem(Coord elCoord);
friend PIX elem(IMAGES img, int i, int j);
friend PIX elem(IMAGES img, int i);
friend PIX elem(IMAGES img, Coord elemGridLoc);

Coord elemLoc(int i, int j);
Coord elemLoc(int i);
friend Coord elemLoc(IMAGES img, int i, int j);
friend Coord elemLoc(IMAGES img, int i);

friend IMAGE erode (IMAGES img, IMAGES structuringElement);
friend IMAGE extadd (IMAGES img, IMAGES img_2);

IMAGE extend(IMAGES img_2);
friend IMAGE extend(IMAGES img, IMAGES img_2);

friend IMAGE extmax (IMAGES img, IMAGES img_2);
friend IMAGE extmin (IMAGES img, IMAGES img_2);
friend IMAGE extmult (IMAGES img, IMAGES img_2);

Domain extractDomain();
friend Domain extractDomain(IMAGES img);

RANGE extractRange();
friend RANGE extract Range(IMAGES img);

friend IMAGE filter (IMAGES img, IMAGES mask);

IMAGE flip();
friend IMAGE flip(IMAGES img);

friend IMAGE intersect(IMAGES img, IMAGES img_2);

IMAGE inv();
friend IMAGE inv(IMAGES img);

int isCardEqual(IMAGES im2);
friend int isCardEqual(IMAGES im1, IMAGES im2);

int isCompImage();
friend int isCompImage(IMAGES img);

int isDomainEqual(IMAGES im2);
friend int isDomainEqual(IMAGES im1, IMAGES im2);

int isHeaderEqual(IMAGES im2);
friend int isHeaderEqual(IMAGES im1, IMAGES im2);
int isNullImage();
friend int isNullImage(IMAGE & img);

Coord loc();
friend Coord loc(IMAGE & img);

PIX max();
IMAGE max(IMAGE & img_2);
friend PIX max(IMAGE & img);
friend IMAGE max (IMAGE & img, IMAGE & img_2);

PIX min();
IMAGE min(IMAGE & img_2);
friend PIX min(IMAGE & img);
friend IMAGE min (IMAGE & img, IMAGE & img_2);

IMAGE minbound();
friend IMAGE minbound(IMAGE & img);

friend IMAGE minksub(IMAGE & img, IMAGE & structuringElement);

Coord nextElemLoc(int gridRow=INT_MIN, int gridColumn=INT_MIN);
Coord nextElemLoc(Coord elemGridLoc);
friend Coord nextElemLoc(IMAGE & img, int gridRow=INT_MIN,
int gridColumn=INT_MIN);
friend Coord nextElemLoc(IMAGE & img, Coord elemGridLoc);

IMAGE ninety();
friend IMAGE ninety(IMAGE & img);

IMAGE ninety2();
friend IMAGE ninety2(IMAGE & img);

IMAGE ninety3();
friend IMAGE ninety3(IMAGE & img);

IntImage normalize(int maxGreyValu=255);
friend IntImage normalize(IMAGE & img, int maxGreyValu=255);

int numCols();
friend int numCols(IMAGE & img);

int numRows();
friend int numRows(IMAGE & img);

friend IMAGE open (IMAGE & img, IMAGE & structuringElement);

IMAGE pow(int exponent=0);
friend IMAGE pow(IMAGE & img, int exponent=0);

void print();
friend void print(IMAGE & img);

void printHeader();
friend void printHeader(IMAGE & img);
IMAGE readImage(char* filename);
friend IMAGE readImage(IMAGE& im1, char* filename);

IMAGE recip();
friend IMAGE recip(IMAGE& img);

IMAGE reflect();

IMAGE setAll(PIX val=STAR);
friend IMAGE setAll(IMAGE& img, PIX val=STAR);

void setElem(int i, int j, PIX val);
void setElem(int i, PIX val);
void setElem(Coord elCoord, PIX val);
friend void setElem(IMAGE& img, int i, int j, PIX val);
friend void setElem(IMAGE& img, int i, PIX val);
friend void setElem(IMAGE& img, Coord elemGridLoc, PIX val);

void setImageLoc(int tt, int rr);
void setImageLoc(Coord imloc);
friend void setImageLoc(IMAGE& img, int tt, int rr);
friend void setImageLoc(IMAGE& img, Coord imloc);

IMAGE sin(int cutoff=0);
friend IMAGE sin(int cutoff=0);

IMAGE sub();
friend IMAGE sub(IMAGE& im);

PIX sum();
friend PIX sum(IMAGE& im);

IMAGE sgr();
friend IMAGE sgr(IMAGE& img);

IMAGE sgrt();
friend IMAGE sgrt(IMAGE& img);

double ssq();
friend double ssq(IMAGE& img);

IMAGE thresh(PIX threshValue);
friend IMAGE thresh(IMAGE& img, PIX threshValue);

IMAGE threshAbove(PIX threshValue);
friend IMAGE threshAbove(IMAGE& img, PIX threshValue);

IMAGE threshBetween(PIX loVal, PIX hiVal);
friend IMAGE threshBetween(IMAGE& img, PIX loVal, PIX hiVal);

IMAGE threshBelow(PIX threshValue);
friend IMAGE threshBelow(IMAGE& img, PIX threshValue);

IMAGE threshBelowEqual(PIX threshValue);
friend IMAGE threshBelowEqual(IMAGE& img, PIX threshValue);

IMAGE threshDefined();
friend IMAGE threshDefined(IMAGE& img);
IMAGE threshEqual(PIX threshValue);
friend IMAGE threshEqual(IMAGE& img, PIX threshValue);

IMAGE tran(int u=0, int v=1);
IMAGE tran(Coord uvpair);
friend IMAGE tran(IMAGE& img, int u=0, int v=1);
friend IMAGE tran(IMAGE& img, Coord uvpair);

IMAGE transpose();
friend IMAGE transpose(IMAGE& img);

IMAGE window(IMAGE& img, PIX background = STAR);
IMAGE window(int R=0, int T=0, int M=1, int N=1, PIX background=STAR);
friend IMAGE window (IMAGE& img, IMAGE& img_2, PIX background = STAR);
friend IMAGE window (IMAGE& img, int R=0, int T=0, int M=1, int N=1, PIX background = STAR);

void writeImage(char* filename);
friend void writeImage(IMAGE& img, char* filename);

/* arithmetic and logical operators */

friend int operator==(IMAGE& img,IMAGE& img_2);
friend int operator!=(IMAGE& img,IMAGE& img_2);
friend IMAGE operator+ (IMAGE& img, IMAGE& img_2);
friend IMAGE operator+ (IMAGE& img, PIX scalar);
friend IMAGE operator+ (PIX scalar, IMAGE& img);

friend IMAGE operator- (IMAGE& img, IMAGE& img_2);
friend IMAGE operator- (IMAGE& img, PIX scalar);
friend IMAGE operator- (PIX scalar, IMAGE& img);
friend IMAGE operator- (IMAGE& img);

friend IMAGE operator* (IMAGE& img, IMAGE& img_2);
friend IMAGE operator* (IMAGE& img, PIX scalar);
friend IMAGE operator* (PIX scalar, IMAGE& img);

friend IMAGE operator/ (IMAGE& img, IMAGE& img_2);
friend IMAGE operator/ (IMAGE& img, PIX scalar);
friend IMAGE operator/ (PIX scalar, IMAGE& img);

friend IMAGE operator% (IMAGE& img, IMAGE& img_2);
friend IMAGE operator% (IMAGE& img, PIX scalar);
friend IMAGE operator% (PIX scalar,IMAGE& img);

friend IMAGE operator! (IMAGE& img);
friend int operator< (IMAGE& img,IMAGE& img_2);
friend int operator<=(IMAGE& img,IMAGE& img_2);
friend int operator> (IMAGE& img,IMAGE& img_2);
friend int operator>=(IMAGE& img, IMAGE& img_2);


This dictionary lists the functions defined for use with the Image classes of the oopsSlip system.

