VISPACK

Ross T. Whitaker

The Scientific Computing and Imaging Institute
University of Utah

Contact information:
E-Mail: whitaker@cs.utah.edu
WWW: http://www.cs.utah.edu/ whitaker

University of Utah, School of Computing, Technical Report: **UUCS 08-0011**

Acknowledgments – This work is supported by the Office of Naval Research grants N00014-97-0227 and N00014-01-10033 and National Science Foundation grant CCR0092065. Input from many students, colleagues and collaborators, including: Josh Cates and Samuel Burgiss Jr.

Abstract

VISPACK (volume-image-surface package) is a C++ library that includes matrix, image, and volume objects and tools for manipulating level-set surface models.

Contents

1 Introduction **6**

2 Design Attributes **6**

2.1 Copy on Write or Data Handles 6

2.2 Direct Data Access 8

2.3 Use of Templating 9

2.4 Use of Standard File Formats 10

2.5 Floating Point File Format 10

2.6 Operator Overloading 10

2.7 Image and Volume Processing Functions and Algorithms 10

3 VISMatrix Library **10**

3.1 Functionality 11

3.1.1 Basic Functionality 11

3.1.2 The VISVector Class 12

3.1.3 Use of LAPACK 12

3.2 Reference for the VISMatrix Library 13

3.2.1 Interacting with the Data 14

3.2.2 Overloaded VISMatrix/VISVector Operators 17

3.2.3 General Functions 19

3.2.4 VISMatrix Specific Functions 20

3.2.5 VISVector Specific Functions 21

3.2.6 Creating VISVectors from Matrices and Vise-a-Versa 21

4 Image Library **22**

4.1 Functionality and Implementation 22

4.1.1 Channels in Images 22

4.1.2 General Functions 23

4.1.3 Image Processing Functions 24

4.2 Reference for Image Library 24

4.2.1 Interacting with the Data 24

4.2.2 Overloaded Image Operators 27

4.2.3 Math Functions 29

4.2.4 General Image Processing Functions 30

4.2.5 Image Processing Algorithms 32

5 Volume Library **33**

5.1 Functionality 33

5.1.1 General Functions 34

5.1.2 Volume Processing Functions 34

5.2 Reference for Volume Library 34

5.2.1 Interacting with the Data 35

5.2.2 Overloaded Volume Operators 37

5.2.3 Math Functions 39

5.2.4 General Volume Processing Functions 40

5.2.5 Volume Processing Algorithms 41

6 Util Library **41**

6.1 Contents 41

6.2 Reference for Util Library 42

6.2.1 Array Object 42

6.2.2 List Object 44

6.2.3 Math Functions 44

7 Level-Set Surface-Modeling Library **45**

7.1 Surface Deformation 46

7.2 Structure and Philosophy of the LSSM Library 47

7.3 The `LevelSetModel` Object 47

7.4 The `VoxModel` Object 48

7.5 Example: 3D Shape Metamorphasis 48

7.6 Morph.h 49

7.7 Morph.C 50

7.8 Main.C 52

A Parameter File Objects **55**

A.1 Introduction 55

A.2 Implementation 55

A.3 Parameter file format 55

A.4 The `ParameterFile` object 56

B Code Examples **59**

B.1 `matrix` example 61

B.2 `shotnoise` example 66

B.3 `fractal` example 67

1 Introduction

VISPACK consists of five libraries: VISMatrix, Image, Volume, Util, and Level-Set Surface-Modeling. These libraries can be used separately or together when creating applications. Because VISPACK itself is divided into libraries we divide our discussion of VISPACK in this same way. Not only is this division natural for the libraries (e.g., a matrix is much different from an image and they should not be included in the same library), but this division also means that users need only understand the workings of the particular library they wish to use. We divide this document according to the libraries so that users can quickly find needed information.

This format results in some redundancy, because there are similarities between the libraries. However, we have written this document so that these redundancies may easily be overlooked. For instance, access methods to ordered data structures (e.g. array or matrix) have similar conventions, but the descriptions of these methods are repeated for each object so that this document can be used as a reference.

2 Design Attributes

We set forth three design goals for this library: usability, efficiency, and functionality. The philosophy behind the usability goals is that the library should provide an environment in which users can easily implement solutions to real problems. Our efficency goals are centered around the idea that the user should have access to all the available resources of his/her machine. Our functionality goals ensure thoroughness in the library's functionality. Table 1 shows our goals and the respective design attributes that support each goal.

Table 1 indicates eight design attributes. In this section we give a discussions of these attributes. The attributes presented are copy on write, direct data access, use of templating, use of standard file formats, floating point file format, operator overloading, and image and volume processing functions and algorithms.

2.1 Copy on Write or Data Handles

We use the idea of copy on write, or data handles, to make VISPACK more efficient for large data sets. The advantage of using this technique is that handles can be passed to and from functions rather than coping all

Table 1: Goals and Design Attributes of VISPACK

Category	Goal	Design Attribute
usability	quick creation of test applications	overloaded operators,copy on write
	creation of large stable applications that maintain effiency	copy on write, direct data access
	easy and accurate saving/loading of data	use of standard file formats floating point image files
efficiency	conservation of memory	copy on write, use of templates
	efficient implementation of multi-access functions	copy on write
	limit speed to that of machine	direct data access
functionality	basic matrix, image, and volume	overloaded operators
	image and volume processing	implement functions and algorithms
	ability to create arrays of matricies, images, and volumes	templating

the data for an object to pass to and from functions. Thus, copies (through either the copy constructor or assignment operator) are can be treated as *deep copies* which are deferred until they become necessary by the copy-on-write protocol.

To better understand this we examine an example. This example is the addition of three matrices and then the assignment of that result to a final matrix. Although this example uses matrices, we use this technique for images and volumes as well. An efficient way to implement the addition is the following [1]:

```
VISMatrix A(3,3),B(3,3),C(3,3);
A=1.0f; //every element in A is 1
B=2.0f; //every element in B is 2
C=3.0f; //every element in C is 3
VISMatrix ans;
for (int r=0;r<A.rows();r++)
    for (int c=0;c<A.cols();c++)
       ans.poke(r,c) = A(r,c)+B(r,c)+C(r,c);
```

This code is efficient but not convenient to implement because the user must write loops for each operation that he/she wishes to perform. One of our goals is to create a library containing methods that allow quick creation of applications. So we create a more intuitive method of implementing this operation by overloading the `=` and `+` operators.

Overloading these two operators allows the code above to be written as simply:

```
VISMatrix A(3,3),B(3,3),C(3,3);
A=1.0f; //every element in A is 1
B=2.0f; //every element in B is 2
C=3.0f; //every element in C is 3
VISMatrix ans;
ans = A+B+C; //pass copies of the data or handles?
ans.operator=((A.operator+(B)).operator+(C));//same as above line
```

Consider the implementation of the line of code `ans=A+B+C` if data handles are not used. Several functions are called; first `A.operator+(B)` is called. The result of this function must be passed on to `operator+(C)`. A copy of the result is created by the compiler and then passed to `operator+(C)`, and after calculations are completed the result from `operator+(C)` is copied and sent to `ans.operator=`.

Suppose that instead of coping all the data to send to the next function we let the computer copy a *handle* to the data and pass this to the next function. We do this by creating a class called `VISMatrixData`. This class contains a reference count (more on this later) and the data of the matrix. The `VISMatrix` object then contains a pointer to a `VISMatrixData` object. Therefore the `VISMatrix` object is simply a handle to the `VISMatrixData` object. The handle is, with this setup, copied and passed from one function to the next (thus, improving the performance of the library–especially for large matricies).

Suppose we have two matricies that are equal to one another. Acting as handles, shouldn't these matricies point to the same `VISMatrixData` object? They are indeed the same. In the implementation, such matrices can share data and maintain consistency using a reference counter.

The `VISMatrixData` class contains a reference count, `refcnt`, and the data for its matrix (or matricies as

[1]In this example the statement `M=f`, where `M` is a matrix and `f` is a floating point value, sets every value in `M` to `f`. The `poke` function assigns a value to the element of the matrix being operated on. The element is indicated by row `r` and column `c`. The `operator()` function returns the value of an element of a matrix: the element is indicated by row `r` and column `c`. (see section 3.2 for further explanation).

Figure 1: Data structures of VISPACK. Figure corresponds to code example.

we will see). The `refcnt` tells how many `VISMatrix` objects are using the `VISMatrixData`. For instance suppose there is an object `VISMatrixData` called `data` for the following code:

```
VISMatrix A(3,3); //A is a handle pointing to data.  data has refcnt=1
A=1.0f; //every element in A is 1
VISMatrix B; //B is a handle pointing to nothing
B=A; //Now both A and B are handles pointing to data.  data has refcnt=2
VISMatrix C; //C is a handle pointing to nothing
C=A; //Now A, B, and C are handles pointing to data.  data has refcnt=3
B.poke(1,1)=5.0f; //Now what happens?
```

The `refcnt` increases by 1 for each new handle that points to `data`. But what happens when a user executes `B.poke(1,1)=5.0f`? Will the (1,1) element of `A` and `C` also be 5.0 or will they remain 1.0? When `poke` is executed a new `VISMatrixData` object must be created for `B` to point to, and then this object can be modified, thus, saving `data` from being changed. Figure 1 shows this process. This figure coinsides with the code example above.

One important aspect of this copy-on-write approach to memory management is that the use must be careful not to use read/write access methods unless they are necessary. That is, virtually every access method comes in a read-only and read/write form, and the read-only form should be used, whenever possible, to prevent unecessary copying of data buffers.

2.2 Direct Data Access

Direct data access in reference to this library indicates giving the user the means to directly access the memory block holding the data. We make direct access available for users so that experienced users, who are familiar with the internal data structures of objects, can create efficient implementations of specific methods or functions.

We provide direct access to the data through functions that return pointers to some place in the data block. For example, a pointer to the begining of the data block. By providing these functions along with explainations of how the data is stored, the overhead of sending and recieving values to/from functions is eliminated.

For example the program shown in section 2.1, which adds three matricies can be rewritten using direct data access:

```
VISMatrix A(3,3),B(3,3),C(3,3);
A=1.0f; //every element in A is 1
B=2.0f; //every element in B is 2
C=3.0f; //every element in C is 3
float* ptrA,ptrB,ptrC; //pointer to a block of float data
ptrA = A.dataPtr;
ptrB = B.dataPtr;
ptrC = C.dataPtr;
VISMatrix ans;
for (int cnt=0;cnt<9;cnt++)
        ans.poke(r,c) = *(ptrA+cnt) + *(ptrB+cnt) + *(ptrC+cnt);
```

In this example, `operator()` is not used, thus, the time taken to send values to/from `operator()` is eliminated. The data is accessed directly via the pointers `ptrA`, `ptrB`, and `ptrC`.

2.3 Use of Templating

We utilize the templating construct of C++ throughout VISPACK. We use templating in two different ways in VISPACK. The first use of templating occurs in the image and volume libraries. This templating allows, for example, the values in an image to be of different data types, e.g., 24-bit color or 16-bit greyscale. The second way in which templating is used is on a larger scale. The `array` and `list` classes are templated such that an array or list may contain any type of object. A discussion of these two types of templating follows.

One case in which templating is usefull in the image and volume classes is in a multi-stage process. As processing proceeds in a mutli-stage, intermediate results might be stored in different formats, for example 8-bit greyscale, 32-bit floating point, or 32-bit color. Well-designed libraries should support all of these different types in a way that conserves memory and computation but gives the programmer maximum flexibility.

The image and volume libraries support 24- and 32-bit color, 8- and 32-bit signed and unsigned integer, and 32- and 64-bit floating point. The templated libraries also give programmers direct access to templated objects, and therefore images of user-defined types are also possible—but they are not supported by the current set of file I/O and data conversion routines.

Image I/O can be handled using the typeless base class. Images can be read from disk and passed to a display device without the programmer ever knowing or having to specify a particular image type. Images can retain their initial data types (e.g. as they had on disk) until they need to be converted for specific operations. On the other hand conversions between types allow programmers to specify which type of data they need to support the algorithms they create and to enforce type checking when necessary. Some tasks, such as anisotropic diffusion, are most effective using floating-point representations and can be implemented using a subclass of the `float`-type, templated, generic image class.

The second type of templating used in the `array` and `list` classes is such that an array or list may contain any type of object. For instance, creation of an array of images consists of a simple statement (`Array<VISImage> image_array;`). An array of `floats` or `ints` is just as easily created. This templating makes the `array` and `list` classes very powerful. Future releases of VISPACK will likely replace these objects with those from the *Standard Template Library* (STL).

2.4 Use of Standard File Formats

When available we use standard file formats. We choose formats that are well known and have publicly available libraries that can be distributed with our libraries. The matrix library uses a simple text format. The image library uses TIFF and FITS file formats. Because no standard format exists for saving volumes of data we do use a *raw* file format.

2.5 Floating Point File Format

We use a floating point file format in order to preserve information. For example, suppose a user wishes to apply a DOG (derivative of Gaussian) filter to an image and save the result. After, another application opens the saved result and performs further processing. The result of applying the DOG filter most likely contains floating point values. Unless these floating point values are saved, the next processing step will begin with inaccurate data. A floating point file format ensures that nothing is lost when saving files during a multi-stage processing. In addition some information is inherently floating point, e.g., range data.

2.6 Operator Overloading

Used properly operator overloading gives users a convenient way to execute operations on an object. Used improperly, operator overloading can cause confusion and lead to poor readability. We, therefore, overload only basic operators, that is, operators with commonly known definitions. In addition, we clearly spell out the functionality of all overloaded operators via inline documentation and in the *Reference* sections of this document (sections 3.2, 4.2, 5.2, 7).

2.7 Image and Volume Processing Functions and Algorithms

This library includes a number of general processing algorithms (e.g. edge detection) for images and volumes. These general processing functions provide a basis upon which algorithms may easily be built. In addition we provide several specific algorithms for the image and volume objects. These specific algorithms allow users to compare their results to the results of well-known algorithms we have implemented. These specific algorithms are also useful as steps in a multi-stage process.

3 VISMatrix Library

The ML (matrix library) contains a matrix class and a vector class. After reading the header file for these classes, anyone familiar with C++ should be able to quickly implement an application to test an idea that requires matrix manipulation. After more work with the library, users should begin to see how to efficiently implement large scale, stable applications. Beyond this, an experienced programmer can, through direct access to data buffers, write efficient implementations.

The ML possesses many features common in linear algebra libraries and software packages. Because many users are already familiar with such systems, we do not discuss in detail the basic functionality of the ML. The purpose of this section is to briefly present the basic functionality of the ML and discuss the details of the implementation of the ML in a manner that will allow users to see how to get the most from the features of the ML.

A few notes on convention:

- In the ML we follow the convention of many linear algebra texts in that we refer to the size of and elements of a matrix in a row-by-column fashion (e.g. 4×1 indicates four rows and one column).
- Indexing in the ML begins at 0 (e.g. the first element of a vector is element number 0).

Include files needed to use the library are:

- `matrix.h`

3.1 Functionality

Many aspects of the functionality of the ML may be familiar to users. The following is a list of some characteristics of the ML:

- Copy on write
- Direct data access
- Easy load/save functions
- Use of LAPACK

In this section we first point users to sources of information concerning the basic functionality of the library. After, we discuss the functionality of the `VISVector` class. We present the `VISVector` class because the implementation of this class my not be obvious to the user. Finally we describe functions in the library that use LAPACK (Linear Algebra PACKage). (We use LAPACK to implement several functions in the ML.)

3.1.1 Basic Functionality

The two main sources for information on the basic functionality of the ML are the `matrix.h` file and section 3.2 of this document. By examining these two sources of information, an experienced C++ programmer should be able to quickly learn about the `VISMatrix` and `VISVector` classes. The main function categories of basic functionality are listed in the table of contents in section 3.2.

3.1.2 The VISVector Class

The `VISVector` is a subclass of `VISMatrix`: vectors are $n \times 1$ matrices. To accomplish this we use inheritance. Because the `VISMatrix` class contains functions that do not apply to a vector we use private inheritance [7]. An example of one such function is the `svd` function, which computes the singular value decomposition.

