Pyvox Reference Manual
Release 0.66

Paul Hughett

March 6, 2002

Contents

List of Tables

Chapter 1

Introduction

1.1 Overview of Pyvox

BBLimage is a set of software tools for medical image processing, particularly skull stripping and segmentation of MR brain images; tools to support other applications may be added later. These tools are intended to support researchers who need to prototype new image analysis algorithms or to develop automated image analysis tools for specific image analysis applications. The sequence of processing operations is specified through a scripting language which can be used interactively or in command files; the language used is an extension of Python.

The BBLimage package contains both Pyvox, an extension to the Python language written in C and designed for efficient processing of volume images, and a collection of older command-line programs written directly in C. Pyvox has proven both efficient and easy to use and is the main current line of development; the older C language programs are vestigial and will probably be replaced by scripts using Pyvox as development proceeds.

Important design criteria for BBLimage and Pyvox include: script files and data files are portable across multiple Unix platforms, including Linux; suitable for rapid prototyping of new algorithms and analysis protocols; suitable for efficient, automated processing of the finished analysis protocols; and easily extensible by programmers outside the original development team.

BBLimage is being distributed under an Open Source license which permits free use, modification, and redistribution provided that proper credit is given.

1.2 Design Goals

BBLimage is a set of software tools being developed for medical image processing with a particular emphasis on brain masking and segmentation of magnetic resonance brain images; tools to support other applications may be added later. These tools are intended to support researchers who need to prototype new image analysis algorithms or to develop automated image analysis tools for specific image analysis applications. The particular sequence of processing operations is specified through a scripting language which can be used interactively or in command files; the language used is an extension of Python.

1.2.1 Medical Image Processing

BBLimage is designed primarily for medical image processing, because that is what the author needs to do most; other applications of volume images are no doubt possible, but their needs come second.

1.2.2 Rapid Prototyping

BBLimage should be suitable for rapid prototyping of new algorithms and analysis protocols. To do this, BBLimage is implemented as a extension to the Python language. Python is a high-level object-oriented scripting language which can be used interactively or in programmed scripts and which is designed to be easily extensible in C.

1.2.3 Efficient Execution

BBLimage should also be suitable for efficient, automated processing of the finished analysis protocols. To do this, the actual image processing functions are written in C, which is more efficient than Python.

1.2.4 Software Portability

The script files that define the analysis protocols and the programs that they invoke should be portable across multiple Unix platforms (including Linux). To meet this requirement, BBLimage is written to comply with the usual standards, including ANSI C, Posix, and the X Window System.

1.2.5 Data Portability

The image files and other data files should also be portable across multiple Unix platforms. In particular, it should be possible to create an image file on a big-endian machine (e.g. Sparc), copy it to a little-endian machine (e.g. Pentium), and further process that image without needing to do any conversion of the file. This is accomplished through a set of portable C functions that can read and write data in specified external formats, converting as necessary to or from the platform-native format.

1.2.6 Open-Source Development

BBLimage should also be easily extensible by programmers outside the original development team. This is accomplished by following good software engineering practice in documenting the software for later maintenance and extensions.

1.3 Architecture

In order to be both efficient and easy to use, BBLimage is designed using a layered approach. The top layer is Pyvox, an image processing extension to the Python programming language. Image analysis scripts are written in Python and work with functions and objects defined by Pyvox. Code at this level is inefficient but usually accounts for only a tiny fraction of the total run time.

The middle layer consists of the Voxel Kit, which is a collection of C-callable functions for high-level image processing operations such as convolution, object extraction, and statistical analysis. Many of these functions are made directly available to the user through Pyvox; others are used only internally. These functions can also be called from C programs; access from languages other than C and Python is possible, but may require writing a set of wrapper functions.

The bottom layer consists of BIPS, the basic image processing subroutines, which are a relatively small set of C functions for elementary image processing functions such as image arithmetic and are written for high efficiency; if needed, these subroutines can be hand optimized for a specific platform. For those familiar with numerical linear algebra, the relationship between the Voxel Kit and BIPS is essentially the same as between LAPACK

and BLAS. BIPS probably accounts for 95% or more of the total run time, so efficiency improvements here have a dramatic effect.

In addition, the quick diagnostic viewer (qdv) provides interactive examination of image files and is implemented using Motif and the X Window System.

1.4 Capabilities

BBLimage is designed to work directly with multi-dimensional image data (up to 8 dimensions) in signed integer, unsigned integer, and floating point formats from 1 to 8 bytes long. Currently supported operations on such arrays include:

- Reading or writing image files in signed integer, unsigned integer, and floating point formats, both big and little endian. For data portability, external files are always written in some specified external format (e.g. big-endian 2-byte two's complement integer or big-endian 4-byte IEEE 754 float), and converted to or from the native format as necessary.
- Image arithmetic, including boolean operators, comparison, transcedental functions, table lookup, and min/max.
- Image resampling to new coordinate systems.
- The morphological operations erode and dilate.
- Univariate and bivariate histograms.
- K-means and nearest neighbor classification.
- Object extraction, where an object is defined as a maximal connected set of voxels.
- Convolution and linear filtering.
- A basic set of matrix operations.
- Interactive image viewing along any coordinate axis with intensity windowing and selection of data format (which is also useful for determining the format of a unknown image file).

Additional capabilities will be added as determined by the needs of BBLimage users. Some areas that are currently under consideration include:

- Improved masking and segmentation algorithms, including handling of shading and partial volume effects.
- Interactive and automated image registration.
- Fourier transforms.
- Tools for validating brain masking and segmentation algorithms.
- Interactive modification of images, especially for manual correction of not-quite-correct automatically masked and segmented images.

1.5 Current Status

BBLimage is still under development, which means that the interface is subject to change without notice when we discover a better way of doing things. Those who want to use it for brain research will need to take care to maintain some stable version themselves and to beware of bugs. We will try to ensure that the NEWS file in the distribution kit will identify any incompatible changes between versions, but you should expect to have to periodically modify old scripts for compatibility with newer versions of BBLimage. Once we reach version 1.0, we will try to keep the interface stable. The "Open Issues" section in the Implementation chapter indicates some areas in which changes are likely.

1.6 Distribution

BBLimage is being distributed under an Open Source license which permits free use, modification, and redistribution provided that proper credit is given. There is no warranty. The file COPYING gives the precise license, which is a variant of the BSD license. People who fix bugs or make generally useful improvements are requested to send the modifications back to the author to be folded into the master copy.

Prerequisites for this software include an ANSI C compiler, make, POSIX libraries, the X Window System, and Python binaries and header files. The

software is known to compile with gcc for both Linux on Intel and Solaris on Sparc. Porting to other compilers and Unix platforms should be straightforward.

BBLimage is still preliminary, alpha software, although there is now enough functionality to support some practical applications. The source code is available from the BBL website at `http://www.med.upenn.edu/bbl` in the Publications and Downloads section.

1.7 Open Source Licensing

We are distributing this code under an open source license (which permits free modification and distribution) for several reasons. First, we believe that software is a form of scientific knowledge and that science advances most rapidly when we can build on each other's work rather than re-implementing the wheel. We hope that the people who find our software useful will reciprocate by contributing bug fixes and other improvements to be folded back into our master copy for future releases. Second, we find that we write better software when we expect that dozens of people will be reading our code than when we are writing just for ourselves. Finally, we would rather spend our time doing science rather than trying to monitor and enforce a more restrictive license.

1.8 Support

BBLimage comes with absolutely NO warranty or support. Nevertheless, it is currently being maintained by Paul Hughett

`<hughett@bbl.med.upenn.edu>` ,

who will simulate pleasure at receiving bug reports and who might actually even do something about them. Bug fixes and other improvements are really welcome.

1.9 How to Contribute

If you have found BBLimage useful and would like to contribute to its further development, there are several things that you can do, most of which will get

your name added to the Credits file:

1. Use BBLimage in a research project and let us know how it worked for you.

2. Use the programs and send us bug reports so that we can fix the bugs in future editions.

3. Fix the bugs yourself, and send us the fixes to be included in future editions.

4. Port BBLimage to another platform, and send us the changes that you had to make to get it to work. If it didn't need any changes, tell us so we can pat ourselves on the back for writing really portable software.

5. Think of some feature that BBLimage really needs, and implement it. Send us the code and documentation.

Chapter 2

Programming with Pyvox

2.1 Data Types

2.1.1 Internal Numeric Types

Pyvox supports essentially all of the numeric types provided by C. (The sole exception is plain char; both signed and unsigned char are supported, however.) The properties of these types are inheritied from the C implementation underlying Pyvox, including their length and arithmetic behavior. Table 2.1 gives the type names defined in the exim module for these types.

2.1.2 External Numeric Types

The exim module defines a set of external data types which describe the format of numeric data stored in files or other external media, and provides tools for converting these external representations to suitable internal numeric types. Each of these external types can be stored in either big- or little-endian byte order; the choice of byte order is specified by a separate flag. A byte is assumed to be exactly 8 bits. Table 2.2 shows the external types that are currently defined; it is possible to add others, but these have been found sufficient for the present.

Table 2.1: Internal numeric types

Pyvox name	C name	Description
none		Undefined or unspecified type
uchar	unsigned char	
ushort	unsigned short	
uint	unsigned int	
ulong	unsigned long	
schar	signed char	
short	(signed) short	
int	(signed) int	
long	(signed) long	
float	float	
double	double	
complex	float[2]	Real, imag pair of floats
dcomplex	double[2]	Real, imag pair of doubles

Table 2.2: External numeric types

Name	Length	Description
uint1	1	Unsigned integer
uint2	2	Unsigned integer
uint4	4	Unsigned integer
int1	1	Two's complement signed integer
int2	2	Two's complement signed integer
int4	4	Two's complement signed integer
float4	4	IEEE-754 floating point
float8	8	IEEE-754 floating point
complex8	8	Real, imag pair of float4s
complex16	16	Real, imag pair of float8s

2.1.3 Pyvox Arrays

The Pyvox array is the principal data type used within Pyvox; it is a homogeneous multi-dimensional array of one of the internal data types listed above. The `type` of the array, is the internal data type stored in the array. The number of dimensions, or `rank`, ranges from zero to eight inclusive. The dimensions themselves are presented as a tuple of numbers, each giving the size of the array along one axis. The coordinates along each axis begin at zero and range up to (but not including) the dimension along that axis. The data elements are stored sequentially, with the last index varying most rapidly. By convention, the last three axes are typically known as the slice, row, and column and denoted z, y, and x. A multi-band image is often stored with the band indexed along the last axis; for example, a three-dimensional RGB image would usually be stored with indices slice, row, column, and band in that order. The type, rank, and dimensions of a Pyvox array are all available as read-only attributes of the array.

The `origin` attribute of a Pyvox array gives the physical coordinates corresponding to the voxel indexed by $(0, 0, \ldots)$; the `space` attribute gives the physical distance between successive elements on each axis. Both the `origin` and `space` attributes may be read and set. If any `space` value is set to zero, it indicates that the physical spacing is unknown or not meaningful; for example, the spacing along the red/green/blue axis should be set to zero. If no particular physical coordinates are defined, then `origin` should be set to zero, and `space` to all ones. The total number of elements in the array may be accessed through the Python function `len()`.

The rank of a Pyvox array may be zero, in which case the array is called a scalar array. A scalar array always contains exactly one element, which is indexed by an empty subscript list. As of Python 1.5.2, however, a empty subscript list is not allowed; as a workaround, a scalar array will accept (and ignore) a single subscript. A Pyvox array of any rank that contains only a single element is known as a singleton. In most contexts where a value is required, a scalar or singleton array acts as if it were a number.

Any desired element of a Pyvox array can be accessed or assigned to using the subscript notation. For example, `a[0,1,2]` evaluates to the element at slice 0, row 1, column 2 of a three-dimensional array; assigning to the same expression will alter the value of that element. Using the subscript expression as an argument to a subroutine will pass the value of the element to the subroutine; assigning to the corresponding formal argument in the

subroutine will not affect the array. (This is consistent with the passing of list elements to subroutines.)

FIXME: Should there be some way to pass a reference to the element to a subroutine? Perhaps a way to create a singleton array slice which is distinguishable from a plain number?

The array type is conventionally used to represent certain types of data, including monochrome and RGB images, points, vectors, histograms, and matrices; these are discussed under Conventions.

2.1.4 Array Slices

An array slice is a subset of a Pyvox array obtained by selecting only certain index values along each axis. The permissible forms for each axis are a single number; a list or tuple of numbers; or a slice object in the form init:limit:stride. If the stride is omitted, it defaults to 1. If the init value is omitted, it defaults to zero. If the limit is omitted, it defaults to the length of that dimension. The colon used alone means all the indices along the given axis. In addition, the ellipsis object "`...`" may be used once in a subscript list to indicate that all index values from the unspecified axes are to be selected.

There are some special cases that the user should be aware of. If `A` and `B` are Pyvox arrays, then `A[...] = B` assigns the elements of `B` to `A`, without changing the shape of `A`, provided that the number of elements in `B` is the same as the number of elements of `A`. Using `A[...]` as an expression yields a copy of `A`, retaining the shape and contents. Logically, if `A` is a scalar, or zero rank array, then subscripting with no arguments, or `A[]`, should yield the number contained in `A`; Python (1.5.2, anyway) is not so logical, so Pyvox allows you to provide any subscript to a scalar array, and ignores it.

FIXME: The implementation of array slices is currently in flux. Evaluating an array slice, or assigning to one will behave correctly. However, passing one as a subroutine argument passes a copy of the slice rather than a reference to it, as is done for lists and other mutable objects. This can be expected to change in future releases, with array slices becoming first-class objects.

FIXME: The semantics of reversed slice objects (limit less than init, or stride less than zero) is not well defined in Python or Pyvox. Expect the behavior of such slices to change, or at least be more carefully specified, in the future.

2.1.5 Affine Transforms

An affine transform is a mapping of the form $y = Ax + b$ where A is a matrix and b is a vector; that is, it is a linear transform plus a constant offset. An affine transform is represented within Pyvox as its own class, containing the matrix A and the offset b. Points to be transformed are represented by column vectors as Pyvox arrays. The difference between two points is a vector and transforms as $v = Au$; the distinction corresponds to two different methods in the affine transform class.

Affine transforms may be composed to yield another affine transform. The order is significant; we will say that A is precomposed with B if they are applied to a point in the order A then B. Conversely, A is postcomposed with B if B is applied first, then A. A flag argument is used to indicate whether to pre- or postcompose; it defaults to postcomposition.

2.1.6 Neighborhood Kernels

A neighborhood kernel (or just kernel) is used to specify the neighboring voxels and coefficients used in a convolution; more generally, it is also used to specify the neighborhood in other neighborhood operations such as dilation and erosion. The rank of a kernel is the number of dimensions. The count of a kernel is the number of voxels in the neighborhood defined by the kernel, possibly including the center voxel. Each neighbor is associated with a coefficient, which convolution multiplies by the value at the neighbor. The position of each neighbor is given by a delta, which is a list of coordinate offsets relative to the center voxel. The bias is a number, which is added to the sum computed by convolution; it can be used to offset the convolution output to fit within a desired range.

FIXME: Check which of these attributes are read-only or read/write.

2.1.7 Objects

Pyvox provides tools for extracting the objects in a 3D image, where an object is defined as a maximal connected set of non-zero voxels (where each voxel is considered to be connected to its 26 nearest neighbors). The implementation of object extraction in the Voxel Kit is relatively complete, and each object is defined by an id, a canonical id, a point on the object, and a count of voxels contained in the object. The implementation at the Pyvox level is

incomplete, and all you can get is a mask showing the voxels contained in the largest object; the issue has been to decide exactly how to represent objects within Pyvox.

