Pyvox Reference Manual
Release 0.71

Paul Hughett

January 23, 2006

Contents

List of Tables

Chapter 1

Introduction

1.1 Overview of Pyvox

Pyvox (formerly known as BBLimage) is a set of software tools for medical image processing, particularly skull stripping, registration, 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.

Important design criteria for 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.

Pyvox 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

Pyvox 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 proto-

type 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

Pyvox 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

Pyvox should be suitable for rapid prototyping of new algorithms and analysis protocols. To do this, it 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

Pyvox 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, Pyvox 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, and easy access should be provided to common medical imaging file formats. 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.

In addition, since some medical image formats are not well standardized, low-level read and write access should be provided to the raw header and image data itself; the user who needs such access, however, may need to work with the internals of Pyvox.

1.2.6 Open-Source Development

Pyvox 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, Pyvox is designed using a layered approach. The top layer consists of several Python extension modules for image processing, written in a combination of Python and C, and effectively creating 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-language 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

Pyvox 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.
- Fourier transforms.
- 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).
- Interactive modification of images, especially for manual correction of not-quite-correct automatically masked and segmented images.
- Automated image registration.

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

- Improved masking and segmentation algorithms, including handling of shading and partial volume effects.
- Tools for validating brain masking and segmentation algorithms.

1.5 Current Status

Pyvox 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 Pyvox. 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

Pyvox 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.

Pyvox 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

Pyvox 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 Pyvox 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 Pyvox 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 Pyvox 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 Pyvox 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 most of the numeric types provided by C. (An exception is plain char; both signed and unsigned char are supported, however.) The properties of these types are inherited 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	
fcomplex	float complex	
dcomplex	double complex	

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 type, rank, and dimensions of an array are available as the read-only attributes *array*`.type`, *array*`.rank`, and *array*`.size`; in addition, the dimensions can be changed by the function *array*`.reshape()`. Most operations on arrays require that the operands have the same type, rank, and dimensions; a plain number is usually acceptable and acts like an array of the appropriate type, rank, and dimensions. The exceptions to this rule are discussed in the descriptions of the individual operations.

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 data elements are stored sequentially, with the last index varying most rapidly. 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. As a special case, providing a plain number as a subscript will be treated as an index into the array considered as a one-dimensional vector. 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.

Origin and Spacing

The `origin` attribute of a Pyvox array gives the physical coordinates corresponding to the voxel indexed by $(0, 0, \dots)$; the `spacing` attribute gives the physical distance between successive elements on each axis. Both the `origin` and `spacing` attributes may be read and set. If any `spacing` 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 `spacing` to all ones. In most cases, all operands to an array operation must have the same `origin` and `spacing` and the result will have the same `origin` and `spacing` as the operands. Matrix operations are generally exempt from this requirement; other exceptions are discussed in the descriptions of the individual operations.

Other Array Metadata

Other metadata is stored in the attributes *array*`.metadata`, *array*`.header`, and *array*`.userdata`, each of which is a dictionary reserved for specific kinds of information. The *array*`.metadata` attribute contains information about the file from which the array was read, or to which it will be written; other sorts of information may be added in the future. In addition, the *array*`.header` attribute contains the image file header in relatively uninterpreted form and can be used (if necessary) for low-level manipulation of the format details; this should not normally be necessary. The *array*`.userdata` attribute is reserved for the user, into which he may store any information that he wishes. These three dictionaries are kept separate to avoid key clashes; otherwise, adding new keys for file information or field names for

new data formats would risk breaking existing code.

None of *array*`.metadata`, *array*`.header`, or *array*`.userdata` is checked or copied during normal array operations, although there is a specific method *array*`.metacopy()` for copying their contents from one array to another. See the appropriate entries in the reference section of this manual for additional details.

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.

A slice object is written in the form *init*:*limit*:*stride*. If *stride* is omitted or `None`, it defaults to 1; it may not be zero. If *init* is omitted or `None`, it defaults to zero if *stride* is positive and to the length of that dimension minus one if *stride* is negative. If *limit* is omitted or `None`, it defaults to the length of that dimension if *stride* is positive and to -1 if *stride* is negative. If `init` or `limit` is not defaulted and is negative, the length of that dimension is added to it. 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. Here are some examples, assuming that the length of the dimension is 10:

```
:        0, 1, 2, 3, 4, 5, 6, 7, 8, 9
::-1     9, 8, 7, 6, 5, 4, 3, 2, 1, 0
1:       1, 2, 3, 4, 5, 6, 7, 8, 9
:9       0, 1, 2, 3, 4, 5, 6, 7, 8
1:9      1, 2, 3, 4, 5, 6, 7, 8
::2      0, 2, 4, 6, 8
::-2     9, 7, 5, 3, 1
```

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 either exactly one or exactly the same as the number of elements of `A`. Using `A[...]` as an expression yields a copy of `A`, retaining the shape and contents. (NOTE that `A[...]` will become an array slice rather than a copy in some future version of Pyvox.) 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.

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 Error Management

Almost all errors detected by Pyvox are converted to Python exceptions which can be caught by the `try ... except` statement; if uncaught, they normally cause Python to terminate. The exceptions are currently of type `PythonError` but may be converted to the various standard Python exceptions in the future; the text string accompanying the exception gives a more detailed description of the error. There are a few errors—mostly assertion failures or other events beyond the control of the Pyvox user—which are fa-

tal and cause the program to abort; these are gradually being converted to exceptions.

2.3 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.

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.3.4 RGBA Images

Two-dimensional images intended for immediate display via the X Window System or other display interface are usually stored as a rank 3 unsigned char array, where the last dimension runs over the components per pixel. The possible values of the last dimension and the interpretation of the components are show in the table below. If the number of components is 1, then it is usually safe to reduce the array to rank 2.

Dimen	Components
1	Luminance (gray level)
3	Red, green, blue
4	Red, green, blue, alpha

2.3.5 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.

2.3.6 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]]`.

2.3.7 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.

2.4 Programming Idioms

The right way to do things in Pyvox is not always obvious. In this section we describe some of the less obvious programming idioms.

2.4.1 Compressed Images

Volume image files are typically large and it is useful to compress them for long-term storage. Pyvox tries to make it easy to handle compressed files transparently. If an image file name given for reading does not exist, a corresponding compressed or gzipped file will be sought, and silently uncompressed as needed. If an image file name given for writing ends in the appropriate extension, it will be automatically compressed after being written. There is, however, one situation where the transparency breaks down. If you read a file such as `foo.img` which exists as the compressed file `foo.img.Z`, reading will silently use the compressed file; but when you attempt to write an updated image file to the original name `foo.img`, it will not be compressed. The `pyvox.truefile()` function can be used to determine the true file name of a possibly compressed file, whereupon you can write back to the true file name.

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 positional arguments are written in *italic* type and followed by an equal sign and the default value. Keyword arguments are written in a `typewriter` font; if optional, they are followed by an equals sign and the default value. In either case, the default value *???* means the default value is given in the description below.

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
`reged`	Generic interactive region editor
`regis`	Image registration classes and functions
`tkphoto`	Access to Tkinter PhotoImage objects

4.1.2 Optional Features

Optional Features	
`pyvox.have_complex`	Are complex types supported?

4.1.3 Types and Classes

Types and Classes	
affine	N-dimensional affine transform
array	Multi-dimensional numeric array
ccmap	Connected components in an image
`exim.extypes`	Tuple of all the external array types
`exim.intypes`	Tuple of all the internal array types
kernel	Generalized convolution kernel
obaffine	Affine transform to be optimized
obfun	Objective function to be optimized
obrigid	Rigid transform to be optimized
poly	N-dimensional polynomial transform
tkphoto	Wrapper around a Tkinter PhotoImage object
type	Pyvox data type (internal or external)

4.1.4 Type Objects

Type Objects	
`pyvox.AffineType`	Class object for an affine transform
`pyvox.ArrayType`	Type of a Pyvox array
`pyvox.KernelType`	Type of a Pyvox kernel
`pyvox.PolyType`	Class object for a polynomial transform

4.1.5 Attributes of Pyvox Data Types

Attributes of Pyvox Data Types	
type`.code`	Numeric type code used in Pyvox C code
type`.complex`	Complex type with equal or better precision
type`.desc`	Description of the type, as a string
type`.epsilon`	Smallest $\epsilon > 0$ such that $1 + \epsilon > 1$
type`.extype`	Natural external type for an internal type
type`.intype`	Natural internal type for an external type
type`.iscomplex`	Is this a complex-valued types?
type`.isfloat`	Is this a floating-point type?
type`.isint`	Is this an integral type?
type`.isreal`	Is this a real-valued type?
type`.isunsigned`	Is this an unsigned type?
type`.name`	Name of type, as a string
type`.nbytes`	Number of bytes per element
type`.real`	Real type with equal precision

4.1.6 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
array.`writepgm()`	Write Pyvox array as a PGM image file
array.`writeraw()`	Write Pyvox array as a raw image file
`exim.dump()`	Dump struct contents from dict
`exim.pack()`	Pack dict into struct per format
`exim.unpack()`	Unpack struct into dict per format
`pyvox.rawread()`	Read a Pyvox array in raw data format
`pyvox.read()`	Read a Pyvox array in various formats
`pyvox.truefile()`	True name of a possibly compressed file

4.1.7 Array Creation

Array Creation	
`pyvox.array()`	Create an array from given data
`pyvox.column()`	Create column vector from given data
`pyvox.const()`	Deprecated alias for `pyvox.array()`
`pyvox.diag()`	Create a diagonal matrix from given data
`pyvox.fftramp()`	Create an FFT frequency ramp
`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.randi()`	Create array of random integers
`pyvox.randn()`	Create array of normal random variates
`pyvox.randu()`	Create array of uniform random variates
`pyvox.vector()`	Create rank-1 vector from given data

4.1.8 Array Attributes

Basic Array Manipulations	
array.`header`	Dictionary with image file header info
array.`metadata`	Dictionary for metadata
array.`origin`	Physical coordinates of origin
array.`rank`	Rank, or number of dimensions
array.`size`	Dimensions of an array
array.`spacing`	Physical spacing of coordinate planes
array.`type`	Type of data in an array
array.`userdata`	Dictionary reserved for the user

4.1.9 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.`metacopy()`	Copy selected metadata
array.`p2i()`	Physical-to-index transform
array.`reshape()`	Change the shape of an array
array.`tuple()`	Elements as a Python tuple
`len(`*array*`)`	Number of elements (Python bug!)

4.1.10 Type Conversions

Type Conversions	
array.`cast()`	Convert to specified type
array.`dcomplex()`	Convert to double complex
array.`double()`	Convert to double
array.`fcomplex()`	Convert to float complex
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

4.1.11 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 `%` *array*	Remainder
`~`*array*	Bitwise negation
array `&` *array*	Bitwise AND
array `|` *array*	Bitwise OR
array `^` *array*	Bitwise XOR
array `==` *array*	Equal
array `!=` *array*	Not equal
array `<` *array*	Strictly less than
array `<=` *array*	Less than or equal
array `>` *array*	Strictly greater than
array `>=` *array*	Greater than or equal
array`.abs()`	Absolute value
`abs(`*array*`)`	Absolute value (deprecated)
array`.carg()`	Complex argument
array`.ceil()`	Ceiling
array`.cmove()`	Conditional move
array`.compare()`	Comparison of two arrays
array`.conj()`	Complex conjugate
array`.floor()`	Floor
array`.fma()`	Fused multiply-add $x * y + z$
array`.imag()`	Imaginary part
array`.max()`	Maximum of two arrays
array`.min()`	Minimum of two arrays
array`.real()`	Real part

4.1.12 Special Functions

Special Functions	
array.`acos()`	Arc cosine
array.`asin()`	Arc sine
array.`atan()`	Arc tangent
array.`atan2()`	Two-argument arc tangent
array.`cbrt()`	Cube root
array.`cos()`	Cosine
array.`cosh()`	Hyperbolic cosine
array.`erf()`	Error function
array.`erfc()`	Complementary error function
array.`exp()`	Exponential
array.`exp2()`	Exponential to base 2
array.`expm1()`	Exponential $e^x - 1$
array.`hypot()`	Hypotenuse $\sqrt{x^2 + y^2}$
array.`lgamma()`	Log of gamma function
array.`log()`	Natural logarithm
array.`log10()`	Logarithm to the base 10
array.`log1p()`	Logarithm $\log(1 + x)$
array.`log2()`	Logarithm to the base 2
array.`pow()`	Power
array.`sin()`	Sine
array.`sinh()`	Hyperbolic sine
array.`sqrt()`	Square root
array.`tan()`	Tangent
array.`tanh()`	Hyperbolic tangent
array.`tgamma()`	Gamma function

4.1.13 Other Elementwise Operations

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

4.1.14 Array Reduction Operations

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

4.1.15 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.correl()`	Weighted correlation of two images
`regis.info()`	Information content of an image
`regis.mutinfo()`	Mutual information of two images

4.1.16 Matrix and Vector Operations

Matrix and Vector Operations	
array.`cholesky()`	Cholesky decomposition
array.`cross()`	Vector cross product in 3-d
array.`det()`	Determinant of a square matrix
array.`diag()`	Extract diagonal elements of a matrix
array.`eigsy()`	Solution of symmetric/Hermitian eigensystem
array.`H()`	Complex conjugate (Hermitian) transpose
array.`inverse()`	Matrix inverse
array.`mmul()`	Matrix multiplication
array.`prinaxes()`	Principal axes transformation
array.`solve()`	Solve the linear system $AX = B$
array.`svd()`	Singular value decomposition
array.`T()`	Transpose of a matrix
array.`trans()`	Transpose of a matrix

4.1.17 Neighborhood Operations

Neighborhood Operations	
array.`convolve()`	Convolution with optional subsampling
array.`dilabel()`	Dilation of a label image
array.`dilate()`	Dilation of a binary image
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
`pyvox.lowpass()`	Create a standard lowpass kernel