Many computer vision applications work with sets of 2D or 3D points. Although the ML can be used in any way to work with sets of points, the ML is designed with a natural way of working with these points as vectors. In the ML, vectors are implemented as column matrices, specifically, a vector is actually a matrix with 1 column. For instance, in the ML, multiplying a 3 element vector by a 3x3 matrix is illegal (run-time error). The ML sees this as multiplying a 3x1 matrix by a 3x3 matrix. The 3x3 matrix must be multiplied by the vector. The following code shows this:

```
VISMatrix M(3,3);
VISVector V(3);
VISVector ans;
ans = V*M; //invalid
ans = M*V; //correct
```

In addition, a natural way to *store* and *recall* sets of 2D or 3D points exists in the ML. VISVectors are stored in a matrix by creating a matrix with the number of columns equal to the number of points stored and the number of rows equal to the number of elements in the vectors. Two functions are used in this *storing* and *recalling* process: `VISMatrix concat(const VISVector& second)` and `VISVector vec(int col) const`. (Other `concat` functions exist, see section 3.2.6.) The `concat` function stores the point (`second`) in the matrix (`first`), and the `vec` function recalls points from the matrix. The `concat` function can be used on an empty matrix; the result is simply a matrix that contains only the vector `second`. This is shown in the example below:

```
VISMatrix M;
VISVector pt3D(3);
for (int i=0;i<10;i++){
     pt3D=(float)i; //3D point is (i,i,i)
     M=M.concat(pt3D); //store the point
}
for (i=0;i<10;i++);{
     pt3D=M.vec(i); //recall the point
     cout << "Here is point number " << i << endl << pt3D;
}
```

3.1.3 Use of LAPACK

LAPACK (Linear Algebra PACKage) provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems [1]. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur) are also provided. We use LAPACK routines in the ML. This section describes how LAPACK is used in the ML and how to set up LAPACK in the ML.

The following functions of the ML (`VISMatrix` object) only are dependent on LAPACK (i.e. they will not work without LAPACK):

- `svd` – singular value decomposition
- `det` – determinant
- `QR` – QR decomposition
- `inv`, `inverseSVD` – inverse using SVD
- `inverseGJ` – inverse using gauss jordan elimination
- `solvLinearEqn`– solve a system of linear equations

We create the above functions using functions that convert a `VISMatrix` to/from LAPACK's data formats. This allows the use of any LAPACK function. For example the `svd` function. We convert the matrix being processed to the LAPACK format. We then apply the LAPACK function `sgesvd_`. Finally we convert the resulting LAPACK matrix back to a `VISMatrix` object and return that object as the result.

The conversion functions metioned above are available for users. For example, suppose a user wishes to implement an algorithm that requires determining the eigenvalues and eigenvectors of a matrix, which does not currently exist in the ML. A user first inherits the matrix class and then adds a function to the inherited class that utilizes the `sgeev_()` function in the C library of LAPACK to find eigenvalues and eigenvectors. The function of the inherited class would:

1. Convert the inherited matrix into a LAPACK format matrix
2. Apply the `sgeev_()` LAPACK function to the LAPACK format matrix
3. Convert the result back to the inherited matrix format

The inheritance is necessary because the conversion functions are protected members of the ML. The functions are protected rather than public because users should add basic functionality as memeber functions and then use this basic functionallity to write functions outside the class.

The functions available for conversion to/from LAPACK format are: `float* VISMatrix::createLAPK() const`,
`void VISMatrix::becomeLAPK(float* lapk, int r, int c)`,
`float* VISVector::createLAPK() const`,
and `void VISVector::becomeLAPK(float* lapk, int n)`. The `createLAPK` functions create a LAPACK matrix/vector from an ML matrix/vector respectively. The `becomeLAPK` functions assign the values from a LAPACK matrix/vector to a ML matrix/vector respectivelly. See section 3.2 for more information on these functions.

We include LAPACK with this release of the library. Users should downlad LAPACK updates and documentation on the World-Wide Web at *http://www.netlib.org/lapack/* .

3.2 Reference for the VISMatrix Library

This reference section lists the functions of the ML. The functions are listed according to category. If only one function declaration is listed the function has the same declaration in the both the `VISMatrix` and VISVector classes. Otherwise, all the declarations are listed.

Table of contents:

3.2.1 Interfacing with the data
    Peeking and Poking
    Direct Access to the Data
    Constructing a VISMatrix/VISVector
    File I/O
    Converting to/from LAPACK Format
3.2.2 Overloaded VISMatrix/VISVector operators
3.2.3 General Functions
3.2.4 VISMatrix Specific Functions
3.2.5 VISVector Specific Functions
3.2.6 Creating VISVectors from VISMatrixices and Vise-a-Versa

3.2.1 Interacting with the Data

This section includes several subsections entitled: Peeking and Poking, Direct Access to the Data, Constructing a matrix, File I/O, and Converting to/from LAPACK Format. Peeking and Poking lists the main functions for accessing data and changing data of a matrix/vector. Direct access to the Data Buffers lists functions that return pointers to the memory block holding the data of a matrix. Constructing a matrix lists ways to construct a matrix, and File I/O lists methods for inputing/outputing matricies and vectors from/to files.

Peeking and Poking

`poke` – `float& VISMatrix::poke(int row,int col)`
`float& VISVector::poke(int n)`
Assign a value to the element of a matrix/vector indicated by `row` and `col` (or `n`). This function is always used on the lhs of an assignment.
Example:
```
VISMatrix M(4,4);
M.poke(2,3)=1.44;
```

`peek -- float VISMatrix::peek(int row,int col)`
`float VISVector::peek(int n)`
Returns the value of the element of a matrix/vector indicated by `row` and `col` (or `n`). Example:
```
VISMatrix M(4,4);
M.poke(2,3)=1.44;
float m=M.peek(2,3);
```

`operator()` – `float VISMatrix::operator()(int row,int col)`
`float VISVector::operator()(int n)`
Same as `peek` but easier to type. Returns the value of the element of a matrix/vector indicated by `row` and `col` (or `n`). Example:
```
VISMatrix M(4,4);
M.poke(2,3)=1.44;
float m=M(2,3);
VISVector V(2);
V.poke(2)=1.55;
float v=V(2);
```

`peekROI` – `VISMatrix peekROI(int startrow, int endrow, int startcol, int endcol) const`
`VISVector peekROI(int startelement,int endelement) const`
These functions returns a region of interest from a matrix/vector. The region is defined by `startrow`, `endrow` and `startcol`, `endcol` OR `startelement`, `endelement`.

`putROI` – `pokeROI(int row,int col,VISMatrix roi)`
`pokeROI(int row,int col,VISVector roi)`
`pokeROI(int row,VISVector roi)`
These function allows a user to place a region of interest into a matrix/vector. The region itself is either a matrix or a vector. The upper left corner of the region of interest is indicated by `row` and `col`.

Direct Access to the Data Buffers

`dataPtr` – `const float* VISMatrix::dataPtr(int i = 0) const`
`const float* VISVector::dataPtr(int i = 0) const`
This function returns a pointer to a block of float data representing a matrix/vector. The function returns, by default, the pointer to the begining of the data block, otherwise, the pointer at the position indicated by `i` is returned. The matrix data is stored in row major order.

`data float data(int i = 0) const`
This function returns the floating point value at location `i` in the data block if `i` is specified. Otherwise, the value at the first position in the data block is returned. We store the matrix data in row major order.

Constructing a VISMatrix These functions are listed according to the input of the function.

default – `VISMatrix::VISMatrix()`
`VISVector::VISVector()`
An *empty* matrix/vector is created.

`other` – `VISMatrix::VISMatrix(VISMatrix& other)`
`VISVector::VISVector(VISVector& other)`
These functions create a matrix/vector from another matrix/vector. The matrix/vector created is the same `other`.

`row,col/n` – `VISMatrix::VISMatrix(int row, int col)`
`VISVector::VISVector(int n)`
These functions create a matrix/vector of a specified size. The size of a matrix is `rowxcol`. The size of a vector is `n`.

`char* filename` – `VISMatrix::VISMatrix(char* filename)`
`VISVector::VISVector(char* filename)`
These functions create a matrix/vector from a file. The file must contain only one matrix/vector where the output format for a `VISMatrix` is:
`numberofrows numberofcols`
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $\ldots$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $\ldots$
$\ldots$ $\ldots$
and the output format for a `VISVector` is:

`numberofelements`

V_0
V_1
V_2
...
Example:

```
VISMatrix M("file.mat");
```

File I/O

`operator<<` – `ostream& operator<<(ostream& os, const VISMatrix& m)`
`ostream& operator<<(ostream& os, const VISVector& m)`
Print a matrix to any `ostream`. Output format for a `VISMatrix` is:
`numberofrows numberofcols`
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ ...
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ ...
... ...
Output format for `VISVector` is:

`numberofelements`
V_0
V_1
V_2
...
Example:

```
VISMatrix M(3,2);
M=2.0f;
cout << M;
ofstream file("output");
file << M;
```

`operator>>` – `istream& operator>>(istream &is, VISMatrix& m)`
`istream& operator>>(istream &is, VISVector& m)`
Read a matrix to any ifstream. Input format is the same as output format (see `operator<<`). Example:

```
VISMatrix M(3,2);
M=2.0f;
ifstream file("input");
file >> M;
```

Converting to/from LAPACK Format

`becomeLAPK` – `void VISMatrix::becomeLAPK(float* lapk, int r, int c)`,
`void VISVector::becomeLAPK(float* lapk, int n)`.
These functions assign the values from a LAPACK format matrix/vector to an ML `VISMatrix`/`VISVector` respectively.

`createLAPK` – `float* VISMatrix::createLAPK() const`,
`float* VISVector::createLAPK() const`,
and These functions create a LAPACK format matrix/vector from an ML `VISMatrix/VISVector` respectively.

3.2.2 Overloaded VISMatrix/VISVector Operators

Some of these functions may have the same name but different inputs and outputs. In these cases the label `other` means an input argument of type `VISMatrix` or `VISVector`; whereas, the label `s` refers to a floating point number.

`operator=(other)` – `VISMatrix& VISMatrix::operator=(const VISMatrix& other)`
`VISVector& VISVector::operator=(const VISVector& other)`
`VISVector& VISVector::operator=(const VISMatrix& other)`
Assigns one matrix/vector to another. This function does not copy the data. (See section 2.1.) A matrix may be assigned to a vector only if the matrix has 1 column. Example:

```
VISMatrix M1(4,4);
M1=1.0f;
VISMatrix M2;
M2=M1;
VISMatrix M3(4,1);
M3=5.0f;
VISVector V1;
V1=M1; //invalid
V1=M3; //o.k.
```

`operator=(s)` – `VISMatrix& VISMatrix::operator=(float s)`
`VISVector& VISVector::operator=(float s)`
Assigns each element in a matrix/vector to the value of the float `s`. Example:

```
VISMatrix A(4,4);
A=3.141592;//Every element of A is equal to 3.141592
```

`operator+(other)` – `VISMatrix VISMatrix::operator+(const VISMatrix& other) const`
`VISVector VISVector::operator+(const VISVector& other)`
Adds two matrices/vectors. The matrices/vectors must be the same size. Example:

```
VISMatrix A(4,4),B(4,4),C;
A=1.0f;
B=2.0f;
C=A+B;//now C=3.0f
```

`operator+(s)` – `VISMatrix VISMatrix::operator+(float s) const`
`VISVector VISVector::operator+(float s) const`
Adds the value of `s` to a matrix/vector.
Example:

```
VISMatrix A(4,4),B;
A=1.0f;
B=A+2.0f;//now B=3.0f;
```

`operator-(other)` – `VISMatrix VISMatrix::operator-(const VISMatrix& other) const`
`VISVector VISVector::operator-(const VISVector& other) const`
Subtracts two matrices/vectors. The matrices/vectors must be the same size.
Example:
```
VISMatrix A(4,4),B(4,4),C;
A=3.0f;
B=1.0f;
C=A-B;//now C=2.0f
```

`operator-(s)` – `VISMatrix VISMatrix::operator-(float s) const`
`VISVector VISVector::operator-(float s) const`
Subtracts the value of `s` from a matrix/vector. Example:
```
VISMatrix A(4,4),B;
A=10.0f;
B=A-2.0f;//now B=8.0f;
```

`operator-()` – `VISMatrix VISMatrix::operator-() const`
`VISVector VISVector::operator-() const`
Negates a matrix/vector. Example:
```
VISMatrix A(4,4),B;
A=3.0f;
B=-A;//now B=-3.0f
```

`operator*(other)` – `VISMatrix VISMatrix::operator*(const VISMatrix& other) const`
`VISMatrix VISVector::operator*(const VISMatrix& other) const`
`VISVector VISMatrix::operator*(const VISVector& other) const`
Multiplies a matrix/vector by a matrix/vector (`other`). Examine the example of `A=B*C`. The number of columns of the `B` must be equal to the number of rows of `C` (`other`). Implementing vectors as matrices with only one column, as we have done, means that if `B` is a vector and `C` a matrix the result, `A`, is a matrix. On the other hand if `B` is a matrix and `C` is a vector the result, `A`, is a vector. Dot product and cross product operations are provided in the vector class. Example:
```
VISMatrix A(4,4),B(4,4),C;
A=3.0f;
B=4.0f;
C=A*B;//now C=48.0f and is 4x4
C=B*A;//still C=48.0f, because A and B are square A*B=B*A
VISVector D(4),E;
D=2.0f;
E=D*B;//error!
E=B*D;//now E=24.0f and has four elements (4x1)
VISVector F
F=E*A;//error!
F=A*E;//now F=288.0f and has four elements (4x1)
```

`operator*(s)` – `VISMatrix VISMatrix::operator*(float s) const`
`VISMatrix operator*(float s,const VISMatrix m)`
`VISVector VISVector::operator*(float s) const`
`VISVector operator*(float s,const VISVector v)`
Multiplies each value in a matrix/vector by `s`. The non-member functions allow the order of multiplication be reversed (i.e. $M * 2.0f$ or $2.0f * M$). Example:
```
VISMatrix A(4,4),B;
A=2.0f;
B=A*2.0f;//now B=4.0f;
```

```
B=4.0f*A;//now B=8.0f;
```

`operator/(s)` – `VISMatrix VISMatrix::operator/(float s) const`
`VISMatrix operator/(float s,const VISMatrix m)`
`VISVector VISVector::operator/(float s) const`
`VISVector operator/(float s,const VISVector v)`
Divides each value in a matrix/vector by `s` or `s` by each value in a matrix/vector.
Example:

```
VISMatrix A(4,4),B;
A=2.0f;
B=A/4.0f;//now B=0.5f;
B=8.0f/A;//now B=4.0f;
```

`operator==(other)` – `VISMatrix VISMatrix::operator==(const VISMatrix& other) const`
`VISVector VISVector::operator==(const VISVector& other) const`
When used as `A==B` this function returns a matrix/vector containing 0/1's indicating equality/inequality respectively of each element of A to the respective element of B. Example:

```
VISMatrix A(4,4),B(4,4),C;
A=2.0f;
B=2.0f;
A.poke(0,0)=1.0f;
C = A==B;//now all elements of C=1.0f except 0,0 which is 0.0f;
```

To determine if every element of A is equal to the respective elements in B use the following:

```
if((A==B).min()) ...  //tests if all elements of A are equal to B
```

`operator==(s)` – `VISMatrix VISMatrix::operator==(float s) const`
`VISVector VISVector::operator==(float s) const`
Returns a matrix/vector of 0/1's indicating equality/inequality respectively of each element to the value of `s`. Example:

```
VISMatrix A(4,4),B;
A=2.0f;
A.poke(0,0)=1.0f;
B = A==2.0f;//now all elements of B=1.0f except 0,0 which is 0.0f;
```

To determine if every element of `A` is equal to `s` use the following:

```
if ((A==2.0f).min()) //tests if all elements of A are equal to k
...                  //where k is a floating point value
```

3.2.3 General Functions

`t` – `VISMatrix VISMatrix::t() const`
`VISMatrix VISVector::t()`
This operator returns the transpose of a matrix/vector. Because a vector has only 1 column, the transpose of a `VISVector` is a `VISMatrix` that has 1 row. Example:

```
VISVector A(4),B;
A=2.0f;//A has 1 column (4x1)
B=A.t();//B is the transpose of A, B has 1 row (1x4)
```

`max` – `float VISMatrix::max() const`
`float VISVector::max() const`
`float VISMatrix::max(int& r,int& c) const`

`float VISVector::max(int& n) const`
These functions return the maximum value in a matrix/vector. In addition, in the case of a matrix, this function assigns the row and column of the location of the maximum value to r and c. In the case of a vector the function assigns the row of the location of the maximum value to n.

`min` – `float VISMatrix::min() const`
`float VISVector::min() const`
`float VISMatrix::min(int& r,int& c)`
`float VISVector::min(int& n)`
These functions return the minimum value in a matrix/vector. In addition, in the case of a matrix, this function assigns the row and column of the location of the minimum value to r and c. In the case of a vector the function assigns the row of the location of the minimum value to n.

`mean` – `VISVector VISMatrix::mean() const`
`VISVector VISMatrix::meanOfCols() const`
`float VISVector::mean() const`
The function `VISMatrix::mean` returns the mean values of each row in a matrix. The function `meanOfCols` returns the mean values of each column in a matrix. The function `VISVector::mean` returns the mean of a vector.