Most fast algorithms for finding objects in an image have the characteristic that they sometimes assign different id numbers to two apparently distinct objects that are later found to be parts of the same object. When this happens, one object id is taken as canonical and the `canon` field of the other is set to this canonical id; furthermore, the `count` field of the non-canonical object is added to that of the canonical object and then set to zero. The canonical object number is distinguished by the fact that its `canon` field is equal to its `ident` field. The `point` coordinates of the non-canonical object are left unchanged, although it's not clear that they are useful for anything. The `ident` (and `canon`) fields are limited to the size of unsigned short to facilitate later table lookup on object idents. If you have more than 65535 objects in an image, you're out of luck, at least with the current version of Pyvox. Ident 0 is always reserved for the background.

FIXME: find a less overloaded name for these? Perhaps blob?

FIXME: The information needed to compute the moments would also be useful, as would a set of run-length codes giving the voxels in the object.

2.2 Programming Conventions

This section describes some conventions which are not required by Pyvox, but which are normally used; in particular, these conventions are used in describing the various functions and methods provided.

2.2.1 Coordinate Systems

Following the usual convention in C (which differs from Fortran), multi-dimensional arrays are stored with the *last* coordinate varying most rapidly. A set of voxels for which the last coordinate is constant is called a row, and a set for which the last two coordinates are constant is called a slice; there isn't any established name for larger aggregates. The last three coordinates are called z, y, and x in that order; note carefully that this differs from the usual mathematical convention.

Pyvox arrays typically represent a rectangular sampling of some physical property and specify the origin and spacing of the samples in some physical

coordinate system; the origin and spacing may be set to zero and one when the physical coordinate system is unknown. An array element may be referred to using either its array indices or its physical coordinates. The first of these is defined as "index" coordinates, and the second as "physical" coordinates.

2.2.2 Uninterpreted (Raw) Data

Data which has been read from a file but not yet interpreted as useful data is usually represented by a rank-1 array of type unsigned char; that is, as raw binary data. For example, an image viewer might read an image file of unknown format as raw data and allow the user to interactively experiment with different interpretation until he finds the one that works. The functions in exim provide the means for interpreting raw data as integers, floats, or whatever.

2.2.3 Image Data

Images which contain a single echo or band of information per pixel are conventionally stored as Pyvox arrays of the appropriate rank and type. In most cases, it is appropriate to set the `origin` and `space` attributes to indicate the relationship between index coordinates and physical coordinates.

Images which contain multiple echoes or bands per pixel are conventionally stored with the last dimension running over the defined bands; note that this does require that all bands be representable in the same data format. In particular, RGB images are usually stored in unsigned char or unsigned short arrays with the last dimension equal to three and running over the red, green, and blue components.

2.2.4 Points and Vectors

The canonical representation of a geometrical point or vector in n dimensions is an $n \times 1$ column vector of type double containing its (physical) coordinates, but most functions and methods that expect points will also accept a row vector, 1-dimensional array, list, or tuple containing numbers of any type whenever its meaning is unambiguous.

There are a few instances, notably in the affine transforms, where it is necessary to distinguish between points and vectors; the distinction is that

a vector is considered to be the difference of two points and is affected only by the matrix part of an affine transform.

2.2.5 Matrices

The canonical representation of a matrix (or linear transform) is an $n \times m$ array of type double using physical coordinates, but most functions and methods that expect matrices will also accept a two-dimensional array of any numeric type, or nested tuples and lists. For example, the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

could be represented by the nested lists `[[1,2], [3,4]]`.

2.2.6 Histograms

The histogram of an usigned char or short image is conventionally represented as a 256- or 65536-element rank-1 array of type unsigned long; histograms of other array types have not yet been implemented but will probably follow a similar pattern.

Chapter 3

Theory

This chapter details the mathematical theory behind some of the more esoteric operations, primarily to establish the notation and its relationship to the softare parameters; a full explanation is left to more tutorial texts. The choice of topics in this chapter is selective, because I don't have time to explain all the background.

3.1 Cubic Spline Transform

Image warps are often defined within Pyvox using a coordinate transform based on cubic Hermite splines. To fix notation, we suppose that such a transform maps from an M-dimensional space to an N-dimensional space. The spline is defined over a rectangular grid containing $R_0 \times R_1 \times \cdots \times R_{M-1}$ nodes in total. At each node $r = (r_0, r_1, \ldots, r_{M-1})$ of this grid is given an offset b_r and matrix A_r which define the local affine transform at that node.

Chapter 4

Pyvox Reference

This chapter describes in detail the functions, methods, and attributes provided by Pyvox. The first section is a listing by category, which is useful for finding the function that provided some desired capability; the second section is in alphabetical order and provides a complete description.

Much of the functionality of Pyvox is provided by methods belonging to objects of some given type or class, and invoked as an attribute of the object. For instance, if *affine* is some expression which evaluates to an instance of an affine transform, then *affine*`.inverse()` computes and returns the inverse of that transform. Such methods are listed here under the name of the class or type; in this example, *affine*`.inverse()`. Similarly, the italicized word *array* indicates an expression which evaluates to a Pyvox array; *kernel* indicates a Pyvox kernel.

Optional arguments are followed by an equal sign and the default value; the value *???* means the default value is given in the description below.

FIXME: Keyword arguments should not be used until I have the chance to ensure that the argument names used here are actually the same as those used in the code.

4.1 Listing by Category

4.1.1 Pyvox Modules

Importable Pyvox Modules	
`exim`	Data export and import
`optim`	Optimization functions and classes
`pyvox`	Pyvox core types and functions
`regis`	Image registration classes and functions

4.1.2 Pyvox Types and Classes

Pyvox Types and Classes	
`pyvox.AffineType`	Class object for an affine transform
`pyvox.ArrayType`	Type of a Pyvox array
`pyvox.KernelType`	Type of a Pyvox kernel

4.1.3 Data Export and Import

Data Export and Import	
array`.print()`	Print contents of a Pyvox array
array`.write()`	Write contents of a Pyvox array
array`.writeppm()`	Write Pyvox array as a PPM image file
`pyvox.rawread()`	Read a Pyvox array from stream or file

4.1.4 Image Display

4.1.5 Array Creation

Array Creation	
`pyvox.array()`	Create an array from given data
`pyvox.column()`	Create column vector from given data
`pyvox.const()`	Create a constant array (deprecated)
`pyvox.diag()`	Create a diagonal matrix from given data
`pyvox.matrix()`	Create a rank-2 matrix from given data
`pyvox.point()`	Create coordinate point from given data
`pyvox.ramp()`	Create array with coordinate indices
`pyvox.vector()`	Create rank-1 vector from given data

4.1.6 Basic Array Manipulations

Basic Array Manipulations	
array`[ ]`	Element or slice
array`[ ] =` *expr*	Assign to element or slice
array`.copy()`	Copy contents and attributes
array`.count()`	Number of elements in an array
array`.i2p()`	Index-to-physical transform
array`.list()`	Elements as a Python list
array`.origin`	Physical coordinates of origin
array`.p2i()`	Physical-to-index transform
array`.spacing`	Physical spacing of coordinate planes
array`.rank`	Rank, or number of dimensions
array`.refcnt`	Reference count
array`.reshape()`	Change the shape of an array
array`.size`	Dimensions of an array
array`.tuple()`	Elements as a Python tuple
array`.type`	Type of data in an array
`len(`*array*`)`	Number of elements (Python bug!)

4.1.7 Type Conversions

Type Conversions	
array`.double()`	Convert to double
array`.float()`	Convert to float
array`.int()`	Convert to int
array`.long()`	Convert to long
array`.short()`	Convert to short
array`.schar()`	Convert to signed char
array`.uchar()`	Convert to unsigned char
array`.uint()`	Convert to unsigned int
array`.ulong()`	Convert to unsigned long
array`.ushort()`	Convert to unsigned short

4.1.8 Arithmetic and Boolean Operations

Elementwise Arithmetic and Boolean Operations	
`-`*array*	Additive inverse
`+`*array*	(Copy?)
array `+` *array*	Addition
array `-` *array*	Subtraction
array `*` *array*	Multiplication
array `/` *array*	Division
`~`*array*	Bitwise negation
array `&` *array*	Bitwise AND
array `|` *array*	Bitwise OR
array `^` *array*	Bitwise XOR
array`.abs()`	Absolute value
`abs(`*array*`)`	Absolute value (deprecated)
array`.ceil()`	Ceiling
array`.floor()`	Floor
array`.max()`	Maximum of two arrays
array`.min()`	Minimum of two arrays
array`.compare()`	Comparison of two arrays

4.1.9 Elementary Functions

Elementary Functions	
array.`acos()`	Arc cosine
array.`asin()`	Arc sine
array.`atan()`	Arc tangent
array.`atan2()`	Two-argument arc tangent
array.`cos()`	Cosine
array.`cosh()`	Hyperbolic cosine
array.`exp()`	Exponential
array.`log()`	Natural logarithm
array.`log10()`	Logarithm to the base 10
array.`pow()`	Power
array.`sin()`	Sine
array.`sinh()`	Hyperbolic sine
array.`sqrt()`	Square root
array.`tan()`	Tangent
array.`tanh()`	Hyperbolic tangent

4.1.10 Other Elementwise Operations

Other Elementwise Operations	
array.`logcomp()`	Log intensity compression
array.`lookup()`	Lookup table
array.`scale()`	Scale by constant gain and bias

4.1.11 Array Reduction Operations

Reduction Operations	
array.`amax()`	Maximum value within array
array.`amin()`	Minimum value within array
array.`aprod()`	Product of array elements
array.`asum()`	Sum of array elements
array.`mean()`	Mean of elements along specified axes

4.1.12 Array and Image Metrics

Array and Image Metrics	
array.`dot()`	Vector dot product of two arrays
array.`norm1()`	Vector 1-norm of an array
array.`norm2()`	Vector 2-norm of an array
array.`normsup()`	Vector sup-norm of an array
`regis.info()`	Information content of an image
`regis.mutinfo()`	Mutual information of two images

4.1.13 Matrix Operations

Matrix Operations	
array.`det()`	Determinant of a square matrix
array.`diag()`	Extract diagonal elements of a matrix
array.`eigsy()`	Solution of real symmetric eigensystem
array.`inverse()`	Matrix inverse
array.`mmul()`	Matrix multiplication
array.`prinaxes()`	Principal axes transformation
array.`solve()`	Solve the linear system $AX = B$
array.`trans()`	Transpose of a matrix

4.1.14 Neighborhood Operations

Neighborhood Operations	
array.`convolve()`	Convolution with optional subsampling
array.`dilate()`	Dilation
array.`erode()`	Erosion
array.`lostat()`	Local mean and variance
array.`lowpass()`	Lowpass filter with optional subsampling
kernel.`bias`	Bias term of a kernel
kernel.`coef`	Coefficients of a kernel
kernel.`count`	Number of neighbors in a kernel
kernel.`delta`	Neighbor offsets for a kernel
kernel.`rank`	Rank of a kernel
`pyvox.kernel()`	Create a kernel

4.1.15 Statistical Operations

Statistical Operations	
array.`bihist()`	Bivariate histogram
array.`histo()`	Univariate histogram
array.`kmeans1()`	Train K-means classifier
array.`lostat()`	Local mean and variance
array.`mean()`	Mean of elements along specified axes
array.`moments()`	Center of gravity and principal moments
array.`mop()`	Arbitrary moments of a product of images
array.`stat()`	Mean and standard deviation of elements
`pyvox.monomials()`	Monomials of degree $\leq n$ in k literals
`regis.info()`	Information content of an image
`regis.mutinfo()`	Mutual information of two images

4.1.16 Voxel Classification

Voxel Classification	
array.`kmeans1()`	Compute K-means classifier
array.`nnclass1()`	Univariate nearest neighbor classifier
array.`nnclass2()`	Bivariate nearest neighbor classifier

4.1.17 Other Image Operations

Other Image Operations	
array.`bigob()`	Extract largest object in image
array.`chamfer()`	Compute chamfer distance transform
array.`zerbv()`	Zero boundary voxels

4.1.18 Affine Transforms

Affine Transforms	
affine.`addparam()`	Add vector of parameters to the transform
affine.`compose()`	Compose with another affine transform
affine.`compose2()`	Compose with given matrix and offset
affine.`i2p()`	Compose with index-to-physical transform
affine.`inverse()`	Inverse
affine.`invert()`	Invert in place
affine.`linear()`	Resample image with linear interpolation
affine.`matrix`	Matrix part of affine transform
affine.`nearest()`	Resample with nearest neighbor interpolation
affine.`norm()`	Norm of an affine transform
affine.`offset`	Constant part of affine transform
affine.`p2i()`	Compose with physical-to-index transform
affine.`param()`	Transform as a vector of parameters
affine.`point()`	Transform a point
affine.`rotate()`	Compose with a rotation
affine.`rotate3d()`	Compose with a rotation around axis in 3-D
affine.`scale()`	Compose with scale transform
affine.`setparam()`	Set transform from a vector of parameters
affine.`shear()`	Compose with elementary shear transform
affine.`translate()`	Compose with translation
affine.`vector()`	Transform a vector
`pyvox.affine()`	Create an affine transformation

4.1.19 Interpolation and Resampling

Interpolation and Resampling	
affine.`linear()`	Resample image with linear interpolation
affine.`nearest()`	Resample with nearest neighbor interpolation
array.`linear()`	Linear interpolation within an image
array.`nearest()`	Nearest neighbor interpolation in image

4.1.20 Image Registration

Image Registration	
`regis.obaffine`	Obfunction for affine registration
`regis.obregis`	Generic obfunction for image registration
`regis.obrigid`	Obfunction for rigid registration

4.1.21 Optimization

Optimization	
`optim.obfunction`	Base class for an objective function
`optim.powell`	Powell direction set method

4.1.22 Graphics and Drawing

Graphics and Drawing	
array`.fill2d()`	Fill a 2D contour with a value

4.2 Full Descriptions

---

affine.`addparam`(*parms*)

Updates an affine transform by adding the contents of the vector *parms* to the elements of the $n \times (n+1)$ matrix defining the affine transform, taking the matrix elements in row-major order. This function is useful for optimization algorithms that generate perturbations to an initial transform.

---

affine.`compose`(*other, pre=0*)

Updates *affine* to contain the composition of the affine transformations *affine* and *other*; returns the updated transform. If *pre* is 1, then precomposes *other* with *affine*; if *pre* is 2 (or 0), then postcomposes *other* with *affine*; if *pre* is -1, then precomposes the inverse of *other* with *affine*; if *pre* is -2, then postcomposes the inverse of *other* with *affine*; any other value of *pre* is an error.

---

affine.`compose2`(*matrix, offset, pre=0*)

Updates the affine transform *affine* by composing it with another affine transform given by *matrix* and *offset*; returns the updated transform. The value of *pre* determines the order in which the two transforms are composed; see *affine*.`compose()` for the permitted values.

---

affine.`i2p`(*origin, spacing, pre=0*)

Updates *affine* by composing it with the index-to-physical transformation given by *origin* and *spacing*; returns the updated transform. The value of *pre* determines the order in which the two transforms are composed; see *affine*.`compose()` for the permitted values.

---

affine.`inverse()`

Returns a new affine transform which is the inverse of the affine transform *affine*.

---

affine.`invert()`

Updates *affine* to contain the inverse of the original transform; returns the updated value.

---

affine.`linear(`*image, dimen)*

Resamples *image* through the affine transform given by *affine* to yield a new Pyvox array with dimensions *dimen* and the same type as *image*. Linear interpolation is used. The affine transform must be given as the destination-to-source transform (not the source-to-destination transform that you might naively expect) and must be given in index coordinates for both the source and destination.