4.1.18 Fourier and Other Transforms

Fourier and Other Transforms	
array.`fft()`	Fast Fourier transform
array.`ifft()`	Inverse fast Fourier transform
`pyvox.fftramp()`	Create an FFT frequency ramp

4.1.19 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.`minmax()`	Minimum/maximum values within array
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.correl()`	Weighted correlation of two images
`regis.info()`	Information content of an image
`regis.mutinfo()`	Mutual information of two images

4.1.20 Voxel Classification

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

4.1.21 Connected Components

Connected Components	
array.`bigob()`	Extract largest object in image
ccmap.`bigob()`	Largest connected component
ccmap.`count()`	Number of pixels in a connected component
ccmap.`map()`	Connected component each pixel belongs to
ccmap.`regions()`	List of connected components found
ccmap.`seed()`	Seed point in a connected component
`pyvox.ccmap()`	Find connected components of an image

4.1.22 Other Image Operations

Other Image Operations	
array.`chamfer()`	Compute chamfer distance transform

4.1.23 Affine Transforms

Affine Transforms	
affine`[]`	Get coefficient of transform
affine`[] =` *value*	Set coefficient of transform
affine `+` *other*	Coefficientwise sum of transforms
affine `-` *other*	Coefficientwise difference of 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`.copy()`	Deep copy
affine`.i2p()`	Compose with index-to-physical transform
affine`.inverse()`	Inverse
affine`.invert()`	Invert in place
affine`.linear()`	Resample image with linear interpolation
affine`.map()`	Coordinate map
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

4.1.24 Polynomial Transforms

Polynomial Transforms	
poly `+` *other*	Coefficientwise sum
poly `-` *other*	Coefficientwise difference
poly `*` *number*	Scalar multiplication
poly`.addparam()`	Add vector of parameters to the transform
poly`.compose()`	Compose with an affine transform
poly`.copy()`	Deep copy
poly`.init()`	Initialize coefficients of transform
poly`.linear()`	Resample image with linear interpolation
poly`.map()`	Coordinate map
poly`.norm()`	Norm of a polynomial transform
poly`.param()`	Transform as a vector of parameters
poly`.point()`	Transform a point
poly`.scale()`	Compose with a scale transform
poly`.setparam()`	Set transform from a vector of parameters
poly`.translate()`	Compose with a translation
poly`.truncate()`	Truncate to specified maximum degree
`pyvox.poly()`	Create an polynomial transformation

4.1.25 Interpolation and Resampling

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

4.1.26 Image Registration

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

4.1.27 Optimization

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

4.1.28 Graphics, Drawing, and Display

Graphics, Drawing and Display	
array`.fill2d()`	Fill a 2D contour with a value
array`.rgba2d()`	Extract 2D RGBA slice from raw data
`reged.reged`	Generic interactive region editor
`tkphoto.tkphoto()`	Construct and wrap a Tkinter PhotoImage
tkphoto`.getimage()`	Extract contents of a Tkinter PhotoImage
tkphoto`.handle`	Get Tk handle of a Tkinter PhotoImage
tkphoto`.name`	Get Tk name of a Tkinter PhotoImage
tkphoto`.putimage()`	Copy array into a Tkinter PhotoImage
tkphoto`.size`	Get dimensions of a Tkinter PhotoImage
tkphoto`.tkphoto`	Get Tkinter PhotoImage from its wrapper

4.1.29 Pyvox Development and Debugging

Pyvox Development and Debugging	
`pyvox.error_action()`	Set action to take if an error occurs
`pyvox.refcnt()`	Reference count of a Python object

4.2 Full Descriptions

---

affine [class instance]

An *affine* class instance represents a N-dimensional affine transform and is typically created by calling `pyvox.affine`.

---

affine`[`*row, col*`]`

Returns the numeric value of the specified entry of the $n \times (n+1)$ element array defining the affine transform, where n is the number of dimensions. The *row* ranges from 0 to $n-1$. The *col* ranges from 0 to n, where $0, \ldots, n-1$ are the rotation matrix elements, and n is the offset.

---

affine`[`*row, col*`] =` *value*

Replaces the specified element of the $n \times (n+1)$ element array which specifies the affine transform, where n is the number of dimensions. The *row* ranges from 0 to $n-1$. The *col* ranges from 0 to n, where $0, \ldots, n-1$ are the rotation matrix elements, and n is the offset.

---

affine `+` *other*

Returns a new transform containing the coefficientwise sum of the transforms *affine* and *other*. If *other* is an affine transform, then the result is an affine transform; if *other* is a poly transform, then the result is a poly transform.

---

affine `-` *other*

Returns a new transform containing the coefficientwise difference of the transforms *affine* and *other*. If *other* is an affine transform, then the result

is an affine transform; if *other* is a poly transform, then the result is a poly transform.

---

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=1, offset=0, pre=0*`)`

Updates the affine transform *affine* by composing it with another affine transform given by *matrix* and *offset*; returns the updated transform. If *matrix* is a scalar, it is converted to a diagonal matrix with that value on the diagonal; in particular, providing `1` yields an identity matrix. If *offset* is a scalar, it is interpreted as a column vector containing that value at each element; in particular, providing `0` yields a zero offset. The value of *pre* determines the order in which the two transforms are composed; see *affine*`.compose()` for the permitted values.

---

affine.`copy()`

Returns a new affine transform containing the contents and attributes of the original *affine* transform but which shares no storage with it.

---

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. The `origin` and `spacing` of the result are set to the results of transforming the `origin` and `spacing` of the *image*; note that this may or may not be appropriate.

(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.`map`(*dimen*)

Returns a new Pyvox array of type double which contains the output coordinates of the transform *affine* computed from index coordinates over a notional source image of dimensions *dimen*. The rank of the result is one higher than the number of dimensions in *affine*; the (new) last dimension ranges over the number of dimensions in *affine* and contains the output coordinates in order.

---

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. The `origin` and `spacing` of the result are set to the results of transforming the `origin` and `spacing` of the *image*; note that this may or may not be appropriate.

---

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 mathematically 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 [type]

An *array* object represents an N-dimensional numeric array and is typically created by calling `pyvox.array` or `pyvox.ramp`. As special cases, it can also represent a matrix or coordinate point. A matrix is typically created by calling `pyvox.column`, `pyvox.diag`, `pyvox.matrix`, or `pyvox.vector`. A point is typically created by calling `pyvox.point`.

---

array`[`*sub0, sub1, ...*`]`

Returns the numeric value of an element if all of the subscripts are numbers, and an array slice for any other valid combination of subscripts. Each subscript may be a integer value, a non-empty list of integers, or a slice object; the number of subscript items must match the rank of *array*. Alternatively, the ellipsis `...` may be used once in a subscript list to mean all the unnamed dimensions; then the number of subscript items must be strictly less than the rank. As a special case, a single number may be provided as subscript to an array of any rank and treats the array as if it were a one-dimensional vector;

the order of elements is taken with the last (regular) subscript varying most rapidly. As another special case, any number of subscripts may be provided to a scalar (rank 0) array and will be ignored; this is to work around a Python bug which prohibits empty subscript lists.

Negative integers from -1 through $-N$, where N is the number of elements along the appropriate dimension, may be used in all subscript items and are mapped to $N-1$ through 0, selecting the last to first elements. Subscripts less than $-N$ or greater than $N-1$ are invalid and will generate an error. A single negative number as the subscript is interpreted the same way, except that N is the total number of elements in the array.

Using a list of integers as a subscript selects the array elements indexed by the elements of the list, along the appropriate dimension of the array.

Using a slice object *start*`:`*limit*`:`*step* as a subscript selects the elements indexed by *start*, *start* + *step*, ... up (or down) to but not including *limit*, along the corresponding dimension. The *step* defaults to 1. If the *step* is positive, the *start* and *limit* default to 0 and $N-1$, where N is the number of elements along the dimension; if the *step* is negative, the *start* and *limit* default to -1 and $-N-1$; a zero *step* is an error. If only a single colon is provided, it is assumed to separate the *start* and *limit*. Any element of the slice object may evaluate to `None`, in which case the appropriate default is used.

The rank of the array slice returned is equal to the rank of *array*, minus the number of subscripts given as integers rather than lists, slices, or ellipses; an array slice of rank 0 decays into a number.

FIXME: The current implementation returns a new array of the same rank as *array* rather than an array slice of reduced rank; this will be fixed in some future version.

FIXME: This does not provide a way to extract a element as a scalar array; this is probably no serious loss, but it might be useful to define some alternate way to get a scalar array.

---

array`[`*sub0, sub1, ...*`]` `=` *expr*

Replaces the elements of *array* indexed by the subscripts with the corresponding elements of *expr*, which must be an array, an array slice, or a numeric value. See *array*`[]` for the permissable subscript items and their interpretation. If *expr* is an array or array slice, the number of elements it

contains must be the same as the number of selected elements in *array*; the shape is not, however, required to match. If *expr* is a numeric value, all the selected *array* elements are replaced by that value. The `origin` and `spacing` of the array are unchanged.

`+`*array*

Returns *array*. Provided for symmetry with `-`*array*.

---

array `+` *array1*

Returns a new array containing the elementwise sum of *array* and *array1*. Either *array* or *array1* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array1* must be the same.

`-`*array*

Returns a new array containing the elementwise additive inverse of *array*.

---

array `-` *array1*

Returns a new array containing the elementwise difference of *array* and *array1*. Either *array* or *array1* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array1* must be the same.

---

array `*` *array1*

Returns a new array containing the elementwise product of *array* and *array1*. Either *array* or *array1* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array1* must be the same.

See also *array*.`mmul(`*array1*`)` for the matrix product.

---

array `/` *array*

Returns a new array containing the elementwise quotient of *array* and *array1*. Either *array* or *array1* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array1* must be the same.

---

array `%` *array1*

Returns a new array containing the elementwise remainder of *array* and *array1*. The remainder is defined as $x - \text{trunc}(x/y) * y$, where trunc denotes rounding toward zero. This operation is valid for both integral and floating-point types.

`~`*array*

Returns a new array each element of which is the bitwise negation of the corresponding element in *array*. Valid for unsigned integral types only.

---

array `&` *array*

Returns a new array each element of which is the bitwise AND of the corresponding elements in *array* and *array1*. Either *array* or *array1* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array1* must be the same. Valid for unsigned integral types only.

---

array `|` *array*

Returns a new array each element of which is the bitwise OR of the corresponding elements in *array* and *array1*. Either *array* or *array1* may

be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array1* must be the same. Valid for unsigned integral types only.

---

array `^` *array*

Returns a new array each element of which is the bitwise XOR of the corresponding elements in *array* and *array1*. Either *array* or *array1* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array1* must be the same. Valid for unsigned integral types only.

---

array `==` *array2*

Returns a new unsigned char array each element of which is 1 if the corresponding elements in *array* and *array2* are equal, and 0 otherwise. Either *array* or *array2* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array2* must be the same.

---

array `!=` *array2*

Returns a new unsigned char array each element of which is 1 if the corresponding elements in *array* and *array2* are unequal, and 0 otherwise. Either *array* or *array2* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array2* must be the same.

---

array `<` *array2*

Returns a new unsigned char array each element of which is 1 if the corresponding element in *array* is strictly less than the corresponding element in *array2*, and 0 otherwise. Either *array* or *array2* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type

of *array*. Otherwise, the types of *array* and *array2* must be the same. Not defined for complex types.

---

array `<=` *array2*

Returns a new unsigned char array each element of which is 1 if the corresponding element in *array* is less than or equal to the corresponding element in *array2*, and 0 otherwise. Either *array* or *array2* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array2* must be the same. Not defined for complex types.

---

array `>` *array2*

Returns a new unsigned char array each element of which is 1 if the corresponding element in *array* is strictly greater than the corresponding element in *array2*, and 0 otherwise. Either *array* or *array2* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array2* must be the same. Not defined for complex types.

---

array `>=` *array2*

Returns a new unsigned char array each element of which is 1 if the corresponding element in *array* is greater than or equal to the corresponding element in *array2*, and 0 otherwise. Either *array* or *array2* may be a numeric value rather than a Pyvox array; in this case, the number is coerced to the type of *array*. Otherwise, the types of *array* and *array2* must be the same. Not defined for complex types.

---

array`.abs()`

Returns a new Pyvox array containing the elementwise absolute value of the *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 floating-point types only.

---

array`.amax(`*axes=All*`)`

This function computes the maximum of the *array* elements along the specified *axes* (defaulting to all axes) and returns either a scalar or an array of reduced rank. 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 copy the array. The values listed in *axes* may be in any order, and duplicates are permitted. The returned result is the same type as *array*.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

---

array`.amin(`*axes=All*`)`

This function computes the minimum of the *array* elements along the specified *axes* (defaulting to all axes) and returns either a scalar or an array of reduced rank. 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 copy the array. The values listed in *axes* may be in any order, and duplicates are permitted. The returned result is the same type as *array*.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

---

array`.aprod(`*axes=All*`)`

This function computes the product of the *array* elements along the specified *axes* (defaulting to all axes) and returns either a scalar or an array of reduced rank. 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 copy the array. The values listed in *axes* may be in any order, and duplicates are permitted. The returned result is the same type as *array*.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

---

array`.asin()`

Returns a new Pyvox array containing the elementwise arc sine of the original array. The returned value is in radians. Valid for floating-point types only.

---

array`.asum(`*axes=All*`)`

This function computes the sum of the *array* elements along the specified *axes* (defaulting to all axes) and returns either a scalar or an array of reduced rank. 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 copy the array. The values listed in *axes* may be in any order, and duplicates are permitted. The returned result is the same type as *array*.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

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 floating-point types 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.

This function has been deprecated; use `pyvox.ccmap(`*image*`).bigob()` in new code, or use other features of the *ccmap* class.

---

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`.carg()`

Returns a new Pyvox array containing the elementwise complex argument (that is, the polar angle) of the original array. Valid for complex types only.

---

array`.cast(`*type)*

Returns a new Pyvox array containing the contents of the original array converted to the specified internal type.