3.2.4 VISMatrix Specific Functions

`transpose` – See function `t` in section 3.2.3.

`inv` – `VISMatrix VISMatrix::inv() const`
`VISMatrix inverseSVD() const`
`VISMatrix inverseGJ() const`
The first two functions compute the inverse of a matrix using SVD (singular value decomposition). The second function computes the inverse of a matrix using Gauss Jordan elimination. These functions will not work without LAPACK. Example:

```
VISMatrix A(4,4),B;
A=2.0f;
B=A.inv();//B is the inverse of A
```

`norm` – `float VISMatrix::normOfCol(int col = 0) const`
`float VISMatrix::normOfRow(int row = 0) const`
The function `normOfCol` returns the 2-norm of a column of a matrix. The function `normOfRow` returns the 2-norm of a row of a matrix. Default is column or row number is 0.

`det` – `float VISMatrix::det() const`
This function finds the determinant of a matrix using SVD. This function will not work without LAPACK.

`svd` – `void svd(VISMatrix &U, VISMatrix &W, VISMatrix &V) const`
This function performs singular value decomposition. The matrices from the decomposition are placed in the matrices U, W, and V that are inputs to the function. This function will not work without LAPACK.

`i` – `VISMatrix i(int size)`
Returns an identity matrix that is has `size` rows and `size` columns.

`cov` – `VISMatrix VISMatrix::cov() const`
For matrices where each row is an observation and each column a variable this function returns the covariance matrix.

`QR` – `void QR(VISMatrix& Q, VISMatrix& R) const`
Returns the QR decomposition of a matrix.

`solvLinearEqn` – `VISVector solvLinearEqn(const VISMatrix& A,const VISVector& b)`
Solves the linear equations:

$$Ax = b, \tag{1}$$

where A is a square matrix, b and x are vectors. The returned result is represented by x in this equation.

3.2.5 VISVector Specific Functions

`norm` – `float VISVector::norm() const`
The function `norm` returns the 2-norm of a vector.

`cross` – `VISVector VISVector::cross(const VISVector& other) const`
This function computes the cross product of two vectors.

`dot(other)` – `float dot(const VISVector& other) const`
This function returns the dot product of two vectors.

3.2.6 Creating VISVectors from Matrices and Vise-a-Versa

`vec` – `VISVector VISMatrix::vec(int col = 0) const`
`VISVector VISMatrix::vecfromcol(int col = 0) const`
`VISVector VISMatrix::vecFromRow(int row = 0) const`
`VISVector VISMatrix::vecFromDiag() const`
These functions create a vector from a matrix. The function `vec` and `vecfromcol` create a vector from a column of a matrix. (These functions are the same.) The default column is 0. The function `vecFromRow` creates a vector from a row of a matrix. The default row is 0. The function `vecFromDiag` creates a vector from the diagonal of a matrix.

`concat` – `VISMatrix concat(const VISMatrix& second)`
`VISMatrix concat(const VISVector& second)`
These functions concatenate the columns of a matrix/vector to the columns of a second matrix/vector. Recall that a vector is treated as a matrix having one column. These functions will work even if one of the matrices/vectors is empty; the matrix/vector containing data is returned by the function.
Example:
Suppose $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ and $B = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$
then `A.concat(B)` $= \begin{bmatrix} 1 & 2 & 3 & 7 & 8 & 9 \\ 4 & 5 & 6 & 10 & 11 & 12 \end{bmatrix}$.

`concatRow` – `VISMatrix concatRow(const VISMatrix& second)`
`VISVector concatRow(const VISVector& second)`
These functions concatenate the rows of a matrix/vector to the rows of a second matrix/vector. Because a vector has only one column there are no functions to concatenate the rows of a vector to the rows of a matrix or vise-a-versa. To concatenate a vector (as a row) onto the rows of a matrix use `M.concatRow(V.t())`, where `M` is a matrix having n columns and `V` is a vector with n elements.
Example:

Suppose $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ and $B = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$

then `A.concat(A)` $= \begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \\ 4 & 5 & 6 \\ 10 & 11 & 12 \end{bmatrix}$.

4 Image Library

The IL (image library) contains two main classes: the `VISImage` class and the `VISImageFile` class. The `VISImageFile` class is used only for file I/O and the `VISImage` class is used for all other functionality. The header files, `image.h` and `imagefile.h`, which include these two classes, provide much information. These header files in conjunction with this document give users complete information.

Our presentation of the IL includes a discussion of some of the functionality and implementation issues of the library and a reference section. Our discussion of functionality and implementation issues describes the general functionality of the library and image processing functionality. The reference provided for the IL lists each function according to category.

In this document we refer to the size of and pixels of an image (matrices are different) in a column-by-row fashion (e.g. 1x4, four columns one row). In addition, the IL follows this same convention.

Include files needed to use the library are:

- `image.h`
- `imagefile.h`
- `imageRGBA.h`

4.1 Functionality and Implementation

We divide the functions of the IL into two categories: general functions and image processing functions. General functions include functions that allow the user to interact with data stored in an image, file I/O functions, overloaded operators, and basic math functions. Image processing functions refer to functions that are more specific to image processing. Filtering, resampling, and distance transform functions are all considered image processing functions. In addition IL includes more complex image processing algorithms; for example, canny edge detection. We divide our discussion of the functionality of this library into the two categories: general functions and image processing functions.

4.1.1 Channels in Images

Images can contain multiple 2D arrays of data. Each 2D array is called a *channel.* By default, all images are of a single channel and all access methods pertain to the first channel. However, channel numbers can be given for constructors and access methods, and virtually all of the processing routines will apply to all of the channels. The use of channels is particularly useful when dealing with unpacked color images.

4.1.2 General Functions

In this section we point users to sources of information that explain the specifics of the general functions in the IL, and we discuss some attributes of the general functions of the IL that may not be obvious from the information given. Here, our discussion of general functions is broken into parts. We describe functions that allow access to the data in an image, file I/O functions, overloaded operators, math functions, image processing functions, and miscellaneous functions. Beyond this information, detailed information for all the functions is available from other sources. Descriptions of the general functions (excluding a file I/O functions) can be found in the `image.h` header file. Detailed information about file I/O can be found in the `imagefile.h` header file. In addition section 4.2 of this document provides descriptions of each function in the IL.

Accessing data The IL uses copy on write as described in section 2.1. This means that there are two different functions for accessing the data of a pixel in an image. One function for use on the left hand side of an assignment (read/write) and another for use on the right hand side of an assignment (read only). An assignment in IL looks like the following:

```
VISImage A(128,128);
VISImage B(128,128); A=1.0f; //every pixel in A is 1
B=2.0f; //every pixel in B is 2
A.at(4,4) = B(4,4); //now pixel 4,4 of A = pixel 4,4 of B, which = 1
//the lhs uses at(column,row) and the rhs uses
//operator(column,row)
```

File I/O The IL uses a second class, `VISImageFile` to accomplish file I/O. Information about this class is contained in the file `imagefile.h`. Here we present a piece of example code that reads a floating point image from a file, adds a value to the image, and then saves the result in a floating point file format:

```
VISIm im; // a typeless base class
VISImage<float> input; //input image
VISImage<float> output; //output image
VISImageFile im_file;
if (!((im=im_file.read("filename")).isValid())){
    fprintf(stderr,"Error reading image file %s.\ n","filename");
    exit(0);}
input=VISImage<float>(im);//Cast input image as a floating point image
output = input + 2.0f; //Add 2 to the input image assign result to output
file.write_fits(output,"out.fit");
```

To read/write images of type other than `float` use functions `read_tiff` and `write_tiff` (see section 4.2.1).

For more usage information see also the file `imagetest/imagetest.cxx` in the source code.

Overloaded Operators The `VISImage` class contains overloaded operators. A value may be added to or subtracted from an image using `+` and `-` respectively. An image may be multiplied by or divided by a value by `*` and `/`, respectively. Applying `+`, `-`, `*`, and `/` to two images adds, subtracts, multiplies, and divides the respective pixels in the two images.

Math Functions Many math functions exist in the IL–for example: square root, natural logarithm, exponential, absolute value, etc. All of these functions have short function names that are designed to be easily recalled.

4.1.3 Image Processing Functions

General image processing functions exist that:

- apply masks to an image
- smooth an image
- determine derivatives of an image
- resample an image
- perform distance transforms on an image
- corupt an image with noise

Details about these functions can be found in the `image.h` header file and in section 4.2.4 of this document. In addition to general image processing functions, several image processing algorithms exist in the IL. Information about image processing algorithms can also be found in the `image.h` header file or section 4.2.5.

4.2 Reference for Image Library

This reference section lists the functions of the IL. The functions are listed according to category.

Table of contents:

4.2.1 Interacting with the data
  Peeking and Poking
  Direct Access to the Data Buffers
  Constructing an Image
  File I/O
4.2.2 Overloaded Image Operators
4.2.3 Math Functions
4.2.4 General Image Processing Functions
4.2.5 Image Processing Algorithms

4.2.1 Interacting with the Data

This section includes several subsections entitled: Peeking and Poking, Direct Access to the Data Buffers, Constructing an Image, and File I/O. All the functions listed here are members of the `VISImage<T>` class unless otherwise noted. Peeking and Poking lists the main functions for accessing data and changing data of an image. We describe the idea of direct data access in section 2.2; here the Direct Data Access description lists functions related to this idea. Constructing an Image lists ways to construct an image, and File I/O lists methods for inputing/outputting images from/to files.

Peeking and Poking

`poke` – `T& poke(unsigned int x, unsigned int y) const`
`T& poke(unsigned int x, unsigned int y, unsigned int ch) const`
Assign a value to the pixel of an image indicated by `x` (column), `y` (row), and `ch` (channel). This function is always used on the lhs of an assignment. Example:

```
VISImage<float> I(4,4);
I.poke(2,3)=1.44;
```

`peek` – `const T& peek(unsigned int x, unsigned int y) const`
`const T& peek(unsigned int x, unsigned int y, unsigned int ch) const`
Returns a value of the pixel of an image indicated by `x` (column), `y` (row), and `ch` (channel). Example:

```
VISImage<float> I(4,4);
I.poke(2,3)=1.44;
float m=I.peek(2,3);
```

`operator()` – `const T& operator()(unsigned int x, unsigned int y) const`
`const T& operator()(unsigned int x, unsigned int y, unsigned int ch) const`
Same as `peek`. Returns a value of the pixel of an image indicated by `x` (column), `y` (row), and `ch` (channel). Example:

```
VISImage I(4,4);
I.poke(2,3)=1.44;
float m=I(2,3);
```

`putChannel` – `void putChannel(VISImage<T>& other)`
`void putChannel(VISImage<T>& other, unsigned int ch)`
These functions place a 2D image into a channel of an image. If `ch` is not specified (first function), a new channel is created, otherwise (second function), the 2D image is placed into channel `ch` of a multi-channel image.

`channel` – `VISImage<T> channel(unsigned int ch)`
This function returns the 2D image at channel `ch` of a multi-channel image.

`interp` – `T interp(float x, float y, unsigned int ch) const`
`T interp(float x, float y) const`
These functions are similar to `peek` and `operator()`, but they interpolate between pixel locations of the image to determine the value at the location represented by `x`, `y`, and `ch`. These functions use linear interpolation.

`interpNoBounds` – `T interpNoBounds(float x, float y, unsigned int ch) const`
`T interpNoBounds(float x, float y) const`
These two functions are the same as the `interp` functions except they do not do any bounds checking. By bounds checking we mean ensuring that the values of `x`, `y`, and `ch` are valid (i.e. within limits). Hence, these functions are faster than the previous two, but are not as safe to use.

`getROI` – `VISImage<T> getROI(unsigned int x_pos, unsigned int y_pos,`
`unsigned int w_roi, unsigned int h_roi)`
This function returns a region of interest from an image. The position (`x_pos`,`y_pos`) indicates the upper left corner of the image. In addition the region has size indicated by w_roi,h_roi– its width and height.

`putROI` – `void putROI(const VISImage<T>& image_in, unsigned int x_pos, unsigned int y_pos)`
This function allows a user to place a region of interest into an image. The region itself is an image and has a position (`x_pos, y_pos`)–its upper left corner.

`printData` – `void printData() const`
  This does a floating point printf on all of the data. Be careful if you have a large image.

Direct Access to the Data Buffers

`rep` – `const VISImageRep<T>* rep(unsigned int ch) const`
  `const VISImageRep<T>* rep() const`
  This routine returns a const pointer to the data of a chosen channel of an image if `ch` is indicated, otherwise, the function returns a const pointer to the data of the first channel of the image.

`repRef` – `VISImageRep<T>* repRef(unsigned int ch)`
  `VISImageRep<T>* repRef()`
  This routine returns a non-const pointer to the data of a chosen channel of an image if `ch` is indicated, otherwise, the function returns a non-const pointer to the data of the first channel of the image. This routine copies the data. (See section 2.1.)

Constructing an Image Most of these functions are listed according to the input of the function.

default – `VISImage()`
  An *empty* image is created.

`image` – `VISImage(const VISImage<T>& image)`
  This function creates an image from another image. The image created is exactly the same as `image`.

`w,h,ch` – `VISImage(unsigned int w, unsigned int h)`
  `VISImage(unsigned int w, unsigned int h, unsigned int ch)`
  These functions create an image of a specified size, where `w` is width, `h` is height, and `ch` is the number of channels.

`buffer VISImage(unsigned int w, unsigned int h, unsigned int ch, T** buf)` This function creates an image `w x h` with `ch` channels, using `buf` as the data. The pointer `buf` is a pointer to an array of pointers to individual images (channels). Each image buffer is a $w \times h$ long 1D array of type `T`.

`createToSize VISImage<T> createToSize() const`
  Returns an image of the same size of the image being operated on. Example:

```
VISImage<float> I1(128,128);
VISImage<float> I2;
I2 = I1.createToSize; //I2 is now 128x128, values in the image are unknown
```

File I/O There is only one read function in the IL. This function automatically determines the file type to read. File formats supported by the read function are discussed below. There are several write functions for the different file formats supported by the IL.

`read VISIm read(const char* fname)`
  This function reads any file type supported by the IL. These file types are tiff and fits. The tiff file format supports most image types, including color, but does not support floating point. The fits file format supports all image types, but does not have a specific type for color. Color images in fits must be handled as multi-channel images. Example (reading a floating point fits format file):

```
VISImage<float> input; //input image
VISImageFile file;
if (!((input=file.read("filename.fit")).isValid())){
    fprintf(stderr,"Error reading image file %s.\ n","filename");
    exit(0);}
input=VISImage<float>(input);//Cast input image as a floating point image
```

`write_tiff` – `int write_tiff(const VISImageRGBA& image, const char* fname)`
`int write_tiff(const VISImage<float>& image, const char* fname)`
`int write_tiff(const VISImage<int>& image, const char* fname)`
`int write_tiff(const VISImage<byte>& image, const char* fname)`
`int write_tiff(const VISImage<short>& image, const char* fname)`
These functions write any image type as a tiff file. The file name is indicated by `fname`.

`write_fits` – `int write_fits(const VISImage<byte>& image, const char* fname)`
`int write_fits(const VISImage<int>& image, const char* fname)`
`int write_fits(const VISImage<float>& image, const char* fname)`
`int write_fits(const VISImage<short>&, const char*)`
These functions write any image type, excluding color images, as a fits format file. The file is named `fname`.

`write_iv` – `int write_iv(const VISImage<float>& im, const char* filename)`
`int write_iv(const VISImage<float>& x, const VISImage<float>& y, const VISImage<float>& z,const char* filename)`
`int VISImageFile::write_iv(const VISImage<float>& im, const VISImageRGBA& im_color, const char* filename)`
These functions create a file that is a surface mesh in open inventor format. The mesh is tesslated on the image grid (2 triangles per group of four pixels). In the first case the image coordinates are treated as x and y positions and the value of the image at each coordinate is treated as z—this gives the *graph* of an image. In the second case, the grid coordinates define the topology of the mesh, but the x, y, and z positions for each image coordinate are given by the values of the 3 input images. The third case is the same as the first but also includes a color (material property) at each vertex from the values given by the color image. For multiple input images, they must be the same size.

4.2.2 Overloaded Image Operators

The input to the functions listed here is either an image or a value. We list functions according to operation name and input type. The label `image` indicates an object of type `VISImage<T>`, and the label `value` refers to a value. (The function name indicates the type of `value`.)