(There are actually two implementations of this method. The `linear0` method is a reference implementation which is not particularly fast but which is simple enough to be verified by inspection; the `linear` method is a faster implementation which has been verified against the reference algorithm.)

---

affine.`matrix`

The matrix part of the affine transform, as a Pyvox array. This attribute may be freely read, but setting it is not recommended.

---

affine.`nearest(`*image, dimen)*

Resamples *image* through the affine transform given by *affine* to yield a new Pyvox array with dimensions *dimen* and the same type as *image*. Nearest neighbor interpolation is used. The affine transform must be given as the destination-to-source transform (not the source-to-destination transform that

you might naively expect) and must be given in index coordinates for both the source and destination.

---

affine.`norm`(*other=None, length=100*)

Computes a simple norm on the vector space of affine transforms; the computed norm is also the maximum sup-norm of the image of any point with sup-norm not exceeding the given *length*. If *other* is given, the norm is computed on the (vector) difference of *self* and *other*; as a special case, setting *other* to 1 compares *self* to the identity transform. For the mathmatically inclined, the norm is defined as

$$\|T\| = \max_r \left(|b_r| + \lambda \sum_c |A_{rc}| \right) \tag{4.1}$$

where T is an affine transform, λ is the specified *length*, A and b are the matrix (or linear) and offset parts of T, and the indices r and c run over the rows and columns of A. (The sup-norm rather than the Euclidean norm on the points is used because the resulting norm on affine transforms is slightly faster to compute.) Note that this norm does not behave properly for composition and so is NOT an operator norm.

---

affine.`offset`

The constant offset part of the affine transform, as a Pyvox column vector. This attribute may be freely read, but setting it is not recommended.

---

affine.`p2i`(*origin, spacing, pre=0*)

Updates *affine* by composing it with the physical-to-index transformation given by *origin* and *spacing*; returns the updated transform. The value of *pre* determines the order in which the two transforms are composed; see *affine*.`compose()` for the permitted values.

---

affine.`param()`

Returns a vector containing the elements of the $n \times (n+1)$ matrix defining the transform, taken in row-major order. This is useful for optimization methods which work on vectors.

---

affine.`point(`x`)`

Transforms a point according to the given affine transform, returning a new Pyvox array.

---

affine.`rotate(`*i, j, angle, pre=0*`)`

Updates *affine* by composing it with an elementary rotation of the form $x[i] = x[i]\cos\theta + x[j]\sin\theta$ and $x[j] = -x[i]\sin\theta + x[j]\cos\theta$; returns the updated transform. The value of *pre* determines the order in which the two transforms are composed; see *affine*.`compose()` for the permitted values.

---

affine.`rotate3d(`*axis, angle, pre=0*`)`

Updates *affine* by composing with a rotation around *axis* by the given *angle* in radians. The axis is presumed to go through the origin, and a positive angle is defined by the righthand rule. The value of *pre* determines the order in which the two transforms are composed; see *affine*.`compose()` for the permitted values.

---

affine.`scale(`*coef, pre=0*`)`

Updates *affine* by composing it with an elementary scale of the form $x[i] = C[i]x[i]$; returns the updated transform. The value of *pre* determines the order in which the two transforms are composed; see *affine*.`compose()` for the permitted values.

---

affine.`setparam(`*parms*`)`

Sets the elements of the $n \times (n+1)$ matrix defining the transform, taken in row-major order, from the contents of the vector *parms*. This is useful for optimization methods which work on vectors.

---

affine.`shear(`*i, j, coef, pre=0*`)`

Updates *affine* by composing it with an elementary shear of the form $x[i] = x[i] + Cx[j]$; returns the updated transform. The value of *pre* determines the order in which the two transforms are composed; see *affine*.`compose()` for the permitted values.

---

affine.`translate(`*delta, pre=0*`)`

Updates *affine* by composing it with a translation of the form $x[i] = x[i] + \Delta[i]$; returns the updated transform. The value of *pre* determines the order in which the two transforms are composed; see *affine*.`compose()` for the permitted values.

---

affine.`vector(`*v*`)`

Transforms a vector (difference of points) according to the affine transform *affine*, returning a new Pyvox array.

`abs(`*array)*

Deprecated; use *array*.`abs()` in new code.

---

array.`abs()`

Returns a new Pyvox array containing the elementwise absolute value of the original array. NOTE: Since Pyvox inherits its arithmetic from the

underlying C implementation, you can get unexpected results taking the absolute value of signed integral types on a two's complement machine; the absolute value of -2^{n-1}, where n is the width of the type in bits, is the same value -2^{n-1}.

---

array.`acos()`

Returns a new Pyvox array containing the elementwise arc cosine of the original array. The returned value is in radians. Valid for types float and double only.

---

array.`amax()`

Returns the maximum value found in the array.

---

array.`amin()`

Returns the minimum value found in the array.

---

array.`aprod()`

Returns the product of the array elements.

---

array.`asin()`

Returns a new Pyvox array containing the elementwise arc sine of the original array. The returned value is in radians. Valid for types float and double only.

---

array.`asum()`

Returns the sum of the array elements. The *array*`.dot()` method may be used to obtain a weighted sum of the elements.

---

array`.atan()`

Returns a new Pyvox array containing the elementwise arc tangent of the original array. The returned value is in radians. Valid for types float and double only.

---

array`.atan2(`*other*`)`

Returns a new Pyvox array containing the elementwise two-argument arc tangent of the original and other arrays. The returned value is in radians. Valid for types float and double only.

---

array`.bigob(`*label=255, other=0*`)`

Returns a new unsigned char Pyvox array containing a mask covering only the largest object in the original image; an object is defined as a maximal connected set of non-zero voxels.

---

array`.bihist(`*other, weight=1*`)`

Returns a new two-dimensional unsigned long Pyvox array containing the bivariate histogram of the two unsigned char arrays *array* and *other*; it also computes the two marginal histograms (which are the univariate histograms of the individual images) as one-dimensional unsigned long Pyvox arrays. An optional unsigned char array of weights may be provided; it must be the same shape as the other two arrays and defaults to weight 1 for each voxel. These results are returned as a tuple containing the bivariate histogram and the two univariate histograms.

---

array.`ceil()`

Returns a new Pyvox array containing the elementwise ceiling of the original array. Valid for types float and double only.

---

array.`chamfer(`*type=???*`)`

Returns a new Pyvox array containing the chamfer distance transform of the original array. The chamfer distance is defined to be zero wherever the original array is non-zero, and the taxicab (L1) distance to the nearest non-zero voxel otherwise. The type of the result may be specified by the caller to be either `exim.uchar` or `exim.ushort`, and will default to the shortest type which is capable of containing the longest possible distance within the original image.

---

array.`compare(`*other, less, equal, more*`)`

Returns a new unsigned char Pyvox array containing the results of comparing *array* element by element to *other*. The result takes one of the values *less*, *equal*, or *more* depending on whether each element of *array* is less than, equal to, or greater than the corresponding element of *other*. The array *other* may be replaced by a number or scalar, which is then used in all the comparisons.

---

array.`convolve(`*kernel, shrink=1*`)`

Convolves *array* with *kernel*, optionally subsamples by the factor *shrink*, and returns the result as a new Pyvox array; the algorithm used avoids computing pixel values that will be omitted by subsampling and is faster than convolution followed by subsampling. The object *array* may be of any rank, type, or shape, except that convolution is not defined for rank zero arrays; the array and kernel must have the same rank. The *shrink* argument may be either a single positive integer, or a list of integers giving the desired shrink factor for each dimension of the array; if the shrink is omitted, no subsampling is done. The convolution is calculated in double precision and

converted back to the original image type; if the original type cannot represent the values, the nearest representable value is used instead.

---

array.`copy()`

Returns a new Pyvox array containing contents and attributes of the original array but which shares no storage with it.

---

array.`cos()`

Returns a new Pyvox array containing the elementwise cosine of the original array. The input is in radians. Valid for types float and double only.

---

array.`cosh()`

Returns a new Pyvox array containing the elementwise hyperbolic cosine of the original array. Valid for types float and double only.

---

array.`count()`

Returns the number of elements in *array*.

---

array.`det()`

Returns the determinant of a square matrix represented as a Python array. Valid for types float and double only.

---

array.`diag()`

Returns a new rank-1 Pyvox array whose elements are the diagonal elements of *array*; the type of the result is the same as the type of *array*. Use the function `pyvox.diag(`*values*`)` to construct a diagonal matrix.

---

array`.dilate(`*kernel=???*`)`

Returns a new Pyvox array containing the morphological dilation of *array*, which must have non-zero rank and any unsigned integral type; the dilation is done bitwise on the voxel values. The neighborbood is specified by a kernel object and defaults to a centered $3 \times 3 \times 3$ neighborhood. The bias and coefficients of *kernel*, if any, are ignored.

---

array`.dot(`*other=None, weight=None*`)`

Returns the vector dot product of *array* and *other*, with optional pixelwise weights provided by *weight*. The source and weight arrays may be of any type and are converted to double for this computation; they must, however, be the same shape. Note that *array*`.dot(`*w*`)` can also be interpreted as the sum of the elements of *array* with weights *w*; setting *w* to `None` will compute the unweighted sum of the elements of *array*.

---

array`.double()`

Returns a new Pyvox array containing the contents of the original array converted to double.

---

array`.eigsy()`

Returns a pair `(val, vec)` containing the eigenvalues and eigenvectors of the original Pyvox array considered as a real symmetric matrix; the results are undefined if the Pyvox array is not actually symmetric. the result `val` is a list of the eigenvalues. The result `vec` is an orthogonal matrix, the rows of which are the eigenvectors; `vec` may or may not be a proper orthogonal matrix (with determinant equal to +1). (If a proper matrix of eigenvalues is required, use *array*`.prinaxes()` instead.) If `A` denotes the original matrix, then `A = vec'*diag(val)*vec`.

---

array`.erode(`*kernel=???*`)`

Returns a new Pyvox array containing the morphological erosion of *array*, which must have non-zero rank and any unsigned integral type; the dilation is done bitwise on the voxel values. The neighborbood is specified by a kernel object and defaults to a centered $3 \times 3 \times 3$ neighborhood. The bias and coefficients of *kernel*, if any, are ignored.

---

array`.exp()`

Returns a new Pyvox array containing the elementwise exponential (to the base e) of *array*. Valid for types float and double only.

---

array`.fill2d(`*points, value)*

Modifies a two-dimensional Pyvox array by setting the voxels inside the contour defined by *points* to the given *value* and returns the modified array. The *points* are given by an $N \times 2$ Pyvox array of (y, x) coordinate pairs. (Other formats may be added later.) If the first and last points are not identical, then the contour is closed by assuming a line segment from the last to the first. A voxel is considered to be inside the contour if a ray to infinity from the center of the voxel (given by integer coordinates) crosses the contour an odd number of times; this is known as the even-odd rule. The ambiguity present when the voxel center falls exactly on the contour are handled by pretending the the voxel center is actually displayed by a positive epsilon along each axis from its nominal position.

---

array`.float()`

Returns a new Pyvox array containing the contents of the original array converted to float.

---

array`.floor()`

Returns a new Pyvox array containing the elementwise floor of the original array. Valid for types float and double only.

---

array.`histo(`*weight=1)*

Returns a new Pyvox array containing the univariate histogram of *array*, which must be an unsigned char image. An optional unsigned char array of weights may be provided; the weight array must be the same shape as *array* and defaults to unit weights for each voxel.

---

array.`i2p()`

Returns the affine transform which maps index coordinates into physical coordinates, as defined by the origin and spacing attributes of *array*.

---

array.`int()`

Returns a new Pyvox array containing the contents of the original array converted to int.

---

array.`inverse()`

Returns a new Pyvox array containing the inverse of the non-singular square matrix represented by the original Pyvox array. Note that it is generally better to use *array*.`solve` for the solution of a system of linear equations.

---

array.`kmeans1(`*cent*`)`

Returns the class centroids computed by the K-means classification algorithm, where *array* is the histogram of an unsigned char image (and must have exactly 256 bins) and *cent* is a list of initial guess at the class centroids. The centroids thus computed can be used later by the *array*.`nnclass1` method to do the actual segmentation. If initial guesses are not available for the

centroids, they should all be set to zero. The number of classes found is set by the number of centroids provided.

---

array.`linear`(*point*)

Returns the pixel value at a given position possibly between samples, using linear interpolation. The argument *point* may be given as a tuple, list, or rank-1 array of coordinate values. Samples outside the image are assumed to be zero.

(There are actually two implementations of this method. The `linear` method invokes the fastest algorithm that is currently considered trustworthy; `linear0` invokes a reference implementation which is simple enough to be verified by inspection and which is used to validate faster but more complex algorithms.)

---

array.`list()`

Returns the elements of *array* as a Python list, in row-major order; the elements of the list will be Python floats, ints, or longs as appropriate.

---

array.`log()`

Returns a new Pyvox array containing the elementwise natural logarithm of the original array. Valid for types float and double only.

---

array.`log10()`

Returns a new Pyvox array containing the elementwise base-10 logarithm of the original array. Valid for types float and double only.

---

array.`logcomp()`

Returns a new unsigned char Pyvox array containing a intensity-compressed version of the orginal image, which must be of type unsigned long. The compression rule is the transformation $y = A \log(1 + x)$, where A is chosen such that the largest voxel value actually present in the image is converted to the value 255.

---

array`.long()`

Returns a new Pyvox array containing the contents of the original array converted to long.

---

array`.lookup(`*lut*`)`

This Pyvox function takes each voxel in *array* and uses it as the index into the lookup table *lut*, saving the result of the lookup as the destination image. The source image must be either unsigned char or unsigned short but may be of any shape. The lookup table must be rank 1 and contain at least as many entries as the largest voxel value in the source image; but it (and thus the destination) may be of any type.

---

array`.lomean()`

Obsolete?

---

array`.lostat()`

This Pyvox function computes the local mean and standard deviation within each $3 \times 3 \times 3$ neighborhood of a volume image and returns them as a list containing two new images of the same shape and type; the first is the local mean and the second is the local standard deviation. Boundary voxels are handled correctly. The standard deviation is multiplied by 2.0 for images of type unsigned char, to better match the possible range of values to the range supported by unsigned char; the scale is left at 1.0 for all other data types.

---

array`.lovar()`

Obsolete. Use *array*`.lostat` instead.

---

array`.lowpass(`*shrink=1*`)`

Lowpass filters *array*, optionally subsamples by the factor *shrink*, and returns the result as a new Pyvox array; the algorithm used avoids computing pixel values that will be omitted by subsampling and is faster than convolution followed by subsampling. The object *array* may be of any rank, type, or shape, except that lowpass filtering is not defined for rank zero arrays. The *shrink* argument may be either a single positive integer, or a list of integers giving the desired shrink factor for each dimension of the array; if the shrink is omitted, no subsampling is done. The convolution is calculated in double precision and converted back to the original image type; if the original type cannot represent the values, the nearest representable value is used instead.

In the current implementation, the *array* must be rank 3. The kernel used is a $3 \times 3 \times 3$ convolution kernel and has the form $2^{-3+|x|+|y|+|z|}$; this kernel will completely surpress the Nyqist frequency along any of the three coordinate axes. (Future versions of this function may operate in any number of dimensions and may allow the bandwidth of the lowpass filter to be specified.)

---

array`.max(`*other*`)`

Returns a new Pyvox array containing the elementwise maximum of the original and other arrays.