The following format is used in the function definitions:

a. **Title:** The bold type denotes the proper name of the routine, to be used in programming. The normal type denotes the full name;

b. **Summary:** The summary description following the title gives the general action of the routine;

c. **Format:** Calling syntax. All allowable format types are given;

d. **Inputs:** Lists the inputs to the routine, as named in the format types;

e. **Outputs:** Lists the outputs from the routine;

f. **Preconditions:** These are the constraints under which a routine will function properly, and the conditions that the routine assumes of the input. The input is not necessarily checked to see if the preconditions are true, and the output may be false if the preconditions are not true;

g. **Postconditions:** These are the properties of the state resulting from the routine's execution. This can be taken as a guarantee of the result of the routine, as long as the preconditions are satisfied;

h. **Description:** A full description of the routine. Any exceptions, warnings or other notes are included here;

j. **Status codes:** Lists the possible values of errno after execution. If none are listed, the value of errno is not changed;

m. **Source file:** Location of source code;

k. **Function prototypes:** Prototype declaration(s);

m. **Example:** Example call.
area
Image Area

Returns the absolute size of the image matrix

Format

imageName.area()
area(imageName)

Input

IMAGE imageName name of image variable

Outputs

int size of matrix (including STARs)

Preconditions

none

Postconditions

No part of the image is changed

Description

The integer returned by area is the number of columns in the image matrix multiplied by the number of rows. If the image is null, the value 0 (zero) is returned.

Status codes

none

Source file

Image.H

Function prototypes

int area();
friend int area(IMAGE& img);

Usage Example

for(int i=0; i< area(); i++) {...}
**avg**

 Pixel average

Find the average value of elements in the image

**Format**

avg(imageName)

**Input**

IMAGE imageName    name of image

**Outputs**

double    maximum element value in image

**Preconditions**

none

**Postconditions**

Input image is not changed

**Description**

The average defined element value is calculated. Undefined elements are ignored. If all elements are undefined, the output is 0.

**Status codes**

none

**Source file**

Stats.C

**Function Prototype**

double avg();
friend double avg(IMAGES& img);

**Usage Example**

double avgval = avg(im1);
**b_close**

Binary Morphological Closing  
Binary closing with a given structuring element

**Format**

\[ b\_close\ (imageName, structuringElementName) \]

**Input**

- IMAGE imageName: name of image to be acted on
- IMAGE structuringElementName: name of structuring element

**Outputs**

- IMAGE: binary closed image

**Preconditions**

none

**Postconditions**

Input image is not changed

**Description**

A grey scale morphological closing is performed on the input image. The output is `threshDefined()` to produce a binary image

**Status codes**

none

**Source file**

Morph.C

**Function Prototypes**

friend IMAGE b_close(IMAGES img, IMAGES structElem);

**Usage Example**

im3 = b_close(a, b)
**b_dilate**

Binary Dilation

**Format**

\[ b\_dilate(imageName, structuringElementName) \]

**Input**

| IMAGE imageName | name of image to be dilated |
| IMAGE structuringElementName | name of structuring element |

**Outputs**

| IMAGE | dilated binary image |

**Preconditions**

none

**Postconditions**

Input image is not changed  
Structuring element is not changed

**Description**

The input image is grey-scale dilated with the structuring element. The output is then threshDefined() to produce a binary image.

**Status codes**

none

**Source file**

Image.H

**Function Prototypes**

friend IMAGE b_dilate(IMAGE& img, IMAGE& structElem)

**Usage Example**

\[ im3 = b\_dilate(im1, se); \]
**b_eroDe**
Brainy Morphological Erosion

**Format**
\[
\text{eroDe}(\text{imageName}, \text{structuringElementName})
\]

**Input**

<table>
<thead>
<tr>
<th>IMAGE imageName</th>
<th>name of image</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE structuringElementName</td>
<td>name of structuring element</td>
</tr>
</tbody>
</table>

**Outputs**

<table>
<thead>
<tr>
<th>IMAGE</th>
<th>eroded binary image</th>
</tr>
</thead>
</table>

**Preconditions**

none

**Postconditions**

Input image is not changed
Structuring element is not changed

**Description**
The input image is grey-scale eroded with the structuring element. If the image and structuring element are binary images, then a binary erosion is performed. The output is `threshDefined()` to produce a binary image.

**Status codes**

none

**Source file**

Image.H

**Function Prototypes**

friend IMAGE b_eroDe(IMAGE& img, IMAGE& structElem)

**Usage Example**

\[
im3 = b_eroDe(im1, se);
\]
**b_open**

Binary Morphological Opening  
Binary opening with a given structuring element

**Format**

\[
b\_open(\text{imageName}, \text{structuringElementName})
\]

**Input**

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE</td>
<td>imageName</td>
</tr>
<tr>
<td>IMAGE</td>
<td>structuringElementName</td>
</tr>
</tbody>
</table>

**Outputs**

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE</td>
<td>binary closed image</td>
</tr>
</tbody>
</table>

**Preconditions**

none

**Postconditions**

Input image is not changed

**Description**

A grey scale morphological opening is performed on the input image. The output is `threshDefined()` to produce a binary image.

**Status codes**

none

**Source file**

Morph.C

**Function Prototypes**

friend IMAGE b_open(IMAGE & img, IMAGE & structElem)

**Usage Example**

\[
im3 = b\_open(a,b)
\]
card

Cardinality
Counts the number of defined pixels in the image

Format
imageName.card()
card(imageName)

Input
IMAGE imageName name of image variable

Outputs
int number of defined pixels

Preconditions
none

Postconditions
Input image is not changed

Description
The number of defined pixels is returned. A null image has a cardinality of 0 (zero).

Status codes
none

Source file
Support.C

Function Prototypes
int card();
friend int card(IMAGES & img);

Usage Example
do (i=0; i < card(test1); i++) [...]
**close**

**Morphological Closing**

Morphological closing with a given structuring element

**Format**

`close(imageName, structuringElementName)`

**Input**

- `IMAGE imageName` name of image to be acted on
- `IMAGE structuringElementName` name of structuring element

**Outputs**

- `IMAGE closed image`

**Preconditions**

None

**Postconditions**

Input image is not changed

**Description**

A grey scale morphological closing is performed on the input image.

**Status codes**

None

**Source file**

Morph.C

**Function Prototypes**

```
friend IMAGE close(IMAGE & img, IMAGE & structElem)
```

**Usage Example**

```
im3 = close(a, b)
```
**comp**

Complement image  Complement the input image

**Format**

imageName.comp()
comp(imageName)

**Input**

IMAGE imageName name of image variable

**Outputs**

IMAGE complement of input image

**Preconditions**

none

**Postconditions**

Image location and size not changed

**Description**

The defined pixels of the input image are set to undefined (*), and the undefined pixels are set to 1 (one). The COMP flag is also set. If a null image is input, a null image is returned.