`operator=(image)` – `VISImage<T>& operator=(const VISImage<T>& from)`
Assigns one image to another. This function does not copy the data. (See section 2.1.) Example:

```
VISImage<float> A(128,128);
A=1.0f; //every pixel in A is 1
VISImage<float> B;
B=A; //B is the same as A
```

`operator=(value)` – `VISImage<T>& operator=(T value)`
Sets each pixel in an image to `value`. Example:

```
VISImage<float> A(128,128);
A=3.141592;//Every pixel of A is equal to 3.141592
```

`operator+(image)` – `VISImage<T> operator+(const VISImage<T>& image) const`
: Adds two images. The images must be the same size. Example:

```
VISImage<float> A(128,128),B(128,128),C;
A=1.0f;
B=2.0f;
C=A+B;//now C=3.0f
```

`operator+(value)` – `VISImage<T> operator+(const VISImage<T>& from, T value)`
: `VISImage<T> operator+(T value,const VISImage<T>& from)`
Adds `value` to an image.
Example:

```
VISImage<float> A(128,128),B;
A=1.0f;
B=A+2.0f;//now B=3.0f
B=3.0f+A;//now B=4.0f
```

`operator-(image)` – `VISImage<T> operator-(const VISImage<T>& image) const`
: Subtracts one image from another image. The images must be the same size. Example:

```
VISImage<float> A(128,128),B(128,128),C;
A=1.0f;
B=2.0f;
C=B-A;//now C=1.0f
```

`operator-(value)` – `VISImage<T> operator-(const VISImage<T>&from, T value)`
: `VISImage<T> operator-(T value,const VISImage<T>& from)`
Subtracts a value from an image or subtracts an image from a value. Example:

```
VISImage<float> A(128,128),B;
A=1.0f;
B=2.0f-A;//now B=1.0f
B=A-3.0f;//now B=-2.0f
```

`operator*(image)` – `VISImage<T> operator*(const VISImage<T>& image) const`
: Multiplies each pixel in an image by the respective pixel in another image, `image`. Example:

```
VISImage<float> A(128,128),B(128,128),C;
A=5.0f;
B=2.0f;
C=B*A;//now C=10.0f
```

`operator*(value)` – `VISImage<T> operator*(const VISImage<T>& from, T value)`
: `VISImage<T> operator*(T value, const VISImage<T>& from)`
Multiplies each pixel in an image by `value`. Example:

```
VISImage<float> A(128,128),B;
A=2.0f;
B=2.0f * A;//now B=4.0f
B=A * -3.0f;//now B=-6.0f
```

`operator/(image)` – `VISImage<T> operator/(const VISImage<T>& image) const`
: Divides each pixel in an image by the respective pixel in `image`. Example:

```
VISImage<float> A(128,128),B(128,128),C;
A=5.0f;
B=2.0f;
C=B/A;//now C=0.4
```

`operator/(value)` – `VISImage<T> operator/(const VISImage<T>& from, T value)`
: `VISImage<T> operator/(T value, const VISImage<T>& from)`

Divides each pixel in an image by `value`. Example:

```
VISImage<float> A(128,128),B;
A=2.0f;
B=10.0f/A;//now B=5.0f
B=A / 4.0f;//now B=0.5
```

`operator+=(image)` – `VISImage<T>& operator+=(const VISImage<T>& image)`
Adds each pixel in an image to the respective pixel in `image` and assigns the result to the image being operated on.

`operator+=(value)` – `VISImage<T>& operator+=(const T& value)`
Adds `value` to each pixel and assigns the result to the image being operated on.

`operator-=(image)` – `VISImage<T>& operator-=(const VISImage<T>& image)`
Subtracts each pixel in `image` from the respective pixel in the image being operated on and assigns the result to the image being operated on.

`operator-=(value)` – `VISImage<T>& operator-=(const T& value)`
Subtracts `value` from each pixel and assigns the result to the image being operated on.

`operator*=(image)` – `VISImage<T>& operator*=(const VISImage<T>& image)`
Multiplies each pixel by the respective pixel in `image` and assigns the result to the image being operated on.

`operator*=(value)` – `VISImage<T>& operator*=(const T& value)`
Multiplies each pixel by `value` and assigns the result to the image being operated on.

`operator/=(image)` – `VISImage<T>& operator/=(const VISImage<T>& image)`
Divides each pixel by the respective pixel in `image` and assigns the result to the image being operated on.

`operator/=(value)` – `VISImage<T>& operator/=(const T& value)` Divides each pixel by `value` and assigns the result to the image being operated on.

4.2.3 Math Functions

`power(int exponent)` – `VISImage<T> power(const int& exponent) const`
This function raises the image to a power.

`sqrt()` – `VISImage<T> sqrt() const`
Performs the square root of an image.

`div_by(value,zeroCondition)` – `VISImage<T> div_by(T value, T zeroCondition) const`
Divides an image by `value`; if division by zero is encountered the pixel value in the returned image is equal to `zeroCondition`.

`div(image,zeroCondition)` – `VISImage<T> div(const VISImage<T>& image, T zeroCondition) const`
Divides the pixels in an image by the respective pixels in `image`; if division by zero is encountered the pixel value in the returned image is equal to `zeroCondition`.

`ln()` – `VISImage<T> ln() const`
Returns natural log of the image.

`exp()` – `VISImage<T> exp() const`
Returns the exponential of the image.

`pos()` – `VISImage<T> pos() const`
: Replaces negative numbers with zeros. Does not change positive numbers.

`neg()` – `VISImage<T> neg() const`
: Replaces positive numbers with zeros. Does not change negative numbers.

`abs()` – `VISImage<T> abs() const`
: Returns the absolute value of the image.

`sign()` – `VISImage<T> sign() const`
: Replaces all zeros with +1. Replaces all positive numbers with +1. Replaces all negative numbers with -1.

`max()` – `T max() const`
: Returns max value in image.

`min()` – `T min() const`
: Returns min value in image.

`max(image)` – `VISImage<T> max(const VISImage<T> &other) const`
: Returns an image containing the max of `other` and the image being operated on.

`min(image)` – `VISImage<T> min(const VISImage<T> &other) const`
: Returns an image containing the min of `other` and the image being operated on.

`sum()` – `float sum() const`
: Returns the sum of all pixels in the image.

`average()` – `float average()`
: Returns the average of all pixels in the image

`scale(value)` – `VISImage<T> scale(float value) const;`
: Multiplies each pixel by `value`.

4.2.4 General Image Processing Functions

`median` – `VISImage<T> median(int window_size) const`
: Median filter where `window_size` indicates the size of the mask used.

`gaus` – `VISImage<T> gauss(float sigma) const`
: Gaussian smoothing. The standard deviation of the Gaussian filter is `sigma`.

`convolve` – `VISImage<T> convolve(const VISImage<T>& kernel) const`
: Convolution with any filter, which is given in the form of an image. The image library contains global functions for constructing images that correspond to Gaussian smoothing filters and discrete derivatives.

`mask` – `VISImage<T> mask(const VISImage<T>& kernel) const`
: Determines the inner product of a mask and an area in an image.

`maskFloat` – `float maskFloat(const VISImage<float>& mask, unsigned int center_x,`
`unsigned int center_y, unsigned int x_pos, unsigned int y_pos) const`
: `Array<float>* maskFloat(const Array< VISImage<float> >& masks,`
`unsigned int x_pos, unsigned int y_pos) const`
Determines the inner product of a mask and an area in an image. In the first function, the center of the mask is indicated by (`center_x`, `center_y`) and the center of the mask is positioned at (`x_pos`,

y_pos) of the image and then the inner product is calculated. The second function does the same as the first except an array of masks is applied to the image, and thus, an array of floats is returned.

`scaleToRGB` – `VISImage<T> scaleToRGB()`
Maps the grey-scale values in an image to the interval [0,255] using a linear map.

`dx` – `VISImage<T> dx() const`
`VISImage<T> dx(unsigned int order) const`
These functions determine the derivative in x direction (along rows). The second function takes the `order` argument indicating the order of the derivative to be taken.

`dy` – `VISImage<T> dy() const`
`VISImage<T> dy(unsigned int order) const`
These functions determine the derivative in y direction (along columns). The second function takes the `order` argument indicating the order of the derivative to be taken.

`derivative` – `VISImage<T> derivative(unsigned int order_x, unsigned int order_y) const`
`VISImage<T> derivative(unsigned int order_x, unsigned int order_y, float scale) const`
These functions take the n-th order derivative in x and y directions. The derivatives can be of different order in the x vs. y direction. In the second function the `scale` parameter indicates the standard deviation of a Gaussian kernel (see [5]) used to compute the derivative.

`derivatives` – `VISImage<T> derivatives(unsigned degree) const`
`VISImage<T> derivatives(unsigned degree, float scale) const`
Similar to the `derivative` functions above but return a multi-channel image with derivatives up to a degree.

`dxHalfForward/dxHalfBack/dyHalfForward/dyHalfBack` – `VISImage<T> dxHalfForward() const`
`VISImage<T> dxHalfBack() const`
`VISImage<T> dyHalfForward() const`
`VISImage<T> dyHalfBack() const`
These functions return "half" derivatives of an image, using forward or backward differences. These are useful in solving differential equations (see method `anisoDiffuse()`) These functions all assume a toroidal topology (periodic boundary conditions).

`zeroCrossings` – `VISImage<T> zeroCrossings() const`
This function places 0's in pixel locations closest to zero crossings along grid lines. Other pixels in the image are undisturbed.

`reduce` – `VISImage<T> reduce(int scale) const`
Reduces the size of an image by sub sampling, taking every `scale`'th pixel.

`reduceAverage` – `VISImage<T> reduceAverage(int scale) const`
Does the same as `reduce` but averages all pixels around the `scale`'th pixel and places the average value in the returned image.

`resample` – `VISImage<T> resample(float the_scale) const`
`VISImage<T> resample(unsigned w, unsigned h) const`
`VISImage<T> resample(float scale_x, float scale_y, float x, float y) const`
These functions scale the size of the image by interpolating between pixels. This change in scaling is specified in three different ways. In the first function, the image size is changed by `scale`, which is a floating point value. In the second function, the image size is changed to match the size of `w` by `h`. In the third function the image size is scaled by `scale_x` in width and scaled by `scale_y` in height. In this function `x` and `y` indicate a position on the image upon which the center of the new image grid will be placed.

`noiseShot` – `VISImage<T> noiseShot(float percent, T low, T high) const`
`VISImage<T> noiseShot(float percent, float sigma) const`
This function adds shot noise to the image being operated on. The `percent` parameter represents the percent of pixels effected by the noise. The `low` and `high` parameters are the limits of the uniformally distributed noise and the `sigma` parameter is the sigma value of the Gaussian distributed noise.

`setBorder` – `VISImage<T> setBorder(T value, unsigned thickness) const`
Sets a strip of pixels of width `thickness` to `value` around the border of the image.

`becomeFlat` – `VISImage<T> becomeFlat() const`
This function creates a single channel image from a multichannel image by shrinking each channel and placing the result into an image, that is, a 'matrix' of the shrunken channel images. This 'matrix' image is the image returned.

`repeatDown` – `VISImage<T> repeatDown(unsigned num) const`
This function creates a bigger image by repeating copies of the image being operated on.

`repeatAcross` – `VISImage<T> repeatAcross(unsigned num) const`
This function creates a bigger image by repeating copies of the image being operated on.

`distanceTrans` – `VISImage<float> distanceTrans() const`
`VISImage<float> distanceTrans(float max_distance) const`
These distance transforms return an image containing distance values. The distance values indicate the Euclidean distance from the respective point in the input image to the closest non-zero point in the input image. The second distance transform function has the input parameter `max_distance`. If the distance is greater than `max_distance` then a zero is placed in the returned image.

`cityBlockDistTrans` – `VISImage<float> cityBlockDistTrans() const`
City block distance transform.

`extrema` – `VISImage<int> extrema() const`
Returns an image of 1's indicating local extrema and 0's elsewhere.

4.2.5 Image Processing Algorithms

`anisoDiffuse` – `VISImage<float> anisoDiffuse(float k) const`
`VISImage<float> anisoDiffuse(VISImage<float> image_dx, VISImage<float> image_dy) const`
These functions return one iteration of an anisotropic diffusion [11, 13]. The parameter `k` is the conductance parameter (expressed in units of RMS of the gradient image). Alternatively, one can compute the conductance in a separate function and pass the two x-offset and y-offset conductance functions, `image_dx` and `image_dy` respectively.

`watershed` – `VISImage<int> watershed(float thresh, float depth, int pad)`
`VISImage<int> watershed(float thresh, float depth, int size, int pad)`
These functions apply the watershed algorithm to an image [2]. The `thresh` parameter is the value of a threshold applied before performing the watershed. This parameter only serves to speed up the algorithm. If set to zero then no pre-thresholding occurs. The `depth` parameter indicates the maximum depth of the watersheds created. The `size` parameter is the minimum size of a region. Regions containing more than `size` pixels are merged with other regions. The `pad` parameter is the size of padding placed around the boarder of the image.

`floodFill` – `VISImage<T> floodFill(T thresh, unsigned x, unsigned y)`
`const VISImage<T>& floodFill(T label_from, T label_to, unsigned x, unsigned y)`
The first function returns an image of ones indicating where the flood fill occurred. All other values

are unchanged. The `x` and `y` parameters are the starting "seed" position and the `thresh` parameter indicates the value of the maximum level filled. The second function does the flood fill and changes all values `label_from` to `label_to` (e.g., `floodFill(1,5,4,4)` means that all the 1's connected to the pixel at 4,4 by a path of 1's are replaced with 5's).

`floodfileBoundBox` – `VISImage<T> floodFillBoundBox(T thresh, unsigned x, unsigned y, unsigned &x_lo, unsigned &y_lo, unsigned &x_hi, unsigned &y_hi)`
This function is similar to the two `floodFill` functions, except that it returns a bounding box on the flooded region, through the parameters `x_lo`, etc.

`cannyEdges` – `VISImage<int> cannyEdges() const`
`VISImage<int> cannyEdges(T threshold) const`
These functions return the zero crossings of the second derivative in the gradient direction (with negative third derivative), i.e., non-maximal edge detection, closely related to the Canny edge detector [4]. The `threshold` parameter is a threshold applied to the gradient magnitude.

5 Volume Library

The two main classes in the volume library (VL) are the `VISVolume` class and the `VISVolumeFile` class. The `VISVolumeFile` class is used only for file I/O and the `VISVolume` class is used for all other functionality. The header files, `volume.h` and `volumefile.h`, which include these two classes, provide much inline information about the classes themselves. These header files in conjunction with this document should give users the information needed to create any type of application.

Our discussion of the VL here consists of a short description of the functionality of the library followed by a reference section. The reference section is similar to the other reference sections in this document in that it contains a listing of all the functions in the VL.

Include files needed to use the library are:

- `volume.h`
- `volumefile.h`

5.1 Functionality

We divide the functions of the VL in the same manner as the functions of the image library: general functions and volume processing functions. The general functions include functions that allow the user to interact with the data of a volume, file I/O functions, overloaded operators, and basic math functions. By volume processing functions we mean functions that involve more than a basic operation. A volume processing function is built using the general functions of the VL. We break our discussion into these two parts.

5.1.1 General Functions

In this section we cite sources of information on the general functions and provide a list describing the different types of general functions of the VL [2]. This information is available from more than one source. Detailed descriptions of the all general functions excluding functions involving file I/O exist found in the `volume.h` file. The functions involving file I/O exist in `volumefile.h`. Section 5.2 of this document also contains descriptions of all the general functions.

Accessing data – These functions involve 'getting' and 'putting' values from/to voxels. They use copy on write (section 2.1).

File I/O – The `VISVolumeFile` provides methods for saving and loading volumes, including floating point volumes.

Overloaded Operators – Many operators have been overloaded for the `VISVolume` class.

Math Functions – Many math function that are not overloaded operators exist in the library.

5.1.2 Volume Processing Functions

General volume processing functions exist that:

- apply masks to a volume
- smooth a volume
- determine derivatives of a volume
- resample a volume
- perform distance transforms on a volume
- corrupt a volume with noise

We include detailed descriptions of these functions in the `volume.h` file. In addition to the functions described in the above list, we have implemented the marching cubes algorithm. This algorithm is described in the `volumefile.h` file. Descriptions of all of these functions and algorithms can be found in the next section.

5.2 Reference for Volume Library

This reference section lists the functions of the VL. The functions are listed according to category.

Table of contents:

5.2.1 Interacting with the data
Peeking and Poking
Direct Access to the Data

[2] In addition section 4.1.2 discusses the general functionality of the image library. The general functionality of the image library is similar to that of the volume library, therefore it may be useful to look at section 4.1.2 if the list given in this section is not sufficient.

Constructing a Volume
Creating Images from Volumes
File I/O
5.2.2 Overloaded Image Operators
5.2.3 Math Functions
5.2.4 General Volume Processing Functions
5.2.5 Volume Processing Algorithms

5.2.1 Interacting with the Data

This section includes several subsections entitled: Peeking and Poking, Direct Access to the Data, Constructing a Volume, Creating Images from Volumes, and File I/O. All the functions listed here are members of the `VISVolume<T>` class unless otherwise noted.