---

array`.cbrt()`

Returns a new Pyvox array of the same type and shape as *array* and which contains the cube root of each corresponding element in *array*. Valid for real floating-point types only, and provided only if the underlying C implementation supports it.

---

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. Not supported for complex types.

---

array`.cholesky()`

Returns a new Pyvox array of the same type as *array* which contains the Cholesky factor of the symmetric or Hermitian positive definite matrix *array*. That is, it returns the lower triangular matrix L such that $array = LL^T$ or LL^H as appropriate. Valid for floating point types only.

---

array`.cmove(`*select*`, `*source*`)`

Modifies *array* in place by replacing each element by the corresponding element of *source* whenever the corresponding element of *select* is non-zero; any elements of *array* corresponding to zero elements of *select* are unchanged. Returns the (modified) *array*. The *select* argument must be unsigned char; the *array* and *source* may be any type but must be the same type. All three arrays must have the same rank, dimensions, origin, and spacing, except that *source* may be a plain number.

---

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. Not defined for complex types.

---

array.`conj()`

Returns a new Pyvox array containing the elementwise complex conjugate of the original array. Valid for complex types only. See also *array*.`H()` for the complex conjugate (Hermitian) transpose of a matrix.

---

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 or double complex and converted back to the original image type; if the original type cannot represent the values, the nearest representable value is used instead. Convolving a real image by a complex kernel is not permitted.

---

array.`copy()`

Returns a new Pyvox array containing the contents and attributes of the original array but which shares no storage with it, except that only shallow copies are made of the `header`, `metadata`, and `userdata` dictionaries.

---

array.`cos()`

Returns a new Pyvox array containing the elementwise cosine of the original array. The input is in radians. Valid for floating-point types only.

---

array.`cosh()`

Returns a new Pyvox array containing the elementwise hyperbolic cosine of the original array. Valid for floating-point types only.

---

array`.count()`

Returns the number of elements in *array*.

---

array`.cross(`*other*`)`

Returns the vector cross product of *array* and *other*. The operands need not be any particular type and shape, but each must contain exactly three elements. The type and shape of the result are taken from *array*. Not defined for complex types.

---

array`.cubic(`*map, type=array*`.type)`

Returns a new array whose elements are computed by cubic Lagrange interpolation in the voxels of the source *array* at the sample points defined by *map*. The type of *map* must be double, and its last dimension must match the rank of *array*. The *type* argument specifies the desired type of the result; it defaults to the type of *array*. The shape of the result is the shape of *map* with the last dimension omitted. Voxels outside *array* are assumed to be zero.

As a special case, the value of *map* may be a tuple, list, or vector of length equal to the rank of *array*; in this case, the result is a plain number rather than an array. If *array* is rank one, and the last dimension of *map* is not one, then a final dimension of length one is effectively added to *map*.

---

array`.dcomplex(`*imag=0*`)`

Returns a new Pyvox array containing the contents of the original array converted to double complex. The *imag* argument, if present, must be a Pyvox array of the same type and shape as *array* and supplies the imaginary part of the result.

---

array`.det()`

Returns the determinant of a square matrix represented as a Python array. Valid for floating-point types 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`.dilabel(`*kernel=???*`)`

Returns a new Pyvox array containing the morphological dilation of a label image *array*, which must have non-zero rank and any unsigned integral type. Each pixel of the result is the smallest nonzero pixel value within the neighborhood defined by the kernel; if all pixels in the neighborhood are zero, then the result is zero. 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.

A common use of this method is to dilate an image in which nonzero pixel values label different regions and a zero pixel value denotes background. This method will dilate the nonbackground regions while preserving the region labels, taking the lower-numbered region in case of ties.

See also *array*`.dilate()`.

---

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

Returns a new Pyvox array containing the morphological dilation of a binary image *array*, which must have non-zero rank and any unsigned integral type. Each pixel of the result is the bitwise OR of the pixel values in the neighborhood defined by the kernel. 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.

See also *array*`.dilabel()`.

---

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

Returns the vector dot product of *array* and *other* along the specified *axes*, with optional pixelwise weights provided by *weight*; the dot product $\langle x, y \rangle_w$ of vectors x and y with weights w is defined as

$$\sum_n w_n \, x_n^* \, y_n \tag{4.2}$$

where n is an index over the elements, and * indicates the complex conjugate. The *array*, *other*, and *weight* arguments may be of any type and are converted to double or double complex 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*. The *weight* argument may be `None`, in which case an unweighted dot product is computed.

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`. The values listed in *axes* may be in any order, and duplicates are permitted.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

---

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 or a complex Hermitian matrix; the results are undefined if the Pyvox array is not actually symmetric or Hermitian. The result `val` is a list of the eigenvalues. The result `vec` is an orthogonal or unitary 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`.erf()`

Returns a new Pyvox array of the same type and shape as *array* and which contains the error function $\mathrm{erf}(x)$ for each element x of *array*. The error function is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \mathrm{e}^{-t^2} \mathrm{d}t \tag{4.3}$$

and is described in most books on special functions. This function is valid for real floating-point types only, and is provided only if the underlying C implemention supports it.

See also *array*`.erfc()`.

---

array`.erfc()`

Returns a new Pyvox array of the same type and shape as *array* and containing the complementary error function $\mathrm{erfc}(x)$ for each element x of *array*. The error function is defined as

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty \mathrm{e}^{-t^2} \mathrm{d}t \tag{4.4}$$

and is described in most books on special functions. This function is valid for real floating-point types only, and is provided only if the underlying C implemention supports it.

See also *array*`.erfc()`.

---

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 of the same type and shape as *array* and containing the value e^x for each element x of *array*. Valid for floating-point types only.

---

array`.exp2()`

Returns a new Pyvox array of the same type and shape as *array* and containing the value 2^x for each element x of *array*. Valid for real floating-point types only and provided only if the underlying C implementation supports it.

---

array`.expm1()`

Returns a new Pyvox array of the same type and shape as *array* and containing the value $e^x - 1$ for each element x of *array*. Valid for real floating-point types only and provided only if the underlying C implementation supports it.

---

array`.fcomplex(`*imag=0*`)`

Returns a new Pyvox array containing the contents of the original array converted to float complex. The *imag* argument, if present, must be a Pyvox array of the same type and shape as *array* and supplies the imaginary part of the result.

---

array.`fft(`*axes=All)*

Returns a new Pyvox array containing the fast Fourier transform of the original *array*, transformed only along the specified *axes*. The original *array* may be of any type; the result is always float or double complex. Each axis on which the transform is to be computed must be a power of 2. The definition of the discrete Fourier transform is not entirely standard; the definition used in this software is, in one dimension,

$$F(k) = \frac{1}{N} \sum_{n=0}^{N-1} f(n)\, \mathrm{e}^{-j2\pi kn/N} \quad . \tag{4.5}$$

An advantage of this choice is that each spectral coefficient $F(k)$ can be read directly as the amplitude of the corresponding complex exponential component $\mathrm{e}^{j2\pi kn/N}$.

---

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 real floating-point types only.

---

array.`fma(`*second, third*`)`

Returns a new Pyvox array containing the elementwise evaluation of $x * y + z$ for the three arrays *array*, *second*, and *third*. This operation is known as a fused multiply-add and is faster and more precise than the equivalent multiply and add operations on most platforms. Valid for floating-point types only.

---

array.`H()`

Returns a new Pyvox array containing the complex conjugate (Hermitian) transpose of the original array. Valid for complex types only.

---

array.`header`

This read/write attribute of a Pyvox array is either a list or a dictionary which is reserved for the contents of the image file header associated with the array. It may either be set by the software which actually reads the array data, or by the user to specify the format and contents of the header file to be written. If never specified, it defaults to an empty list or dictionary. The elements of the list or keys into the dictionary are determined by the particular image file format used.

---

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

Returns a new Pyvox array containing the univariate histogram of *array*, which must be an unsigned char or unsigned short 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.

A useful trick to know is that, if *array* is a histogram, then *array*`.moments()` will compute the total counts, mean, and variance of the index variable of the histogram.

---

array`.hypot(`*other*`)`

Returns a new Pyvox array with the same type and shape as *array* and *other* and which contains the value $\sqrt{x^2 + y^2}$ for the corresponding elements x in *array* and y in *other*. Valid for real floating-point types only, and provided only if the underlying C implementation supports it.

---

array`.i2p()`

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

---

array`.ifft(`*axes=All)*

Returns a new Pyvox array containing the inverse fast Fourier transform of the original *array*, transformed only along the specified *axes*. The original *array* may be of any type; the result is always float or double complex. Each axis on which the transform is to be computed must be a power of 2. The definition of the inverse discrete Fourier transform is not entirely standard; the definition used in this software is, in one dimension,

$$f(n) = \sum_{n=0}^{N-1} F(k)\, \mathrm{e}^{j2\pi kn/N} \quad . \tag{4.6}$$

Note that each spectral coefficient $F(k)$ can be read directly as the amplitude of the corresponding complex exponential component $\mathrm{e}^{j2\pi kn/N}$.

---

array.`imag()`

Returns a new Pyvox array containing the elementwise imaginary part of the original array. Valid for complex types only.

---

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.`kmeans2(`*cent1*`,` *cent2*`)`

Returns a tuple (*newcent1*, *newcent2*) containing two lists of class centroids computed by the K-means classification algorithm, where *array* is the bivariate histogram of two unsigned char images (and must have exactly 256×256 bins), and *cent1* and *cent2* are two lists of initial guesses at the

class centroids. The centroids thus computed can be used later by the *array*`.nnclass2` 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`.lgamma()`

Returns a new Pyvox array with the same type and shape as *array* and containing the value $\log \Gamma(x)$ for each element x in *array*. The function $\Gamma(x)$ is defined by

$$\Gamma(x) = \int_0^\infty \mathrm{e}^{-t} t^{x-1} \, \mathrm{d}t \tag{4.7}$$

Valid for real floating-point types only, and provided only if the underlying C implementation supports it.

See also *array*`.tgamma()`.

---

array`.linear(`*map, type=array*`.type)`

Returns a new array whose elements are computed by linear interpolation in the voxels of a source *array* at the sample points defined by *map*. The type of *map* must be double, and its last dimension must match the rank of *array*. The *type* argument specifies the desired type of the result; it defaults to the type of *array*. The shape of the result is the shape of *map* with the last dimension omitted. Voxels outside *array* are assumed to be zero.

As a special case, the value of *map* may be a tuple, list, or vector of length equal to the rank of *array*; in this case, the result is a plain number rather than an array. If *array* is rank one, and the last dimension of *map* is not one, then a final dimension of length one is effectively added to *map*.

See also *affine*`.linear()` and *poly*`.linear()` for linear interpolation of an array subject to an affine or polynomial transform; using either of these functions, when appropriate, is faster than computing the map and using *array*`.linear()` to interpolate.

Prior to 2005-07-18, the type of the result was always double.

---

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 floating-point types only.

---

array.`log10()`

Returns a new Pyvox array containing the elementwise base-10 logarithm of the original array. Valid for real floating-point types only.

---

array.`log1p()`

Returns a new Pyvox array with the same type and shape as *array* and which contains the value $\log(1 + x)$ for each corresponding element x in *array*. Supported for real floating-point types only, and provided only if the underlying C implementation supports it.

---

array.`log2()`

Returns a new Pyvox array with the same type and shape as *array* and which contains the value $\log_2 x$ for each corresponding element x in *array*. Supported for real floating-point types only, and only if the underlying C implementation supports it.

---

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 all the results of the lookup as a Pyvox array. The source image *array* must be either unsigned char or unsigned short but may be of any shape. The `origin` and `spacing` of the result are the same as *array*; they are ignored for the *lut* itself. The *lut* must be rank 1 or 2 and contain at least as many rows as the largest voxel value in *array*; but it (and thus the result) may be of any type. If the *lut* is rank 1, then the result is the same shape as the original *array*. If the *lut* is rank 2, then the result has rank one higher than the *array* and the last dimension ranges over the columns of the *lut*. For example, a *lut* with 256×3 elements could be used to expand a monochrome image into an RGB image.

---

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 as *array*; 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.

If you want only the local mean, you can compute it with *array*`.convolve()`.

---

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.

The kernel used is a $3 \times 3 \times \ldots$ convolution kernel and has the form $2^{-n-\sum |x_i|}$, where n is the rank and x_i are the coordinates; this kernel will completely surpress the Nyquist frequency along any of the coordinate axes.

---

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

Returns a new Pyvox array containing the elementwise maximum of the original and other arrays. Not defined for complex types.

---

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 or double complex. 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.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

---

array`.metacopy(`*other, options...*`)`

This method updates *array* by copying into it selected metadata from the *other* array. The option `metadata=1` copies into *array*`.metadata` each *key*: *value* pair from *other*`.metadata`. The option `filedata=1` copies into *array*`.metadata` each *key*: *value* pair descriptive of the external file from *other*`.metadata`. (The `metadata=1` and `filedata=1` options are currently synonomous, but this may change in the future.) The option `header=1` copies into *array*`.header` each *key*: *value* pair from *other*`.header`. The option `physdata=1` copies *other*`.origin` and *other*`.spacing` into *array*`.origin` and *array*`.spacing`. The option `userdata=1` copies into *array*`.userdata` each *key*: *value* pair from *other*`.userdata`. If no *options* are specified, then all attributes are copied; otherwise only the specified attributes are copied.

---

array`.metadata`

This attribute of a Pyvox *array* is a dictionary in which is stored metadata that is not specific to the image file header or defined by the user.

The only information currently stored in this dictionary describes the file from which the *array* was read, or to which it will be written. These *key*: *value* pairs are automatically set when the *array* is read from an external file but may also be set by the user to specify how *array* is to be written to an external file.

The value corresponding to the key `'bigend'` is an integer that specifies the byte order used in the external file. The value 0 specifies little-endian; 1 specifies big-endian. Note that some image file formats specify a particular byte order, in which case this attribute is ignored.

The value corresponding to the key `'compress'` is a string that specifies the name of the compression program to be used for the external file. Note that some image file formats specify a particular compression algorithm, in which case this attribute is ignored. If the image file format provides separate header and data files, then only the data file is compressed. The currently supported compression programs are `'gzip'` and `'compress'`.