**Status codes**

none

**Source file**

Arith.C

**Function Prototypes**

IMAGE comp();
friend IMAGE comp(IMAGE& img);

**Usage Example**

b = b.comp();
## convolute

**Convolute**

Convolute the input image

### Format

```
convolute(imageName, kernel)
```

### Input

- `IMAGE imageName`: name of image variable
- `IMAGE kernel`: convolution kernel

### Outputs

- `IMAGE`: convolution of input image

### Preconditions

none

### Postconditions

Input image not changed

### Description

The input image is convolved with the kernel. If a null image is input, a null image is returned.

### Status codes

- `W_NULIMGINPUT`: (warn) Null image input - no action

### Source file

Filter.C

### Function Prototypes

```
friend IMAGE convolute(IMAGE& img, IMAGE& kernel);
```

### Usage Example

```
b = convolute(b, c);
```
**COS**
Cosine
Get cosine of elements

**Format**
imageName.cos(cutoff)  
cos(imageName,cutoff)

**Input**
IMAGE imageName  name of image variable  
int cutoff  cutoff value for series expansion (optional)

**Outputs**
IMAGE  image with cos of elements

**Preconditions**
none

**Postconditions**
none

**Description**
Each element is replace by its cosine (radians). If the element is undefined, no action is done.

If the cutoff value is not given, or is 0, the C++ cos routine is used. Otherwise a series expansion to the given number of terms is performed.

**Status codes**
W_NULIMGINPUT  (warn) Null image input - no action

**Source file**
Arith.C

**Function Prototypes**
IMAGE cos(int cutoff=0);  
friend IMAGE cos(IMAGES img, int cutoff=0);

**Usage Example**

b = b.cos();
CROSS
Matrix Cross product  Compute the matrix product

Format

cross (imageName1, imageName2)

Input

IMAGE imageName1  name of image 1
IMAGE imageName2  name of image 2

Outputs

PIX  dot matrix product

Preconditions

Number of columns of im1 = number of rows in im2

Postconditions

Image locations and sizes not changed

Description

The cross product of the images is computed. If the number of columns in the first image is not the same as the number of rows in the second, a null image is returned and errno is set.

Status codes

E_DOMNOTVALID  (error) Domains not valid

Source file

Matrix.C

Function Prototypes

friend IMAGE cross(IMAGE& img1, IMAGE& img_2);

Usage Example

crossprod = cross(im1, im2);
dilate
Dilation or Minkowski Addition

Dilate image with structuring element

Format

dilate(imageName, structuringElementName)

Input

IMAGE imageName name of image to be dilated
IMAGE structuringElementName name of structuring element

Outputs

IMAGE dilated image

Preconditions

none

Postconditions

Input image is not changed
Structuring element is not changed

Description

The input image is grey-scale dilated with the structuring element.

Status codes

none

Source file

Morph.C

Usage Example

b = dilate(b,se);
dot
Dot matrix product

Compute the dot matrix product

Format

\[ \text{dot} (\text{image}1, \text{image}2) \]

Input

\[
\begin{align*}
\text{IMAGE} & \quad \text{name of image 1} \\
\text{IMAGE} & \quad \text{name of image 2}
\end{align*}
\]

Outputs

\[
\text{PIX} \quad \text{dot matrix product}
\]

Preconditions

Images are same domain, in same location in space and contain no undefined elements

Postconditions

Image locations and sizes not changed

Description

The dot matrix product of images \( f \) and \( g \) defined over a common domain \( D \) is:

\[
\sum_{(u,v) \in D} f(u,v) \times g(u,v)
\]

Both images must be at the same location in space, have the same domain and contain no undefined elements, or the value 0 (zero) is returned and \( \text{errno} \) is set.

Status codes

- \( E\text{\_DOMNOTEQUAL} \) (error) Domains not equal
- \( E\text{\_POSNOTSAME} \) (error) Image positions not identical
- \( E\text{\_UNDEFELEM} \) (error) Undefined elements in image matrix

Source file

Matrix.C

Function Prototypes

\[
\text{friend PIX} \quad \text{dot} (\text{IMAGE} & \text{ img1}, \text{ IMAGE} & \text{ img}_2);
\]

Usage Example

\[
\text{dotprod} = \text{dot} (\text{im}1, \text{ im}_2);
\]
Show Element Value

Show the value of a certain element

Format

imageName.pix(row, column)
imageName.pix(elementNumber)
ImageName.pix(elementGridLoc)
elem(imageName, row, column)
elem(imageName, elementNumber)
 elem(imageName, elementGridLoc)

Input

IMAGE imageName name of image
int row row number of element (image coord)
int column column number of element (image coord)
int elementNumber nbr of element (product of row*column)
Coord elementGridLoc absolute grid location of element

Outputs

PIX value of element

Preconditions

none

Postconditions

Image is not changed

Description

This routine shows a particular image element. The identification of the element is either by its (row, column) designation, by a count, or by its absolute grid coordinates. The count notation is useful when cycling through all the elements of an image, and denotes the i'th element when traversing the image in the normal traversal order (left to right, top to bottom). Undefined elements are included and are allowed to be shown.

If the desired element is outside the image boundaries, value STAR is returned, and an errno value is set.

Status codes

E_OUTSIDEIMAGE (error) Outside the image

Source file

Support.C

Function Prototypes

PIX elem(int i, int j);
PIX elem(int i);
PIX elem(Coord elCoord);
friend PIX elem(IMAGE* img, int i, int j);
friend PIX elem(IMAGE* img, int i);
friend PIX elem(IMAGE* img, Coord elemGridLoc);

Usage Example

offset = iml.tran(se.elemLoc(i)) + se.elem(i);
elemLoc
Show Element Grid Location

Format

imageName. elem(row, column)
imageName. elem(elementNumber)
elem(imageName, row, column)
elem(imageName, elementNumber)
elem(imageName, elementGridLoc)

Input

IMAGE imageName name of image
int row row number of element (image coord)
int column column number of element (image coord)
int elementNumber number of element (product of row*column)

Outputs

Coord absolute grid location of the element

Preconditions

none

Postconditions

Image is not changed

Description

This routine shows the grid location of a particular image element. The identification of the element is either by its (row, column) designation, or by a count. The count notation is useful when cycling through all the elements of an image, and denotes the i'th element when traversing the image in the normal traversal order (left to right, top to bottom). Undefined elements are included and are allowed to be shown.

If the desired element is outside the image boundaries, its grid location is returned, but an errno value is set.

Status codes

E_OUTSIDEIMAGE (error) Outside the image

Source file

Support.C

Function Prototypes

Coord elemLoc(int i, int j);
Coord elemLoc(int i);
friend Coord elemLoc(IMAGES img, int i, int j);
friend Coord elemLoc(IMAGES img, int i);

Usage Example

offset = iml.tran(se.elemLoc(i)) + se.elem(i);
erode
Morphological Erosion  Erosion of the input image with a structuring element

Format  erode(imageName, structuringElementName)

Input  IMAGE imageName  name of image
       IMAGE structuringElementName  name of structuring element

Outputs  IMAGE  eroded image

Preconditions  none

Postconditions  Input image is not changed
                Structuring element is not changed

Description  The input image is grey-scale eroded with the structuring element.