Peeking and Poking

`poke` – `T& poke(unsigned int x, unsigned int y, unsigned int z)`
Assign a value to the voxel of a volume indicated by `x`, `y`, and `z`. This function is always used on the lhs of an assignment. Example:
```
VISVolume<float> V(4,4,4);
V.poke(2,3,3)=1.44;
```

`peek` – `const T& peek(unsigned int x, unsigned int y, unsigned int z) const`
Returns a value of the voxel of a volume indicated by `x`, `y`, and `z`. Example:
```
VISVolume<float> V(4,4,4);
V.poke(2,3,3)=1.44;
float m=V.peek(2,3);
```

`operator()` – `const T& operator()(unsigned int x, unsigned int y, unsigned int z) const`
Same as `peek`. Returns a value of the voxel of a volume indicated by `x`, `y`, and `z`. Example:
```
VISVolume V(4,4,4);
V.poke(2,3,3)=1.44;
float m=V(2,3,3);
```

`interp` – `T interp(float x, float y, float z) const`
This function is similar to `peek` and `operator()`, but it interpolates between voxel locations of the volume to determine the value at the location represented by `x`, `y`, and `z`. This is a linear interpolation.

`getROI` – `VISImage<T> getROI(unsigned int x_pos, unsigned int y_pos, unsigned int z_roi, unsigned int w_roi, unsigned int h_roi, unsigned int d_roi)`
This function returns a region of interest from a volume. The region has a position (`x_pos`,`y_pos`,`z_pos`): its upper left corner. In addition the region has a size (w_roi,h_roi,d_roi): its width, height, and depth.

`putROI` – `void putROI(const VISVolume<T>& volume_in, unsigned int x_pos, unsigned int y_pos, unsigned int z_pos)`
This function allows a user to place a region of interest into a volume. The region itself is a volume and has a position (`x_pos,y_pos,z_pos`): its upper left corner.

`printData` – `void printData() const`
This does a floating point `printf` on all of the data. Be careful if you have a large volume.

Direct Access to the Data

`repRef` – `VISVolumeRep<T>* repRef()`
  This routine returns a non-const pointer to the data of a volume

Constructing a Volume Most of these functions are listed according to the input of the function.

default – `VISVolume()`
  An *empty* volume is created.

`volume` – `VISVolume(const VISVolume<T>& volume)`
  This function creates a volume from another volume. The volume created is exactly the same as `volume`.

`image` – `VISVolume(const VISImage<T>& image)`
  This function creates a volume from a multi-channel image. The volume created is exactly the same as `image`.

`w,h,d` – `VISVolume(unsigned int w, unsigned int h, unsigned int d)`
  This function creates a volume of a specified size, where `w` is width, `h` is height, and `d` is the depth, which correspond to the `x`, `y`, and `z` directions respectively.

`buf` – `VISVolume(unsigned int w, unsigned int h, unsigned int ch, T** buf)`
  This function creates a volume `w x h x d`, using `buf` as the data. The `buf` buffer is a pointer to an array of pointers to image-like buffers. This is used to create volumes from stacks of images.

`createToSize` – `VISVolume<T> createToSize() const`
  Returns a volume of the same size as the volume being operated on. Example:

```
VISVolume<float> V1(128,128,128);
VISVolume<float> V2;
V2 = V1.createToSize; //V2 is now 128x128x128, values in the volume are unknown
```

Creating Images from Volumes

`image` – `VISImage<T> image()`
  `VISImage<T> image(unsigned int slice)`
  The first function creates a multi-channel image that matches the volume. Each channel in one “slice” in depth (z direction). The second function creates a single-channel image corresponding to the number given in `slice`.

`imageRef` – `VISImage<T> imageRef()`
  `VISImage<T> imageRef(unsigned int slice)`
  The “`ref`” forms return images which contain actual pointers to the same buffers used by the volume. The resulting images are only valid as long as the associated volume, and the buffers are not cleaned up when the images are destroyed.

File I/O The `VISVolumeFile` class contains all the file I/O functions including two read functions and two write functions. These functions read and write floating point and binary format files. In addition the class includes a function that executes the marching cubes algorithm and saves the result in an ivview file format. We list this marching cubes function in our discussion of volume processing algorithms, rather than here, because this is not primarily a file I/O function.

`read_float` **and** `write_float` – `VISVol read_float(const char* fname)`
`int write_float(const VISVolume<float>& volume, const char*) const`
These functions read/write a "raw" floating point file. This file has an ascii header. We use a "raw" format because no good, widely-used volume file format exists. The header has the following format:
`volume.width() volume.height() volume.depth()`
Example (reading a floating point format file):

```
VISVolume<float> input; //input volume
VISImageFile file;
if (!((input=file.read_float("volumein.vol")).isValid())){
    fprintf(stderr,"Error reading image file %s.\n","volumein.vol");
    exit(0);}
input=VISVolume<float>(input);//Cast input volume as a floating point
output = input + 5.0f;
file.write_float(output,"volumeout.vol"); //write the volume as a "raw"
                                          //floating point file
```

`read_binary` **and** `write_binary` – `VISVol read_binary(const char*) const`
`int write_binary(const VISVolume<byte>& volume, const char*) const`
These functions read/write a volume of type `byte`. The volume is assumed to be binary and the values are stored in the file as bits. The header is binary. The header has the following format(where the values are stored in the file as unsigned short's):
`volume.width() volume.height() volume.depth()`

5.2.2 Overloaded Volume Operators

In some cases, functions may have the same name but different inputs and outputs. In these cases the labels `volume` and `from` indicate an object of type `VISVolume<T>`; whereas, any other label ,such as `value`, refers to a value. Labels for values are type `T` unless otherwise noted.

`operator=(from)` – `VISVolume<T>& operator=(const VISVolume<T>& from)`
Assigns one volume to another. This function does not copy the data. (See section 2.1.) Example:
```
VISVolume<float> A(128,128,128);
A=1.0f; //every pixel in A is 1
VISVolume<float> B;
B=A; //B is the same as A
```

`operator=(value)` – `VISVolume<T>& operator=(T value)`
Assigns each voxel in a volume to the value of `value`. Example:
```
VISVolume<float> A(128,128,128);
A=3.141592;//Every element of A is equal to 3.141592
```

`operator+(volume)` – `VISVolume<T> operator+(const VISVolume<T>& volume) const`
Adds two volumes. The volumes must be the same size. Example:
```
VISVolume<float> A(128,128,128),B(128,128,128),C;
A=1.0f;
B=2.0f;
C=A+B;//now C=3.0f
```

`operator+(value)` – `VISVolume<T> operator+(const VISVolume<T>& from, T value)`
`VISVolume<T> operator+(T value,const VISVolume<T>& from)`
Adds `value` to a volume.
Example:

```
VISVolume<float> A(128,128,128),B;
A=1.0f;
B=A+2.0f;//now B=3.0f
B=3.0f+A;//now B=4.0f
```

`operator-(volume)` – `VISVolume<T> operator-(const VISVolume<T>& volume) const`
Subtracts one volume from another volume. The volumes must be the same size. Example:

```
VISVolume<float> A(128,128,128),B(128,128,128),C;
A=1.0f;
B=2.0f;
C=B-A;//now C=1.0f
```

`operator-(value)` – `VISVolume<T> operator-(const VISVolume<T>&from, T value)`
`VISVolume<T> operator-(T value,const VISVolume<T>& from)`
Subtracts a value from a volume or subtracts a volume from a value. Example:

```
VISVolume<float> A(128,128,128),B;
A=1.0f;
B=2.0f-A;//now B=1.0f
B=A-3.0f;//now B=-2.0f
```

`operator*(volume)` – `VISVolume<T> operator*(const VISVolume<T>& volume) const`
Multiplies each voxel in a volume to the respective voxel in `volume`. Example:

```
VISVolume<float> A(128,128,128),B(128,128,128),C;
A=5.0f;
B=2.0f;
C=B*A;//now C=10.0f
```

`operator*(value)` – `VISVolume<T> operator*(const VISVolume<T>& from, T value)`
`VISVolume<T> operator*(T value, const VISVolume<T>& from)`
Multiplies each voxel in a volume by `value`. Example:

```
VISVolume<float> A(128,128,128),B;
A=2.0f;
B=2.0f*A;//now B=4.0f
B=A * -3.0f;//now B=-6.0f
```

`operator/(volume)` – `VISVolume<T> operator/(const VISVolume<T>& volume) const`
Divides each voxel in a volume by the respective voxel in `volume`. Example:

```
VISVolume<float> A(128,128,128),B(128,128,128),C;
A=5.0f;
B=2.0f;
C=B/A;//now C=0.4
```

`operator/(value)` – `VISVolume<T> operator/(const VISVolume<T>& from, T value)`
`VISVolume<T> operator/(T value, const VISVolume<T>& from)`
Divides each voxel in a volume by `value` or divides a value by each voxel in a volume. Example:

```
VISVolume<float> A(128,128,128),B;
A=2.0f;
B=10.0f/A;//now B=5.0f
B=A/4.0f;//now B=0.5
```

`operator+=(volume) – VISVolume<T>& operator+=(const VISVolume<T>& volume)`
: Adds each voxel in a volume by the respective voxel in `volume` and assigns the result to the volume being operated on.

`operator+=(value) – VISVolume<T>& operator+=(const T& value)`
: Adds `value` to each voxel and assigns the result to the volume being operated on.

`operator-=(volume) – VISVolume<T>& operator-=(const VISVolume<T>& volume)`
: Subtracts each voxel in `volume` from the respective voxel in the volume being operated on and assigns the result to the volume being operated on.

`operator-=(value) – VISVolume<T>& operator-=(const T& value)`
: Subtracts `value` from each voxel and assigns the result to the volume being operated on.

`operator*=(volume) – VISVolume<T>& operator*=(const VISVolume<T>& volume)`
: Multiplies each voxel by the respective voxel in `volume` and assigns the result to the volume being operated on.

`operator*=(value) – VISVolume<T>& operator*=(const T& value)`
: Multiplies each voxel by `value` and assigns the result to the volume being operated on.

`operator/=(volume) – VISVolume<T>& operator/=(const VISVolume<T>& volume)`
: Divides each voxel by the respective voxel in `volume` and assigns the result to the volume being operated on.

`operator/=(value) – VISVolume<T>& operator/=(const T& value)`
: Divides each voxel by `value` and assigns the result to the volume being operated on.

5.2.3 Math Functions

`power(int power) – VISVolume<T> power(const int& power) const`
: This function raises the volume to a power.

`sqrt() – VISVolume<T> sqrt() const`
: Performs the square root of each voxel in a volume.

`div(volume, value) – VISVolume<T> div(const VISVolume<T>& volume, T value`
: A divide method that returns `value` for any voxel that is zero in `volume`.

`abs() – VISVolume<T> abs() const`
: Returns the absolute value of the volume.

`max() – T max() const`
: Returns max value in volume.

`min() – T min() const`
: Returns min value in volume.

`sum() – float sum() const`
: Returns the sum of all voxels in the volume.

`average() – float average()`
: Returns the average of all voxels in the volume

`scale(value) – VISVolume<T> scale(float value) const;`
: Multiplies each voxel by `value`.

5.2.4 General Volume Processing Functions

`gauss` – `VISVolume<T> gauss(float sigma) const`
Gaussian smoothing. The standard deviation of the Gaussian filter is `sigma`.

`convolve` – `VISVolume<T> convolve(const VISVolume<T>& kernel) const`
Convolution with any filter you choose.

`maskFloat` – `float maskFloat(const VISVolume<float>& mask, unsigned int center_x, unsigned int center_y, unsigned int center_z, unsigned int x_pos, unsigned int y_pos, unsigned int z_pos) const`
`Array<float>* maskFloat(const Array< VISVolume<float> >& masks, unsigned int x_pos, unsigned int y_pos const, unsigned int z_pos) const`
Determines the inner product of a mask and a region in a volume. In the first function, the center of the mask is indicated by `center_x`, `center_y`, `center_z` and the center of the mask is positioned at `x_pos`, `y_pos`, `z_pos` of the volume. The second function does the same as the first except it applies an array of masks to the volume, and thus, returns an array of floats.

`dx` – `VISVolume<T> dx() const`
`VISVolume<T> dx(unsigned int order) const`
Determines the derivative in x direction (along width). The second function takes the `order` argument indicating the order of the derivative to be taken.

`dy` – `VISVolume<T> dy() const`
`VISVolume<T> dy(unsigned int order) const`
Determines the derivative in y direction (along height). The second function takes the `order` argument indicating the order of the derivative to be taken.

`dz` – `VISVolume<T> dz() const`
`VISVolume<T> dz(unsigned int order) const`
Determines the derivative in z direction (along depth). The second function takes the `order` argument indicating the order of the derivative to be taken.

`derivative` – `VISVolume<T> derivative(unsigned int order_x, unsigned int order_y, unsigned int order_z) const`
These functions take the n-th order derivative in x, y, and z directions. The derivatives can be of different order in the x, y, and z directions.

`zeroCrossings` – `VISVolume<T> zeroCrossings() const`
This function places 0's in voxel locations closest to zero crossings along grid lines. Other voxels in the image are undisturbed.

`reduce` – `VISVolume<T> reduce(int scale) const`
Reduces the size of a volume by sub sampling, taking every `scale`'th voxel.

`reduceAverage` – Reduces the size of a volume by sub sampling, taking every `scale`'th voxel and averaging over a footprint of the same size (i.e. reduces aliasing artifacts as it subsamples).

`resample` – `VISVolume<T> resample(float the_scale) const`
`VISVolume<T> resample(unsigned w, unsigned h, unsigned d) const`
`VISVolume<T> resample(float scale_x, float scale_y, float scale_z, float x, float y, float z) const`
These functions scale the size of the volume by interpolating between voxels. The change in scaling is specified in three different ways. In the first function, the volume size is changed by `scale`, which is a floating point value. In the second function, the volume size is changed to match the size of `w` x `h` x `d`. In the third function the volume size is scaled by `scale_x` in width, scaled by `scale_y` in height, and

by `scale_z` in depth. In this function `x`, `y`, and `z` indicate a position on the volume upon which the center of the new volume grid will be placed.

`noise` – `VISVolume<float> noise() const`
This function returns a volume with random values drawn from a uniform distribution ranging zero to one.

`setBorder` – `void setBorder(T value)`
`void setBorder(T value, unsigned w)`
The first function sets a strip of voxels (1 voxel wide) to `value` around the border of the volume. The second function sets a strip of voxels of width `w` to `value` around the border of the volume.

`floodFill` – `VISImage<T> floodFill(T thresh, unsigned x, unsigned y, unsigned z)`
This function returns a volume of ones indicating where the flood fill occurred. All other values are unaffected. The `x`, `y`, and `z` parameters are the starting "seed" position, and the `thresh` parameter indicates the value of the maximum level filled.

`cityBlockDistTrans` – `VISVolume<float> cityBlockDistTrans() const`
City block distance transform. Values returned are on the interval [0,1].

`extrema` – `VISVolume<int> extrema() const`
Returns a volume of 1's indicating local extrema and 0's elsewhere.

5.2.5 Volume Processing Algorithms

We have implemented only one volume processing algorithm. This function is a member of the `VISVolumeFile` class because it creates a file as its output.

`write_marchingCubes` – `write_marchingCubes(const VISVolume<float>& volume,`
`float iso_value, char* fname) const`
Applies the marching cubes algorithm to a volume [6]. The `iso_value` parameter indicates the value that the algorithm uses to determine the surface location in the volume. The `fname` parameter is the name of the ivview file created by the function. This file contains the surface resulting from the marching cubes algorithm.

6 Util Library

The UL (util library) contains several classes that are simply utilities. These utilities support other parts of VISPACK and can be used directly by users. In fact we encourage users to look through the functionality of the UL before beginning to program with VISPACK in order to be aware of the functions available in the UL.

6.1 Contents

The UL contains two main classes and a collection of stand-alone functions. The two main classes are a templated array class, `Array<T>`, and a templated list class, `List<T>`. The useful stand-alone functions in the UL are found in th `mathutil.h` file. (In this section we refer to the functions in `mathutil.h` as math functions.) In most cases the math functions in the UL have special functionality or special implementation (e.g., improved efficiency over traditional functions).

6.2 Reference for Util Library

This reference section lists the functions of the UL. The functions are listed according to category.

Table of contents:

6.2.1Array Object
Constructing Arrays
Determining Size
Interfacing with the Elements
6.2.2List Object
6.2.3 Math Functions

6.2.1 Array Object

Here we list the functions of the `Array<T>` class. These functions are listed by category.

Constructing Arrays Most of these functions are listed according to the input of the function.

default – `Array()`
An *empty* array is created.