---

array`.mean(`*weight=None, axes=All*`)`

This function computes the *weight*ed mean of the *array* elements along the specified *axes* (defaulting to all axes) and returns either a scalar or an array of reduced rank. The *weight* is normally an array but may be `None` or

a plain number to specify an unweighted mean. The *axes* argument may be an integer in the range $[0, n-1]$, where n is the rank of the array, meaning a single axis counting from the left; an integer in the range $[-1, -n]$, meaning a single axis counting from the right; or a tuple or list of such integers, meaning all the axes listed. The default is to reduce on all axes to yield a scalar, and can be obtained either by omitting the *axes* argument or setting it to `None`. On the other hand, setting *axes* to a list or tuple with no elements is just a expensive way to cast the array to type double. The values listed in *axes* may be in any order, and duplicates are permitted. The *array* and *weight* arguments may be of any type but must be the same shape; the returned value is always type double. If you're foolish enough to let the weights sum to zero for some output element, you get whatever value the underlying C implementation provides for division by zero; this should normally be nan for IEEE-754 platforms.

---

array`.min(`*other*`)`

Returns a new Pyvox array containing the elementwise minimum of the original and other arrays.

---

array`.mmul(`*other*`)`

Returns a new Pyvox array containing the matrix product of *array* and *other*, both of which must be rank-2 arrays of type float or double and compatible for multiplication. As a special case, if *other* is a rank-1 array of the appropriate length, it will be treated as a column vector.

FIXME: The current version requires that both arrays are of the same type.

---

array`.moments()`

Returns a tuple containing the total mass, center of gravity, and second central moments of a volume image; the values returned are the total mass (as a Python float), the center of gravity (as a Pyvox array), and the moments

(as a Pyvox array). The center of gravity and moments are in physical units, as defined by the origin and spacing of the volume image.

---

array.`mop`(*moments, array2=None, array3=None*)

This function computes and returns arbitrary non-central moments in index coordinates for the elementwise product of up to three arrays. The requested moments are defined by *moments*, which must be a Pyvox array of type int with any number of rows and a number of columns equal to the dimension of *array*; each row of *moments* specifies one moment to be computed, while each column specifies the power to which the corresponding coordinate is to be raised. (The `pyvox.monomials` function may be useful in constructing the *moments* argument.) The requested moments are returned in a one-dimensional Pyvox array of type double and are in the same order as the rows of *moments*. The three arrays must be the same dimension and shape, but may be of any type or types; the product is always done in double. (It often takes less time and memory to let this function compute the products on the fly rather than computing them outside; this is especially true for arrays of integral type.)

---

array.`nearest`(*point*)

Returns the pixel value at a given position possibly between samples, using nearest neighbor interpolation. The point may be given as a tuple, list, or rank-1 array of coordinate values. Samples outside the image are assumed to be zero.

---

array.`nnclass1`(*clids, cents*)

Returns a new unsigned char Pyvox array containing the classification of each voxel of the original image using a univariate nearest neighbor classifier; the original image must be of type unsigned char. The arguments are the class ids and the class centroids. Different centroids may be assigned to the same class number. There must be exactly as many class ids as class centroids.

FIXME: The current calling sequence was adopted because it was fairly easy to implement quickly, but it's not clear that it's the best option; so it might change in the future.

---

array`.nnclass2(`*other, clids, cents1, cents2*`)`

Returns a new unsigned char Pyvox array containing the classification of each voxel using a bivariate nearest neighbor classifier on corresponding voxels of the original and other arrays, both of which must be of type unsigned char. The remainings arguments are the class ids and the class centroids for each array. Different centroids may be assigned to the same class number. There must be exactly as many class ids as class centroids.

FIXME: The current calling sequence was adopted because it was fairly easy to implement quickly, but it's not clear that it's the best option; so it might change in the future.

---

array`.norm1(`*other=0, weight=1*`)`

Computes the vector 1-norm of *array*, or the difference between *array* and *other*, with optional pixelwise weights given by *weight*. The argument *other* must either be the same shape and type as *array*, or must be the scalar 0, or must be omitted. The argument *weight*, if provided, must be same shape as *array* but may be of any type; however, types other than float and double should be avoided pending a decision as to whether integral types will be interpreted as integer or scaled fraction values. Note that it is better to use `foo.norm1(bar)` rather than `(foo-bar).norm1()` because it avoids potential overflow and wraparound problems in the subtraction. Note also that the weighted norm of a single array may be computed as *array*`.norm1(0,` *weight*`)`.

---

array`.norm2(`*other=0, weight=1*`)`

Computes the vector 2-norm of *array*, or the difference between *array* and *other*, with optional pixelwise weights given by *weight*. The argument *other* must either be the same shape and type as *array*, or must be the scalar

0, or must be omitted. The argument *weight*, if provided, must be same shape as *array* but may be of any type; however, types other than float and double should be avoided pending a decision as to whether integral types will be interpreted as integer or scaled fraction values. Note that it is better to use `foo.norm2(bar)` rather than `(foo-bar).norm2()` because it avoids potential overflow and wraparound problems in the subtraction. Note also that the weighted norm of a single array may be computed as *array*`.norm2(0,` *weight*`)`.

---

array`.normsup(`*other=0, weight=1*`)`

Computes the vector sup-norm of *array*, or the difference between *array* and *other*, with optional pixelwise weights given by *weight*. The argument *other* must either be the same shape and type as *array*, or must be the scalar 0, or must be omitted. The argument *weight*, if provided, must be same shape as *array* but may be of any type; however, types other than float and double should be avoided pending a decision as to whether integral types will be interpreted as integer or scaled fraction values. Note that it is better to use `foo.normsup(bar)` rather than `(foo-bar).normsup()` because it avoids potential overflow and wraparound problems in the subtraction. Note also that the weighted norm of a single array may be computed as *array*`.normsup(0,` *weight*`)`.

---

array`.origin`

This attribute of a Pyvox array is a list which specifies the physical coordinates corresponding to zero index coordinates and is represented as a list of numbers. Assigning to this attribute changes the origin. As a special case, a plain number may be assigned as the origin of a rank-1 array; it will, however, always be returned as a list.

---

array`.p2i()`

Returns the affine transform which maps physical coordinates into index coordinates, as defined by the origin and spacing attributes of *array*.

---

array.`pow(`*other*`)`

Returns a new Pyvox array containing the elementwise power function of *array* and *other*; that is, `pow(x, y)` or x^y for each element. Valid for types float and double only.

---

array.`prinaxes()`

Returns a pair `(val, vec)`, where `val` is a vector containing the principal moments and `val` is a proper orthogonal matrix containing the principal axes of the original array considered as a real symmetric matrix. The results are undefined if the matrix is not actually symmetric. This method is equivalent to *array*.`eigsy`, except that the matrix of eigenvalues is guaranteed to have determinant +1.

---

array.`print(`*stream*`)`

(Not yet implemented) Writes a human-readable representation of the contents to the indicated stream.

---

array.`rank`

Reports the rank of the Pyvox array. This attribute may not be changed directly, although the reshape method will modify it.

---

array.`refcnt`

Reports the reference count of a Pyvox array. (Mainly useful for debugging Pyvox itself.)

---

array.`reshape(`*newshape*`)`

Changes the shape of a Pyvox array without copying or modifying any of its elements. It is not permitted to change the total number of elements by this method; that is, the product of the new dimensions must equal the product of the old dimensions. Similarly, it is not permitted to change the type of the data.

FIXME: Once array views are implemented, this will probably change to return a new view of the array, leaving the old array intact but sharing it.

---

array.`scale`(*gain=1.0, bias=0.0*)

Returns a new Pyvox array of the same type as the original, with each element rescaled by multiplying by the gain and adding the bias; the rescaling is done in double and then rounded and limited to the destination type. This operation can be done with other functions, but this function makes better use of cache and is faster for arrays of integral type.

---

array.`schar()`

Returns a new Pyvox array containing the contents of the original array converted to signed char.

---

array.`short()`

Returns a new Pyvox array containing the contents of the original array converted to short.

---

array.`sin()`

Returns a new Pyvox array containing the elementwise sine of the original array. The input is in radians. Valid for types float and double only.

---

array.`sinh()`

Returns a new Pyvox array containing the elementwise hyperbolic sine of the original array. Valid for types float and double only.

---

array`.size`

This attribute of a Pyvox array is a tuple containing its dimensions. The rank is obviously the length of this list. This attribute may not be changed directly, although the *array*`.reshape()` method will modify it. FIXME: This should probably be renamed as *array*`.shape()` for consistency.

---

array`.solve(`*rhs*`)`

Returns the solution X of the linear system $AX = B$ for a square matrix A given by *array* and a general matrix B given by *rhs*. Both *array* and *rhs* must be the same type, either float or double, and must be compatible in shape.

---

array`.spacing`

This attribute of a Pyvox array is a line which specifies the spacing between coordinate planes in each axis and is represented as a list of numbers. Assigning to this attribute changes the spacing. As a special case, a plain number may be assigned as the spacing of a rank-1 array; it will, however, always be returned as a list.

---

array`.sqrt()`

Returns a new Pyvox array containing the elementwise square root of the original array. Valid for types float and double only.

---

array`.stat(`*weight=None*`)`

Returns a tuple containing the estimated mean $\bar{x}$ and standard deviation s of the elements of *array*, optionally weighted by the contents of *weight*. The two arrays may be of any type or shape, but must be the same shape. The results are defined by

$$p_i = \frac{w_i}{\sum_i w_i} \tag{4.2}$$

$$\bar{x} = \sum_i p_i x_i \tag{4.3}$$

$$s^2 = \frac{\sum_i p_i (x_i - \bar{x})^2}{1 - \sum_i p_i^2} \tag{4.4}$$

where i runs over the elements of *array*, x_i is the ith element of *array*, w_i is the ith element of *weight*, and p_i is the weight w_i converted to a probability. In the case that *weight* is omitted or set to `None`, these simplify to

$$\bar{x} = \frac{1}{N} \sum_i x_i \tag{4.5}$$

$$s^2 = \frac{1}{N-1} \sum_i (x_i - \bar{x})^2 \tag{4.6}$$

where N is the number of elements in *array*.

---

array`.tan()`

Returns a new Pyvox array containing the elementwise tangent of the original array. The input is in radians. Valid for types float and double only.

---

array`.tanh()`

Returns a new Pyvox array containing the elementwise hypebolic tangent of the original array. Valid for types float and double only.

---

array`.trans()`

Returns a new Pyvox array containing the transpose of the original array. Valid for an array of any type.

---

array.`tuple()`

Returns the elements of *array* as a Python tuple, in row-major order; the elements of the tuple will be Python floats, ints, or longs as appropriate.

---

array.`type`

This attribute of a Pyvox array is the internal type used for the contents of the Pyvox array. This attribute may not be modified.

---

array.`uchar(`*lower=0, upper=255*`)`

Returns a new Pyvox array containing the contents of the original array converted to unsigned char. The caller may optionally specify the lower and upper limits of the range to be linearly compressed to fit the 0..255 range of unsigned char voxels; if no limits are specified, then the input range 0..255 will be converted to the output range 0..255.

---

array.`uint()`

Returns a new Pyvox array containing the contents of the original array converted to unsigned int.

---

array.`ulong()`

Returns a new Pyvox array containing the contents of the original array converted to unsigned long.

---

array.`ushort()`

Returns a new Pyvox array containing the contents of the original array converted to unsigned short.

---

array.`write(`*filename, extype=???, bigend=1*`)`

Writes the contents of a Pyvox array to an external file in some specified external format. If a string is given as the file argument, then that file is opened for writing, the data are written to it, and the file is closed; if a file object is given, then the data are written to the file starting at its current position but the file is neither closed nor repositioned after writing. If no external data type is specified, the data are written in the "natural" data type corresponding to the type of the Pyvox array; note that careless use of this feature can produce output files that you don't know how to read back in.

---

array.`writeppm(`*filename*`)`

Writes the contents of a Pyvox array to an external file in the binary Portable Pixel Map format. The array must be unsigned char and rank 2; the last dimension must be 3 and contain the red, green, and blue components of each pixel in that order.

---

array.`zerbv(`*thresh*`)`

Returns a new Pyvox array in which voxels whose local variance exceeds the given threshold have been set to zero; the threshold is specified as the standard deviation. The intent is that the output image should contain only voxels in the interior of regions with approximately uniform tone, and that voxels on the boundary between regions should be zeroed; this should make histogram-based segmentation algorithms more robust by eliminating partial volume effects from the analysis. Voxels at the edges of the image are boundary voxels by definition, to avoid the problems of having an incomplete neighborhood. The threshold value is given as the standard deviation, or

the square root of the threshold variance, and is a compromise between the intensity variance of uniform regions and the difference in gray level between uniform regions. As a rough rule of thumb, take the threshold to be 2 or 3σ, where σ is the standard deviation of the intensity variation within a uniform region; if this value is comparable to or larger than the intensity differences between regions, this algorithm probably won't produce any useful results. The minimum and maximum possible values of the standard deviation are approximately 0.19 and 127.5 for a 3D unsigned char image.

---

kernel.`bias`

The bias attribute of a kernel object is the value to which is added the sum of coefficient times voxel intensity for each voxel in the neighborhood. This attribute may be either read or written.

---

kernel.`coef`

Returns a list of the kernel coefficients, one for each neighbor given in the kernel and corresponding to the delta list. This attribute may not be written, to guarantee consistency between the coefficients and the deltas.

---

kernel.`count`

Returns the number of neighbors defined in the kernel.

---

kernel.`delta`

Returns a list of kernel deltas. Each delta is a list of coordinate offsets relative to the center of the neighborhood.

---

kernel.`rank`

Returns the rank of the kernel, which is the same as the number of dimensions in which it is defined.

`optim.obfunction` [base class]

This base class defines an objective function to be minimized using the optimization functions defined in the `optim` module, and also contains the parameters that control the minimization process. It may be used by itself when the function to be optimized is simple, or as a base class for more complex optimizations. Note that the precise interpretation of the optimization parameters is determined by the particular optimizer used, although the following descriptions should be typical.

An instance of an objective function has the following attributes:

obfun`.npar` is the number of dimensions in the parameter space, or the number of arguments to the objection function.

obfun`.funct` is the function to be optimized and will be invoked when the instance is used as a function, as in *obfun*`(foo)`. The user or derived class may elect to do strange things by overriding the `__call__` method, in which case the `funct` attribute is traditionally set to None.

obfun`.xtol` is a vector of tolerances in the abscissae; optimization stops when the dimensions of the box which the optimizer is currently exploring become less than the tolerances.

obfun`.ftol` is the tolerance in the objective function; optimization stops when the range of the objective function within the region the optimizer is currently exploring becomes less than this tolerance.

obfun`.xopt` is the best abscissa found so far, or None if no optimization has been attempted yet. It is usually a vector, but will be a scalar for one-dimensional optimization.

obfun`.fopt` is the value of the objective function at the abscissa `xopt`.

obfun`.step` is a vector of "reasonable" step sizes defining a box within which the optimizer should start exploring for a minimum; the optimizer is not, however, prohibited from moving outside this initial region.

obfun`.niter` is the maximum number of iterations that the optimizer should attempt. Generally speaking, an iteration consists of a complete cycle through all parameters, but the precise definition depends on the particular optimizer.

The `obfunction` class itself does not define any user-callable methods, although its derived classes often do.

`optim.powell(`*obfun*`)`

This function in the `optim` module implements the Powell direction set optimization algorithm. It takes one argument, which is an objective function (that is, an instance of the `optim.obfunction` class or a derived class) which defines the function to be minimized and the the parameters for the optimizer. The location and value of the minimum found by the optimizer is stored in the `xopt` and `fopt` attributes of the objective function.

`pyvox.affine(`*ndim, matrix=identity, offset=0*`)`

Creates a new affine transform in the given number of dimensions with the given matrix and offset. If the matrix is omitted, an identity matrix is used; if the offset is omitted, zero is used.