Status codes  none

Source file  Morph.C

Function Prototype  friend IMAGE erode(IMAGE& img, IMAGE& structElem)

Usage Example  

    c = erode(a,b);
extend
Extension

Format

imageNameDominant. extend(imageNameSubordinate)
extend(imageNameDominant, imageNameSubordinate)

Input

IMAGE imageNameDominant name of dominant image
IMAGE imageNameSubordinate name of subordinate image

Outputs

IMAGE output image

Preconditions

none

Postconditions

Dominant image is extended, subordinate image is unchanged

Description

Extension returns an image whose elements are the dominant input image, with the elements of the subordinate image added to the areas of the domain of the dominant image where the two images do not intersect. If the images do not intersect, the result is simply the union of the two domains.

Status codes

l_NOINTERSECT (Info) Images do not intersect

Source file

Extend.C

Function Prototypes

IMAGE extend(IMAGE& img_2);
friend IMAGE extend(IMAGE& img, IMAGE& img_2);

Usage Example

c = a.extend(b);
extmax
Extended Maximum

Format
extmax(imageNameF, imageNameG)

Input
IMAGE imageNameF name of first image
IMAGE imageNameG name of second image

Outputs
IMAGE output image

Preconditions
none

Postconditions
Input images are not changed

Description
An extended maximum is performed on the two images. Extended maximum of two images \( f \) and \( g \) defined on domains \( A \) and \( B \) is the maximum of two images on their intersection, unioned by the remaining domain of each of the two images:

\[
(f \oplus g)(i,j) = \begin{cases} 
  f(i,j) & (i,j) \in A - B \\
  \max(f(i,j),g(i,j)) & (i,j) \in A \cap B \\
  g(i,j) & (i,j) \in B - A 
\end{cases}
\]

If the images do not intersect, the result is simply the extension of image \( f \) by image \( g \).

Status codes
L_NOINTERSECT (Info) Images do not intersect

Source file
Image.H

Function Prototypes
friend IMAGE extmax(IMAGE& img, IMAGE& img_2);

Usage Example
c = extmax(im1, im2);
extmin
Extended Minimum

Format
extmin(imageNameF, imageNameG)

Input
IMAGE imageNameF    name of first image
IMAGE imageNameG    name of second image

Outputs
IMAGE       output image

 Preconditions
    none

Postconditions
    Input images are not changed

Description
An extended minimum is performed on the two images. Extended minimum of two images f and g defined on domains A and B is the minimum of two images on their intersection, unioned by the remaining domain of each of the two images:

\[
(f \preceq g)(i,j) = \begin{cases} 
  f(i,j) & (i,j) \in A - B \\
  \min(f(i,j), g(i,j)) & (i,j) \in A \cap B \\
  g(i,j) & (i,j) \in B - A
\end{cases}
\]

If the images do not intersect, the result is simply the extension of image f by image g.

Status codes
  l_NOINTERSECT (Info) Images do not intersect

Source file
Image.H

Function Prototypes
friend IMAGE extmin(IMAGE& img, IMAGE& img_2);

Usage Example
c = extmin(im1,im2);
**extmult**
Extended Multiplication

*Format*

`extmult(imageNameF, imageNameG)`

*Input*

| IMAGE imageNameF | name of first image |
| IMAGE imageNameG | name of second image |

*Outputs*

| IMAGE | output image |

*Preconditions*

none

*Postconditions*

Input images are not changed

*Description*

An extended multiplication is performed on the two images. Extended multiplication of two images f and g defined on domains A and B is the maximum of two images on their intersection, unioned by the remaining domain of each of the two images:

\[
(f \leftrightarrow g)(i,j) = \begin{cases} 
  f(i,j) & (i,j) \in A - B \\
  f(i,j) \times g(i,j) & (i,j) \in A \cap B \\
  g(i,j) & (i,j) \in B - A 
\end{cases}
\]

If the images do not intersect, the result is simply the extension of image f by image g.

*Status codes*

| L_NOINTERSECT | (Info) Images do not intersect |

*Source file*

Image.H

*Function Prototypes*

friend IMAGE extmult(IMAGE& img, IMAGE& img_2);

*Usage Example*

\[
c = extmult(im1, im2);
\]
**filter**

Filter

Filter the input image

**Format**

filter(imageName, mask)

**Input**

IMAGE imageName     name of image variable
IMAGE mask          filter mask

**Outputs**

IMAGE             filtered image

**Preconditions**

none

**Postconditions**

Input image not changed

**Description**

The input image is filtered with the mask. If a null image is input, a null image is returned.

**Status codes**

W_NULIMGINPUT       (warn) Null image input - no action

**Source file**

Filter.C

**Function Prototypes**

friend IMAGE filter(IMAGE& img, IMAGE& kernel);

**Usage Example**

```c
b = filter(b, c);
```
flip
Flip image over origin

Format
imageName.flip()
flip(imageName)

Input
IMAGE imageName name of image variable

Outputs
IMAGE flipped image

Preconditions
none

Postconditions
none

Description
Implementation of basis flip operator. Element values are not modified. If a null image is entered, a null image is returned.

Status codes
W_NULIMGINPUT (warn) Null image input - no action

Source file
Arith.C

Function prototypes
IMAGE flip();
friend IMAGE flip(IMAGE& img);

Usage Examples
im2 = im1.flip() + im2;
**intersect**

*Intersection*
Find the intersecting domain of two images

*Format*

```c
intersect(imageName1, imageName2)
```

*Input*

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE</td>
<td>name of first image</td>
</tr>
<tr>
<td>IMAGE</td>
<td>name of second image</td>
</tr>
</tbody>
</table>

*Outputs*

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE</td>
<td>intersecting area of input images</td>
</tr>
</tbody>
</table>

*Preconditions*

none

*Postconditions*

Input images are not changed

*Description*

The intersecting domain of the two images is calculated. The returned output is an image whose elements are all undefined (STARs), but of the location and dimensions (minimal bound matrix) of the intersecting area. If the images do not intersect, a null image is returned.

*Status codes*

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_NOINTERSECT</td>
<td>Images do not intersect</td>
</tr>
</tbody>
</table>

*Source file*

Support.C

*Function prototype*

```c
friend IMAGE intersect(IMAGE& img, IMAGE& img_2);
```

*Usage Example*

```c
imout = intersect(a,b);
```
inv
Invert matrix

**Format**

- `imageName.inv()`
- `inv(imageName)`

**Input**

- `IMAGE imageName` name of image variable

**Outputs**

- `IMAGE` inverted matrix

**Preconditions**

- Image contains no undefined (*) elements

**Postconditions**

- Input image size and location not changed

**Description**

Inverts the image matrix. If a null image is entered, a null image is returned. If there are undefined elements, a null image is returned.

**Status codes**

- `W_NULIMGINPUT` (warn) Null image input
- `E_UNDEFELEM` (error) Undefined elements

**Source file**

- `Matrix.C`

**Function prototypes**

- `IMAGE inv();`
- `friend IMAGE inv(IMAGE& img);`

**Usage Examples**

- `im2.inv();`
**isCardEqual**
**isDomainEqual**
**isHeaderEqual**

Comparison operators

**Format**

```c
imageName1.isCardEqual(imageName2)

isCardEqual(imageName1, imageName2)

imageName1.isDomainEqual(imageName2)

isDomainEqual(imageName1, imageName2)

imageName1.isHeaderEqual(imageName2)

isHeaderEqual(imageName1, imageName2)
```

**Input**

| IMAGE imageName1 | name of first image |
| IMAGE imageName2 | name of second image |

**Outputs**

int
TRUE (=1) or FALSE (=0)

**Preconditions**

none

**Postconditions**

Input images are not changed

**Description**

The corresponding features (card, domain and image header) are compared. TRUE is returned if the features are equal. This construct is used to clarify if statements.