`array` – `VISArray(const VISArray<T>& s)`
This function creates an array from another array. The array created is exactly the same as `array`.

`size` – `VISArray(unsigned size0)`
This function creates an array containing a specified number of elements. The `size0` parameter specifies the number of elements.

`x0, x1, x2, x3` – `Array(T x0,T x1)`
`Array(T x0,T x1,T x2)`
`Array(T x0,T x1,T x2,T x3)`
These functions create a 2 element to 4 element sized array containing some of `x0`, `x1`, `x2`, and `x3` (depending on the function used).

Determining Size

`n` – `unsigned n() const`
This function returns the number of elements in an array.

`size` – `unsigned size() const`
This function does NOT return the number of elements in an array. We include this function in the reference only to point out that it should NOT be used to find the number of elements in an array. Example:

```
Array<float> A; //size()=0 n()=0
A.appenditem(0.0f); //size()=8 n()=1
```

Interacting with the Elements

`peek` – `const T& peek(unsigned idx) const`
: Returns the contents of element number `idx` in the array. (First element in the array is `idx`=0.)
Example:

```
Array<VISMatrix> m_array(4);
VISMatrix M(2,2);
M=2.0f;
m_array.poke(2)=M;
VISMatrix N;
N=m_array.peek(2);//Now N is the same as M
```

`operator()` – `const T& operator()(unsigned idx) const`
: Same as `peek`. Returns the contents of element number `idx` in the array. (First element in the array is `idx`=0.) For an example see `peek`.

`poke` – `T& poke(unsigned idx)`
: Returns a reference to the data in element number `idx` of the array. This function is always used on the lhs of an assignment.
Example:

```
Array<VISMatrix> m_array(4);
VISMatrix M(2,2);
M=2.0f;
m_array.poke(2)=M;
```

`refAt` – `T& refAt(unsigned idx) const`
: This routine returns a const pointer to the data of a chosen element of an array.

`appendItem` – `void appendItem(const T& t)`
: Adds an item onto the end of the array.

`prependItem` – `void prependItem(const T& t)`
: Adds an item onto the beginning of the array.

`insertItemAt` – `void insertItemAt(const T& t,unsigned idx)`
: Inserts an item at index `idx`.

`replaceItemWith` – `void replaceItemAtWith(unsigned idx, const T& t)`
: Replaces an item in an array with a new item at index `idx`.

`removeItemAt` – `void removeItemAt(unsigned idx)`
: Removes item from array at index `idx`.

`reverse` – `void reverse()`
: `void reverse(unsigned start,unsigned end)`
The first function reverses the order of the entire array. The second function reverses the elements between indeces `start` and `end`.

`sort` – `void sort()`
: This function performs a (descending) bubble sort, relying on the "¡" and "¿" operators for the template class object.

`operator=` – `Array& operator=(const Array& s)`
: Assigns one array to another. This function copies the array.

`sizeTo` – `virtual void sizeTo(unsigned new_size)`
: Changes the size of an array to `new_size`.

`copy` – `Array<T>* copy() const`
    Returns a copy of an array.

6.2.2 List Object

This list object is a simple implementation of a doubly-linked list. It consists of templated `Link<T>` and `List<T>` objects. This `List` is not a full-blown object, and must be handled by pointer (i.e. the copy is not deep). The `Link` contains pointers to "before" and "after" objects on the list and a reference to data of type `T`. The link has methods for access to data and inserting new elements either before or after, and has a constructor which takes data of type `T`. The list contains a pointers to the head and tail and methods for inserting, removing, and getting access to specific links.

The `Link` object contains the following methods:

`next` – `Link<T> *next() const`
    Returns a pointer to next link on list.

`prev` – `Link<T> *prev() const`
    Returns a pointer to previous link on list.

`append` – `virtual void append(Link<T>* link)`
    Inserts the link after the current one.

`prepend` – `virtual void prepend(Link<T>* link)`
    Inserts the link before the current one.

`data` – `virtual T& data()`
    `virtual void data(T data)`
    Returns or sets data in link.

The `List` object contains the following methods:

`head/tail Link<T> *head() const`
    `Link<T> *tail() const`
    Return the head/tail of the list, respectively.

`append/prepend Link<T> *appendItem(T data)`
    `Link<T> *prependItem(T data)`
    Puts link containing data at the end/beginning of list, respectively.

`insertItem Link<T> *insertItemBefore(T data,Link<T> *link)`
    `Link<T> *insertItemAfter(T data,Link<T> *link)`
    Puts a link containing data before/after the given link, respectively.

6.2.3 Math Functions

`tabs` – `T tabs(const T& a)`
    This function returns the absolute value of `a`.

`power` – `float power(float a, int n)`
  This function raises `a` to the power of `n`.

`min` – `const T& min(const T& a,const T& b)`
  `const T& min(const T& a,const T& b,const T& c)`
  `const T& min(const T& a,const T& b,const T& c,const T& d)`
  `const T& min(const T& a,const T& b,const T& c,const T& d,const T& e,const T& f)`
  The first function returns `a` if `a<b` else `b` is returned. The second function returns `min(a,min(b,c))`. The third function returns `min(min(a,b),min(c,d))`. The fourth function returns `min(min(a,b,c,d),min(e,f))`.

`max` – `const T& max(const T& a,const T& b)`
  `const T& max(const T& a,const T& b,const T& c)`
  `const T& max(const T& a,const T& b,const T& c,const T& d)`
  `const T& max(const T& a,const T& b,const T& c,const T& d,const T& e,const T& f)`
  The first function returns `a` if `a>b` else `b` is returned. The second function returns `max(a,max(b,c))`. The third function returns `max(max(a,b),max(c,d))`. The fourth function returns `max(max(a,b,c,d),max(e,f))`.

`rad_to_deg` – `VISReal rad_to_deg(VISReal radians)`
  Converts radians to degrees.

`deg_to_rad` – `VISReal deg_to_rad(VISReal degrees)`
  Converts degrees to radians.

`rand1` – `float rand1()`
  Returns a uniformly distributed deviate on the [0,1] interval.

`gasdev` – `float gasdev()`
  Returns a normally distributed deviate with zero mean and unit variance, using ran1() as the source of uniform deviates.

`sqr` – `T sqr(T x)`
  Returns the square of `x` (i.e. x^2).

`cub` – `T cub(T x)`
  Returns the cube of `x` (i.e. x^3).

`gaussFast` – `float gaussFast(float a)`
  Computes a normalized Gaussian on `a` using a look up table.

`gaussCumFast` – `float gaussCumFast(float a)`
  Computes a normalized cumulative (integral of Gaussian) on `a` using a look up table.

`cosFast` – `float cosFast(float a)`
  Fast cosine function using look up tables. Returns the cosine of `a` in radians.

`acosFast` – `float acosFast(float a)`
  Fast arccosine function. Returns the arccosine of `a` in radians using a look-up-table.

7 Level-Set Surface-Modeling Library

The Level-Set Surface-Modeling (LSSM) Library is an implementation of the level-set technique [8, 9] specifically for deforming surface models embedded in volumes. The implementation uses the sparse-field method described in [10]. The library implements all of the basic numerical algorithms and handles all of the data structures required to perform LSSM. The strategy for using this library is to subclass the object `VoxModel`, set some parameters, define a set of simple virtual functions that control the deformation process, initialize the model, tell the model to iteratively deform according to those equations. This document assumes the reader is somewhat familiar with level-set techniques, their numerical and LSSM as described in [12]. We begin with a mathematical description of what we are doing and how it relates to the object in the library; this is a short review that does not substitute for a basic familiarity with the literature on the subject. We also present an example of using level-set models to do 3D shape metamorphosis as described in the literature [10, 3].

7.1 Surface Deformation

The LSSM library allows one to solve for surface deformations, as a function of time, for surface movements of the following form:

$$\frac{\partial S}{\partial t} = \alpha F(S, N(S)) + \beta G(S, N(S))N(S) + \gamma N(S) + \eta E\left(H(S), K(s)\right), \tag{2}$$

where $H(S)$ and $K(S)$ are the mean and Gaussian curvatures, respectively, of the surface at a point S. This equation is solved by representing the surface as the Kth level set of an implicit function $\phi(x,t) : \Re^3 \times \Re^+ \mapsto \Re$. This gives

$$\frac{\partial \phi}{\partial t} = \alpha F(x, \nabla\phi)) \cdot \nabla\phi + \beta G(x, \nabla\phi)|\nabla\phi| + \gamma|\nabla\phi| + \eta E(D\phi, \mathrm{D}^2\phi), \tag{3}$$

where $\mathrm{D}\phi$ and $\mathrm{D}^2\phi$ are collections first and second derivatives of ϕ, respectively. In practice equation 3 is solved on a discrete grid using an *up-wind* scheme gradient calculations, centralized differences for the curvature, and forward finite differences in time. The updates for the nth iteration on the discrete grid $u^n_{i,j,k}$ take the form

$$u^{n+1}_{i,j,k} = \Delta u^n_{i,j,k} \Delta t. \tag{4}$$

The LSSM library uses the *sparse-field* method [10], and therefore updates to not take place on the entire grid $u_{i,j,k}$ at each iteration, but rather only on a finite subset of grid points that lie near the *zero level set*, which serves as the surface model. This set of grid points is called the active set.

Thus, the LSSM library offers the following capabilities:

1. Creates an initial model (with associated active set) from a volume.
2. Calculates $\Delta u^n_{i,j,k}$ and Δt using virtual functions (defined by subclasses) that describe F and G, and parameters (values set by the subclass) α, β, γ, and η.
3. Performs an update on the values of $u^n_{i,j,k}$.
4. Maintains the list of active grid points and updates the *layers* around those points in order to maintain a neighborhood from which to calculate subsequent updates.
5. Provides access to the volume that defines $u^n_{i,j,k}$ and the linked list of active grid points.

Given the volume defining $u^n_{i,j,k}$, one can then rely on the functionality of the Volume Library (see Section 5) for subsequent processing, file I/O, or surface extraction.

Figure 2: The base class `LevelSetModel` keeps track of the active set and surrounding layers, while the derived class `Voxmodel` performs updates on the active set from a set of virtual functions.

7.2 Structure and Philosophy of the LSSM Library

The library is organized (mostly for ease of development) into a base class, `LevelSetModel`, and a derived class, `VoxModel`, as shown in Figure 2. The base class does all of the book keeping associated with the active set and surrounding *layers*, the link lists associated with those sets, and initializing the model. Thus it adds and removes voxels from the active set (and surrounding layers) in response to an update operation. The base class assumes that the subclasses know how to update individual voxels.

The subclass, `VoxModel`, performs update on the grid points in the active set of the form given in Equation 4, using functions F and G and parameters α, β, γ, and η. It also calculates the maximum Δt that ensures stability.

Thus, a user who wishes to perform a surface deformation using the LSSM library, would create subclass of `VoxModel` and define the appropriate virtual functions and set the parameters to achieve the desired behavior.

Note, these two classes are not *full fledged* objects, but rather heavy-duty structure. The copy constructors and assignment operators for these classes are not well behaved, and therefore these classes should be passed by reference when sharing them between other routines/modules.

7.3 The `LevelSetModel` Object

The `LevelSetModel` contains a volume of values, a volume of status flags, five lists (one active list, two inside lists, and two outside lists), and three parameters that determine the origin of the coordinate system form which the model performs its calculations.

There are two constructors, `LevelSetModel()` and `LevelSetModel( const VISVolume<float> &)`. The first simply initializes the data structure, and the second also set the values of the model volume (`_values`) to the input. Once the values have been set, one can create an initial volume from those values by calling `constructLists()`, which can also take a floating-point argument that controls the scaling of the input relative to a local distance transform near the zero set.

The list that keeps track of the active set, called `_active_list`, keeps track of the location of those grid points and a single floating-point value, which stores the change in their values from one iteration to the next.

Another important methods for users of this object is `update(float)`, which changes the grey-scale values of the grid for the active set according to the values stored in `_active_list`, and updates the status of elements on the active list as well as the values and status of nearby layers (2 inside and 2 outside). The floating point argument is the value of Δt from Equation 4, and the return value is the maximum change that occurred on the active set. Finally, the method `iterate()` calls the virtual method `calculate_change`, a virtual function which sets the values of $\Delta u^n_{i,j,k}$ and returns the maximum value of Δt for stability, and then calls `update`. For this object the function `calculate_change` performs some trivial (i.e., useless) operation.

7.4 The `VoxModel` Object

The `VoxModel` object is a subclass of `LevelSetModel`, and it add three things to the base class.

1. `calculate_change()` is redefined to implement the surface deformation described in Equation 3.
2. The virtual functions are declared for F (called `force`) and G (called `grow`). These functions are defined to return null values for this object.
3. Parameters are introduces which control the relative influence of the various terms. A routine `load_params` is defined which simply reads these parameter from a file.
4. A method `rescale(float)` is defined, which resamples the volume of grid-point values into a new volume with different resolution and redefines the lists (and thereby the model) in this new volume. This method is for performing coarse-to-fine deformation procedures.

7.5 Example: 3D Shape Metamorphasis

The `Morph` object allows one to construct a sequence of volumes or surface meshes using the 3D shape metamorphasis technique described by

Breen and Whitaker []. This technique relies distance transforms for both the source and target objects and uses a LSSMs to manipulate the shape of the source so that it coincides with the target. The surface deformation that describes this behavior is

$$\frac{\partial S}{\partial t} = \beta G\left(T(S)\right) N(S), \tag{5}$$

where $G(x)$ is simply the distance transform (or some monotonic function thereof) of the target, and T is a coordinate transformation that aligns the source and target objects. The level-set formulation of this is

$$\frac{\partial \phi(x,t)}{\partial t} = \beta G\left(T(S)\right) |\nabla \phi|. \tag{6}$$

The morphing process consists of several steps:

1. Read in distance transforms (in the form of volumes) for both source and target.
2. Initialize the LSSM by fitting it to the zero set of the source distance transform.
3. Update the LSSM according to Equation 6.
4. Save intermediate volumes/surfaces at regular intervals.

The remainder of this section lists the code and comments for three files, morph.h (which declares the `Morph` object), morph.C (which defines the methods) and main.C (which performs all of the I/O and uses the `Morph` object to construct a sequence of shapes.

7.6 Morph.h

```
//
// morph.h
//
//

#ifndef iris_morph_h
#define iris_morph_h

#include "voxmodel/voxmodel.h"
#include "matrix/matrix.h"

#define INIT_STATE 0
#define MORPH_STATE 1
//
// This is the morph object.  It uses all of the machinery of the base
// class to manipulate level sets.  It needs to have an initial volume
// and a final volume (which would typically be the distance transform,
// it might need a 3D transformation, and it needs to redefine the
// virtual function "grow", which takes 6 floats as input, the position
// followed by the normal vectors (all will calculated and passed into
// this method by the base class).  It might also have a state, that
// indicates whether or not it's been initialized.
//
// Functions not defined here should be defined in "morph.C"
//
class Morph: public VoxModel
{
  protected:
    VISVolume<float> _dist_source;
    VISVolume<float> _dist_target;
    VISMatrix _transform;
//
// This is the function that is used by the base class to manipulate the
level
// set.  You can define it to by anything you want.  For this object, it
will
// return a value from the distance transform of the target.
//
    virtual float grow(float x, float y, float z,
         float nx, float ny, float nz);

// There are two states.  In the first state, the model is trying to fit
// to the input data.  In this way the models starts by looking just like

// the input data
    int _state;

  public:

    Morph(const Morph& other)
    {
 _dist_target = other._dist_target;
 _initial = other._initial;
 _state = MORPH_STATE;
 _transform = VISVISMatrix(3, 3);
 _transform.identity();
// initialize();
    }

    Morph(VISVolume<float> init, VISVolume<float> d)
 :VoxModel()
    {
 _dist_target = d;
 _initial = init;
 _state = MORPH_STATE;
 _transform = VISVISMatrix(3, 3);
 _transform.identity();
// initialize();
    }

    void initialize();

// for this object I assume that the transform is just a matrix.
// but it could be anything
    void transform(const VISVISMatrix& t)
    { _transform = t;}

    const VISVISMatrix& transform()
    { return(_transform);}

    void distance(const VISVolume<float> d)
    { _dist_target = d;}
    VISVolume<float> distance()
    { return(_dist_target);}

};
#endif
```