The value corresponding to the key `'filename'` is a string that specifies the filename used for the external file. A filename extension indicating compression is normally omitted, and the compression specified under the `'compress'` key is used. A filename extension indicating the data format is normally included, but will be overridden by the `'format'` value if different.

The value corresponding to the key `'format'` is a string that specifies the data format to be used for the external file. The values currently supported are shown in the table below. If the `'format'` value is undefined or `None`, the format defaults to the value implied by the filename extension.

`format`	Extension	File Format
`'avw'`	`.hdr`	Analyze View format
`'analyze'`	`.hdr`	Analyze View format
`'pgm'`	`.pgm`	Portable Gray Map (binary)
`'ppm'`	`.ppm`	Portable Pixel Map (binary)
`'raw'`	`.img`	Raw data

The value corresponding to the `'seek'` key is an integer that specifies the position in the external file from which the data were read or to which a write operation should seek before actually writing the data from *array*. It is currently implemented only for reading raw format data, and is not stored in the metadata. Note that if this attribute is set and is not `None`, then the file being written to must already exist and will be updated. Zero is the

default and means that the desired data begins with the first byte of the file. A non-negative value means that the desired data is preceded by the given number of bytes. A negative value $-N$ means that the data is followed by $N - 1$ bytes; in particular, `seek=-1` means that the desired data is last in the file.

---

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

Returns a new Pyvox array containing the elementwise minimum of the original and other arrays. Not defined for complex types.

---

array`.minmax(`*axes=All*`)`

This function computes the minimum and maximum of the *array* elements along the specified *axes* (defaulting to all axes) and returns either a list of two scalars or a list of two arrays of reduced rank. 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`. The values listed in *axes* may be in any order, and duplicates are permitted. The returned result is the same type as *array*. Not defined for complex types.

The `origin` and `spacing` of the results are copied from *array*, except that the named *axes* are omitted.

---

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 floating-point type 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. Not supported for complex types.

---

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 of any integral or real floating-point type. Not implemented for complex types. 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 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. 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, axes=All*)

Computes the vector 1-norm of *array*, or of the difference between *array* and *other*, with optional pixelwise weights given by *weight*, and returns either a scalar or an array of reduced rank and type double. The argument *other*, if provided, must be `None`, the scalar 0, or an array of any type but the same shape as *array*. The argument *weight*, if provided, must be `None`, the scalar 1, or an array of any type but the same shape as *array*.

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`. The values listed in *axes* may be in any order; duplicates will be used only once.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

Note that using `foo.norm1(bar)` is better 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, axes=All*`)`

Computes the vector 2-norm of *array*, or of the difference between *array* and *other*, with optional pixelwise weights given by *weight*, and returns either a scalar or an array of reduced rank and type double. The argument *other*, if provided, must be `None`, the scalar 0, or an array of any type but the same shape as *array*. The argument *weight*, if provided, must be `None`, the scalar 1, or an array of any type but the same shape as *array*.

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`. The values listed in *axes* may be in any order; duplicates will be used only once.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

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, axes=All*`)`

Computes the vector sup-norm of *array*, or of the difference between *array* and *other*, with optional pixelwise weights given by *weight*, and returns either a scalar or an array of reduced rank and type double. The argument *other*,

if provided, must be `None`, the scalar 0, or an array of any type but the same shape as *array*. The argument *weight*, if provided, must be `None`, the scalar 1, or an array of any type but the same shape as *array*.

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`. The values listed in *axes* may be in any order; duplicates will be used only once.

The `origin` and `spacing` of the result are copied from *array*, except that the named *axes* are omitted.

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 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. Supported for floating-point types 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. Note that this function is not defined for complex Hermitian matrices, the determinant of which can lie anywhere on the unit circle.

---

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.`real()`

Returns a new Pyvox array containing the elementwise real part of the original array.

---

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

Changes the shape of a Pyvox array in place without copying or modifying any of its elements. As a special case, omitting the *newshape* argument will simply remove any dimensions along which the number of elements is one; for example, a $1 \times 3 \times 1 \times 5$ array would be reduced to a 3×5 array. If *newshape* is given explicitly, then the `origin` and `spacing` attributes are set to their

default values; otherwise they are retained for the retained dimensions only. 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`.rgba2d(dimen, extype=exim.uint1, bigend=1, coords=[0,...], axes=[0,1], chan=[0,0] )`

This function is intended to support interactive display programs that allow the user to dynamically change the assumed external format of an image file of unknown characteristics. It assumes that the external file has been read as a one-dimensional array of unsigned char bytes, reinterprets those bytes as an array of given dimensions and external type, selects a specified 2-dimensional slice, further selects specified input channels as red, green, blue, and alpha, intensity windows the input data, and converts into an unsigned char array suitable for display; all this is, of course, done efficiently so that the user doesn't have to wait too long. The result is always a rank three array, although the last dimension may be one.

The `dimen`, `extype`, and `bigend` arguments specify how the raw bytes of the input file are to be imported into a notional "input image" such as would be created by the `pyvox.rawread()` command. The `coords`, `axes`, and `chan` arguments specify how extract and prepare for display a two-dimensional slice of the input image. All arguments are keyword arguments.

The `dimen` argument is required and specifies the assumed dimensions of the input image. The total number of bytes implied may not exceed the length of *array*. The last dimension may optionally contain the channels of a multichannel image.

The `extype` argument specifies the assumed external type of the input image; it defaults to `exim.uint1`. Note that an external type rather than an internal type is required here.

The `bigend` argument specifies whether the input image is assumed to be big-endian (the default) or little-endian.

The `coords` argument specifies the coordinates of the desired slice in the input image which is to be mapped into output image. It includes the channel dimension (if any) and defaults to a list of all zeros.

The `axes` argument specifies the axes of the input image which are to be mapped to the output image. A list of two elements specifies the axes to become the y and x axes of a single-channel output image. A list of three elements specifies the axes to become the y, x, and channel axes of a multichannel output image. This argument defaults to `(0,1)`.

The `chan` argument specifies the input channel which maps to each output channel. It defaults to `(0,0,0)`, which maps a luminance image into an RGB image.

The output image may be either a single- or multichannel image. A single-channel output image is obtained by setting the `axes` argument to a two-element list and setting the last element of the `coords` argument to the desired channel from the input image. A multichannel output image is obtained by setting the `axes` argument to a three-element list, the last of which specifies the input axis which varies over the bands; then the `chan` argument specifies the coordinate values on this axis which map to the output channels.

---

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 or double complex 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 floating-point types only.

---

array`.sinh()`

Returns a new Pyvox array containing the elementwise hyperbolic sine of the original array. Valid for floating-point types 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 floating-point type and must be compatible in shape.

---

array`.spacing`

This attribute of a Pyvox array 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 floating-point types only.

---

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

Returns a tuple containing the estimated mean $\bar{x}$ and standard deviation s of the elements of *array* along the specified *axes*, optionally weighted by the contents of *weight* array. 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`. The values listed in *axes* may be in any order, and duplicates are permitted.

The *array* and *weight* arguments may be of any type or shape, but must be the same shape. The results are defined by

$$p_i = \frac{w_i}{\sum_i w_i} \tag{4.8}$$

$$\bar{x} = \sum_i p_i x_i \tag{4.9}$$

$$s^2 = \frac{\sum_i p_i |x_i - \bar{x}|^2}{1 - \sum_i p_i^2} \tag{4.10}$$

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. The alternative definition

$$\bar{x} = \frac{\sum_i w_i x_i}{\sum_i w_i} \tag{4.11}$$

$$s^2 = \frac{\sum_i w_i}{\left(\sum_i w_i\right)^2 - \sum_i w_i^2} \cdot \left[\sum_i w_i |x_i|^2 - |\bar{x}|^2 \sum_i w_i\right] \tag{4.12}$$

is often more convenient for computation. In the case that *weight* is omitted

or set to `None`, these simplify to

$$\bar{x} = \frac{1}{N}\sum_i x_i \tag{4.13}$$

$$s^2 = \frac{1}{N-1}\sum_i |x_i - \bar{x}|^2 \tag{4.14}$$

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

The `origin` and `spacing` of the result arrays are copied from *array*, except that the named *axes* are omitted.

---

array`.svd()`

Given an *array* A of $m \times n$ elements, returns a tuple (U, S, V^T) such that U is an orthogonal or Hermitian matrix of $m \times m$ elements, S is a vector of $\min(m, n)$ elements, V^T is an orthogonal or Hermitian matrix of $n \times n$ elements, and

$$A = U \cdot \mathrm{diag}(S) \cdot V^T \tag{4.15}$$

where $\mathrm{diag}(S)$ is an $m \times n$ matrix with the elements of S along its principal diagonal. The elements of S are the singular values of A and are in decreasing order; the rows of U and the columns of V^T are the corresponding left and right singular vectors of A. Note that this function returns the transpose V^T rather than V itself. Valid for floating-point types only.

---

array`.T()`

Returns a new Pyvox array containing the transpose of the original array. Valid for an array of any type. Same as *array*`.trans()`. See also *array*`.H()` for the complex conjugate (Hermitian) transpose.

---

array`.tan()`

Returns a new Pyvox array containing the elementwise tangent of the original array. The input is in radians. Valid for floating-point types only.

---

array`.tanh()`

Returns a new Pyvox array containing the elementwise hypebolic tangent of the original array. Valid for floating-point types only.

---

array`.tgamma()`

Returns a new Pyvox array with the same type and shape as *array* and which contains the value $\Gamma(x)$ for each corresponding element x in *array*. The function $\Gamma(x)$ is defined by

$$\Gamma(x) = \int_0^\infty \mathrm{e}^{-t} t^{x-1}\,\mathrm{d}t \tag{4.16}$$

Supported for real floating-point types only, and provided only if the underlying C implementation supports it.

See also *array*`.lgamma()`.

---

array`.trans()`

Returns a new Pyvox array containing the transpose of the original array. Valid for an array of any type. Same as *array*`.T()`. See also *array*`.H()` for the complex conjugate (Hermitian) transpose.

---

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=???, upper=???*`)`

Returns a new Pyvox array containing the contents of the original array converted to unsigned char. The conversion is done the same way as the underlying C implementation; one consequence is that casting from an integral type is done modulo the size of the destination.

The caller may optionally specify the lower and upper limits of a range to be linearly compressed to fit the 0..255 range of unsigned char voxels; if this is done, then the results are limited to fit the range rather than cast modulo the destination size. Either both or neither of *lower* and *upper* should be provided; the results are unpredictable if only *lower* is provided.

NOTE: Prior to 2003-08-25, the results were limited to the destination min/max even if no lower and upper limits were specified.

---

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.`userdata`

This read-only array attribute is a dictionary reserved for the user, who may save here any information which he wishes to associate with the array.

---

array.`ushort()`

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

---

array.`write(`*filename*`,` *options...*`)`

Writes the contents of a Pyvox *array* to an external file named *filename* as prescribed by the *options*.

In many cases, only the *filename* need be specified, and the *options* will default to the right values. Specifically, the parameters used to write the file take the first-defined of: an explicit argument or keyword option; the values implied by the *filename* extension; the values contained in *array*.`metadata`; or uncompressed, `'raw'`, and the natural external type corresponding to the actual Pyvox array type. But if an explicit argument or keyword option is incompatible with the *format*, it will be silently ignored. See *array*.`metadata` for further information on the parameters and their possible values.

As a special case, the PGM and PPM formats will ignore any extra dimensions equal to 1.

Encapsulated PostScript is supported (for output only) and can be requested by using the `.eps` suffix in the filename or requesting `format = 'eps'`. The keyword options `height=`*height* and `width=`*width* specify the desired height and width, in inches, of the output image. If only one of *height* and *width* is specified, then the other is set to maintain the aspect ratio of the image; if neither is specified, then they are set to provide 72 samples per inch.

The *filename* argument specifies the filename to which the data are to be written; it is optional and defaults to *array*.`metadata['filename']`.

The keyword option `format=`*format* specifies the image file format.

The keyword option `bigend=`*bigend* specifies whether the data are to be stored in big-endian or little-endian byte order.

The keyword option `compress=`*compress* specifies the compression to be used on the file.

(Not yet implemented) The keyword option `exact=1` specifies that the contents of *array*.`header` are preferred to any other attributes of *array*, particularly *array*.`origin` and *array*.`spacing`. This permits certain non-conformant AnalyzeView headers to be copied without modification.

(Not yet implemented) The keyword option `seek=`*seek* specifies that an existing file is to be updated in place, rather than a new file written and specifies the position within the file to which the data are to be written; this preserves any information in the file outside the region written. If this option is used with any value other than `None`, the file must already exist.

(Not yet implemented) The keyword `extype=`*extype* is permitted only for `'raw'` format and specifies the external data type to be used for the data; it defaults to the natural external type corresponding to the actual internal type of *array*. Using this option implies `format='raw'`.

For compatibility with some older versions of Pyvox, the calling sequence *array*`.write(`*filename*`, `*format*`, `*bigend*`, `*options...*`)` may also be used, but this will be deprecated and removed in the future.

---

array`.writepgm(`*filename*`)`

Writes the contents of a Pyvox array to an external file in the binary Portable Gray Map format. The array must be unsigned char or unsigned short, and exactly two dimensions must be greater than 1. Use of this function is discouraged; in most cases, using *array*`.write()` with the appropriate arguments produces the same results and is more flexible. As

---

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 or unsigned short; exactly 3 dimensions must be greater than 1; the last dimension must be 3 and contain the red, green, and blue components of each pixel in that order. Use of this function is discouraged; in most cases, using *array*`.write()` with the appropriate arguments produces the same results and is more flexible.

---

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

Writes the contents of a Pyvox array to an external file in some specified external raw 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 the given string ends in `.Z` or `.gz`, the file is compressed or gzipped after being written. If a file object is given as the filename argument, 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. Use of this function is discouraged; in most cases, using *array*`.write()` with the appropriate arguments produces the same results and is more flexible.