`pyvox.AffineType`

Returns the Python class object for an affine transform, which can be used with the Python function `isinstance` to check if some object is a Pyvox affine transform.

`pyvox.array(`*dimen, type=double, data=0*`)`

This method constructs a new Pyvox array with given shape and type, and fills it with data from a list. The data must be a list of numbers, and the total number of data provided must exactly match the total number of elements in the array to be created. As a special case, the given data may be a scalar value, in which case the entire array is filled with that value; the default is to fill the array with zeroes. The internal type defaults to double, which is larger than is necessary in many applications but can contain any value (except complex).

`pyvox.ArrayType`

Returns the Python type object for a Pyvox array, which can be used with the Python function `isinstance`() to check if some object is a Pyvox array.

`pyvox.column(`*data, n=None*`)`

Creates an $n \times 1$ Pyvox array of type double from the given *data*, which may be an int, float, tuple, list, or a Pyvox array of any shape; if *data* is an int or float, then n must be specified and the value of *data* is used for all elements of the column vector. If n is not specified, it defaults to the actual number of elements in *data*. If n is specified but differs from the actual number of elements in *data*, or if those elements are not numeric, then an exception is thrown.

The function `pyvox.point()` is identical to this one but may more clearly express the user's intent in some contexts.

`pyvox.const(`*dimen, intype=double, value=0*`)`

Creates a constant Pyvox array with the given dimensions, internal type, and constant value. The value defaults to zero. The internal type defaults to double, which may be much larger than is necessary in many applications but can contain any value (except complex).

This method is now deprecated; use `pyvox.array()` instead.

`pyvox.diag(`*data, n=None*`)`

Creates a new $n \times n$ Pyvox array of type double whose diagonal elements are taken from the given *data*, which may be an int, float, tuple, list, or Pyvox array of any shape. If *data* is an int or float, then n must be specified and the value of *data* is used for all diagonal elements. If n is not specified, it defaults to the actual number of elements in *data*. If n is specified but

differs from the actual number of elements in *data*, or if those elements are not numeric, then an exception is thrown. Use the method *array*`.diag()` to extract the diagonal elements of a matrix.

`pyvox.kernel(`*deltas, coefs=None, bias=0*`)`

Returns a new Pyvox kernel with the given neighbors (defined by coordinate offsets or deltas), the coefficient for each neighbor, and the constant bias for the convolution. The coefficients are optional; if omitted, the kernel defines a neighborhood only. The bias is also optional; if omitted, it defaults to zero. Once constructed, the kernel cannot be changed, although this restriction may be lifted in later versions. The number of neighbors is determined by the length of the outer delta list; the length of the coefficient list must be the same, or zero. The rank is determined by the length of the inner delta lists; it is an error if these are not all the same. Kernels containing no neighbors are invalid by decree, since it is not possible to determine the rank in that case.

`pyvox.KernelType`

Returns the Python type object for a Pyvox kernel, which can be used with the Python function `isinstance`() to check if some object is a Pyvox kernel.

`pyvox.matrix(`*data, nr=None, nc=None*`)`

Creates a $nr \times nc$ Pyvox array of type double and fills it with the contents of *data*, which may be a Pyvox array of any shape, a list (tuple) of lists (tuples) of numbers, an int, or a float; if *data* is an int or float, then *nr* and *nc* must be specified and a diagonal matrix with the value of *data* along the diagnonal is returned. If *nr* and *nc* are omitted they default to the values implied by *data*; if *nr* is provided but *nc* is not, then *nc* is set equal to *nr*; if they are provided but fail to match the values implied by *data*, then an exception is thrown. An exception will be thrown if the contents of *data* are not all numeric. (Note that this function will not allow you to create a

matrix from a flat list of numbers; for that, you should use `pyvox.array()` with appropriate arguments.

`pyvox.monomials(`*n, k*`)`

Returns a new Pyvox array defining the monomials in k literals of total degree less than or equal to n. Each row of the table defines one possible monomial; each column gives the power to which the corresponding literal is to be raised. For example, the following array gives the monomials of degree ≤ 2 in 2 literals, which we shall take to be y and x.

0	0	means	1
0	1	means	x
0	2	means	x^2
1	0	means	y
1	1	means	xy
2	0	means	y^2

The order of rows in the table is not guaranteed to be consistent from one version of Pyvox to the next.

This function may be useful in generating a set of nth order moments in k dimensions for use with the *array*`.mop()` function.

`pyvox.point(`*data, n=None*`)`

Creates an $n \times 1$ Pyvox array of type double from the given *data*, which may be an int, float, tuple, list, or Pyvox array of any shape; if *data* is an int or float, the n must be specified and the value of *data* is used for all coordinates of the point. If n is not specified, it defaults to the actual number of elements in *data*. If n is specified but differs from the actual number of elements in *data*, or if those elements are not numeric, then an exception is thrown.

This function is identical to `pyvox.column()` but may more clearly express the user's intent in some contexts.

`pyvox.ramp(`*shape, type=double, axis=Last*`)`

Returns a new Pyvox array of the specified shape and type, each voxel of which contains its own coordinate along the specified axis, converted to the array type according to the usual C rules. The type defaults to double, and the axis defaults to the last axis.

`pyvox.rawread(`*filename, dimen, extype=uint1, bigend=1*`)`

Return a new Pyvox array initialized with raw image data read from an external file in some specified external numeric format and converted to the most natural internal format. The arguments are the file to read from, the desired shape of the array (as a tuple or list), the external type (which defaults to unsigned char), and an optional flag to mark bigendian (which defaults to bigendian).

If the given filename ends in `.gz` or `.Z`, it is assumed to be a compressed file and is uncompressed into a temporary file and then read. If there is no image file with the given name, but there is a file with either `.gz` or `.Z` appended, then that file is assumed to be the image file compressed, which is uncompressed on the fly and read instead.

The first element of the dimen array may be zero, in which case the number of slices is determined by the size of the (uncompressed) image file.

`pyvox.read(...)`

A deprecated name for what is now called `pyvox.rawread`, and will soon be replaced by a read function for multiple formats.

`pyvox.vector(`*data, n=None*`)`

Creates an n-element rank-1 Pyvox array of type double from the given *data*, which may be an int, float, tuple, list, or Pyvox array of any shape; if *data* is an int or float, the n must be specified and the value of *data* is used for all elements of the vector. If n is not specified, it defaults to the actual number of elements in *data*. If n is specified but differs from the actual

number of elements in *data*, or if those elements are not numeric, then an exception is thrown.

`regis.info(`*image, mask=None*`)`

This function in the `regis` module computes the information content, in bits per voxel, of an unsigned char image of any rank and shape. Zero voxels are assumed to be background and are ignored. The optional *mask* image defines a per-pixel weight; it must be unsigned char and the same shape as *image*. The *mask* defaults to one. The information content I in bits per voxel is defined by

$$p_k = \sum_{x_i=k} w_i \Big/ \sum_i w_i \tag{4.7}$$

$$I = \sum_k p_k \log_2 p_k \tag{4.8}$$

where i ranges over the voxels in the image, k runs over the intensity levels 1 through 255, x_i is the intensity of the ith voxel, and w_i is the weight for the ith voxel.

`regis.mutinfo(`*image1, image2, mask=None*`)`

This function in the `regis` module computes the mutual information of two unsigned char images, in bits per voxel. The two images *image1* and *image2* may be of any rank and shape but must be the same rank and shape. The optional *mask* image defines a per-pixel weight; it must be unsigned char and the same shape as the other two images. The *mask* defaults to one. The mutual information I in bits per voxel is defined by

$$p_{kl} = \sum_{x_i=k;\ y_i=l} w_i \Big/ \sum_i w_i \tag{4.9}$$

$$p_k = \sum_l p_{kl} \tag{4.10}$$

$$p_l = \sum_k p_{kl} \tag{4.11}$$

$$I = \sum_{kl} p_{kl} \log_2 \left(\frac{p_{kl}}{p_k p_l} \right) \tag{4.12}$$

where i ranges over the voxels in each image, k and l run over the intensity levels 0 through 255, x_i and y_i are the intensities of the ith voxel in the first and second images, and w_i is the weight for the ith voxel.

`regis.obaffine(`*source, target, init=1)*

This class invocation creates an instance of the `obaffine` class, which is used to define an objective function for the affine registration of two images. Its arguments are the *source* and *target* images to be registered, plus an optional initial guess *init* at the transform that will register the two images. The initial guess defaults to the identity transform and may be an affine transform, an *obaffine* instance, or an *obrigid* instance. In either of the latter two cases, the `metric`, `scenter`, and `tcenter` attributes are also copied into the new obaffine object.

The class is derived from `regis.obregis` and inherits the attributes and methods of that class. The following additional attributes and methods are defined.

The `ndim` attribute is the number of dimensions in the images to be registered. Both images must be the same number of dimensions. This attribute is set automatically when the instance is initialized and should not be altered.

The `npar` attribute is the number of parameters needed to define an affine transform; it is set automatically and should not be modified.

The `xtol`, `ftol`, and `step` attributes of the `obregis` class are set to reasonable default values and do not usually need to be set by the user.

The `itgt2src()` method returns the best transform found so far in index coordinates, as an instance of the `affine` class. The transform returned is in index coordinates for the unshrunk images and is the desired mapping from target coordinates to source coordinates.

The `ptgt2src()` method returns the best transform found so far in physical coordinates, as an instance of the `affine` class. The transform returned is in physical coordinates for the images and is the desired mapping from target coordinates to source coordinates.

The `ctgt2src()` method returns the best transform found so far in centered physical coordinates, as an instance of the `affine` class. Centered physical coordinates have the same spacing as physical coordinates, but the origin is placed at the specified center of the source or target. The optimizer works in centered physical coordinates, since this generally gives the most rapid convergence.

The deprecated `initrigid(`*obrigid*`)` method initializes the optimizer to the best rigid transform found by an earlier *obrigid* rigid registraton objective function; the *init* argument to the `obaffine` constructor should be used in new code.

`regis.obregis` [base class]

This base class is derived from `optim.obfunction` and is used to derive objective function classes for image registration. It is not usually used on its own but defines the following attributes and methods in addition to those defined by `optim.obfunction`.

The `source` and `target` attributes define the source and target images to be registered. Both images must have the same rank and both must (currently) be of type unsigned char.

The `scenter` and `tcenter` attributes are the nominal centers of rotation and scaling for the source and target images. They are not strictly necessary, but choosing good values here often yields faster convergence; setting them to the center of gravity of each image is a good default choice.

The `sspacing` and `tspacing` attributes are the physical pixel spacings in the source and target images. They are used to correct for aniotropic sampling and default to the spacing attributes of the source and target images; few user will need to set the explicitly.

The `metric` attribute selects the metric to be used to measure the quality of the match between the two images. The following choices are currently supported: "correl" uses a Pearson product moment correlation; "mutinfo" uses the mutual information between the two images; and "norm2" uses the L2 norm or RMS error.

The `scale(`*shrink, smooth=0*`)` method sets shrink and smoothing levels for multi-scale registration. The values set remain effective until new values are set with this method; the best transform found so far is automatically modified to suit the new scale. When the shrink parameter is not zero (which

is its default value when an instance is created), the metric is evaluated on images shrunk by the factor 2^{shrink}; thus if the shrink is 2, the metric is evaluated on an image reduced by a factor of 4 in each dimension. When the smooth parameter is not zero (its default value), the images being registered are lowpass filtered smooth times after being shrunk and before being registered. (The original unshrunk and unsmoothed images are saved for later use.) Initially registering with shrunk and smoothed images and progressively unshrinking and unsmoothing them often provides a faster and more robust algorithm than attempting to register using only the original images.

`regis.obrigid(`*source, target, init=1)*

This class invocation creates an instance of the `obrigid` class, which is used to define an objective function for the rigit registration of two images. Its arguments are the *source* and *target* images to be registered, plus an optional initial guess *init* at the transform that will register the two images. The initial guess defaults to the identity transform and may be an affine transform, an *obrigid* instance, or an *obaffine* instance. In either of the latter two cases, the `metric`, `scenter`, and `tcenter` attributes are also copied into the new *obrigid* instance. Note that, while it is possible to set the initial transform to other than a rigid transform, it is rarely sensible to do so.

The class is derived from `regis.obregis` and inherits the attributes and methods of that class. The following additional attributes and methods are defined.

The `ndim` attribute is the number of dimensions in the images to be registered. Both images must be the same number of dimensions. This attribute is set automatically when the instance is initialized and should not be altered.

The `npar` attribute is the number of parameters needed to define an affine transform; it is set automatically and should not be modified.

The `xtol`, `ftol`, and `step` attributes of the `obregis` class are set to reasonable default values and often do not need to be set by the user.

The `itgt2src()` method returns the best transform found so far in index coordinates, as an instance of the `affine` class. The transform returned is in index coordinates for the unshrunk images and is the desired mapping from target coordinates to source coordinates.

The `ptgt2src()` method returns the best transform found so far in physical coordinates, as an instance of the `affine` class. The transform returned is in physical coordinates and is the desired mapping from target coordinates to source coordinates.

The `ctgt2src()` method returns the best transform found so far in centered physical coordinates, as an instance of the `affine` class. Centered physical coordinates have the same spacing as physical coordinates, but the origin is placed at the specified center of the source or target. The optimizer works in centered physical coordinates, since this generally gives the most rapid convergence.

Chapter 5

Application Reference

In addition to Pyvox itself, BBLimage also contains various command-line appliction programs written either as Python scripts or directly in C. These are listed briefly below and are fully described by man pages.

5.1 Data File Formats

File Formats	
`cdata(5)`	Commented data files
`mri_data(5)`	MRI data formats used at BBL
`param(5)`	Parameter files

5.2 Applications

Command-line Programs	
`binnseg(1`	Bivariate nearest-neighbor segmentation
`conseg(1)`	Compute concordance of two segmented images
`decomment(1)`	Remove comments from a commented data file
`imstack(1)`	Stack slice files into a volume image file
`inleav(1)`	Interleave 2 single-echo images into a dual-echo image
`lovar(1)`	Compute local variance of an unsigned char volume image
`qdv(1)`	Image viewer for gray-scale volume images
`rpsamp(1)`	Choose random set of points within an image
`skmiv(1)`	Shaded K-means segmentation on interior voxels
`swab(1)`	Swap bytes according to a pattern
`usb2uc(1)`	Convert unsigned short big-endian image to unsigned char
`vibihist(1)`	Compute bivariate histogram of two volume images
`vihist(1)`	Compute univariate histogram of a volume image

Chapter 6

Installation

6.1 Prerequisites

In short, BBLimage requires a Unix-compatible operating system, the Gnu C compiler, Posix-compatible C libraries, Python 1.5, the X Window System with 24-bit true color visuals, the LAPACK and BLAS libraries, and Lesstif or Motif. If this describes your system, there is a pretty good chance that you can compile and install BBLimage without having to do anything special. If not, or if you run into problems, the sections below discuss possible solutions.

6.1.1 ANSI C Compiler

BBLimage is intended to work with any ANSI (1989) C compiler, but for the moment that compiler must be the Gnu C compiler (gcc). This is considered a bug but we haven't yet figured out how to portably build shared libraries (which are required for Python extensions). If you manage to get it working with another compiler, please let us know how.

6.1.2 X and Motif

The `qdv` image viewer requires X and Motif with a 24-bit true color visual. As long as the header and library files are in reasonably standard places, no special steps should be needed; if not, use the `--with-c-header-path` and `--with-library-path` configure options to indicate the right place to look. If you don't have X and Motif at all, use the `--without-x` option to leave

out the `qdv` viewer. The requirement for 24-bit true color could be removed in theory, but doesn't seem to us to be worth the effort.