**Status codes**

none

**Source file**

Image.H  (isCardEqual)
Support.C

**Usage Example**

```c
if (im1.isHeaderEqual(im2)) {...}
```
**isComplImage**

**isNullImage**

**Verification operators**

**Format**

- `imageName.isComplImage`
- `isComplImage(imageName)`

- `imageName.isNullImage`
- `isNullImage(imageName)`

**Input**

- IMAGE `imageName` : name of image

**Outputs**

- `int` : TRUE (=1) or FALSE (=0)

**Preconditions**

- none

**Postconditions**

- Input image is not changed

**Description**

The image feature is checked. This construct is used to clarify if statements.

- `isComplImage` returns the value of the image COMP flag. TRUE means the image is a complemented image (all elements outside matrix are assumed to be equal to 1).

- `isNullImage` first checks if the image is equal to one of the nullIMAGE structures. If it is, or if all elements are undefined (even if the matrix has dimensions other than 0 rows and columns), the function returns TRUE.

**Status codes**

- none

**Source file**

- Image.H (isComplImage)
- Support.C (isNullImage)

**Usage Example**

```c
if (im1.isComplImage) {...}
```
loc
Show Image Location  Show image location in the absolute grid

Format
imageName.loc()
loc(imageName)

Input
IMAGE      imageName   name of image variable

Outputs
Coord      (row, column) location of image

Preconditions
none

Postconditions
Input image is not changed

Description
The (row, column) location of the image is returned.

Status codes
none

Source file
Image.H

Function prototype
Coord loc();
friend Coord loc(IMAGES img);

Usage Example
RealImage im2(im1.loc(), 0);
**max**

Image maximum

Find the maximum value of elements on the intersection

**Format**

- `imageName1.max (imageName2)`
- `max (imageName1, imageName2)`
- `imageName1.max ()`
- `max (imageName1)`

**Input**

- `IMAGE imageName1` name of image 1
- `IMAGE imageName2` name of image 2

**Outputs**

- `IMAGE` maximum image
- `PIX` maximum element value in image

**Preconditions**

none

**Postconditions**

Input images are not changed

**Description**

Two types are functions are possible, depending on the number of input images:

1. **2 input images (image maximum)**: The output is the maximum pixel values between the two images, where they intersect. If they do not intersect, a null image is returned.

2. **1 input image (maximal element value)**: The output is the largest element value in the image matrix.

**Status codes**

- L_NOINTERSECT (info) Images do not intersect

**Source file**

- `Arith.C` (image maximum)
- `Stats.C` (maximum element value)

**Function prototypes**

```c
PIX max ();
IMAGE max (IMAGE& img_2);
friend PIX max (IMAGE& img);
friend IMAGE max (IMAGE& img, IMAGE& img_2);
```

**Usage Example**

```c
RealImage maximg = max (im1, im2);
double maxval = max (im1);
```
**min**

**Image minimum**

Find the minimum value of elements on the intersection.

**Format**

\[
\begin{align*}
\text{imageName1.min (imageName2)} \\
\text{min (imageName1, imageName2)} \\
\text{imageName1.min ()} \\
\text{min (imageName1)}
\end{align*}
\]

**Input**

\[
\begin{align*}
\text{IMAGE imageName1} & \quad \text{name of image 1} \\
\text{IMAGE imageName2} & \quad \text{name of image 2}
\end{align*}
\]

**Outputs**

\[
\begin{align*}
\text{IMAGE} & \quad \text{minimum image} \\
\text{PIX} & \quad \text{minimum element value in image}
\end{align*}
\]

**Preconditions**

none

**Postconditions**

Input images are not changed

**Description**

Two types are functions are possible, depending on the number of input images:

2 input images (image minimum): The output is the minimum pixel values between the two images, where they intersect. If they do not intersect, a null image is returned.

1 input image (minimal element value): The output is the largest element value in the image matrix.

**Status codes**

l_NOINTERSECT (info) Images do not intersect

**Source file**

Arith.C (image minimum)
Stats.C (minimum element value)

**Function prototypes**

\[
\begin{align*}
\text{PIX min ();} \\
\text{IMAGE min (IMAGE\& img_2);} \\
\text{friend PIX min (IMAGE\& img);} \\
\text{friend IMAGE min (IMAGE\& img, IMAGE\& img_2);} \\
\end{align*}
\]

**Usage Example**

\[
\begin{align*}
\text{RealImage minimg = min (im1, im2);} \\
\text{double minval = min (im1);} \\
\end{align*}
\]
minbound
Minimum bound matrix    Find the corresponding minimum bound matrix

Format
imageName.minbound()      minbound(ImageName)

Input
IMAGE imageName           name of image

Outputs
IMAGE                    minimum bound matrix

Preconditions
none

Postconditions
Input image elements are not changed
Dimensions of the input matrix are changed to that of the bound matrix
Image location is not changed (but image grid coordinates changed to
reflect new image matrix dimensions)

Description
The minimum bound matrix of an image is the minimum matrix that
contains all defined elements of the image.
If a null image is input, a null image is output.
If an image is already a minimum bound matrix, no action is done.

Status codes
W_NULLIMGINPUT            (warning) Null image was input - no action

Source file
Support.C

Function Prototype
IMAGE minbound();
friend IMAGE minbound(IMAGE& img);

Usage Example
    c = c.minbound();
**minksub**

**Minkowski Subtraction**  
Do a minkowski subtraction on the image

**Format**

\[
\text{minksub}(\text{imageName}, \text{structuringElementName})
\]

**Input**

- \text{IMAGE}\ image\Name\  \text{name of image}
- \text{IMAGE}\ structuringElementName\  \text{name of structuring element}

**Outputs**

- \text{IMAGE}\  \text{output image}

**Preconditions**

none

**Postconditions**

Inputs are not changed

**Description**

Minkowski subtraction is an erosion where the structuring element is rotated 180 degrees about the origin before the operation.

**Status codes**

none

**Source file**

Image.H

**Function Prototype**

friend \text{IMAGE}\ minksub(\text{IMAGE} & img, \text{IMAGE} & structElem)

**Usage Example**

\[
c = \text{minksub}(a, b);
\]
nextElemLoc
Next Element Location  Find the grid location of the next defined element

Format

imageName.nextElemLoc(gridRow, gridColumn)
imageName.nextElemLoc(GridLocation)
nextElemLoc(imageName, gridRow, gridColumn)
nextElemLoc(imageName, GridLocation)

Input

IMAGE imageName   name of image
int gridRow   absolute row number of present grid location
int gridColumn  absolute column number of present grid location
Coord gridLocation  grid coordinates of present location

Outputs

Coord   absolute coordinates of next defined element

Preconditions

none

Postconditions

Input image is not changed

Description

This routine returns the absolute grid location of the next defined element, following the grid coordinates (or row and column number) input. Image traversal is done row by row, column by column (i.e. left to right, top to bottom).
If no coordinates are input, then the routine returns the coordinates of the first defined element.
If no further elements are defined past the given coordinates, and the given coordinates are in the bound matrix domain, the first element location is returned, and a warning flag is set.
If the given coordinates are outside the image domain, the following procedure is followed:
- the row is traversed until a defined element is found in that row;
- if there are no defined elements in the row, the first defined element in the traversal order is returned;
- if the given coordinates are below the image and outside the bound matrix domain, the first defined element location is given.

This procedure is used in traversing through the domain of an image.