---

ccmap [class instance]

A *ccmap* object represents a map of the connected components (regions for short) found in a 2D or 3D unsigned char *image* and is created by calling `pyvox.ccmap(`*image*`)`.

---

ccmap`.bigob()`

This convenience method returns a new unsigned char array the same shape as the original image in which each pixel has the value 255 if it is contained in the largest connected component and zero otherwise. If there are two or more maximal regions of the same size, one is chosen at random.

---

ccmap`.count(`*region=1*`)`

This method returns the number of pixels contained in the region with the given *region* number, or zero if there is no such *region*. If *region* is omitted, it defaults to the nonbackground region with the greatest number of pixels. Region 0 may be specified.

---

ccmap`.map(`*region=???*`)`

If *region* is specified, this method returns a new unsigned char array the same shape as the original image in which pixels belonging to the specified *region* number have value 255 and all other pixels have value 0.

Otherwise, this method returns a new unsigned char or unsigned short array the same shape as the original image. The value of each pixel in that array is the number of the region which contains that pixel; pixels in the background region have the value 0.

---

ccmap`.regions()`

This method returns a sorted list of (*region*, *count*, *seed*) triples for the connected components found in the image, where *region* is the region number, *count* is the number of pixels in the region, and *seed* is a seed point on the boundary of the region. The background region is omitted.

---

ccmap`.seed(`*region=1*`)`

This method returns the coordinates of a pixel on the boundary of the specified *region*. If *region* is omitted, it defaults to the nonbackground region with the greatest number of pixels. Region 0 is not permitted.

`exim` [module]

This module implements various functions for converting data between internal (platform-dependent) and external (platform-independent) formats; it also includes type descriptors for the internal and external types which it supports.

`exim.dump(`*dict, format, file*`=sys.stdout)`

This function dumps the contents of a dictionary *dict* to the specified *file* according to the struct description *format*. The meaning of *format* is described under `exim.pack`.

`exim.extypes`

Is a list of all the external data types known to the `exim` module.

`exim.intypes`

Is a list of all the internal Pyvox array types known to the `exim` module.

`exim.pack(`*dict, format, bigend=1*`)`

This function packs the contents of the dictionary *dict* into a byte string containing a struct in some external *format*, with byte order as specified by *bigend*. The *format* descriptor is a list or tuple which contains one entry for each field in the format, in the order which they appear. Each field descriptor is a list or tuple of three or four elements. The first is the name of the field, and will be used as a key into the *dict* to find the corresponding value. The second is the default value, which will be used if the *dict* lacks the corresponding key; an error will be reported if the default is `None` and the *dict* fails to provide a value. The third is the external type of the field, given as one of the external types from the `exim` package; the special value `exim.string` requests a string. The value or default provided must be compatible with requested type. The fourth, if provided, is the number of elements in the field or string.

`exim.unpack(`*data, format, bigend=1*`)`

This function unpacks the contents of the byte string *data* containing a struct in some external *format*, with byte order given by *bigend*, into a dictionary and returns the dictionary. The meaning of *format* is described under `exim.pack()`.

---

kernel [type]

A *kernel* object represents a generalized convolution kernel and is typically created by calling `pyvox.kernel`.

---

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 deltas for the *kernel*. 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.

---

kernel`.type`

Returns the type of the *kernel*, which, in the current implementation, will be either `exim.double` or `exim.dcomplex`.

---

obaffine [class instance]

An *obaffine* instance represents an affine transform as an objective function to be optimized and is typically created by calling `regis.obaffine`. The class is derived from `optim.obfunction` and inherits most of the behavior of an *obfun* instance.

---

obfun [class instance]

An *obfun* instance represents an objective function to be optimized and can be created by calling `optim.obfunction`; more commonly, however, more specific classes are derived from the `optim.obfunction` base class and inherit its attributes and methods.

---

obrigid [class instance]

An *obrigid* instance represents a rigid transform as an objective function to be optimized and is typically created by calling `regis.obrigid`. The class is derived from `optim.obfunction` and inherits most of the behavior of an *obfun* instance.

`optim` [module]

This module implements various class and functions to support multi-dimensional optimization.

`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.

---

poly [class instance]

An *poly* class instance represents a N-dimensional polynomial transform and is typically created by calling `pyvox.poly()`.

---

poly `+` *other*

Returns a new *poly* transform which contains the coefficientwise sum of *poly* and *other*, which may be an *affine* or *poly* transform.

---

poly `-` *other*

Returns a new *poly* transform which contains the coefficientwise difference of *poly* and *other*, which may be an *affine* or *poly* transform.

---

poly `*` *number*

Returns a new *poly* transform which contains the scalar product of *poly* and *number*. The form *number* `*` *poly* is equivalent.

---

poly`.addparam(`*parms*`)`

Updates an polynomial transform by adding the contents of the vector *parms* to the coefficients which define the polynomial transform. This function is useful for optimization algorithms that generate perturbations to an initial transform.

---

poly.`compose(`*other, pre=2*`)`

Updates *poly* to contain the composition of *poly* and the affine or polynomial transformation *other*; returns the updated transform. If *pre* is 1, then precomposes *other* with *poly*; if *pre* is 2 or omitted, then postcomposes *other* with *poly*; if *pre* is -1, then precomposes the inverse of *other* with *poly*; if *pre* is -2, then postcomposes the inverse of *other* with *poly*; any other value of *pre* is an error. However, polynomial transforms are not invertible (into polynomial transforms), so attempting to compose with the inverse of a polynomial transform will fail.

---

poly.`copy()`

Returns a new polynomial transform containing the contents and attributes of the original polynomial transform *poly* but which shares no storage with it.

---

poly.`init(`*data*`)`

Initializes the transform *poly* from the given *data* and returns the modified *poly*. The value of *data* must be an *affine* transform, another *poly* transform, a list or tuple, a dictionary, the value 0, or the value 1. If *data* is the value 0, then all coefficients are set to zero. If *data* is the value 1, then *poly* is initialized to an identity transform. If *data* is a list or tuple, it must contain one entry for each term to be defined; each entry must be a three-element list containing the output index, the multi-index which define the exponents to which each input coordinate must be raised, and the value of the coefficient itself. For example, the argument `data=[[0, [0, 0], 2.0], [0, [1, 2], 3.0], [1, [3, 0], 4.0]]` defines the transform

$$\begin{aligned} y_0 &= 2.0 + 3.0\, x_0 x_1^2 \\ y_1 &= 4.0\, x_0^3 \quad . \end{aligned}$$

If *data* is a dictionary, then it must contain one *key*: *value* pair for each term to be defined. The *key* must be a tuple (not a list) containing the total degree of the term, the output index, and the multi-index; the *value* must be the coefficient. For example, `data={(0, 0, (0, 0) :  2.0, (3, 0, (1,`

`2) : 3.0, (3, 1, (3, 0)) : 4.0`} would define the same transform as shown above.

Any existing terms in *poly* not named in *data* are retained but their coefficients are set to zero. Any new terms specified in *data* are added.

---

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

Resamples *image* through the polynomial transform given by *poly* to yield a new Pyvox array with dimensions *dimen* and the same type as *image*. Linear interpolation is used. The polynomial 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. The `origin` and `spacing` of the result are set to their default values, and may be reset by the user to the desired values.

(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.)

---

poly`.map(`*dimen*`)`

Returns a new Pyvox array of type double which contains the output coordinates of the transform *poly* computed from index coordinates over a notional source image of dimensions *dimen*. The rank of the result is one higher than the number of dimensions in *poly*; the (new) last dimension ranges over the number of dimensions in *poly* and contains the output coordinates in order.

---

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

Computes a simple norm on the vector space of polynomial 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 *poly* and *other*; as a special

case, setting *other* to 1 compares *poly* to the identity transform. For the mathmatically inclined, the norm is defined as

$$\|T\| = \max_i \sum_k |c_{ik}| \, \lambda^{|\alpha_k|} \tag{4.17}$$

where T is a polynomial transform, λ is the specified *length*, i runs over the output coordinates, k runs over the terms for each output coordinate i, α_k is the multi-index of exponents for the kth term, and $|\alpha_k|$ is the sum of the exponents in the kth term. (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.

---

poly`.param()`

Returns a vector containing the coefficients which define the transform. This is useful for optimization methods which work on vectors.

---

poly`.point(`*x*`)`

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

---

poly`.scale(`*coef, pre=2*`)`

Updates *poly* by composing it with an elementary scale transform of the form $x[i] = C[i]x[i]$, and returns the updated transform. If *coef* is a plain number, then isotropic scaling is done. The value of *pre* determines the order in which the two transforms are composed; see *poly*`.compose()` for the permitted values.

---

poly`.setparam(`*parms*`)`

Sets the coefficients which define the polynomial transform from the contents of the vector *parms*. This is useful for optimization methods which work on vectors.

---

poly.`translate`(*delta, pre=2*)

Updates *poly* 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 *poly*.`compose()` for the permitted values.

---

poly.`truncate`(*maxdegree*)

Updates *poly* by removing all terms of total degree higher than the specified *maxdegree*. Note that simple truncation does NOT provide the polynomial transform of degree *maxdegree* that most closely approximates the original transform. (The method that will has not yet been written.)

`pyvox` [module]

This module implements the core types, class, and functions of the Pyvox extension to Python.

`pyvox.affine`(*ndim, matrix=1, offset=0*)

Creates a new affine transform in the given number of dimensions with the given *matrix* and *offset*. If *matrix* is omitted, an identity matrix is used; if *offset* is omitted, zero is used. Providing a plain number for *matrix* yields a *diagonal* matrix; providing a plain number for *offset* yields a column vector all of whose elements are that number.

`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.ccmap(`*image*`, minsize=0, maxreg=None)`

Scans a two- or three-dimensional unsigned char *image* to find its connected components and returns a new *ccmap* object which the user can query for more information.

A *connected component*, or region, of a digital image is defined to be a maximal connected set of nonzero pixels, where two pixels are *connected* if none of their corresponding coordinates differ by more than one. The *boundary* of a region is defined to be those pixels contained in the region that are connected to some pixel outside the region. (Note that this definition differs from the one used in topology, in which a boundary point need not be contained in the region.)

Each nonbackground region found is assigned a region number, starting from 1, in order by decreasing number of pixels; if there are two regions of

the same size, the order is indeterminate. Region 0 always represents the background and includes all pixels that were zero in the original *image*, that were contained in a region of less than the specified minimum size, or that were not contained in the *maxob* largest regions. Note that the background pixels need not form a single connected region.

If the optional keyword argument `minsize` is provided, then any region containing fewer than the specified number of pixels will be omitted, and its pixels added to the background object.

If the optional keyword argument `maxreg` is provided, then only the specified number of nonbackground regions will be included; all other regions will be added to the background.

See *ccmap* and its methods for more information.

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

Creates an $n \times 1$ Pyvox array of type double or double complex from the given *data*, which may be a number, tuple, list, or a Pyvox array of any shape; if *data* is a number, 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*`)`

This method is a deprecated alias for the `pyvox.array()`.

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

Creates a new $n \times n$ Pyvox array of type double or double complex whose diagonal elements are taken from the given *data*, which may be a number, tuple, list, or Pyvox array of any shape. If *data* is a number, 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.

See also *array*`.diag()` to extract the diagonal elements of a matrix.

`pyvox.error_action(`*what*`)`

This function sets the action that Pyvox should take if an error is encountered within the C extension code; it is likely to be useful only to those who are actually debugging the C code. The possible values are 0 to generate a Python error, 1 to exit immediately from the program, and 2 to abort and dump core.

`pyvox.fftramp(`*size, type=double, axis=Last*`)`

Returns a new Pyvox array of the specified shape and type, each voxel of which contains the FFT frequency index corresponding to its own coordinate index 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. If the requested type is complex, the real part contains the index and the imaginary part is zero. This function is not defined for unsigned types. If there are N elements on the specified axis, then the frequency n' corresponding to element $n \in \{0, \ldots N-1\}$ is

$$n' = \begin{cases} n & \text{if } n \leq N/2 \\ n - N & \text{otherwise.} \end{cases} \tag{4.18}$$

`pyvox.have_complex`

This attribute is 1 if complex types are supported in this build of Pyvox, and 0 otherwise. The complex types `exim.fcomplex` and `exim.dcomplex` are always *defined*, but operations on complex types will be supported only if `pyvox.have_complex` is true.

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

Returns a new Pyvox kernel with given neighbors, coefficients, and bias. The *deltas* argument is a list of the neighbors, defined by an offset in each coordinate direction. The *coefs* argument is optional; if omitted, the kernel defines a neighborhood only. Otherwise *coefs* is a list of numbers giving the coefficient at each neighbor. The *bias* argument is also optional; if omitted, it defaults to zero. The *bias* and *coefs* may be either real or complex. The kernel itself will be double complex if any of the *coefs* is complex, and double otherwise. 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.lowpass(`*rank*`)`

Returns a new Pyvox kernel which contains a standard lowpass filter in the specified number of dimensions. The kernel generated is a $3 \times 3 \times \ldots$ convolution kernel with the form $2^{-n-\sum |x_i|}$, where n is the rank and x_i are the coordinates; this kernel will completely surpress the Nyquist frequency along any of the coordinate axes.

`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 or double complex 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, or a plain number; if *data* is a plain number, 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, or for generating a polynomial transform using `pyvox.poly()`.

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

Creates an $n \times 1$ Pyvox array of type double or double complex from the given *data*, which may be a number, tuple, list, or Pyvox array of any shape; if *data* is a number, then 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.poly(`*ndim, options...*`)`

Creates a new polynomial transform in multiple dimensions. The parameter *ndim* specifies the number of dimensions.

The keyword option `degree=`*degree* specifies the degree of the transform to be created. For example, 1 specifies an affine transorm (linear and constant terms only) and 2 specifies a quadratic transform (constant, linear, and quadratic terms). The `degree` defaults to `None`, in which case only the terms specified in the `init` argument are provided.