6.1.3 LAPACK and BLAS

BBLimage uses the LAPACK and BLAS libraries for numerical linear algebra; some platforms may also require the f2c library or the F77 and I77 libraries. Note that these libraries must be shared libraries; Python extenstions such as Pyvox cannot be implemented as statically linked code. If you have these libraries installed in a reasonably standard place, then the configure script should be able to find them automatically and nothing special needs to be done. If you have them in some non-standard place, then you will need to use the `--with-lapack` option to specify where; see the section on configuration options for more details. If the configure script cannot find these libraries (which must be shared libraries), or if you specify the `--without-lapack` option, then it will use its own internal light-weight version.

It should be noted that LAPACK and BLAS libraries tuned for your specific platform are generally much better if you care at all about numerical linear algebra; this light-weight code is provided only as a convenience for users who are primarily interested in image processing and don't want to spend a lot of time getting BBLimage up and running.

6.1.4 Python

BBLimage currently requires version 1.5 of Python; we'll get around to extending to version 2.x any day now. There is a `--without-python` option to `configure` that compiles only the C language bindings but it has not been tested for some time now.

6.1.5 Miscellaneous

The `/usr/bin/env` command is required; the Python scripts use this to find the Python executable without knowing its exact path. If you don't have it for some reason, you will need to modify the first line of each Python script to indicate where the Python executable is found.

If `gzip` is visible on the path during configuration, then it will be used to support automatic uncompression of image files for reading.

6.2 Procedure

BBLimage has been sucessfully compiled and run under Red Hat Linux 6.1 and later on Intel and under Solaris 2.7 and later on Sparc. We will be interested to hear about other successes or failures, and very interested to receive fixes that will yield success on other machines and operating systems.

The following instructions should work on most systems. If they don't, or if one of the special caveats below applies to you, see the other sections in this chapter.

If you are upgrading from BBLimage 0.62 or earlier, see the the section "Upgrading Old Installations" below on manually fixing some incompatibilies between the old and new versions.

If you intend to install both BBLimage and segm, you should install segm first, then BBLimage; there are a few installed files that are shared between the two packages, and BBLimage is most likely to have the current version.

The following steps will usually suffice:

1. Unpack the tar file and cd into the source tree.

2. Run `./configure` to guess the right parameters for your machine. See the section "Configuration Options" below for possible options to this command.

3. Run `make` to compile and link everything.

4. If you're a programmer, run `make tags` to create the `TAGS` file for emacs; if you're a vi partisan, make the obvious change to the Makefile first.

5. As root, run `make install` to install all the executable binaries and man pages, usually in `/usr/local/bin` and `/usr/local/man`. See the configuration options below if you want to install it elsewhere.

6. Several Python extension modules are installed in the directory

   `$PREFIX/lib/python1.5/site-packages`;

   you will need to add this directory to your `PYTHONPATH` if it is not already there.

7. Similarly, the shared library `libbbli.so` is installed in `$PREFIX/lib`; you may need to add this directory to the search path for shared libraries. The details for doing this will depend on your operating system, its setup, and your choice of shell; a few systems are described in the section "Particular Systems" below.

6.3 Upgrading Old Installations

If you are upgrading from BBLmage version 0.62 or earlier, you may need to make the following changes by hand.

- The default location for installing the Python modules has changed from `$PREFIX/lib/python1.5` to `$PREFIX/lib/python1.5/site-packages`. You should remove the old files `pyvox.so` and `exim.so` from the old location.
- The pyvox module is now defined by a Python file `pyvox.py` and a shared library `pyvoxC.so`; it was previously defined by a shared library `pyvox.so`. You should remove the old file `pyvox.so` to prevent it from shadowing `pyvox.py`.
- Versions of Pyvox prior to 0.63 advocated creating a link from `/usr/local/bin/python` to `/usr/bin/python` if needed; this is no longer necessary and should be removed unless needed for some other reason.

6.4 Particular Systems

6.4.1 Linux

You can make the new shared libraries available by adding the the directory `$PREFIX/lib` to `/etc/ld.so.conf` and running `ldconfig`, or by adding the directory to `LD_LIBRARY_PATH` in `/etc/profile` or other shell init file. The most convenient way to declare `PYTHONPATH` for all users is to add the line

```
export PYTHONPATH=$PREFIX/lib/python1.5
```

to `/etc/profile`, substituting the proper value of `PREFIX` where appropriate.

6.4.2 Solaris

Some of the X headers provided with Solaris omit the type declaration on many of the functions they declare, letting it default to 'int'; this yields a page or two of warning messages, which can be ignored.

6.5 Configuration Options

The following options may be provided to the configuration script to provide for special needs. To change the options, you should run `make distclean` to clean up the source tree before running configure with the new options.

1. The machine-independent files (only the man pages, at the moment) are installed in `$prefix/man`, where `prefix` defaults to `/usr/local`. You can specify another location `PATH` by using the

   ```
   --prefix=PATH
   ```

   option to the configure command.

2. The machine-dependent files are installed in the location specified by `$exec_prefix`, which defaults to `$prefix`. More specifically, the executable programs and scripts are installed in `$exec_prefix/bin`; the libraries (except the Python modules) are installed in `$exec_prefix/lib`; and the Python modules are installed in `$exec_prefix/lib/python1.5`. The

   ```
   --exec-prefix=PATH
   ```

   option to the configure script can be used to specify another location.

3. In most cases, the configure script will automatically find the necessary header and library files. If not, the configure options

   ```
   --with-c-include-path=PATH
   --with-library-path=PATH
   ```

may be used to indicate the directories where they may be found. For example, `--with-c-include-path=/img/prog/include` will cause that directory to be added to the list of directories searched for include files. Multiple directories may be specified and are separated by colons. The environment variables `C_INCLUDE_PATH` and `LIBRARY_PATH` work the same way.

4. Motif-compatible headers and library files are required to compile the qdv viewer. They will be found automatically if present in the usual place within the X11 directory tree. If they are actually somewhere else, use the `--with-c-include-path` and `--with-library-path` options described above to indicate where.

5. If you don't have X and Motif, or don't want to use them, the configure option

   ```
   --without-x
   ```

   will omit compilation of all the programs and libraries that use X.

6. BBLimage will try to find and use the platform-specific versions of the LAPACK and BLAS libraries if they exist, but will use its own generic light-weight version if they cannot be found. The option

   ```
   --with-lapack=OPTIONS
   ```

   allows the user to specify any necessary `-L` and `-l` loader options to get the platform-specific libraries, including libf2c or libF77 and libI77 if necessary; the options string will need to be quoted if it contains blanks. (The `-L` options could also be specified through the `--with-library-path` configure option.) The option

   ```
   --without-lapack
   ```

   forces BBLimage to use its own light-weight libraries.

7. The configure script attempts to find an existing installation of Python 1.5 automatically. If it fails, the options

`--with-python=DIR`
`--with-python-exec[=XDIR]`

can be used to specify the location of the Python machine-independent and -dependent files. The header files are expected to be found in `$DIR/include/python1.5`, the executables in `$XDIR/bin`, and the libraries in `$XDIR/lib/python1.5`. The value of `XDIR` defaults to `$DIR`.

6.6 Make Targets

This section summarizes the targets for the make command that are useful for the installer and user; see the same-named section in the Implementation chapter for additional targets useful for developers.

The `all` target compiles (but does not install) all the components of BBLimage that are needed for the specified configuration.

The `clean` target deletes all the compiled and generated files and some related files but does not modify the configuration.

The `distclean` target deletes all compiled and generated files, plus the files that define the configuration. In general, it attempts to restore the directory to its "as-distributed" state.

The `install` target installs the compiled code into the locations specified by configuration. In general, you must be root to install BBLimage.

The `dvi` and `pdf` targets regenerate the dvi and pdf forms of the documentation from their original TeX files. You will need to have LaTeX installed for the `dvi` target, and both LaTeX and ps2pdf installed for the `pdf` target. The documentation is distributed in pdf format, so most users will never need to do this.

Chapter 7

Implementation

This Chapter is intended primarily for the developers of Pyvox itself, although other users may find the discussion of design decisions interesting. (The decisions are ordered from basic to technical, so begin at the beginning and read until it becomes too technical.) Even this Chapter does not give all the details; for that you must consult the source code. But it does try to give you enough orientation that you understand the architecture, can easily find the right source code to read, and understand why things were done as they were. A final section discusses some design issues that are still open.

7.1 Some History

A bit of history may be helpful in understanding the organization of the software. BBLimage was originally designed as a toolkit of image processing functions intended to be called from C, plus a set of command-line programs that would call the lower-level toolkit to provide user-level functionality.

The results were not entirely satisfactory. Building complete image analysis protocols for end users by using shell scripts to connect CLI programs was just plain painful and involved constantly reading and writing images to disk; on the other hand, writing complete protocols in C involved getting a lot of fussy little details straight that distracted from the image processing algorithm itself.

The current approach is to encapsulate the image processing library as an extension of the Python language, which was chosen because it is a full-featured, high-level programming language which is very easily extended in

C. This approach makes it easy to program new analysis protocols in Python while still permitting the lower-level functions to be written in highly efficient C. There are, however, still many command-line programs that have not yet been converted into Python scripts.

BBLimage previously contained the programs BrainMask, Kmean_3Dseg, and AdpKmean_3Dseg_Ebeta, which were originally developed by Michelle Yan for use with specific MR imaging protocols used at the Brain Behavior Lab. These programs are heavily used at BBL for image analysis, but have not proven adaptable to other imaging protocols. They have been moved into the segm package (which is made available to the public but is not recommended for general use) and are no longer included in BBLimage.

7.2 Design Decisions and Rationale

7.2.1 Target Audience

The primary audience for Pyvox is image analysts in neuroscience and related research groups who need to develop automated image analysis protocols and apply them to hundreds of large images. The key quality criteria for this group include rapid development and validation, efficiency, and robustness. Portability will be important to any analysts who have a platform other than the few that Pyvox is being developed on. Ease of learning, pretty graphical user interfaces, and elegant code are definitely secondary issues; anyone who needs to process hundreds of images can be assumed to be willing to spend some time learning how to do it efficiently and to want effective automation more than a pretty graphical interface. Clarity, maintainability, and extensibility of the source code, while of little interest to the image analyst, are of considerable interest to the developers; they will be given high priority but may be sacrificed if necessary to the primary virtues of rapid development, efficiency, and robustness. By the way, rapid development refers to the rapid development of applications using Pyvox, not necessarily to the development of Pyvox itself.

The reason for choosing this audience is that it's the itch *I* need to scratch. Researchers who need to do rapid prototyping of algorithms without worrying about efficiency in applying them, and students who want to experiment with medical image processing will not be deliberately excluded, but if it comes down to inconveniencing them or inconveniencing my primary audience, I'll

focus on the needs of my primary audience.

7.2.2 Target Platform

Pyvox is generally optimized for a modern scientific or engineering workstation or high-performance personal computer, say a system with dual 500 MHz or faster processors, 256 MB or more of RAM, 20 GB or more of fast hard disk, 1280×1024 or better display resolution with 24-bit color, and a 19-inch monitor or better. The software is designed to be portable, but there is a definite bias toward Linux and Unix platforms, because that's what I'm most experienced and comfortable with; volunteers to get Pyvox to run well on Windows or Macintosh will be gratefully received. Pyvox will probably run successfully on smaller and slower platforms (assuming enough swap space and hard disk) but s-s-l-l-o-o-w-w-l-l-y-y. As for larger and faster platforms—if someone would like to donate a supercomputer and its upkeep, I'll be happy to make Pyvox work on it.

7.2.3 Open Source License

Pyvox is distributed under an Open Source license (which permits free modification and distribution) for several reasons. First, I believe that software is a form of scientific knowledge and that science advances most rapidly when we can build on each other's work rather than re-implementing the wheel. I hope that the people who find this software useful will reciprocate by contributing bug fixes and other improvements to be folded back into the master copy for future releases. Second, I find that we write better software when I expect that dozens of people will be reading my code than when I am writing just for myself. Finally, I would rather spend my time doing science rather than trying to monitor and enforce a more restrictive license. (Note: The only reason that last paragraph says "I" rather than "we" is simply that I'm the only developer so far.)

Since Pyvox is funded in large part by federal research grants, I do not feel that it is ethical to prohibit for-profit organizations (who do, after all, pay some of the taxes which support Pyvox) from using this code. Thus I use a license similar to the BSD license rather than the Gnu General Public License and do *not* use GPLed code within Pyvox to avoid its viral property.

Note also that I am an academic, for whom publications and citations are often worth more than money (at least, they can often be converted into

tenure and money), so I really do want to see citations of this work.

7.2.4 Large Images

Pyvox is optimized for "large" images, by which we mean images that will fit comfortably into main memory, but not into L2 cache. A modern 32-bit workstation can typically support up to 3-4 GB of virtual memory; physical memory may be somewhat smaller. L2 cache is typically about 256 to 2048 KB. An MRI volume image containing $256 \times 256 \times 256$ voxels of 8- or 16-bit data, or a data set of several such images, would be representative. Efficient processing of large images requires careful attention to locality and blocking to avoid unnecessary traffic between the cache and main memory. On the good side, Pyvox does not need to be particularly careful about limiting the size of image headers; any reasonable information may be included in the header without noticeably increasing the total memory requirements.

Pyvox will support "small" images that fit into L2 cache but will not exploit their small size for improved efficiency. The actual payload in a "tiny" image such as a 4×4 array used to represent an affine transformation will likely be overwhelmed by the size of the header; since relatively few of these are expected to be used, the cost in memory and computer time should be acceptable.

On the other hand, "huge" images that will not fit into main memory (or a single disk file) and must be processed in pieces introduce a whole new set of problems that Pyvox will not attempt to handle, at least yet.

7.2.5 Image Operations

Pyvox emphasizes the use of operations that work on entire images, with operations on individual voxels an anomaly. This viewpoint will be familiar to experienced Matlab programmers, but will seem bizarre and uncomfortable to C and Fortran programmers. Rest assured, however, that the effort of recasting algorithms into operations on entire images pays off in efficiency, because much of the overhead in dealing with single voxels can then be amortized over the entire image; this is especially true in comparing pixelwise operations written in Pyvox to imagewise operations written as a C extension to Python.

7.2.6 Focus on the Core Engine

Pyvox focuses on the computational engine for image processing, and is designed to be used from a scripting language rather than interactively; it provides a graphical user interface (GUI) only when human intervention is absolutely necessary. GUIs are nice for interactive experimentation but do not lend themselves to batch processing or reproducible analysis protocols. For our target audience, the effort put into a GUI would usually be better spent in improving the computational engine.

A slightly more subtle issue is that Pyvox focuses on the core image processing algorithms such as convolution, resampling, etc rather than attempting to implement the wide variety of segmentation, registration, etc. algorithms currently available in the literature. The idea is that the wider variety of complete algorithms can be written in Python using the efficient core functions provided by Pyvox; they can thus be both concise and efficient.

7.2.7 Installation Prerequisites

Pyvox is designed for the serious user who intends to process many images with it; such a user is assumed to be willing to expend a little additional effort in installation to obtain more efficient operation. Thus we recommend that the user take the extra time to install the best available libraries (currently just LAPACK and BLAS) before installing Pyvox, although we also provide an internal lightweight version for the impatient.

7.2.8 Moderate Portability

Pyvox is designed to be moderately portable; this means that it should compile and run with at most minor modifications on a wide variety of modern platforms, especially Unix system. It does not, however, attempt to handle every possible perversion permitted by the relevant standards. A general rule is that code should be written to be platform-independent whenever feasible and reasonably efficient; but if portability requires platform-specific code that we don't have sample platforms to test on, we'll just go ahead and be non-portable to those platforms.