Status codes

L_BACKTOTOP (info) One pass through image, returning to top

Source file

Support.C
Function prototypes

Coord nextElemLoc(int gridRow=INT_MIN,
                   int gridColumn=INT_MIN);
Coord nextElemLoc(Coord elemGridLoc);
friend Coord nextElemLoc(IMAGES img,
                         int gridRow=INT_MIN, int gridColumn=INT_MIN);
friend Coord nextElemLoc(IMAGES img,
                         Coord elemGridLoc);

Usage Example

Coord here=iml.nextElemLoc(); /* gets 1st location */
for(int i=0;i<iml.card();
    i++, here=iml.nextElemLoc(here))
{ /* cycles through domain of image */ }
Rotation by 90, 180 or 270 degrees about the grid origin

Format

imageName.ninety()
ninety(imageName)

imageName.ninety2()
ninety2(imageName)

imageName.ninety3()
ninety3(imageName)

Input

IMAGE imageName name of image

Outputs

IMAGE rotated image

Preconditions

none

Postconditions

Input image element values are not changed (although locations are)

Description

The image matrix is rotated 90, 180 or 270 degrees about the absolute grid origin.

Status codes

none

Source file

Image.C (ninety)
Image.H (ninety2, ninety3)

Function Prototypes

IMAGE ninety();
friend IMAGE ninety(IMAGE& img);

IMAGE ninety2();
friend IMAGE ninety2(IMAGE& img);

IMAGE ninety3();
friend IMAGE ninety3(IMAGE& img);

Usage Example

c = c.ninety();
**normalize**

Requantize an image

Adjust the grey values from 0 to a given max value

**Format**

imageName.normalize(maxGreyValue)

normalize(imageName, maxGreyValue)

**Input**

<table>
<thead>
<tr>
<th>IMAGE</th>
<th>imageName</th>
<th>name of image</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>maxGreyValue</td>
<td>maximal desired grey value</td>
</tr>
</tbody>
</table>

**Outputs**

| IntImage | normalized image |

**Preconditions**

Image is non-nul

**Postconditions**

Image location and size not changed

**Description**

The grey values of the image are distributed linearly between 0 and maxGreyValue. If this value is not specified, the default is 255.

Any undefined elements (*) are set to 0 in the output.

This function is used to process images before saving to a file for display.

**Status codes**

none

**Source file**

Fileops.C

**Function Prototypes**

IntImage normalize(int maxGreyValue=255);

friend IntImage normalize(IMAGES img,
int maxGreyValue=255);

**Usage Example**

```c
imout = imout.normalize(512);
```
**numCols**  
**numRows**  
Number of Columns and Rows  

**Format**  
```
imageName.numCols()  
numCols(imageName)  
```
```
imageName.numRows()  
numRows(imageName)  
```

**Input**  
```
IMAGE imageName  
name of image
```

**Output**  
```
int  
number of columns or rows in the image
```

**Preconditions**  
```
none
```

**Postconditions**  
```
Input image is not changed
```

**Description**  
```
umCols() returns the number of columns in the image (corresponds to the value N in the bound matrix notation). Columns consisting only of undefined pixels (*) are also included, if the image is not minbound.
```
```
umRows() returns the number of rows in the image (corresponds to the value M in the bound matrix notation). Rows consisting only of undefined pixels (*) are also included, if the image is not minbound.
```

**Status codes**  
```
none
```

**Source file**  
```
Image.H
```

**Function Prototypes**  
```
int numCols();  
friend int numCols(IMAGES img);  

int numRows();  
friend int numRows(IMAGES img);  
```

**Usage Example**  
```
int area = im1.numRows() * im1.numCols()
```
open
Morphological Opening
Opening with a given structuring element

Format
open (imageName, structuringElementName)

Input
IMAGE imageName name of image to be acted on
IMAGE structuringElementName name of structuring element

Outputs
IMAGE closed image

Preconditions
none

Postconditions
Input image is not changed

Description
A grey scale morphological opening is performed on the input image.

Status codes
none

Source file
Morph.C

Function Prototypes
friend IMAGE open(IMAGES img, IMAGES structElem)

Usage Example
im3 = open(a,b)
print
println

Print
Print the image in a number matrix form

Format

imageName.print()
print(imageName)

imageName.printHeader()
printHeader(imageName)

Input
IMAGE imageName name of image

Output
no values returned
sends output to standard output

Preconditions
none

Postconditions
Input image is not changed

Description
print()
The address, header and element values of the image are printed. The image is presented in matrix form. The output is sent to whatever device is designated as standard output. If image exceeds 16 rows by 16 columns, only the image header is printed.

printHeader()
The address, location, size of image and values of image flags (the image header) is printed.

Status codes
none

Source file
Support.C

Function Prototypes
void print();
friend void print(IMAGE& img);

void printHeader();
friend void printHeader(IMAGE& img);
Usage Example

test1.print();  /* message call format */

Output:

IntImage at -> c0020330
R= 3 T= 4 COMP flag = 0
matrix at -> 80883304 M= 2 N = 3
-1 -1 1
-1 -1 -1
**readImage**

**Read image from file**

**Format**

readImage(filename)

**Input**

char* filename  
Name of file to be read

**Outputs**

IMAGE  
image read from file

**Preconditions**

none

**Postconditions**

none

**Description**

This procedure creates an image from the data in a file created by writeImage(). The file format is described in the documentation for writeImage().

If the image in the file is of a different type than the type desired by the function call, conversion is made for each element according to the conversion rules for C++ (see [Str86]).

If there is an error in reading the image, an error flag is set and a null image is returned.

**Status codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_CANTOPENFILE</td>
<td>(error) Can't open file, null image returned</td>
</tr>
<tr>
<td>E_BADINFOINFILE</td>
<td>(error) Bad information in file, null image returned</td>
</tr>
<tr>
<td>E_DIMDONTMATCH</td>
<td>(error) Dimensions exceed given number of elements, null image returned</td>
</tr>
</tbody>
</table>

**Source file**

Fileops.C

**Function Prototypes**

```
IMAGE readImage(char* filename);
frend IMAGE readImage(IMAGE& im1, char* filename);
```

**Usage Example**

```
im1.readImage("im1file.chr");
```
recip
Reciprocal
Sets all element values to their reciprocal value

Format
imageName.recip()
recip(imageName)

Input
IMAGE imageName name of image variable

Outputs
IMAGE all elements set to reciprocal value

Preconditions
none

Postconditions
Input image size and location not changed

Description
All defined elements in the input image are set to their reciprocal value. Any undefined elements are not modified. If a null image is entered, a null image is returned.

Status codes
none

Source file
Arith.C

Function prototypes
IMAGE recip();
friend IMAGE recip(IMAGE& img);

Usage Examples
im2 = iml.recip() * im2;
reflect
Grey Scale Reflection  Rotate image 180 degrees and inverse

Format
imageName.reflect()
reflect(imageName)

Input
IMAGE imageName  name of image variable

Outputs
IMAGE  reflected image

Preconditions
none

Postconditions
Number of defined elements stay the same

Description
Grey scale reflection consists of rotating the image 180 degrees about the absolute grid origin, then finding its additive inverse (sub()). If a null image is input, a null image is returned and errno is set.