7.7 Morph.C

```
#include "morph.h"
#include "util/geometry.h"
#include "util/mathutil.h"


//
// this is the virtual function, that  is the guts of it all.
//

float Morph::grow(float x, float y, float z,
    float nx, float ny, float nz)
```

```
{

// this says you are in the morph state (things have been initialized)
    if (_state == MORPH_STATE)
 {
 float xx, yy, zz;
 VISPoint p(4u);
 p.at(0) = x;
 p.at(1) = y;
 p.at(2) = z;
 p.at(3) = 1;
 VISPoint p_tmp;
// this is where you could put some other transform.
 p_tmp = _transform*p;

 xx = p_tmp.x();
 yy = p_tmp.y();
 zz = p_tmp.z();

// make sure you are not out of the bounds
// of your distance volume.
 if (_dist_target.checkBounds(xx, yy, zz))
// if not, get the distance (use trilinear interpolation).
     return(_dist_target.interp(xx, yy, zz));
 else
     return(0.0f);
    }
else
    {
// if you are still initializing, then move toward the zero set of
// your initial case
 if (_initial.checkBounds(x, y, z))
     return(_initial.interp(x, y, z));
 else
     return(0.0f);
    }
}

// this makes the model look like the input.
#define INIT_ITERATIONS 5
void Morph::initialize()
{
    _values = _initial;
    int state_tmp = _state;
    _state = INIT_STATE;
    construct_lists(DIFFERENCE_FACTOR);
// these couple of iterations are required to make sure that the zero
// sets of the model match the zero sets of the
//
    for (int i = 0; i < INIT_ITERATIONS; i++)
 {
// limit the dt to 1.0 so that the model settles in to a solution
```

```
     update(::min(calculate_change(), 1.0f));
 }
    _state = state_tmp;
}
```

7.8 Main.C

```
#include "vol/volume.h"
#include "vol/volumefile.h"
#include "image/imagefile.h"
#include "morph.h"
#include <string.h>


const int V_HEIGHT = (40);
const int V_WIDTH = (40);
const int V_DEPTH = (40);

#define XY_RADIUS (12)  // this matches the 2.5D data generated in
torus.C
#define T_RADIUS (4)  // this matches the 2.5D data generated in torus.C
#define S_RADIUS (12)  // radius of a sphere

#define B_WIDTH (20.0f)
#define B_HEIGHT (60.0f)
#define B_DEPTH (20.0f)

#define B_CENTER_X (12.0f)
#define B_CENTER_Y (32.0f)
#define B_CENTER_Z (12.0f)

float sphere(unsigned x, unsigned y, unsigned z);
float torus(unsigned x, unsigned y, unsigned z);
float cube(unsigned x, unsigned y, unsigned z);


// This is a program that does the morph.  If you give it two
// arguments, it reads the initial model and the dist trans for the
// final model from the two file names given, otherwise, it makes a
sphere
// and deforms it into a torus

main(int argc, char** argv)
{

VISVolume<float> vol_source, vol_target;
VISVolumeFile vol_file;
int i;
char fname[80];
```

```
vol_source = VISVolume<float>(25,65,25);
vol_source.evaluate(cube);

if (argc > 2)
    {
// read in the sourceing model
 vol_source = VISVolume<float>(vol_file.read_float(argv[1]));
// read in the dist trans of the final model
     vol_target = VISVolume<float>(vol_file.read_float(argv[2]));
    }
else
// make up some volumes
    {
 vol_source = VISVolume<float>(V_WIDTH, V_HEIGHT, V_DEPTH);
 vol_source.evaluate(sphere);
 vol_target = VISVolume<float>(V_WIDTH, V_HEIGHT, V_DEPTH);
 vol_target.evaluate(torus);
    }

// create morph object
Morph morph(vol_source, vol_target);
// loads in some parameters (for morphing these are all zero but one)
// i.e.
//
//
//
//
morph.load_parameters("morph_params");
morph.initialize();
vol_file.write_float(morph.values(), "morph0.flt");

float dt;

// do 150 iterations for your model to get from start to finish
// probably don't need this many iterations

for (i = 0; i < 150; i++)
    {
 dt = morph.calculate_change();
// limit dt to 0.5 so that model never overshoots goal
 dt = min(dt, 0.5f);
 morph.update(dt);

 printf("iteration %d dt %f\n", i, dt);

 if (((i + 1)%10) == 0)
 {
// save every tenth volume
     sprintf(fname, "morph_out.%d.dat", i + 1);
     vol_file.write_float(morph.values(), fname);
 }
    }
```

```
// save a surface model (i.e. marching cubes).
vol_file.march(0.0f, morph.values(), ‘‘morph_final.iv’’);

printf("done\n");

}
```

A Parameter File Objects

A.1 Introduction

When implementing algorithms, it is often the case that a significant number of values are needed to initialize various constants, specify input and output files, or otherwise tune the performance of the algorithm. Specifying increasingly more and more parameter values as command line arguments becomes unwieldy for both the developer and the user of the software. The `ParameterFile` object is intended to simplify the process by defining a standard file format for specifying parameters and a simple protocol for reading those files.

This document discusses the `ParameterFile` object design and its user interface. Examples are provided.

A.2 Implementation

The `ParameterFile` object encapsulates the process of opening, reading, and parsing a file into the instantiation of a single high level object with a limited public interface. The implementation consists of three components: a token scanner, a parser, and a syntax tree. The parser queries the token scanner for input and constructs a syntax tree of objects whose root node is the `ParameterFile` object.

The token scanner (`yylex()`) is created using the `flex` utility, a standard lexical analyzer generator. The set of tokens T,

$$T = (\mathbf{identity}, \mathbf{string}, \mathbf{integer}, \mathbf{decimal}, \mathbf{rparen}, \mathbf{lparen})$$

recognized by the scanner are encoded in the file `param.l` as regular expressions.

The parser is generated with the standard `yacc` utility and then compiled into a function `yyparse()`. When invoked, `yyparse()` calls `yylex()` for input and attempts to identify syntactic structures of the grammar $G^3 = (V, T, P, \mathbf{ParameterFile})$ where V is the set of variables, $V = (\mathbf{ParameterFile}, \mathbf{Parameter}, \mathbf{Value})$, and P is the set of productions:

$$\begin{array}{l} \mathbf{ParameterFile} \Rightarrow\ \mathbf{Parameter}^{*} \\ \mathbf{Parameter} \Rightarrow\ \mathbf{lparen\ identity\ Value}^{*}\ \mathbf{rparen} \\ \mathbf{Value} \Rightarrow\ (\ \mathbf{string} \mid \mathbf{integer} \mid \mathbf{decimal}\ ) \end{array}$$

As productions in the grammar are identified, `yyparse()` constructs a "syntax tree" of objects representing those productions. The tree is constructed by the parser from the bottom up. All the objects are subclassed from a pure virtual `VPF::node` object, thereby enabling run-time polymorphism of node object methods.

A.3 Parameter file format

A parameter file as described by the grammar G, is an ASCII text file supplying a list of parameters of the form,

[3] G is defined in the file `param.y`.

```
(parameter_name optional_value1 optional_value2 ... optional_valueN)
```

The parameter naming conventions are similar to C language variable naming conventions. A parameter name must begin with an alphanumeric character and may consist of any of the letters `A-Z`,`a-z`; numbers `0-9`; and the underscore character _.

The optional values may be either integer, decimal, or string values. Decimal numbers can be either floating point or double precision[4]. A number value lacking a decimal point character is assumed to be an integer value.

A string value is specified by enclosing any characters (except the newline and " characters) in quotation marks ".

```
(parameter_name "string value number 1.1" "stri%$ng value number 2")
```

The value types for a parameter can be mixed. For example,

```
(mixed_values  45.23 1 0 2 3234.99999 "string")
```

C++-style commenting is supported. C-style commenting is not supported. All characters after double forward slashes `//` up to a newline character are ignored.

A.4 The `ParameterFile` object

Introduction When implementing algorithms, it is often the case that a significant number of values are needed to initialize various constants, specify input and output files, or otherwise tune the performance of the algorithm. Specifying increasingly more and more parameter values as command line arguments becomes unwieldy for both the developer and the user of the software. The `ParameterFile` object is intended to simplify the process by defining a standard file format for specifying parameters and a simple protocol for reading those files.

This document discusses the `ParameterFile` object design and its user interface. Examples are provided.

Implementation The `ParameterFile` object encapsulates the process of opening, reading, and parsing a file into the instantiation of a single high level object with a limited public interface. The implementation consists of three components: a token scanner, a parser, and a syntax tree. The parser queries the token scanner for input and constructs a syntax tree of objects whose root node is the `ParameterFile` object.

The token scanner (`yylex()`) is created using the `flex` utility, a standard lexical analyzer generator. The set of tokens T,

$$T = (\mathbf{identity}, \mathbf{string}, \mathbf{integer}, \mathbf{decimal}, \mathbf{rparen}, \mathbf{lparen})$$

recognized by the scanner are encoded in the file `param.l` as regular expressions.

[4]Most variants on these types are also supported (long double, etc). Check the header files for a complete list of types supported.

The parser is generated with the standard `yacc` utility and then compiled into a function `yyparse()`. When invoked, `yyparse()` calls `yylex()` for input and attempts to identify syntactic structures of the grammar $G^5 = (V, T, P, \textbf{ParameterFile})$ where V is the set of variables, $V = (\textbf{ParameterFile}, \textbf{Parameter}, \textbf{Value})$, and P is the set of productions:

$$\begin{array}{l} \textbf{ParameterFile} \Rightarrow\ \textbf{Parameter}^* \\ \textbf{Parameter} \Rightarrow\ \textbf{lparen identity Value}^*\ \textbf{rparen} \\ \textbf{Value} \Rightarrow\ (\ \textbf{string} \mid \textbf{integer} \mid \textbf{decimal}\ ) \end{array}$$

As productions in the grammar are identified, `yyparse()` constructs a "syntax tree" of objects representing those productions. The tree is constructed by the parser from the bottom up. All the objects are subclassed from a pure virtual `VPF::node` object, thereby enabling run-time polymorphism of node object methods.

Parameter file format A parameter file as described by the grammar G, is an ASCII text file supplying a list of parameters of the form,

```
(parameter_name optional_value1 optional_value2 ... optional_valueN)
```

The parameter naming conventions are similar to C language variable naming conventions. A parameter name must begin with an alphanumeric character and may consist of any of the letters `A-Z`,`a-z`; numbers `0-9`; and the underscore character _.

The optional values may be either integer, decimal, or string values. Decimal numbers can be either floating point or double precision[6]. A number value lacking a decimal point character is assumed to be an integer value.

A string value is specified by enclosing any characters (except the newline and `"` characters) in quotation marks `"`.

```
(parameter_name "string value number 1.1" "stri%$ng value number 2")
```

The value types for a parameter can be mixed. For example,

```
(mixed_values  45.23 1 0 2 3234.99999 "string")
```

C++-style commenting is supported. C-style commenting is not supported. All characters after double forward slashes `//` up to a newline character are ignored.

The `ParameterFile` object The `ParameterFile` object is a "nice" object, meaning that the copy constructor and assignment operators are defined to perform deep copies. Instances of `ParameterFile` objects can be safely passed by reference or by value.

A `ParameterFile` object can be instantiated with a character string, a copy constructor, or by assignment,

[5] G is defined in the file `param.y`.

[6] Most variants on these types are also supported (long double, etc). Check the header files for a complete list of types supported.

```
VPF::ParameterFile p1("filename");  // filename
VPF::ParameterFile p2(p1);     // copy constructor
VPF::ParameterFile p3;  p3 = p2;    // assignment
```

Note the use of the scope resolution operator. The `ParameterFile` objects are enclosed in a namespace `VISParameterFile` aliased to `VPF`.

When instantiated with a character string argument, the constructor attempts to open and parse the file whose name is represented in the argument, calling the yyparse() method. On success, a pointer to the resulting syntax tree is held in the `ParameterFile` object. The `ParameterFile` object contains the following public methods:

`bool valid()`	Returns `true` if object contains a valid syntax tree, `false` otherwise.
`int size()`	Returns the number of parameters contained in the syntax tree.
`bool empty()`	Returns `true` if the object's valid syntax tree is empty, `false` otherwise.
`void clear()`	Clears the syntax tree and frees all associated memory.
`void print()`	Prints a formatted list of everything in the syntax tree to standard out. For debugging.

The syntax tree of the `ParameterFile` object can be thought of as an ordered list of `Parameter` objects. The `[]` operator has been overloaded to allow (read) access to the parameters. Supplying an integer argument i returns the i^{th} parameter in the list, as ordered in the input file. No bounds checking is performed by this operation. Supplying a character string argument returns the the parameter whose name matches the argument. If no match is found, a "null parameter" is returned whose validity is `false` (see next section).

The `Parameter` object A `Parameter` object is also a "nice" object in the sense that it can be passed by value or reference and its assignment operator is overloaded. A `Parameter` object has the following methods:

`bool valid()`	Returns `true` if the parameter is valid, `false` otherwise.
`int size()`	Returns the number of values contained in the parameter.
`bool empty()`	Returns `true` if the object's value list is empty, `false` otherwise.
`const char *getName()`	Returns the name of the parameter.
`void print()`	Prints the parameter name and its values to standard out. For debugging.

A `Parameter` object contains a character string name and an ordered list of `Value` objects. The `[]` operator has been overloaded to allow (read) access to this list. No bounds checking is performed by the `[]` operator. Note that since a `Value` object may be of several types, the assigment operator cannot be overloaded for the `Value` object. Hence the value contained in the object can only be accessed by downcasting to the appropriate class and using that class's `getValue` method (see example section). The preferred method for accessing a value is through the overloaded `set` method described in the next section.

The `set` method The `VPF::set` method has been overloaded for all the supported variable types. It takes as its arguments a variable of the appropriate type and a `Value` object and returns `true` if the operation

```
// This is a sample file to demonstrate the format readable by the
// ParamterFile object

// Parameters are enclosed in parentheses and take the form:
//
// (parameter_name optional_value1 optional_value2 .... optional_valueN)
//
// C++ style comments can be used.  Note C-style comments are NOT supported.
//

(bounding_box1 0 100 50 -50) (k .15)

(forsythia "the tree with the" "lights in it" "edgy" "2394 sljdkf09128 ...sdfk238"
"321s lskdl 02302...&*&%$#!" "right now strings cannot be more"
"than VPF::MAXSTRLEN characters long" )

(flag_value)  // No parameter values is OK, too.

(mixed_types 0 "string_value" 34.2222 -0.987) // mixes types are OK, too

// SOME COMMENTS

(d 1.13425835)  // MORE COMMENTS
```

Figure 3: sample.txt

was successful and `false` otherwise. The declaration of the set method has the following form:

```
VPF::bool set(<variable type> &, VPF::Value &)
```

The method works by first checking the type of the supplied `Value` object to make sure it is consistant with the requested variable type. Then the value contained in the `Value` object is cast to the appropriate type and the assignement is made.

Error handling At the time of this writing, C++ exception handling is not used in the `ParameterFile` code. Some run-time warnings are printed to standard error and true exceptions such as "File not found" result in a call to `exit()`. Syntax and scanner errors are reported to standard error, but the parser cannot usually recover from such errors and will cause a `ParameterFile` instantiation to fail with a call to `exit`.

Examples Figure 3 shows a sample input file that is readable by the `ParameterFile` parser. Figures 4 shows a small sample program that reads in some of these values.

B Code Examples

The code examples shown here can be found in the `examples` directory. There are three examples: `matrix` uses the Matrix Library, `shotnoise` uses only the Image Library, and `fractal` uses both the Volume and Image Libraries. Examine the `README` files in the directories of these examples to learn the function of

```
#include <iostream.h>
#include "param.h"

main(int argc, char *argv[]) {
  int x1, x2, y1, y2;
  float k;
  double d;
  char string[VPF::MAXSTRLEN];
  VPF::ParameterFile F("sample.txt");

  if ( ! F.valid() ) { cerr << "Problems reading input file." << endl; exit(1); }
  if ( F.empty()   ) { cerr << "Input file is empty" << endl; exit(2); }

  if (F["bounding_box1"].valid() && F["bounding_box1"].size() == 4)
{  if ( ! VPF::set(x1, F["bounding_box1"][0]) )
cerr << "Argument 0 of \"bounding box\" is not an integer value" <<
  endl;
  if ( ! VPF::set(x2, F["bounding_box1"][1]) )
   cerr << "Argument 1 of \"bounding box\" is not an integer value" <<
  endl;
  if ( ! VPF::set(y1, F["bounding_box1"][2]) )
   cerr << "Argument 2 of \"bounding box\" is not an integer value" <<
  endl;
  if ( ! VPF::set(y2, F["bounding_box1"][3]) )
   cerr << "Argument 3 of \"bounding box\" is not an integer value" <<
  endl;
}
  else
{ cerr << "\"bounding_box1\" is not a valid parameter name" << endl;
  exit(3);
}
  // Set a floating point value
  if (F["k"].valid() && F["k"].size() > 0)
{ if ( !VPF::set(k, F["k"][0]) )
  cerr << "Argument 1 of \"k\" is not a decimal value" << endl;
}
  else { cerr << "\"k\" is not a valid parameter name" << endl; }

  // Set a double value
  if (F["d"].valid() && F["d"].size() > 0)
{ if ( !VPF::set(d, F["d"][0]) )
  cerr << "Argument 1 of \"d\" is not a decimal value" << endl;
}
  else { cerr << "\"d\" is not a valid parameter name" << endl; }

  // Set a string value.
  // Note that the string set function assumes a buffer of at least VPF::MAXSTRLEN
  // characters.

  if (F["forsythia"].valid() && F["forsythia"].size() > 4)
{
  if ( !VPF::set(string, F["forsythia"][3]) )
cerr << "Argument 3 of \"forsythia\" is not a string value" <<endl;
}
  else { cerr << "\"forsythia\" is not a valid parameter name" << endl; }
}
```

Figure 4: example.C

these programs. Also, the examples each include a `Makefile`. These Makefiles show how to include all the necessary libraries and directories to create an application.