For example, some of the code in `exim` assumes that signed integers are represented in two's complement format, and that the value -2^{n-1} has a valid representation. Since handling one's complement machines would require

special-case code which cannot be tested on any machine we have, we simply don't try to handle one's complement machines. Similarly, we don't try to handle platforms on which the char type is not exactly 8 bits, or the character representation is not ASCII.

On the other hand, both big- and little-endian platforms are supported. (Middle-endian platforms are not.) Any platform which supports ANSI C (1989) and Posix should be able to comple and run Pyvox with little or no modification.

Floating point in other than IEEE 754 format and ints shorter than 32 bits are intermediate cases. They are supported in principle, but we don't develop on any platforms that don't support these possibilities, so some dependencies may have crept in without detection.

7.2.9 Efficiency Tradeoffs

An emphasis on run-time efficiency tends to degrade ease of use, robustness, maintainability, extensibility, generality, and all those other virtues; Pyvox is by no means exempt from this trade-off, and it is necessary to decide when efficiency should dominate and when the other virtues should be more important.

Operations in Pyvox can be roughly classified by how frequently they are executed: Per-image operations are done once or only a few times per image. Per-pixel operations are done once (or more) for each pixel in an image; image addition or histogramming are good examples. Per-neighbor operations are done once (or more) for each neighbor of each pixel in an image; convolution is the canonical example. Per-neighbor and per-pixel operations will normally constitute the largest fraction of the computer time spent in an algorithm, with per-image operations as a minor contributor. As a rough estimate, we might say that per-neighbor operations constitute 80% of the run time, per-pixel operations about 15%, and per-image about 5%. It follows that efficiency matters enormously in per-neighbor and per-pixel operations, and hardly at all in per-image operations.

The general policy is thus to emphasize efficiency in per-neighbor and per-pixel operations, even if it requires sacrificing clarity, generality, and ease of use. On the other hand, per-image operations should emphasize clarity, generality, and ease of use.

7.2.10 Parallel Processing

Pyvox is being written to be thread-safe wherever possible, to facilitate the possible future use of multiple processors; however, there are no current efforts to actively exploit multiple processors except by running multiple copies of Pyvox in parallel (which is soon limited by available memory).

Pyvox will probably never try to exploit the parallelism possible in a network of workstations. If you've got multiple workstations, the most effective way to use them (for our target audience) is usually to process separate images in parallel on separate workstations and there is little benefit to the complex coordination and data communication required to harness multiple workstations to process a single image.

7.2.11 Data Typing

The header for a Pyvox arrays includes a field encoding the data type contained in that array. The functions in BIPS switch on this field to determine which efficient loop to use. Most higher-level functions are then implemented to handle any of the defined data types and do not even need to examine the type field; the only common exception to this rule is that some operations are meaningful only for floating point data. This approach facilitates simple, generic high-level routines, at the cost of messy (but efficient) code at the lower levels. It is also consistent with Python's data typing, which is attached to data items rather than to the variables that contain them.

7.2.12 Limited Number of New Types

There are two basic approaches to choosing the new types (or classes) to be implemented in a package. The "splitting" approach is to embody even fine distinctions between object usages into distinct types or classes; the "lumping" approach is to introduce distinct new types only where it seems unavoidable, and otherwise overloading existing types with specific interpretations. Pyvox has generally chosen to lump, on the grounds that this is probably the best choice for a small, compact package that nevertheless intends to be used in a wide variety of applications.

7.2.13 Short Function Names

Similarly, function and method names can be either short and abbreviated, or long and explicit. Pyvox has generally chosen to use short, mnemonic names at the user level rather than long, explicit names for essentially the same reasons that it introduces only a few new types—it seems unnecessarily complex to use long names for a small, compact package developed by a single programmer. Lower-level functions, on the other hand, often have longer descriptive names.

7.2.14 External Data Formats

For simplicity in transfering data files between platforms with possibly different internal data representations, Pyvox recommends and supports writing and reading data files in defined external formats rather than in the native formats; the code that actually imports or exports the data is written to be platform-independent. The recommended formats are the ones most commonly used internally: two's complement signed integers, and IEEE 754 floating point; other external formats may be added as needed. The choice between big and little endian is essentially arbitrary; BBL has standardized on big-endian because most of our data was originally written in that byte order.

The current version of Pyvox supports only raw pixel data in raster order; extensions to handle DICOM are almost certain but have not yet been implemented. Other extensions could be added; the only ones that are likely in the near future are Analyze and NRIA formats. Some tools for reading legacy data are also provided in exim.

7.2.15 Internal Data Formats

The performance of many operations on large images is limited by main memory bandwidth and can be improved by using a data type no longer than is necessary to represent the data values. To facilitate this, Pyvox supports essentially the full set of data types provided by the underlying C language. The sole exception is plain char, although both signed and unsigned char are supported; the reasoning is that a type of unknown signedness is worthless for numerical work and the Pyvox array type is not intended for text processing.

Float and double complex types are planned, mostly to support the FFT

and frequency domain processing; the set of supported operations is likely to be limited compared to the native types.

7.2.16 C

Most of Pyvox, and all of the low level functions, are written in ANSI C (1989) because the language is well standardized, lends itself to efficient software, and good open-source optimizing compilers are readily available for a variety of different platforms; the fact that I am experienced with and comfortable with C was also a consideration. Once compilers for the 1999 C standard become available, it is likely that Pyvox will be modified to match; some features such as the restrict and inline keywords are already used when the compilers support them as extensions.

C++ might have been a possibility except that I didn't have any experience with it, and it was not clear that C++ would be compatible with Python. C++ also seems to encourage inefficient programming, which is not a good thing for this application. C++ does support some tempting capabilities, so an upgrade to C++ is still possible.

Fortran 77 might have been more efficient for the lowest-level operations (because the aliasing rules permit better optimization) but does not support data structures or structured programming; Fortran 90 does but open source compilers are not available. Other languages including Ada and Java were excluded simply because I have no experience with them.

7.2.17 Python

Although Pyvox is written primarily in C, its applications will normally use an interactive scripting language to support rapid development. The language chosen is Python, because it is well-suited to C extensions, has a well-defined syntax and sematics, supports a reasonable implementation of objects, and is portable to a variety of platforms including Unix, Windows, and Macintosh. Perl was rejected because it does not facilitate extensions in C and because its semantics is rather ad hoc; any language in which experimentation is necessary to determine how to do an operation is not well suited to a large software project. Tcl does not support objects and has a rather limited semantic model. Java was rejected because I have little experience with it and it seems more suited for compilation than interactive design.

It is also important to note that Pyvox has been designed to support a *single* scripting language. Attempting to support multiple scripting languages requires either restricting capabilities to the common subset of the languages or providing multiple variants suited to each language.

7.2.18 LaTeX

Useful software requires documentation, and documentation requires choosing a word processor or document compiler. I standardized on TeX/LaTeX more than a decade ago because it is unsurpassed at typesetting mathematics; while that particular virtue is largely irrelevant to this project, I see no reason to change a winning strategy. TeX has also proven highly portable and stable. Its major disadvantage is that it takes a long time to learn how to use effectively.

7.2.19 LAPACK and BLAS

Volume image analysis requires some numerical linear algebra, including eigenvalues, least squares, solution of linear systems, and so on. Various packages are available for this, even some in C. The current state of the art package for linear algebra is LAPACK, supported by BLAS. The major disadvantage of LAPACK and BLAS is that they are written in Fortran, and portably interfacing C and Fortran code is messy at best. It is possible using f2c to convert Fortran into C, but many platforms will have optimized versions of BLAS and it seems foolish not to take advantage of these.

The policy that I've adopted is to use the platform-specific LAPACK and BLAS libraries if available, but to use an internal light-weight version generated using f2c if not. This decision will be reconsidered if things become too messy, but it seems to be working reasonably well so far.

Additional packages for optimization and special functions are likely to be needed in due course; once C/Fortran interfacing is worked out, it becomes possible to use Fortran packages for this purpose as well.

7.2.20 Vectorization over Rows

As has already been discussed, it is inefficient to do most operations pixel-by-pixel; arranging to spread the overhead over many pixels works better. The BIPS layer essentially provides an abstract vector processor for this purpose.

If multiple operations must be done, it is also inefficient to vectorize over an entire large image because the image must then be brought into cache multiple times; it is better to bring into cache a portion of the image and perform the multiple operations on that portion, before bringing in the next portion of the image. While it is theoretically possible to optimize the size of the portion brought into cache, it is difficult to do well in practice. Pyvox generally compromises by bringing in one row or scanline of an image at a time; a row is defined as a set of voxels with the same first $n-1$ coordinates. A row typically contains 128–1024 voxels, which is enough to amortize the loop overhead without overflowing the cache.

A typical loop nest for a point operation looks like this:

```
Setup for the entire image
Loop over the rows of the image
    Setup for the current row
    Loop over the voxels in the row
```

A typical loop nest for a neighborhood operation looks like this:

```
Setup for the entire image
Loop over the rows of the image
    Setup for the current row
    Loop over the voxels of the neighborhood
        Setup for the current neighbor and row
        Loop over the voxels in the current row and neighbor
```

7.2.21 FIXME Notes

Pyvox is (and probably never will be) incomplete; thus it is inevitable that there will be sections of code that are unfinished, have known or suspected bugs, do not handle special cases, and could be made faster or otherwise improved. All such unresolved issues are marked with the special string `FIXME` in the source code so that they may be easily found by a search command; a few such issues appear in the user documentation are are marked by the same string. The principle here is to be honest about the state of the code.

An early attempt was made to distinguish between bugs and enhancements, marking the latter with the string `ADDME`, but it proved too difficult to make the distinction consistently.

7.2.22 Signed Sizes and Indices

Array sizes and indices are stored as signed rather than unsigned longs. The advantage of using unsigned values is that you can specify counts and indices that are twice as large, but there is a formidable set of disadvantages: you cannot easily and safely take the difference of two indices, and the difference may exceed the range of values representable in either signed or unsigned longs; and you cannot safely compare a count or index to a signed value without treating a negative signed value as a special case. Considering that the size of virtual memory and disk files is usually no more than twice `LONG_MAX`, the extra trouble of unsigned counts and indices doesn't seem worth it.

7.3 Open Issues

7.3.1 Error Handling

7.3.2 Image Objects

...meaning objects as connected sets of non-zero voxels, not Python objects

7.3.3 Image Views

7.3.4 Huge Images

7.3.5 64-bit Platforms

7.4 Development Prerequisites

Those who want to participate in developing Pyvox itself will need a few more tools than are necessary to compile and install it. Gnu autoconf and m4 are used to create the configuration script. LaTeX, dvips, and ps2pdf are used for the Reference Manual; groff or some equivalent is needed for the man pages. The f2c Fortran-to-C converter is used to construct the internal lightweight LAPACK/BLAS implementation so that it can be compiled with only a C compiler; a few of the f2c runtime library functions are also needed. Either etags (for emacs) or ctags (for vi) is helpful for rapidly finding particular functions. Of course, substantial knowledge of the Python C API, scientific

programming, C programming, and image processing algorithms wouldn't hurt any.

7.5 Directory Layout

The source directory for BBLimage contains the following subdirectories for specific types of files.

The `bin` directory is reserved for the compiled object files and programs.

The `bitmaps` directory contains the X Window icons used by the `qdv` image viewer.

The `doc` directory contains the original TeX files for the documentation, plus their conversions into dvi and pdf formats.

The `examples` directory contains a variety of image processing scripts written using Pyvox, plus man pages.

The `include` directory contains the C header files used by BBLimage.

The `lib` directory contains the compiled object library files, and various Python modules that implement Pyvox. These make rather strange bedfellows and this directory may get split into two.

The `lite` directory contains the lightweight LAPACK and BLAS implementation used when BBLimage cannot find a native implementation. Its contents are divided into `lapack` and `f2clib` subdirectories to distinguish components taken from different sources.

The `local` directory is reserved for site-specific files and is guaranteed never to be touched by configuration or any of the `make *clean` targets.

The `man` directory contains the man pages for BBLimage proper; man pages for the examples are included in the `examples` directory.

The `src` directory contains the C source code for BBLimage proper, including applications but excluding examples and test scripts. Header files are kept in the `include` directory.

The `test` directory contains test programs and scripts for testing BBLimage.

7.6 Architecture and Code Organization

7.6.1 Voxel Kit

The Voxel Kit is a collection of higher level tools for volume image processing, at roughly the level treated in image processing textbooks; it includes the files `voxel.c`, `voxel.h`, and `vxli.h`. Most of the per-pixel operations within the Voxel Kit are actually done by BIPS, which can be optimized to specific platforms. The Voxel Kit may be called directly from C language programs, or used from Python via the Pyvox extension.

7.6.2 Pyvox

Pyvox is an image processing extension to the Python language written partly in C and partly in Python.

PyvoxC is the C language part of Python; it consists of the files `pyvox.h`, `pyvox.c`, and `parray.c`, encapsulates the voxel kit as a Python extension, and is compiled into a shared library `pyvoxC.so`.

The rest of Pyvox is written in Python; it includes the files `pyvox.py`, which exposes `pyvoxC.so` to the Python user and implements a set of core image processing methods; `optim.py`, which implments a set of optimization algorithms; and `regis.py` which implements a basic set of image registration methods.

7.6.3 BIPS

The BIPS (Basic Image Processing Subroutines) level defines a relatively small set of image processing primitives from which higher level operations can be built and which can reasonably be hand-optimized for specific target platforms; the functions in the Voxel Kit are built from BIPS routines and should not have to be modified for efficiency on different platforms. The relationship between the Voxel Kit and BIPS is essentially the same as between LAPACK and BLAS for those familiar with numerical linear algebra. BIPS includes the files `bips.c` and `bips.h`.

7.6.4 Exim

Exim is a set of functions for translating between external (file) and internal (native) representations of data and is used to permit Sparc-format data to be read on any platform; it includes the files `exim.h` and `exim.c`. It is designed to accomodate many different external data formats, although only the most common formats are currently supported. Pyexim, consisting of the file `pyexim.c`, encapsulates exim as a Python extension (although only the data type names are currently implemented).

7.6.5 Numerical Methods

C wrappers for LAPACK are defined in `clap.h` and `clap.c`; these have not yet been made truly portable.

7.6.6 Language Extensions

The files `errm.c`, `errm.h`, `memm.c`, and `memm.h` encapsulate the standard C error handling and memory management facilities into something more convenient. The files `rand.c` and `rand.h` implement a high-quality random number generator. The files `cdata.c`, `cdata.h`, and `decomment.c` implement some functions for processing "commented data"; that is, data files with embedded comments that can be usefully read by either a person or a computer. The file `dstring.c` implements some functions for dynamically allocated strings, which seem much less useful now that the facilities of Python can be used.

7.6.7 Applications

...both Python scripts and CLI C programs

The files `bblanz.c`, `bblanz.h`, and `dumpbblanz.c` use exim to support reading, dumping, and writing a BBL variant of the Analyze VW image header format.

The files `binnseg.c`, `conseg.c`, `imstack.c`, `inleav2.c`, `lovar.c`, `rowcol.c`, `rpsamp.c`, `skmiv.c`, `swab.c`, `usb2uc.c`, `vibihist.c`, and `vihist.c` implement command-line programs for particular useful image processing functions. Most or all of these are relicts of the the command-line days prior to Pyvox and will eventually be replaced by Python scripts.

The file `qdv.c` implements an interactive image viewer.

7.6.8 Examples

The directory `examples` contains a few Python scripts for image processing; these are intended more as examples (or particular functions that we needed at BBL) than as finished user applications.