Status codes
W_NULIMGINPUT  (warning) Null image input - no action taken

Source file
Image.H

Function Prototype
IMAGE reflect();

Usage Example
c = b + a.reflect();
**setAll**  
Set all elements in the image to a given value

**Format**

imageName.setAll(elementValue)  
setAll(imageName, elementValue)

**Input**

IMAGE imageName name of image  
PIX elementValue value at which all elements must be set

**Outputs**

IMAGE reset image

**Preconditions**

none

**Postconditions**

Input image dimensions and location are not changed

**Description**

This routine sets all the elements in the image (including STARs) to the value elementValue. If no value is given, all elements are set to STAR. If the input image is null, a null image is returned and an errno flag set.

**Status codes**

W_NULIMGINPUT (warning) Null image input - no action

**Source file**

Support.C

**Function Prototype**

IMAGE setAll(PIX val=STAR);  
friend IMAGE setAll(IMAGE& im1, PIX val=STAR);

**Usage Example**

IntImage im1;  
im1.setAll(im2, 0);
**setElem**  
Set Element Value  
Set a given element to a certain value

**Format**  

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imageName.setElem(row, column, elementValue)</td>
<td>Set element to a certain value within the image specified by the row and column coordinates.</td>
</tr>
<tr>
<td>imageName.setElem(elementNumber, elementValue)</td>
<td>Set element to a certain value by specifying its number in the image, calculated as row*column.</td>
</tr>
<tr>
<td>imageName.setElem(elementGridLoc, elementValue)</td>
<td>Set element to a certain value by specifying its absolute grid location.</td>
</tr>
<tr>
<td>setElem(imageName, row, column, elementValue)</td>
<td>Set element to a certain value within a given image.</td>
</tr>
<tr>
<td>setElem(imageName, elementNumber, elementValue)</td>
<td>Set element to a certain value by specifying its number in the image.</td>
</tr>
<tr>
<td>setElem(imageName, elementGridLoc, elementValue)</td>
<td>Set element to a certain value by specifying its absolute grid location.</td>
</tr>
</tbody>
</table>

**Input**  

- **IMAGE**  
  - imageName: name of image  
  - int row: row number of element  
  - int column: column number of element  
  - PIX elementValue: value to set element at  
  - int elementNumber: number of element (product of row*column)  
  - Coord elementGridLoc: absolute grid location of element

**Outputs**

- none (element value is set)

**Preconditions**

- none

**Postconditions**

- Image dimensions and location is not changed

**Description**

This routine sets a particular image element to the given value. The identification of the element is either by its (row, column) designation, by a count, or by its absolute grid coordinates. The count notation is useful when cycling through all the elements of an image, and denotes the ith element when traversing the image in the normal traversal order (left to right, top to bottom). Undefined elements are included and are allowed to be set.

If the desired element is outside the image boundaries, the element is not set, and an errno value is set.

**Status codes**

- E_OUTSIDEIMAGE (error) Outside the image - no action taken

**Source file**

- Support.C

**Function Prototype**

- void setElem(int i, int j, PIX val);
- void setElem(int i, PIX val);
- void setElem(Coord elCoord, PIX val);
- friend void setElem(IMAGES img, int i, int j, PIX val);
- friend void setElem(IMAGES img, int i, PIX val);
- friend void setElem(IMAGES img, Coord elemGridLoc, PIX val);

**Usage Example**

- im2.setElem(i, j, STAR);
setImageLoc
Set Image Location  Set image coordinates

Format

```
imageName.setImageLoc(gridRow, gridColumn)
imageName.setImageLoc(gridLocation)

setImageLoc(imageName, gridRow, gridColumn)
setImageLoc(imageName, gridLocation)
```

Input

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE</td>
<td>imageName</td>
</tr>
<tr>
<td>int</td>
<td>gridRow</td>
</tr>
<tr>
<td>int</td>
<td>gridColumn</td>
</tr>
<tr>
<td>Coord</td>
<td>gridLocation</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(image location coordinates set)</td>
</tr>
</tbody>
</table>

Preconditions

none

Postconditions

Input image dimensions and contents are not changed

Description

This routine resets the image location coordinates (R and T in mound matrix notation) to the new values given.

Status codes

none

Source file

Image.H

Function prototypes

```
void setImageLoc(int tt, int rr);
void setImageLoc(Coord imloc);
friend void setImageLoc(IMAGE& img, int tt, int rr);
friend void setImageLoc(IMAGE& img, Coord imloc);
```

Usage Example

```
Coord here = se.nextElemLoc();
iml.setImageLoc(here);
```
**sin**

Sine

Get sine of elements

**Format**

imageName.sin(cutoff)

sin(imageName,cutoff)

**Input**

<table>
<thead>
<tr>
<th>IMAGE imageName</th>
<th>name of image variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>int cutoff</td>
<td>cutoff value for series expansion (optional)</td>
</tr>
</tbody>
</table>

**Outputs**

| IMAGE | image with sin of elements |

**Preconditions**

none

**Postconditions**

none

**Description**

Each element is replace by its sine (radians). If the element is undefined, no action is done.

If the cutoff value is not given, or is 0, the C++ sin routine is used. Otherwise a series expansion to the given number of terms is performed.

**Status codes**

W_NULIMGINPUT (warn) Null image input - no action

**Source file**

Arith.C

**Function Prototypes**

IMAGE sin(int cutoff=0);
friend IMAGE sin(IMAGE& img, int cutoff=0);

**Usage Example**

b = b.sin();
<table>
<thead>
<tr>
<th>Additive inverse</th>
<th>Negate all defined elements in image</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format</strong></td>
<td><strong>imageName</strong>.sub()</td>
</tr>
<tr>
<td></td>
<td><strong>sub</strong>(imageName)</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td><strong>IMAGE imageName</strong> name of image variable</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
<td><strong>IMAGE</strong> additive inverse of input</td>
</tr>
<tr>
<td><strong>Preconditions</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Postconditions</strong></td>
<td>All defined values are replaced by their negative value</td>
</tr>
<tr>
<td></td>
<td>Undefined elements are not changed</td>
</tr>
<tr>
<td></td>
<td>The location and dimensions of the image is not changed</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The additive inverse is the image where each defined element is replaced by its product with (-1).</td>
</tr>
<tr>
<td><strong>Status codes</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Source file</strong></td>
<td>Arith.C</td>
</tr>
<tr>
<td><strong>Function prototype</strong></td>
<td><strong>IMAGE sub();</strong></td>
</tr>
<tr>
<td></td>
<td>friend <strong>IMAGE sub</strong>(IMAGE&amp; im1);</td>
</tr>
<tr>
<td><strong>Usage Example</strong></td>
<td><strong>c = extadd(a, sub(b));</strong></td>
</tr>
</tbody>
</table>
**Sum**

Pixel sum

Sum all defined elements in image

**Format**

imageName.sum()

sum(imageName)

**Input**

IMAGE imageName name of image variable

**Outputs**

PIX sum of elements in input

**Preconditions**

none

**Postconditions**

The location and dimensions of the image is not changed

**Description**

All defined values are summed. Undefined elements are ignored. If image is a null image, 0 is returned

**Status codes**

W_NULIMGINPUT (warn) Null image input

**Source file**

Arith.C

**Function prototype**

PIX sum();
friend PIX sum(IMAGE& im1);

**Usage Example**

c = extadd(a, sub(b));
Sqr
Sqrt
Square, Square Root

Format

imageName.sqr()
sqr(imageName)

imageName.sqrt()
sqrt(imageName)

Input

IMAGE imageName name of image

Outputs

IMAGE output image

Preconditions

sqrt: no elements have a negative value

Postconditions

Input image dimensions and location are not changed
Undefined elements are not touched

Description

The defined elements are replaced by their squared value, for \texttt{sqr()},
and their square root, for \texttt{sqrt()}. Images of type \texttt{IntImage} will have
truncated square roots (recommended to avoid use).