The keyword option `init=`*init* specifies the multi-indices and corresponding coefficients and works as described under *poly*`.init()`. The *init* defaults to `None`, in which case only the terms specified by *degree* are included, with zero coefficients.

If both `degree` and `init` are supplied, then any terms omitted from `init` default to zero, while any terms listed in *init* with degree higher than *degree* are added to the transform. If neither `degree` nor `init` are specified, then a zero transform is created; this transform has no terms and maps every input point to zero.

`pyvox.PolyType`

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

`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 index 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. If the requested type is complex, the real part contains the index and the imaginary part is zero.

`pyvox.randi(`*shape, N*`)`

Returns a new Pyvox array of the specified shape and type int, which contains a set of newly generated random integers in the range $0, \ldots, N-1$. The default random number generator is used.

`pyvox.randn(`*shape, mean=0, sdev=1*`)`

Returns a new Pyvox array of the specified shape and type double, which contains a set of newly generated normal random variates with the given *mean* and *sdev*. The default random number generator is used.

`pyvox.randu(`*shape, a=0, b=1*`)`

Returns a new Pyvox array of the specified shape and type double, which contains a set of newly generated uniform random variates in the interval (a, b). The default random number generator is used. The generated variates should never exactly equal a or b when the default a and b are used, but, due to roundoff error, this cannot be guaranteed for user-supplied values.

`pyvox.rawread(`*filename, dimen, extype=uint1, bigend=1, seek=0*`)`

Returns 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), an optional flag to mark bigendian (which defaults to bigendian), and the position in the file to start reading from.

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. This may not be combined with a non-zero *seek* argument.

If the *seek* is positive, it is relative to the beginning of the file; if negative, relative to the end of the file; if zero, no seek is done.

NOTE: This function is deprecated and should be replaced by `pyvox.read` in new code.

`pyvox.read(`*filename*`, `*options...*`)`

Reads an external file named *filename* with specified keyword *options* as described below and returns the contents as a new Pyvox array. In many cases, only the *filename* need be specified and the *options* will default to the right values according to the file format defined by the extension part of the filename. Specifically, the parameters used to read the file will be taken as the first-defined of: an explicit argument or keyword option; the values implied by the magic number in the file itself; or the values implied by the *filename* extension. But if an explicit argument or keyword option is incompatible with the *format*, it will be silently ignored.

The *array*`.metadata` attribute of the returned array will contain the filename and many of the other parameters used to read the file. See *array*`.metadata` for additional information.

The *array*`.header` attribute of the returned array will contain the information extracted from the file header; the specific keys and values provided depend on the selected data *format*.

The *filename* argument specifies the name of the file to be read. If the image *format* specified uses separate header and data files, the name of the header file should be specified here and the name of the corresponding data file will be either read from the header file or automatically generated. The *filename* extension indicates the data format but can be overriden by the `format` option. If the given *filename* does not exist, but appending a compression-indicating suffix to it yields a real file, then that file will be silently uncompressed.

The keyword option `format=`*format* specifies the image file format used; see *array*`.metadata` for the currently supported values.

The keyword option `bigend=`*bigend* specifies whether the data is stored in big-endian or little-endian byte order; it defaults to 1 (big-endian) but is ignored if the byte order is implicit in the format, as in the PGM and PPM formats.

The keyword option `seek=`*seek* specifies the position of the desired data in the file. See *array*`.metadata` for additional information. This option is supported for raw mode only.

(Not yet implemented) The keyword option `origin=`*origin* is a number or list of numbers which specify the physical coordinates which correspond to index $(0, \ldots)$; it will be overridden by the origin specified in the header file, if any. A single number will be replicated as needed for the actual number of dimensions. The actual *origin* will be saved as the *array*`.origin` attribute.

(Not yet implemented) The keyword option `spacing=`*spacing* is a number or list of numbers which specify the physical spacing between voxels; it will be overridden by the spacing specified in the header file, if any. A single number will be replicated as needed for the actual number of dimensions. The actual *spacing* will be saved as the *array*`.spacing` attribute.

The following keyword options apply to only a few image file formats. In fact, they apply only to the `'raw'` format, but this might change in the future.

The keyword option `dimen=`*dimen* is permitted and required only for the `'raw'` format and specifies the dimensions of the data to be read; as a special case, the first dimension may be specified as `0`, in which case as many slices or rows as possible will be read from the file. In all other formats, the dimensions are specified in the file itself.

The keyword option `extype=`*extype* is permitted and required only for the `'raw'` format and specifies the external data type used for the data. The use of this option implies `format='raw'`. The internal type of the Pyvox array returned defaults to the natural internal type corresponding to the external type but may be overriden by the `intype=`*intype* option.

(Not yet implemented) The keyword option `intype=`*intype* is permitted only for the `'raw'` format and specifies the internal data type to which the data are to be converted. It defaults to the natural internal type corresponding to the specified *extype*.

For compatibility with some older versions of Pyvox, the calling sequence `pyvox.read(`*filename*`,` *dimen*`,` *format*`,` *bigend*`,` *options...*`)` may also be

used, but this will be deprecated and removed in the future.

`pyvox.refcnt(`*ob*`)`

Returns the current reference count of a Python object; mainly useful for debugging Pyvox itself.

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

Creates an $1 \times n$ Pyvox array of type double or double complex from the given *data*, which may be a number, tuple, list, or a Pyvox array of any shape; if *data* is a number, 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.

`pyvox.truefile(`*filename)*

This function returns the true file name of a possibly compressed file. That is, if the given file name exists and is readable, it returns the given file name. If not, but if the given file name does not already end in `.Z` or `.gz` and appending one of these yields the name of a valid and readable file, it returns the modified file name. Otherwise, it returns `None`.

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

Creates an n-element rank-1 Pyvox array of type double or double complex from the given *data*, which may be a number, tuple, list, or Pyvox array of any shape; if *data* is a number, 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.

`reged` [module]

This module implements a generic interactive region editor for volume images. It is still being developed and is highly subject to change; if you feel brave or ambitious, look at the files `lib/reged.py` and `scripts/bredit`.

`regis` [module]

This module implements various classes and functions for the registration of N-dimensional images.

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

This function in the `regis` module computes the (optionally weighted) Pearson product-moment correlation of two images. The two images *image1* and *image2* may be of any type and shape but must be the same shape. The optional *mask* image defines a per-pixel weight; it must be the same shape as the other two images. The *mask* defaults to one. The correlation r is computed as

$$s_x^2 = \sum_n w_n x_n^2 - \frac{\left(\sum_n w_n x_n\right)^2}{\sum_n w_n} \tag{4.19}$$

$$s_y^2 = \sum_n w_n y_n^2 - \frac{\left(\sum_n w_n y_n\right)^2}{\sum_n w_n} \tag{4.20}$$

$$r = \left(s_x^2 s_y^2\right)^{-1/2} \left[\sum_n w_n x_n y_n - \frac{\left(\sum_n w_n x_n\right)\left(\sum_n w_n y_n\right)}{\sum_n w_n}\right] \tag{4.21}$$

where n ranges over the voxels in each image, x_n and y_n are the intensities of the nth voxel in the first and second images, and w_n is the weight for the nth voxel.

`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

$$p_k = \sum_{x_i=k} w_i \Big/ \sum_i w_i \tag{4.22}$$

$$I = \sum_k p_k \log_2 p_k \tag{4.23}$$

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

$$p_{kl} = \sum_{x_i=k;\ y_i=l} w_i \Big/ \sum_i w_i \tag{4.24}$$

$$p_k = \sum_l p_{kl} \tag{4.25}$$

$$p_l = \sum_k p_{kl} \tag{4.26}$$

$$I = \sum_{kl} p_{kl} \log_2 \left(\frac{p_{kl}}{p_k p_l} \right) \tag{4.27}$$

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 progres-

sively 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.

`tkphoto.tkphoto(`*tkphoto=None, **options*`)`

This function wraps the Tkinter PhotoImage provided as argument in a Pyvox `tkphoto` object and returns the new `tkphoto` object. Alternatively, if no positional arguments are provided, this function creates a new Tkinter PhotoImage object using any keyword arguments provided, and returns that wrapped in a `tkphoto` object. It is intended that a `tkphoto` object should support all the methods of the enclosed `PhotoImage` object, and be usable anywhere that a `PhotoImage` object can be used; but don't count on it just yet.

---

tkphoto [type]

A *tkphoto* object is a wrapper around the Tkinter PhotoImage object and is typically created by calling `pyvox.tkphoto`.

---

tkphoto`.getimage()`

This method of a `tkphoto` object copies the contents of the image it contains into a new Pyvox array object and returns the new array. The new array always has type unsigned char and dimensions *height* $\times$ *width* $\times$ 4, where the last dimension ranges over red, green, blue and alpha.

---

tkphoto`.handle`

This attribute of a `tkphoto` object is the Tk handle of the Tkinter PhotoImage it contains.

---

tkphoto`.name`

This attribute of a `tkphoto` object is the name of the Tkinter PhotoImage it contains, as assigned by Tk.

---

tkphoto`.putimage(`*data, position=(0,0), size=all, zoom=1, subsample=1)*

This method of a `tkphoto` object copies the contents of the Pyvox array *data* into the *tkphoto* image. The Pyvox array must have type unsigned char and dimensions *height* × *width* × 4, where the last dimension ranges over red, green, blue and alpha; the last dimension may also be omitted entirely (gray level only), 1 (gray level), or 3 (RGB). The *position* of the upper left corner defaults to the upper left of the `tkphoto` image. The *size* defaults to the size of *data*, adjusted for *zoom* and *subsample*. The *zoom* and *subsample* arguments may be a single number to apply equally to x and y, or a list of numbers for anisotropic zoom and subsampling. If the image *data* contains an alpha channel, note that the `Tkinter Canvas` widget seems to distinguish only between zero (transparent) and non-zero (opaque). This method is essentially a wrapper for the Tk function `Tk_PhotoPutZoomedBlock` and the documentation of that function should be consulted for the finer details.

---

tkphoto`.size`

This attribute of a `tkphoto` object describes the dimensions of the Tkinter PhotoImage that it contains. The dimensions are always *height* × *width* × 4, where the last dimension ranges over red, green, blue and alpha.

---

tkphoto`.tkphoto`

This attribute of a `tkphoto` object is the Tkinter PhotoImage that it contains.

---

type`.code`

This attribute of an internal or external *type* is the numeric code used to designate it within the Pyvox C code. This is unlikely to be of any interest to the `Pyvox` user, but is used internally.

---

type`.complex`

This attribute of an internal *type* is the complex type with equal or better precision; for example, `exim.double.complex` evaluates to `exim.dcomplex`. Any complex internal type maps to itself.

---

type`.desc`

This attribute of an internal or external *type* is a short (half-line) string describing the type.

---

type`.epsilon`

This attribute of an internal *type* is the smallest value $\epsilon >$ such that $1+\epsilon > 1$ when computed in the given *type*. Integral types are assumed to have $\epsilon = 1$; complex types are assumed to have the same ϵ as the corresponding real type. This value is useful in estimating the available precision of computations in the given *type*; see a book on numerical analysis for further details.

---

type`.extype`

This attribute of an internal *type* is the external type that most naturally corresponds to *type*. Note that the correspondence is not guaranteed to be constant across platforms, or different versions of `Pyvox`. Returns `None` for an external type, although this might change in the future.

---

type`.intype`

This attribute of an external *type* is the internal type that most naturally corresponds to *type*. Note that the correspondence is not guaranteed to be constant across platforms, or different versions of `Pyvox`. Returns `None` for an internal type, although this might change in the future.

---

type.`iscomplex`

This attribute of an internal or external *type* is 1 if the type is complex-valued and 0 if it is real-valued.

---

type.`isfloat`

This attribute of an internal or external *type* is 1 if the type is a real or complex floating point type and 0 otherwise.

---

type.`isint`

This attribute of an internal or external *type* is 1 if the type is integral and 0 otherwise.

---

type.`isreal`

This attribute of an internal or external *type* is 1 if the type is real-valued and 0 if it is complex-valued.

---

type.`isunsigned`

This attribute of an internal or external *type* is 1 if the type is unsigned and 0 otherwise.

---

type.`name`

This attribute of an internal or external *type* is a one-word string naming the type.

---

type.`nbytes`

This attribute of an internal or external *type* is the number of bytes of storage required by each element.

---

type.`real`

This attribute of an internal *type* is the real type with the same precision; for example, `exim.dcomplex.real` evaluates to `exim.double`. Any real integral or floating-point type maps to itself.

Chapter 5

Applications and Examples

The Pyvox package also contains various command-line application programs written either as Python scripts or directly in C. These are listed briefly below and are fully described by man pages. There are also several example programs written in Python, and usually documented by man pages.

5.1 Examples

The example programs are implemented in Python and contained in the `examples` directory; they are intended as examples of the use of Pyvox and are not installed as part of the Pyvox package. Their usage is described by man pages in the same directory, and their implementation in the source code. There may be other, experimental, programs in this directory that have not been documented yet. On the other hand, some of these program may move into the applications when they grow up.

Example programs	
`agate`	Generate agate-like fractal image
`brmask4`	An experimental brain masking algorithm
`colorcube`	Generate volume image with all 24-bit colors
`conseg(1)`	Compute concordance of two segmented images
`ellipsoid`	Generate test image with ellipsoidal region
`fresnel`	Generate test image with Fresnel diffraction pattern
`kmsegm`	Univariate or bivariate K-means segmentation
`lovar`	Compute local mean and variance of an image
`paregis`	Principal axes registration
`regis`	Affine registration and resampling
`rpsamp(1)`	Choose random set of points within an image
`vihist`	Univariate or bivariate histogram

5.2 Applications

Applications are implemented either in Python or C and appear in either the `scripts` or `src` directories; they are intended to be useful programs and are installed with the rest of the Pyvox package. Their usage is described in man pages, and their implementation in the source code.