7.6.9 Test Scripts

The `test` directory contains various scripts and programs for testing BBLimage; most of these are intended as regression tests to verify that BBLimage works correctly rather than as diagnostic tests to determine the precise location of a bug.

7.7 Make Targets

This section summarizes the make targets that are useful for the developer; see also the same-named section in the Installation chapter for additional targets useful for users and installers.

The `realclean` target deletes everything that can be regenerated from other source files in the distribution. You will need f2c, ps2pdf, autoconf, and m4 installed to regenerate the deleted files, so don't do this unless you really mean it.

The `tags` target will regenerate the TAGS file used by emacs to rapidly find the definitions of given names. If you prefer vi, make the obvious changes to Makefile or Makefile.in.

The `loc` runs a program to count the total number of lines of code in BBLimage, not including code borrowed from other sources (e.g. LAPACK). This requires the `loc` program, which can be obtained from the same place you got BBLimage itself.

The `f2c` target regenerates C code for the lightweight LAPACK/BLAS implementation from the original Fortran implementation. You will need to have the `f2c` program installed for this to work.

There are additional targets to make specific subsets of BBLimage; see the Makefile for details.

7.8 Coding Style

7.8.1 Rationale

The Open Source movement has removed some of the legal and social barriers to the widespread distribution and reuse of source code but it has not directly addressed the problem of ensuring that the available source code is *worth* finding, understanding, and reusing. The following style rules used at BBL are an attempt to make it as easy as possible for a prospective code recycler to understand and evaluate the code that we write, and to provide as much portability as possible, without imposing an unreasonable burden on the author. None of these rules are dogma, but they are a good starting point; you probably shouldn't violate them without a pretty good reason. On the other hand, the spirit of the rule is almost always more important than the letter, except in a couple of areas where diversity seems to create too much confusion.

For a (tongue-in-cheek) contrarian view, see the "Old Regime" section below.

These rules are definitely C- and Unix-centric, because those are the language and operating system that I use by choice. Feel free to modify for your own preferences.

It is perhaps worth noting that these rules are *not* armchair theorizing; they are the rules that I actually follow (most of the time) when writing code that others may see or that I expect to be still using myself a year or two from now. I thus have a real incentive to make the rules as simple as possible, consistent with communicating what the reader needs to know.

...audiences for reuse: use program as black box; fix bugs in the black box; use program as initial approximation to the desired program; use selected functions; study algorithms and style;

...uses: black box; baseline; component store; education; bug fixing; good (or bad) example;

...components: whole program; major modules; functions; data structures; algorithms; documentation

7.8.2 The Rules

• The package should be distributed as a tar file which is named in the format `bblimage-1.0.src.tgz`, containing the name of the package, the

version number, the fact that it is source, and the file format (gzipped tar file). It should unpack into a directory `bblimage-1.0`, using both the name and version number.

• The distribution package should contain README, INSTALL, and NEWS text files giving a introduction to the package, installation instructions, and notes on recent changes. In the installation instructions are short, they may be included in the README file.

• A man page or similar documentation should be included for each independent program or important file format. Alternatively, a full-fledged reference or user's manual may be done in TeX.

• Gnu autoconf should be used to automatically configure the programs to the user's system. There should be a Makefile (or Makefile.in) with at least targets all and install.

• The source code should be POSIX-compatible wherever possible. OS-specific coding should not be used, unless there is no other way to get the job done; if unavoidable, it should be wrapped in appropriate ifdefs or bundled into architecture-specific files.

• Standard (ISO) C should be used wherever possible. Any exceptions should be commented and justified.

• You may assume IEEE 754 floating point and two's complement integers if you need to; it would be nice to comment these for the benefit of the poor sod that doesn't have a nice computer.

• Library functions that might be useable in other, possibly non-interactive programs should be kept in separate source files and should not depend on XView or other GUI functions.

• Each source file should begin with a banner comment similar to the example below that gives the name of the file and briefly describes its contents– enough that a reader can quickly decide if *this* file is likely to contain the bug he's currently trying to track down, or the algorithm that he wants to study and copy. The first line, as illustrated, should give the name of the file and a one-line description; the exact format shown should be used, so that the one-line description can be automatically extracted for a table of contents. The same format can be used for a major block section within a source file, such as a group of functions for handling linked lists.

```
/************************************************************
defenst.c - Defenestrate a randomly chosen programmer
```

```
Author:  A. L. Fanatic

This program examines /etc/group to obtain a list of users
belonging to the prog group, randomly selects one user from
that list, remotely examines any desktop machines owned by
the user, and forcibly replaces MS Windows by Linux.  See the
man page for the complete gory details.
**********************************************************/
```

• The banner comment for a source file containing the main program for a standalone program should also describe in user-oriented terms what the program does and what arguments it takes; or it should refer to the man page or other documentation that provides this information. If it's necessary to read the program source to decide how to use it (or what it does), you need more comments.

• Each function, or tightly connected group of functions should have a banner comment in the form given below, which is slightly less emphatic. It should begin with a one-line function name and description, and describe the purpose and calling sequence of the function. Information contained in the comments describing each argument need not be repeated.

```
/*------------------------------------------------------------
fat2ext2fs - Remotely convert FAT file system to 2nd extended fs

Given the IP address of a Windows system, this function converts
each FAT file system on that computer to a Linux 2nd extended file
system. The names and contents of the files are unchanged; the
ownership and permissions are set according to the parameters
described below.
------------------------------------------------------------*/
```

• The format for the banner comments should be followed to the letter, to make it easier for automatic collection and indexing of those one-line descriptions. (We'll get around to writing that automatic indexing program real soon now.)

• The type declaration and argments for each function are also written in a stylized format, to convey as much information as possible with the least programming effort. The ideal is that a programmer who reads the banner

description and argument descriptions can successully call the function in question without having to examine any of the code.

```
int                         /* 1 => success; 0 => failure */
fat2ext2fs(
    unsigned char ip[4],    /* IP address of remote system */
    char *passwd)           /* Adminstrator's password */
{
    ...definition of the function...
}
```

• The function name and the beginning and ending braces should be flush left, again to facilitate automatic processing; no other braces should be flush left, to avoid confusing our hypothetical automatic processor.

• Any of the fifteen standard indentation styles is acceptable, provided that at least three spaces are used per level. But please try not to switch styles more than five times per page.

• Use the string FIXME to mark known bugs and other potential problems that you are brushing under the rug. If you're really brave, add your initials so we know who to blame.

• If you're making a possibly dangerous change that might break something, include an explanatory comment, your initials, and the date; this might make life much easier for the poor sod that has to figure out what broke. This is *in addition* to whatever comments you provided to the source control system; if you insist on living dangerously, please leave some conspicuous cues behind for those of us that might have to clean up after you.

• The type of each function should always be explicitly declared, even if int or void.

• Function prototypes should always be used, and placed in header files for functions used outside a single file. Functions used only in a single file should be declared static.

• The code should compile without warnings under

```
gcc -Wall -Wmissing-prototypes
```

• You may assume that the reader understands C and common algorithms, but not that he can read your mind to determine what variable names, data structures, etc. mean.

• Dividing a function definition into paragraphs headed by an brief explanatory comment can be very helpful to the reader.

• An explanatory comment should be attached to any variable declaration, except for temporary variables of obvious meaning.

• If you're not sure whether it's obvious, it's not!

• More than a few comments tacked onto the ends of executable source lines usually means that you need to rethink your algorithm, or that you don't trust the reader to understand C.

• White space (i.e. blank lines) is a useful way to visually break up your code into meaningful blocks without being heavy-handed. Putting several blank lines between function definitions makes it far easier to tell where one ends and the next begins.

• A textbook example of a well-written (if useless) function is given below.

```
/------------------------------------------------------------
area_in_common - Compute intersection of many bounding boxes

Given a linked list of bounding boxes, this function counts
the number of boxes on the list and computes the area of
their intersection.
------------------------------------------------------------*/

struct bounding_box_rec {
   struct bounding_box_rec *next;     /* Pointer to next record, or zero */
   double left, right, top, bottom;   /* Boundaries of the box; */
   } ;                                /*    origin is to the top and left */

double
area_in_common(
   struct bounding_box_rec *list,     /* Head of the linked list */
   double *area)                      /* Area of intersection */
{
   int count;                         /* Number of boxes on list */
   double left, right, top, bottom;   /* Limits of intersection so far */

   /* Initialize before walking over the list */
   /* FIXME:  This function will break if the list is empty. */
   count = 1;
```

```
   left   = list->left;
   right  = list->right;
   top    = list->top;
   bottom = list->bottom;

   /* Walk the list, keeping track of intersection */
   for (list = list->next; list != NULL; list = list->next)  {
      count++;
      if (left   < list->left)    left   = list->left;
      if (right  > list->right)   right  = list->right;
      if (top    < list->top)     top    = list->top;
      if (bottom > list->bottom)  bottom = list->bottom;  }

   /* Compute the area, which might be zero */
   if (left > right || top > bottom)
      *area = 0.0;
   else
      *area = (right - left) * (bottom - top);

   return count;
}
```

7.8.3 The Old Regime

• Comments are for wimps; I can keep all the documentation in my head, since no one else will ever need to read or modify the program.

• Anyone that needs to learn how to use the program can learn from me directly.

• Anyone that needs to know what the file formats are can read the source code and figure out how the input/output routines work.

• No one will ever want to use any feature of this program without going through the GUI.

• No one will ever want to port this program to another platform. Even if they do, everything that they need to know is in the source code.

• It works under *my* compiler. Why should I worry about POSIX compatibility?

7.9 Coding Issues

This section contains some minor coding issues that require explanation and which don't lend themselves to documentation in the code itself.

7.9.1 Bugs in Python 1.5.2

There are a number of arguable bugs in Python 1.5.2, but since current Python development is working on version 2.x, it is pointless to report these and expect them to be repaired; so we just work around them. Upgrading to Python 2.x is the right thing to do, but there appear to be enough changes to the API that it will probably break the existing code, which we don't want to do until Python 2.x becomes the de facto standard version. Anyway, here are the "bugs":

- `foo[]` is invalid syntax, even though it's the logically consistent expression to get the value of a scalar array. We handle this by permitting and ignoring any scalar subscript.
- `math.sqrt(-1)` produces an OverflowError rather than a NaN or a DomainError. This doesn't really affect anything and is ignored.
- The C API for the `len()` function returns type int rather than long; this may not be big enough for a voxel array on a platform where int is only 16 bits. We claim to support only 32-bit platforms.
- Python does not appear to support IEEE 754 even if the underlying C implementation does. FIXME: What the heck did I mean by this?
- `/usr/local` is not on the default `sys.paths`; this doesn't affect Pyvox itself but might confuse the user.
- `x[i:j]` always tries to call the `sq_slice` method even if it doesn't exist and the `mp_getitem` method does; similarly `x[i:j] = v` always calls `sq_ass_slice` rather than `mp_setitem`. We work around this by providing the `sq_slice` and `sq_ass_slice` methods even though they are logically redundant.
- `PyNumber_Check(ob) == true` does not guarantee that ob is a built-in number type, nor that it supports all of the number functions.

7.9.2 Upcalls

The usual practice is that the Python layer calls functions and methods defined in the C layer, but it is occasionally necessary or useful for functions written in C to call functions or methods written in Python; such calls are referred to as "upcalls" since they go from the lower level to the upper.

An upcall to a Python function is done using the `PyObject_CallFunction` of the Python C API; see the function `upcall_function` in `pyvox.c` for an example. It may be necesary to map the name of the function or method into a Python object; this is done using the C API function `PyDict_GetItemString` with the Python dictionary of the class or module in which the function is defined. The dictionary of the `pyvox` module is passed down to the C level during initialization of the module calling the `set_pyvox_dict` function, which caches the dictionary in the global C variable `pyvox_dict`. Dictionaries for other classes or modules can be added as necessary.

Similarly, an upcall to a Python method uses the `PyObject_CallMethod` of the C API; see `upcall_method` in `pyvox.c` for an example. In this case, the API expects the method name as text, so it is not usually necessary to look up the name.

Variables in the Python layer can presumably be looked up by name in the appropriate dictionary to obtain a Python object, but there are as yet no actual examples of this.

7.9.3 The /usr/bin/env Hack

The location at which Python is installed depends on the platform: It is usually installed at `/usr/bin` for systems for which it is included as a standard feature (e.g. Linux) but at `/usr/local/bin` when it is not standard and is added later by the system adminstrators. To handle this variability, the first line of scripts should be set to `#! /usr/bin/env python`, which will find `python`, wherever it may be in the path and call it. The `env` is intended to make temporary changes in the environment, but may be adapted for the present purpose.

7.9.4 Inlineable Functions

The `inline` keyword, which is standard in C++ and C99 and often available as an extension in C89, causes the function definition which it prefixes to be

expanded inline rather than called when that function is invoked; this usually improves performance and can be significant for heavily used functions.

The configuration script checks whether or not the compiler accepts the `inline` keyword or some equivalent and defines a C macro in the file `config.h` appropriately. For an inlineable function defined and used within a single file, it is sufficient to qualify that function definition as `static inline`.

Handling an inlineable function used in more than one source file is trickier. If `inline` is supported, then the function definition should be qualified as `static inline` and included in each source file that needs it; if not, then the function definition must be qualified as `extern` and contained in exactly one source file. The BBLimage configuration script handles the complexity by defining two additional macros in `config.h`. The macro `HAVE_INLINE` is defined if the compiler supports `inline` or an equivalent, and undefined otherwise; it is used to control where the inlineable functions are actually defined. The macro `inlineable` is defined as either `static inline` or `extern` under the same condition and is used to qualify inlineable functions appropriately for the compiler.

7.9.5 Solaris isalpha Bug

There is a bug in the Solaris implementation of the function `isalpha` and its relatives. The ANSI standard specifies that these functions take an argument of type int; under the usual promotion rules, or under the standard prototype which specifies an argument of type int, an argument of type char should be cast to type int. Sun, however, implements these functions as a macro that does a table lookup *without* doing the cast first; the gcc compiler detects this and generates a cryptic warning message `subscript has type 'char'` when it sees a expression of the form `isalpha(c)` where c is an expression of type char. To avoid these warning messages, we use `isspace((int)c)` to include the necessary cast explicitly. This should not cause problems on other systems, but is explained here to avoid puzzling any human programmer who might be reading the code.

7.9.6 getsubopt Bug

The Gnu and Solaris C libraries both provide an implementation of `getsubopt` and the two implementations are compatible. However `gcc` under Linux fails to provide a prototype for this function in `stdlib.c` while `gcc` under Solaris

does provide a prototype. Any programs that use `getsubopt` must contain various hackery that attempt to provide a prototype only when it is needed; BBLimage does not currently contain any such programs but might in the future.

7.10 Release Checklist

The following release checklist is of little interest to anyone except the BBLimage maintainer, but this is a convenient place to store it.

1. Update the `NEWS` and `README` files as needed. Commit them to the repository.
2. Choose the new version number and update the `bblimage.ver` file.
3. Regenerate the files `doc/pyvox.pdf` and `./configure` if necessary and commit to the repository.
4. If necessary, regenerate and commit the C source files for the LAPACK lite code.
5. In a working directory, check that all the files seem to be there and are consistent with the repository. Then use them with a tag in the form `bblimage-nn-nn`, using the `tag` command.
6. Export the new release to a working directory (`~/src` is a good place for this) and name the top directory with the version number, e.g. `bblimage-nn.nn`.
7. Tar and gzip, in the form `bblimage-nn.nn.src.tgz`.
8. Try making the exported version, just as a quick check that eveything is there.
9. Put one copy of the tgz file in `mercur:/export/devel/distrib`.
10. Add to the BBL website and update the web page.
11. Email announcements as appropriate.