If the input image to \texttt{sqrt} has negative element values, a null image is
returned.

Status codes

E_NEGELEMVAL (error) Negative element values

Source file

Image.H (sqr)
Arith.C (sqrt)

Function prototypes

IMAGE sqr();
friend IMAGE sqr(IMAGE& img);

IMAGE sqrt();
friend IMAGE sqrt(IMAGE& img);

Usage Example

sumofsqrs = sum(im1.sqr());
thresh
threshAbove
threshBelow
threshBelowEqual
threshBetween
threshDefined
threshEqual

Threshold
Threshold the image at a certain level

Generic Format

imageName.thresh (threshValue)
thresh (imageName, threshValue)

imageName.threshBetween (threshValue1, threshValue2)
threshBetween (imageName, threshValue1, threshValue2)

Input

IMAGE imageName name of image
PIX threshValue value to be thresholded at

Outputs

IMAGE thresholded image

Preconditions

none

Postconditions

Input image dimensions and location are not changed

Description

The thresh series of functions sets to 1 all elements that meet the requirements, and sets to 0 all other elements. Undefined elements are not affected:

thresh all values >= threshValue
threshAbove all values > threshValue
threshBetween all values > threshValue1 and < threshValue2
threshBelow all values < threshValue
threshBelowEqual all values <= threshValue
threshDefined all defined values (no argument)
threshEqual all values = threshValue

Status codes

none

Source file

Arith.C

Function prototypes

IMAGE thresh(PIX threshValue);
friend IMAGE thresh(IMAGE& img, PIX threshValue);
IMAGE threshAbove(PIX threshValue);
friend IMAGE threshAbove(IMAGES img, PIX threshValue);

IMAGE threshBetween(PIX loVal, PIX hiVal);
friend IMAGE threshBetween(IMAGES img, PIX loVal, PIX hiVal);

IMAGE threshBelow(PIX threshValue);
friend IMAGE threshBelow(IMAGES img, PIX threshValue);

IMAGE threshBelowEqual(PIX threshValue);
friend IMAGE threshBelowEqual(IMAGES img, PIX threshValue);

IMAGE threshDefined();
friend IMAGE threshDefined(IMAGES img);

IMAGE threshEqual(PIX threshValue);
friend IMAGE threshEqual(IMAGES img, PIX threshValue);
**tran**  
**Translate**  
Translate image

**Format**

\[ \text{imageName}.\text{tran}(\text{nbrOfRows, nbrOfColumns}) \]
\[ \text{imageName}.\text{tran}(\text{rowColumnPair}) \]

\[ \text{trans} (\text{imageName, nbrOfRows, nbrOfColumns}) \]
\[ \text{trans} (\text{imageName, rowColumnPair}) \]

**Input**

- IMAGE imageName name of image
- int nbrOfRows number of rows to translate (+up, -down)
- int nbrOfColumns number of columns to translate (+right, -left)
- Coord rowColumnPair row/column coordinate pair

**Outputs**

- IMAGE translated image

**Preconditions**

none

**Postconditions**

Input image dimensions and contents are not changed

**Description**

This routine moves the image to a new location in the absolute grid by adding the nbrOfRows and nbrOfColumns values to the image coordinates.

**Status codes**

none

**Source file**

Image.H

**Function Prototypes**

\[ \text{IMAGE \ trans}(\text{int u=0, int v=1}); \]
\[ \text{IMAGE \ trans}(\text{Coord uvpair}); \]
\[ \text{friend IMAGE \ trans}(\text{IMAGE& img, int u=0, int v=1}); \]
\[ \text{friend IMAGE \ trans}(\text{IMAGE& img, Coord uvpair}); \]

**Usage Example**

b.\text{tran}(\text{here});
**window**

**Window**

Restrict the image to a certain window

**Format**

imageName.window(restrictingImage)
imageName.window(newColNbr, newRowNbr, nbrOfRows, nbrOfCols)

window(imageName, restrictingImage)
window(imageName, newColNbr, newRowNbr, nbrOfRows, nbrOfCols)

**Input**

<table>
<thead>
<tr>
<th>IMAGE</th>
<th>imageName</th>
<th>name of image</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE</td>
<td>restrictingImage</td>
<td>image to restrict to the domain of</td>
</tr>
<tr>
<td>int</td>
<td>newColNbr</td>
<td>new coordinates of image</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(grid column number of top left-hand corner)</td>
</tr>
<tr>
<td>int</td>
<td>newRowNbr</td>
<td>new coordinates of image</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(grid row number of top left-hand corner)</td>
</tr>
<tr>
<td>int</td>
<td>nbrOfRows</td>
<td>number of rows in new image</td>
</tr>
<tr>
<td>int</td>
<td>nbrOfCols</td>
<td>number of columns in new image</td>
</tr>
</tbody>
</table>

**Outputs**

| IMAGE | imageName | windowed image |

**Preconditions**

none

**Postconditions**

none

**Description**

window() restricts the image domain to that of a second given image, or to the given coordinates and dimensions. If the given image/dimensions are larger than the input image, then the remaining area is filled with undefined elements (STARS). The restricting image does not have to be fully defined, window() will only transfer the actual domain of the image.

**Status codes**

none

**Source file**

Support.C

**Function Prototypes**

IMAGE window(IMAGES img, PIX background = STAR);
IMAGE window(int R=0, int T=0, int M=1, int N=1, PIX background=STAR);
friend IMAGE window (IMAGE& img, IMAGE& img_2, PIX background = STAR);
friend IMAGE window (IMAGE& img, int R=0, int T=0, int M=1, int N=1, PIX background = STAR);

**Usage Example**

```
b = a.window(c);
```
**writelImage**

**Write Image**

Write image matrix to a file.

**Format**

```plaintext
imageName.writeImage(filename)
writelImage(ImageName, filename)
```

**Input**

- `IMAGE imageName` name of image
- `char* filename` name of file to write to

**Outputs**

None (image matrix written to stdout)

**Preconditions**

None

**Postconditions**

Input image is not changed.

**Description**

This routine writes an image matrix to a file as a stream of ASCII characters. The first four numbers (must be integers) are the row and column grid coordinates of the image, followed by the number of rows in the image and the number of columns in the image.

The element values then follow separated by spaces.

Undefined elements are stored as *

**Status codes**

- `E_CANTOPENFILE` (error) Cant open file, no image written
- `E_WRITERR` (error) Unknown writing error, no image written

**Source file**

Fileops.C

**Function prototype**

```plaintext
void writeImage(char* filename);
friend void writeImage(Image& img, char* filename);
```

**Usage Example**

```plaintext
iml.writeImage("testfile.chr");
```
REFERENCES


DGS89 Dougherty, Edward R., D.N. Gupta and Paramjit Sehdev. "Structured Language for Image Processing." Electronic Imaging West '89, Pasadena, CA, pp. 301-305


Linowes, Jonathan S. "It's an Attitude." Byte, August 1988, pp. 219-224

Lippman, Stanley B. C++ Primer. Reading, MA: Addison-Wesley, 1989

Lougheed, Robert M. "An Overview of Greyscale Morphological Filters." Electronic Imaging West '89, Pasadena, CA, pp. 285-289


Marineau, Peter and Michael M. Skolnik. "Using Object-Oriented Languages for Mathematical Morphology." Electronic Imaging 89, Boston, 1989, pp. 284-288