Application Programs	
`anonavw(1)`	Anonymize an AnalyzeView header file
`bredit(1)`	Interactive brain mask editor
`dumpavw(1)`	Dump contents of an Analyze View header file
`editavw(1)`	Edit contents of an Analyze View header file
`makeavw(1)`	Create an Analyze View header file
`qdv(1)`	Image viewer for gray-scale volume images
`reged(1)`	Generic interactive region editor

Chapter 6

Installation

6.1 Prerequisites

In short, Pyvox requires a Unix-compatible operating system, the Gnu C compiler with C99 support, Posix-compatible C libraries, Python 2.1 or later, the X Window System with 24-bit true color visuals, Tcl/Tk, 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 Pyvox 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 (1999) Compiler

Pyvox is intended to work with any ANSI (1999) C compiler [4], 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.

Compilation under a C89 C compiler is possible in theory, but scant effort goes into making sure that it works; it's probably more economical for you to upgrade your compiler than to patch Pyvox to match.

Pyvox will compile under a C compiler that lacks support for complex numbers, but accomplishes this by omitting any feature that uses complex numbers. The recommended solution is to upgrade your compiler.

6.1.2 Posix C Libraries

C libraries compatible with Posix [3, 5] are required.

6.1.3 Python

Version 2.1 or later of Python [2] is required. In theory, you can compile with Python 2.0 but all the features that depend on a later version will be omitted. If the Python executable is not on the path, or if it is not named `python`, use the `--with-python=`*EXEC* to specify the location of the desired Python executable. In this case, you should carefully check the generated `Makefile` for errors and use `make -n install` before installing for real.

6.1.4 X11

Several GUI programs in Pyvox require the X11 Window System with a 24-bit true color display. Pyvox will (probably) compile without X11, but will omit all the GUI programs.

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 at all, use the `--without-x` option to leave out the GUI programs.

6.1.5 Tcl/Tk and Tkinter

The current implementation of Pyvox uses the `Tkinter` extension module to support GUIs, which is a wrapper around the Tcl/Tk libraries. This is still somewhat experimental, since Tk is not very good at handling and displaying images generated internally by a program rather than read from the disk; we might eventually wind up using some other GUI package, perhaps GTK.

If no working `Tkinter` module is found during configuration, then the Pyvox `tkphoto` module will not be created and none of the modules or scripts that depend on it will work.

6.1.6 Pmw: Python Mega Widgets

The GUI programs other than `qdv` require the Python Mega Widgets package; version 1.2 is known to work and is available from

`http://pmw.sourceforge.net/` .

6.1.7 Motif/Lesstif

The `qdv` image viewer requires X and Motif with a 24-bit true color visual. Pyvox should compile even without these, but `qdv` will not be built. Pyvox is expected to eventually eliminate any use of Motif/Lesstif in favor of a more modern GUI toolkit, but this is not yet a high priority item.

6.1.8 LAPACK and BLAS

Pyvox uses the LAPACK and BLAS libraries [1] for numerical linear algebra; some platforms may also require the f2c library or the F77 and I77 libraries to support LAPACK and BLAS. 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 Pyvox up and running.

6.1.9 Miscellaneous

A recent `make` command is required. Gnu `make`, often installed as `gmake` on platforms with a vendor-supplied `make`, is known to work. Many, but not all, other versions of `make` will work.

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 Particular Platforms

Pyvox is developed primarily on Linux for the x86 and x86_64 platforms, and works best on these platforms. Development and testing on other platforms is still experimental and probably buggy. 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.

6.2.1 Linux

Pyvox should build and install out of the box on any recent RedHat or Fedora distribution, provided that you have the appropriate tools and libraries installed. It is likely to work well on most other distributions, but we do not attempt to test them.

Some (older) versions of RedHat and Fedora Linux do not install `tcl-devel` and `tk-devel` rpm packages by default, so you may need to install these by hand.

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/pythonN.N
```

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

6.2.2 Darwin (Mac OS X)

The Developer's Toolkit is required to build Pyvox. If you want to use any of the GUI programs, you will also need to install X11 support and the X11 SDK. However, we don't yet have the GUI programs working reliably on Darwin.

In at least some versions of Darwin, version 2.95.2 of the C compiler does not handle `__restrict` correctly but fails to set the return code to indicate

this, which causes the `config.h` file to be set up incorrectly. The symptom is that compiling `bips.c` generates several pages of syntax errors complaining of a missing ':' after '__restrict'. A workaround for this error is to manually edit `include/config.h` after running `./configure` to define `restrict` as an empty string.

In at least some versions of Darwin, the `math.h` header file for version 2.95.2 of the C compiler does not include a proper ANSI prototype for the `cabs` function and the compiler will complain about it with the warning levels as usually set by Pyvox; this warning can be ignored without consequence.

6.2.3 Solaris

The GUI programs do not yet build and work reliably on Solaris. Nor do complex numbers, although this may be an artifact of an outdated or misconfigured C compiler on our test system.

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.

6.2.4 IRIX

This platform is *really* experimental; don't try it unless you're willing to debug it. The rest of this section consists of my notes from trying to get it to work correctly.

The GUI programs do not yet build and work reliably. Nor do complex numbers.

...Python, Tk, gcc in `/usr/freeware`; not known if this is vendor-supplied or a local modification.

...multiple executable file formats; you must use the same one for Python, Pyvox, Tk, and other shared libraries files, and must compile pyvox in this same format. For the one sample system, the default gcc format was mips-3 and n32; the appropriate library files were found in `/usr/lib32` and `/usr/freeware/lib32`. Use the –with-library-path option on configure to set the path correctly.

...Typing plain `python` on the command line gets version 1.5.2, which is no longer compatible with Pyvox. There also exists `python2`, which appears to be version 2.1. Use `--with-python=‘which python2‘` to get the right

version. Also define `ln -s ‘which python2‘ /bin/python` so that the `/usr/bin/env` hack works correctly.

...For at least one example, the following works:

`./configure --with-python=‘which python2‘ --with-library-path=/usr/freeware/lib`

...On my test machine, gcc is configured with –disable-c99, which means no complex numbers and thus no real Pyvox support. I notice also –disable-shared, although that doesn't seem to have stopped me from using the -shared option successfully.

6.3 Installation Locations

The configuration script attempts to guess where the various components of Pyvox should be installed; its guesses are printed out at the end of its run. You should review these to make sure that they are appropriate for your system. If they are not, you have the following options: (1) Use the `--prefix` and `--exec-prefix` options to `./configure`. (2) Modify the generated `Makefile` by hand. (3) Modify the `configure.in` file to make a better guess. If you do the latter, please send the improved script to us.

One particular point should be noted. If you have Python installed in `/usr` rather than in `/usr/local`, which includes most platforms on which Python is installed with the system rather than being a later add-on, the configure script will typically install the new Python modules under

```
/usr/local/lib/pythonN.N/site-packages
```

provided that this directory exists, rather than under

```
/usr/lib/pythonN.N/site-packages
```

This is for consistency with the fact that all the other components are installed in `/usr/local` rather than `/usr`. If this is what you want, then you should make sure that the former directory exists (and is included on your `PYTHONPATH` environment variable); if not, then you should make sure that that directory does not exist, or modify the `Makefile` by hand.

6.4 Procedure

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 0.67 or earlier, see the the section "Upgrading Old Installations" below on manually fixing some incompatibilies between the old and new versions.

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. Examine the installation locations list at the end of the run and make sure they are what you want.

4. Run `make` to compile and link everything. A side effect of this is to run `make tags` to create the `TAGS` file for emacs; if you're a vi partisan, make the obvious change to the Makefile.

5. Run `make regress` to run the regression tests. These don't yet test everything, but are sufficient to give a reasonably good smoke test.

6. 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.

7. Several Python extension modules are installed in the directory

   `$PREFIX/lib/pythonN.N/site-packages`;

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

8. Similarly, the shared libraries `libpyvox.so` and `libvoxkit.so` are 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" above.

6.5 Upgrading Old Installations

If you are upgrading from BBLmage version 0.67 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/pythonN.N/site-packages`

  where N.N is the Python version. You should remove the files `pyvox.so` and `exim.so` from the old location.

- The old command line programs `anonbblanz`, `binnseg`, `conseg`, `dumpbblanz`, `imstack`, `inleav2`, `lovar`, `rpsamp`, `skmiv`, `swab`, `usb2uc`, `vibihist`, and `vihist` have been converted to examples in Python or removed entirely; you should remove the old programs and their man pages from their installed locations.

- 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 recommended that the installer should create a link from `/usr/local/bin/python` to `/usr/bin/python`; this is no longer necessary and should be removed unless needed for some other reason.

- Versions 0.67 and earlier installed a single shared library `libbbli.so` in `$PREFIX/lib`; later versions have replaced this with two libraries `libvoxkit.so` and `libpyvox.so`. You should remove the older shared library.

6.6 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/pythonN.N`. 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. Pyvox 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 Pyvox to use its own light-weight libraries.

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

   ```
   --with-python=EXEC
   ```

   can be used to specify the location of the Python executable.

6.7 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 Pyvox 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 `realclean` target deletes all files that can be regenerated by the builder.

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

The `regress` target runs a regression test on the code in the build directory (not the installed code)

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, the predecessor of Pyvox, 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.

The original BBLimage package also 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 Pyvox.

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 One Glue Language

Pyvox is intended to be used with a single glue language—Python—rather than trying to support C++, Python, Perl, Tcl, and perhaps a few other scripting languages. While this limits the number of people who will be willing to try Pyvox, it has some compensating advantages. The user interface can be designed to exploit and support the special features of the chosen glue language, rather than the lowest common denominator of several languages. The parts of the package that are not critical to performance can be written in Python rather than C, making them simpler to implement. The effort needed to write several sets of wrappers for different glue languages can be devoted instead to making one wrapper better.

Nonwithstanding the above arguments, the high-performance algorithms in Pyvox are all written in C and can be called from C or C++ programs by any programmer who is willing to look into the source code to determine the calling sequences; the documentation is done carefully, just not extracted into a separate manual.

7.2.8 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, BLAS, and perhaps libf2c) before installing Pyvox, although we also provide an internal lightweight version for the impatient.

7.2.9 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 systems. 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 (1999) and Posix should be able to comple and run Pyvox with little or no modification. Most development and testing has been done on a 32-bit platform.

Both 32-bit and 64-bit platforms are supported; however, less development and testing has been done on 64-bit platforms and there are likely to be residual bugs.

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.

A probably incomplete list of such portability assumptions follows. Some of these might actually be guaranteed by the C standard, but it might take a language lawyer to be sure; even so, a compiler implementor might get them wrong. (Adding tests in the configuration script might not be a bad idea.)

- A C `double complex` value and a Python `Py_complex` have the same layout in memory; that is, two contiguous `double` values containing the real and imaginary parts in that order with no internal or external padding.

- Addresses within an array can be computed by converting pointers to unsigned char pointers, doing arithmetic in units of the original size, and converting back to pointers of the original type.

7.2.10 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. This means that per-neighbor and per-voxel operations are almost always implemented in C, while per-image operations may be implemented in either C or Python, whichever is more convenient.

7.2.11 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). Note that the current implementation of error management is NOT thread-safe, nor are a few other functions; most of these should be marked with FIXME comments.

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.

7.2.12 Data Typing

The header for a Pyvox array 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 exceptions to this rule are that some operations are meaningful only for unsigned, floating point, or complex types. 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.

7.2.13 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.

7.2.14 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.

7.2.15 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 raw pixel data in raster order, plus AnalyzeView and Portable Bit Map formats; extensions to handle DICOM are almost certain but have not yet been implemented. Other extensions could be added, but none are currently planned. The `exim` module provides some tools for packing and unpacking data in other formats.

7.2.16 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.

7.2.17 Random Variates

Fully specifying random variates to be generated can be very complex, but many users need only a common set of distributions.

To support the casual user, Pyvox supplies a default high-quality random number generator with simple calling sequences but which is initialized from the current time and supports only discrete uniform (`pyvox.randi`), continuous uniform (`pyvox.randu`), and normal (`pyvox.randn`) distributions.

(Not yet implemented.) For the more advanced user, Pyvox provides a new Python type *random*, which is a random number generator initialized by a seed supplied by the user and which can support a larger class of distributions. (It might also be reasonable to allow the user to specify the type, but the crystal ball is not yet clear on this point.)

Pyvox does not directly support the generation of complex-valued random variates, since any serious user needs to decide how the variance is to be distributed between the real and imaginary parts, and the covariance of the real and imaginary parts; more generally, the user must define the joint probability distribution for the real and imaginary parts. A calling sequence that supports this complexity is conceptually no simpler than simply generating the real and imaginary parts separately and combining them into a complex array. Related to this is the fact that there seems to be no clear definition of what is meant by a "standard" normal or uniform distribution in complex variables. There is, of course, some loss of efficiency in this approach, but there doesn't seem to be any compelling use case for what is only a minor efficiency improvement.

For somewhat similar reasons, Pyvox also provides floating-point random variates only in type double; as before, the complexity added to the calling sequence does not seem justified by a use case.

7.2.18 Error Management

The error management mechanisms are intended to serve three categories of users: Pyvox users writing in Python; Pyvox developers writing in C and Python; and BIPS/Voxel Kit users writing in C. For each category of user, an error report should show the context in which it occurs and a description

of the error at the appropriate language level. Pyvox developers and perhaps BIPS/Voxel Kit users from C may also wish to abort execution and do stack traces using a debugger.

There are also various constraints on the implementation. BIPS and the Voxel Kit should not depend on Python or Pyvox, so that they can be used from C, but any errors found should be converted into Python exceptions for Python users. The `setjump/longjump` facility of C should be avoided, since its portability is doubtful. Multi-threading should be supported (but is not yet implemented).

There are three distinct levels of error severity. Warnings indicate anomalies that can probably be safely ignored; the error handler may be configured to silently ignore warnings, print a message and continue, or convert warnings into errors. Errors indicate that the requested operation failed to complete successfully. The handler may be configured to pass the error back to the caller to be handled there; print an error message and call `exit()` with a failure status; or to call `abort()`, which normally dumps core for inspection by a debugger. Panics indicate that the software cannot safely continue, perhaps because permanent data structures have been damaged; the handler always prints an error message and calls `abort()`.

The `errm.h` file defines a set of numeric error codes which specify different classes of errors. When an error occurs, one of these codes is saved in a thread-local variable for use by the caller. None of the existing code ever looks at this variable, but this may change in the future; in particular, they may be used to select a specific Python exception to be raised. A second thread-local variable is set to a descriptive string which describes the error for a human user. (NOTE that the error management code is not yet thread-safe.)

Pyvox functions called directly from Python convert the error code and message into a Python exception and return that to their callers in the usual fashion. There are several macros defined in `errm.h` to facilitate this. The `ReturnPyvoxError` macro constructs an exception using an error code and message given explicitly, for errors detected in a top-level Pyvox function, and returns NULL to the caller to indicate an error; the `XReturnPyvoxError` does the same thing, except that it returns a non-zero value to indicate an error. The `ReturnPyvoxLibError` and `XReturnPyvoxLibError` are similar but use an error code and message already set by a lower-level function. There are two other functions `set_error_context` and `clear_error_context` which are used to set the name of the top-level function as seen by the Python

programmer; the context string provided is included in the Python exception generated and serves to define the error context in terms intelligible to the Python programmer.

Lower-level Pyvox functions, plus all Voxel Kit and BIPS functions, use different macros that do not depend on Python. The `NZReturnError` and `ZReturnError` macros set the error code and message from explicit arguments and return non-zero and NULL values respectively; the `NZReturnLibError` and `ZReturnLibError` macros are parallel but use an error code and message already set by a lower-level function.

Destructors get special handling, because they are called at random times that are apparently unrelated to the code currently being executed. The errors detected by a destructor are treated as warnings and normally just print an error message before continuing; the developer who wishes to investigate further can configure warnings to convert to errors and work with a core dump.

BLAS and LAPACK currently return an error flag to their callers but do not set the thread-local error code and message; this can be expected to change in the future.

There are also three older error reporting functions `warning`, `fatal`, and `panic` which print an error message and then continue or terminate the program. These are suitable only for the top level of command line programs and are being gradually eliminated from all other code, except that `panic` is still sometimes used to report assertion failures and similar severe problems.

7.2.19 Regression Tests

The `test` directory contains a set of scripts for regression testing and a master script `test_all` that invokes all the non-interactive tests in turn and summarizes the results at the end; the master script is invoked by `make regress` for easy testing. There are a few tests for measuring the performance of the Pyvox code, and for testing either interactive functions or low-level C functions which are independent of the master test script.

For the most part, the C extensions are tested only through Python rather than being tested independently.

The tests are intended for regression and smoke testing, not for detailed diagnosis of problems. The idea is that the built-in tests detect errors; the developer or porter must then write his own custom tests to diagnose the error.

One common motif for testing, especially in the resampling code, is to compare the answers from a slow reference implementation that can be easily verified by inspection against a fast but obscure implementation which is used for production; to help ensure testing coverage, the test cases are often generated randomly.

7.2.20 C

Most of Pyvox, and all of the low level functions, are written in ANSI C (1999) 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.

The implementation is theoretically still compatible with the older 1989 ANSI C standard, but certain features will be omitted. However, no effort is being put into testing and maintaining C89 compatibility, and you should expect to have to work at actually getting Pyvox to compile and run under C89.

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, such as exceptions, 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.

7.2.21 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. A second and perhaps more important advantage of supporting only one scripting language is that non-performance-critical portions of Pyvox can then be written in that language rather than in C.

7.2.22 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.

7.2.23 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. However, many platforms will have optimized versions of BLAS and it seems foolish not to take advantage of these where available.

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, which is a subset of CLAPACK, a translation of LAPACK into C. 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.

7.2.24 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
        Adjust loop limits for image boundaries
        Loop over the voxels in the current row and neighbor
```