Tips about Makefiles

- Notice in the `matrix` example there are several libraries included that are used by LAPACK, i.e., `-llapack -lF77 -lblasirix` (see 3.1.3 for explaination of LAPACK).
- On our compilers the order of the libraries is significant. For example, in the `fractal` example: `-lvol -limage -lutil -ltiff -lcfitsio -lm` may work while `-lm -lcfitsio -ltiff -lutil -limage -lvol` will cause a runtime linking error. If you have unresolvable symbol problems at runtime try changing the order of your libraries.

B.1 `matrix` example

```
\begin{tt}
#include <stream.h>
#include <stdlib.h>
#include "matrix.h"

/////////////////////////////////////////////////////////////////////////
//    Summary: This program contains several 'main' functions.  Each   //
//             one tests a different aspect of the matrix library.     //
//             By commenting/uncommmenting portions of the code,       //
//             different aspects of the matrix library may be examined //
//    Output:  usually stdout
//    Author:  Samuel G. Burgiss Jr.                                   //
/////////////////////////////////////////////////////////////////////////

//test of operator=
main(char** argv, int argc){
    cout << "m1" << endl;
    VISMatrix M1(3,3);
    M1=1.0f;
    cout << "m2" << endl;
    VISMatrix M2;
    cout << "m2made" << endl;
    M2=M1;
    cout << "m3" << endl;
    VISMatrix M3;
    M3=3.0f;
    cout << "m3made" << endl;
    M1=M3;
    cout << M2 << endl;
}

//test of det()
/*main(char** argv, int argc){
    VISMatrix A("input4");
    cout << A << endl;
    cout << A.det() << endl;
```

```
}*/

//test of operator()
/*main(char** argv,int argc){
    VISMatrix A("input4");
    cout << A(1,1) << endl;
    cout << A(2,2) << endl;
}*/

//test of operator==()
/*main(char** argv,int argc){
    VISMatrix A("input4");
    VISMatrix B,C,D;
    B = A;
    B.poke(1,1) = 100.0f;
    C = A==A;
    D = A==B;
    cout << A << endl;
    cout << B << endl;
    cout << C << endl;
    cout << D << endl;
}*/

/*
//test pokeROI and peekROI
main(char** argv,int argc){
    VISMatrix A("input6");
    VISMatrix B("input5");
    cout << "A" << endl << A << endl;
    cout << "B" << endl << B << endl;
    cout << "peekROI(0,2,4,5) - startrow,endrow,startcol,endcol" << endl;
    cout << A.peekROI(0,2,4,5) << endl;
    A.pokeROI(1,1,B);
    cout << "A.pokeROI(1,1,B);" << A << endl;
    VISVector a;
    a=A.vec(2);
    cout << "a" << endl << a << endl;
    cout << "a.peekROI(2,3)" << endl  << a.peekROI(2,3) << endl;
    VISVector b;
    b = a.peekROI(2,3);
    cout << a.pokeROI(8,b) << endl;
}*/

//test of +=, -=, *=
/*main(char** argv,int argc){
    VISMatrix A("input4");
    VISMatrix B("input5");
    cout << "A" << endl << A << endl;
    cout << "B" << endl << B << endl;
    A+=B;
    cout << "A+=B" << endl << A << endl;
    cout << "B" << endl << B << endl;
    A+=A;
    cout << "A+=A" << endl << A << endl;
    A-=A;
    cout << "A-=A" << endl << A << endl;
    A+=B;
    A+=B;
    cout << "A+-B;A+=B" << endl << A << endl;
    A*=A;
    cout << "A*=A" << endl << A << endl;
    cout << "A/5.0f" << endl << (A/5.0f) << endl;
    A/=5.0f;
    cout << "A/=5.0f" << endl << A << endl;
    A*=5.0f;
    cout << "A*=5.0f" << endl << A << endl;
}*/

//test for vector class
/*main(char** argv, int argc){
    VISVector v1;
    VISMatrix m(6,1);

    for (int i=0;i<6;i++)
m.poke(i,0)=(float)i;

    cout << "Assignment and creation of a vector." << endl;
    cout << "m:" << endl << m << endl;

    v1=m+m;

    cout << "v1=m+m:" << endl << v1 << endl;

    cout << "Dot product." << endl;
    cout << "v1 dot v1: " << v1.dot(v1) << endl;

    VISVector v2(3);
    cout << "v2" << endl << v2 << endl;
    VISVector v3(3);
    cout << "crap" << endl;
    v2=1.0f;
    cout << "v2" << endl << v2 << endl;
    v2.poke(1)=2.0f;

    cout << "v2:" << endl << v2 << endl;

    v3=3.0f;

    cout << "v3:" << endl << v3 << endl;

    v2.cross(v3);

    cout << "v2 X v3:" << endl << v2.cross(v3) << endl;
    cout << v2 << v3 << endl;

    VISVector v4;
    cout << "v2+v3" << endl << v2+v3 << endl;
    v4 = v2+v3;

    cout << "v2" << endl << v2 << endl;
    cout << "v3" << endl << v3 << endl;
    cout << "v2+v3" << v2+v3 << endl;
    cout << "ans" << endl << v4 << endl;
}*/

//test the concat algorithms
/*main(char** argv, int argc){
    VISMatrix a("input");
    cout << "a" << endl << a << endl;
    VISVector b(4);
    b=2.0f;
    b.poke(0) = 3.0f;
    b.poke(1) = 4.0f;
    cout << "b" << endl << b << endl;
    cout << "a.concatcol(b)" << endl << a.concatcol(b) << endl;
    cout << "b.concatrow(b)" << endl << b.concatrow(b) << endl;
    cout << "a.concatrow(a)" << endl << a.concatrow(a) << endl;

}*/

//testing of the svd decomposition and
//the covariance function
//fitting a plane

/*main(char** argv, int argc){
    VISMatrix C,U,W,V;
    VISVector n;
    float minval,d;
    int r;
    VISMatrix b("pts3");
    C=(b.t()).cov();
    cout << "b.cov()" << endl << C << endl;
    C.svd(U,W,V);
    cout << "U" << endl << U << endl;
    cout << "W" << endl << W << endl;
    cout << "V" << endl << V << endl;
    minval = (W.vecfromdiag()).min(r);
    cout << "minval  " << minval << endl;
    cout << "r       " << r << endl;
    n = -(V.vec(r));
    cout << "n" << endl << n << endl;
    d = -1.0f*((n.t()*(b.t()).mean()).peek(0));
    cout << "d       " << d << endl;
}*/
```

```
//test the min,max stuff.
/*main(char** argv, int argc){
    VISMatrix b("input3");
    cout << "b" << endl << b << endl;
    int r,c,n;
    cout << "b.max()  " << b.max() << endl;
    n=b.max(r,c);
    cout << "n  " << n << endl;
    cout << "r  " << r << endl;
    cout << "c  " << c << endl;
}*/

//test transpose
/*main(char** argv, int argc){
    VISVector v(1,2,3);
    cout << "v" << endl << v << endl;
    cout << "v.t()" << endl << v.t() << endl;
}*/

// testing the fstream stuff
// writing to and reading from a file
/*main(char** argv, int argc){
    VISMatrix temp_matrix;
    ifstream ultrafile("input");
    if(!ultrafile){
        cout << "Unable to open the file:  input" << endl;
        exit(0);
    }
    ultrafile >> temp_matrix;
    cout << "temp_matrix" << endl << temp_matrix << endl;
}*/

//test inverse functions
/*main(char** argv, int argc){
    float **nra,**nrb;

    int i,j;
    int m=3;
    int n=3;

    VISMatrix a(m,n),b;

    a.poke(0,0)=2.0f;
    a.poke(0,1)=4.0f;
    a.poke(0,2)=2.0f;
    a.poke(1,0)=13.0f;
    a.poke(1,1)=23.0f;
    a.poke(1,2)=24.0f;
    a.poke(2,0)=4.0f;
    a.poke(2,1)=23.0f;
    a.poke(2,2)=7.0f;
    cout << "a: " << endl << a << endl;
    b = a.inversesvd();
    cout << "b: " << endl << b << endl;
    cout << "inversegj" << endl << a.inversegj() << endl;
}
*/

\end{tt}
```

B.2 shotnoise example

```
\begin{tt}
#include "image/image.h"
#include "image/imagefile.h"
#include "util/mathutil.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

//////////////////////////////////////////////////////////////////////////
//    Summary: Adds shot noise to an image.                             //
//    Usage:   shotnoise percent min max input_filename output_filename//
//    Output:  fits file (filename chosen by user)                      //
//    Author:  Samuel G. Burgiss Jr.                                    //
//////////////////////////////////////////////////////////////////////////


int process(int num);

main(int argc, char** argv)
{

    char          infilename[80];
    char                outfilename[20];
    float                 min,max;
    FILE           *fp;
    VISIm              image;
    VISImage<float>   output_image;
    VISImage<float>   input_image,deriv_x,deriv_y,added;
    VISImageFile       input_file,output_file;
    float percent;

    if (argc<4){
cout << "summary:  Adds shot noise of particular characteristics" << endl;
cout << "          to an image." << endl;
cout << "usage:    shotnoise percent min max input_filename output_filename" << endl;
cout << "          percent = percent of pixels affected" << endl;
cout << "          min = min of the noise amplitude" << endl;
cout << "          max = max of the noise amplitude" << endl;
       exit(0);
```

```
        }
    // Load in the arguments
    percent = (float)atof(argv[1]);
    min = (float)atof(argv[2]);
    max = (float)atof(argv[3]);
    strcpy (infilename,argv[4]);
    strcpy (outfilename,argv[5]);
    // Open the file and check to see that it is valid.
    fp = fopen(infilename, "r");
    if (fp == NULL){
cout << "Unable to open " << infilename << endl;
    exit(0);
    }
    if (!((image=input_file.read(infilename)).isValid())){
cout << "Error reading image file " << infilename << endl;
        exit(0);
    }
    input_image=VISImage<float>(image);
    // Print out the characteristics of the noise.
    cout << "percent = " << percent << endl;
    cout << "min =     " << min << endl;
    cout << "max =     " << max << endl;
    // Add the noise to the image
    output_image=input_image.noiseShot(percent, min, max);
    // Write the output file in fits format
    output_file.write_fits(output_image,outfilename);

}

\end{tt}
```

B.3 fractal example

```
\begin{tt}
#include "image/image.h"
#include "image/imagefile.h"
#include "volume.h"
#include "volumefile.h"

#include <math.h>
#include <string.h>
#include <iostream.h>
#include <fstream.h>

//////////////////////////////////////////////////////////////////////////
//    Summary: Creates a 3D surface by applying the marching cubes      //
//             algorithm to a volume of data that is created using the //
//             mandelbroth set.                                         //
//    Output:  fits file (filename chosen by user)                      //
//    Author:  Samuel G. Burgiss Jr.                                    //
//////////////////////////////////////////////////////////////////////////
```

```
void write_volume_flat(VISVolume<float> V, char* filename);

main(int argc, char** argv){
    float tempza;
    float begin_c,begin_ci;
    float end_c,end_ci;
    int number_slices;
    float begin_w;
    float end_w;
    int window_w;
    float begin_h;
    float end_h;
    int window_h;
    float increment_w,increment_h,increment_c,increment_ci;
    int volume_index_w,volume_index_h,slice_index;
    float cnt_w,cnt_h,cnt_c,cnt_ci;
    float za,zb;
    int iter,max_iter;
    float march_thresh;

    begin_c = -2.0f;
    begin_ci = 0.0f;
    end_c = 2.0f;
    end_ci = 0.0;
    number_slices = 100;
    begin_w = -2.0f;
    end_w = 2.0f;
    window_w = 100;
    begin_h = -2.0f;
    end_h = 2.0f;
    window_h = 100;
    max_iter = 30;
    march_thresh = 5.5;


    VISVolume<float> V(window_w+3,window_h+3,number_slices+3);
    V=0.0f;

    increment_w = (end_w-begin_w)/(float)window_w;
    increment_h = (end_h-begin_h)/(float)window_h;
    increment_c = (end_c-begin_c)/(float)number_slices;
    increment_ci= (end_ci-begin_ci)/(float)number_slices;

    volume_index_w=1;
    slice_index=1;
    cnt_ci=begin_ci;
    cout << "Making fractal volume";
    for (cnt_c=begin_c;cnt_c<end_c;cnt_c=cnt_c+increment_c){
cout << ".";
volume_index_w=1;
for (cnt_w=begin_w;cnt_w<end_w;cnt_w=cnt_w+increment_w){
    volume_index_h=0;
    for (cnt_h=begin_h;cnt_h<end_h;cnt_h=cnt_h+increment_h){
za=cnt_w;
zb=cnt_h;
iter=0;
//cout << za << "     " << zb << "    " << iter << endl;
while ((iter<max_iter) && (sqrt(za*za+zb*zb)<2)){
    tempza=(za*za)-(zb*zb)+cnt_c;
    zb=2*za*zb+cnt_ci;
    za=tempza;
    iter++;
    //cout << za << "     " << zb << "    " << iter << endl;
}
//cout << "putting it at:  " << volume_index_w <<  "   " << volume_index_h << "   " << slice_index <
//V.at(volume_index_w,volume_index_h,slice_index)=iter;
V.poke(volume_index_w,volume_index_h,slice_index)=iter;
volume_index_h++;
    }
    volume_index_w++;
}
slice_index++;
cnt_ci=cnt_ci+increment_ci;
    }
    cout << endl << "Creating a flat image." << endl;
    //create a flat from all the slices
    write_volume_flat(V,"fractal_flat.fit");
    cout << "Applying marching cubes algorithm." << endl;
    //marching cubes
    VISVolumeFile _iv_file;
    _iv_file.write_marchingCubes(V,march_thresh,"fractal.iv");

}



void write_volume_flat(VISVolume<float> V, char* filename){

    int slice,row,col;
    int v_width,v_height,v_depth;

    VISImage<float>           image_volume(V.width(),V.height(),V.depth());
    VISImage<float>           temp_image(V.width(),V.height());
    VISImageFile              im_file;

    v_width = V.width();
    v_height = V.height();
    v_depth = V.depth();

    for(slice=0;slice<v_depth;slice++){
for(col=0;col<v_width;col++)
    for(row=0;row<v_height;row++)
temp_image.at(col,row)=V(col,row,slice);
```

```
image_volume.putChannel(temp_image,slice);
    }
    im_file.write(image_volume.becomeFlat(), filename);
}
\end{tt}
```

References

[1] Lapack++(v1.1). http://math.nist.gov/lapack++/.

[2] M. Baccar, L. A. Gee, R. C. Gonzalez, and M. A. Abidi. Segmentation of range images via data fusion and morphological watersheds. *Pattern Recognition*, 29(10):1671–1685, 1996.

[3] D. Breen and R. Whitaker. A level-set approach to 3d shape metamorphosis. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):173–192, 2001.

[4] J. Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[5] Tony Lindeberg. Scale-space for discrete signals. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12(3):234–254, 1990.

[6] W. Lorenson and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1982.

[7] Scott Meyers. *Effective C++*. Addison Wesley Longman, 2 edition, 1998.

[8] S. Osher and J. Sethian. Fronts propogating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[9] J. A. Sethian. *Level Set Methods: Evolving Interfaces in Gometry, Fluid Mechanics, Computer Vision, and Material Sciences*. Cambridge University Press, 1996.

[10] R. Whitaker and D. Breen. Level-set models for the deformation of solid objects. In *The Third International Workshop on Implicit Surfaces*, pages 19–35. Eurographics, 1998.

[11] R. T. Whitaker. Geometry-limited diffusion in the characterization of geometric patches in images. *CVGIP: Image Understanding*, 57(1):111–120, 1993.

[12] R. T. Whitaker. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision*, October(3):203–231, 1998.

[13] R. T. Whitaker and Stephen M. Pizer. A multi-scale approach to nonuniform diffusion. *CVGIP: Image Understanding*, 57(1):99–110, 1993.