7.2.25 FIXME Notes

Pyvox is (and probably will always 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. The principle here is to be honest about the state of the code. 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 `TODO` file is a higher-level list of open problems, ideas, and issues.

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.

7.2.26 Generic Types and Pointers

Much of the C code in Pyvox is written to apply generically to most or all of the C numeric types, and makes heavy use of generic pointers and occasional use of generic variables. The following rules should make generics easier to work with (but are not guaranteed to be consistently followed). Pointers to variables or arrays of generic type should normally be passed as `void *` and converted to `unsigned char *` for address arithmetic; in the usual case, arrays will be densely packed with no space between successive elements. There is a special union type `anytype` declared in `exim.h` which defines a variable large enough to hold a single value of any numeric type; it is possibly to define arrays of `anytype`, but they will be sparsely packed and are not compatible with most other functions, which expect densely packed arrays.

7.2.27 Data Conversion

Numeric values in Pyvox can exist in any of four domains, and functions are available to convert between these domains.

The C domain is the most primitive. The data type and format are implicit in the variable name (via its declaration) and cannot vary over time.

The Voxel Kit domain support homogeneous arrays of uniform type; the type is defined by an auxillary variable, either contained in the voxel array struct or in some separate variable. All the numeric types defined by C are supported. (But pointers and chars with indefinite signedness are not supported.) There are three subdomains. The voxel array contains a homogeneous array of values plus a `voxel_array` struct that describes the array and specifies the data type. A scalar voxel array is a voxel array that has

Table 7.1: Parsing Python Arguments. These functions convert specific types of arguments in Python into a form easily useable in C code.

Name	Function
`PyvoxValue_Check`	Convert `PyObject*` to `anytype*`
`PyvoxLong_Check`	Convert `PyObject*` to `long`
`PyvoxLongSeq_Check`	Convert `PyObject*` to `long[ ]`
`PyvoxDouble_Check`	Convert `PyObject*` to `double`
`PyvoxDoubleSeq_Check`	Convert `PyObject*` to `double[ ]`
`PyvoxDoubleSeq_AsDouble`	Convert `PyObject*` to `double*`
`PyvoxComplex_Check`	Convert `PyObject*` to `double complex`
`PyvoxComplexSeq_Check`	Convert `PyObject*` to `double complex[ ]`
`PyvoxArray_Check`	Convert `PyObject*` to `voxel_array*`
`PyvoxArray_CheckSrc`	Convert `PyObject*` to `voxel_array*`
`PyvoxArray_CheckDest`	Convert `PyObject*` to `voxel_array*`
`Pyvox_CheckAxes`	Convert `PyObject*` to array axes info
`Pyvox_ParseShape`	Convert `PyObject*` to array shape info
`PyvoxIntype_Check`	Convert `PyObject*` to internal type code
`PyvoxExtype_Check`	Convert `PyObject*` to external type code

rank 0 and contains only a single value. The `anytype` union can contain a single value of any type; the type must be passed as a separate parameter to any function that supports this union. In addition, a pointer to an `anytype` union may be cast to the appropriate pointer type and passed to a pure C function that expects a pointer.

The Python domain supports a subset of the C numeric types (long, double, and double complex) and are always wrapped in a struct that specifies the type of the data.

The external domain supports numeric data in external data files in standard bit lengths and formats.

Tables 7.1 through 7.6 summarize the functions available for converting between domains; the C/Python conversion functions included the the Python C API are not listed here. See the source code for usage and calling sequences. Note that the names and exact functionality are subject to change, now that I see what a mess the current version is.

Table 7.2: Converting between C and Python. These functions will convert between a `PyObject *` and any of a C array element, C scalar variable, or an `anytype*`.

Name	Function
`pyvox_get_value`	Convert any C type to `PyObject*`
`pyvox_set_value`	Convert `PyObject*` to any C type

Table 7.3: Converting between C and the Voxel Kit. These functions support conversions between between elements of voxel arrays and C variables.

Name	Function
`bips_locate`	Compute pointer to element of a C array
`exim_set_value`	Set element of a C array from a `double`
`exim_get_value`	Get element of a C array as a `double`
`vxl_locate`	Compute pointer to element of a voxel array
`vxli_locate`	Compute pointer to element of a voxel array
`vxl_get_voxel`	Get element of a voxel array as a `double`
`vxl_set_voxel`	Set element of a voxel array from a `double`
`vxl_create_scalar`	Create scalar voxel array from a `double`
`vxl_store_scalar`	Initialize a scalar array from a `double`

Table 7.4: Converting between Python and the Voxel Kit. These functions convert between Python numbers and elements of voxel arrays.

Name	Function
`parray_get_voxel`	Get voxel array element as a `PyObject*`
`parray_set_voxel`	Set voxel array element from `PyObject*`
`varray_create_scalar`	Create scalar voxel array from `PyObject*`
`varray_init_scalar`	Initialize scalar voxel array from `PyObject*`
`PyvoxType_FromInt`	Convert numeric type code to `PyObject *`

Table 7.5: Converting between Python and external data formats

Name	Function
`pyexim_export`	Convert Python number to an external format
`pyexim_import`	Convert external format to a Python number

Table 7.6: Converting between C and external data formats

Name	Function
`exim_export`	Convert C values to an external format
`exim_import`	Convert external data to C
`exim_swap_bytes`	Change byte order in external data

7.2.28 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.

7.2.29 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.

7.2.30 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 Pyvox 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.

7.2.31 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.

7.3 Open Issues

7.3.1 Merging Pyvox and Voxel Arrays

The current distinction between Pyvox arrays and voxel arrays (i.e. between Pyvox and the Voxel Kit) is a historical artifact due to the original design decision to support use of the Voxel Kit from C, without requiring any use of Python. Given that quite a bit of functionality is now written in Python, it may be time to relegate this design decision to the dustbin of history and combine the `pyvox_array` and `voxel_array` structs into a single struct that combines their functions.

7.3.2 Commented Data Files

The ancestral C version of Pyvox had a notion of "commented data files," which were essentially plain text files including comments that could easily be ignored or removed. The idea is not used anymore in the new Python-based code but might be useful enough to reimplement in Python; on the other hand, it could be killed entirely. The relevant files are `src/cdata.c`, `src/decomment.c`, `include/cdata.h`, `man/man1/decomment.1`, and `man/man5/cdata.5`.

7.3.3 Parameter Files

The ancestral C version of Pyvox had a notion of "parameter files," which were essentially plain text files for specifying the parameters used for multiple programs in an image analysis protocol. The idea is not used anymore in the new Python-based code but might be useful enough to reimplement in

Python; on the other hand, it could be killed entirely. The relevant files are `src/param.c`, `param.h`, and `man/man5/param.5`.

7.3.4 Image Views

It is potentially useful to allow two voxel arrays to share the same data, possibly with different indexing schemes. The (so far hypothetical) mechanism for enabling this is called *image views*. Another application would be to allow access to the data of another array-like object (such as a PIL image or a Numeric array) without needing to copy that data.

There are some nasty problems involved in doing this well, and we haven't yet put much time into it. For example, changing the order of dimensions will create massive inefficiencies in a naive rowwise iteration. Another issue is that array-like objects usually assume that they manage their own storage and might move the shared data without notice.

7.3.5 Huge Images

We define a *huge* image as one which will not fit into virtual memory but must be handled as a mosaic of pieces defined by disk files. Pyvox does not attempt to handle these automatically, but it is an idea for the future. (Assuming that 64-bit machines do not obviate the need.)

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. 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 Pyvox 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 Pyvox.

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 Pyvox 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 the installable programs in Pyvox; man pages for the examples are included in the `examples` directory.

The `scripts` directory contains various small image processing applications written using Pyvox and which will be installed with the Pyvox package. It also contains a non-installable script `pycompile` which is used to byte-compile the Pyvox libraries.

The `src` directory contains the C source code for Pyvox 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 Pyvox.

7.6 Architecture and Code Organization

7.6.1 Voxel Kit

The Voxel Kit is a collection of higher level C functions for volume image processing, at roughly the level treated in image processing textbooks, plus various related functions. The Voxel Kit is instantiated in the shared library `libvoxkit.so` and may be called directly from C language programs, or used from Python via the Pyvox extension.

The image processing functions within the Voxel Kit are defined in 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.

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`.

Exim

Exim is a set of functions for translating between external (file) and internal (native) representations of data and is used to permit external two's complement binary integers, unsigned binary integers, and IEEE 754 floating point data to be read or written on any ANSI C 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).

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.

7.6.2 Pyvox

Pyvox is an image processing extension to the Python language written partly in C and partly in Python. It calls on the Voxel Kit and other libraries for much of the actual processing.

Pyvox defines the two public Python modules `pyvox.py` and `exim.so`. The first of these provides access to the image processing functions; the second to the data import/export functions of `exim`. In addition, there are two pure Python modules `optim.py`, which implments a set of optimization algorithms; and `regis.py` which implements a basic set of image registration methods.

The `pyvox` extension module defines part of its contents in Python, and calls a C language extension module `pyvoxC` to define the rest. PyvoxC 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`.

Since the `pyvoxC` and `exim` modules need to share some of their internals, they are thin wrappers that call a common shared library `libpyvox.so`.

7.6.3 Numerical Methods

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

7.6.4 Applications

...both Python scripts and CLI C programs

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

7.6.5 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.

7.6.6 Test Scripts

The `test` directory contains various scripts and programs for testing Pyvox; most of these are intended as regression tests to verify that Pyvox 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 `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` target runs a program to count the total number of lines of code in Pyvox, 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 Pyvox itself.

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

7.8 The Testmode Script

Once the `configure` script has been run, the home directory for this package contains a script `testmode` which will set the various path variables in the environment to point to the appropriate places within the home directory. Thus, after you source this script using the right shell command, the compiled but uninstalled programs, scripts, and libraries in the home directory will be preferred to any installed versions. This is a great convenience for testing and debugging the code.

7.9 Coding Style

7.9.1 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

7.9.2 The Rules

• The package should be distributed as a tar file which is named in the format `pyvox-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 `pyvox-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 to 2nd ext2fs file system

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;
}
```

7.9.3 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?

7.10 Coding Hacks

This section describes some known bugs in the code that Pyvox relies on, and how Pyvox arranges to work around them.

7.10.1 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 we just haven't actually done it yet; some preliminary work suggests that there will be few problems in doing so, except for Red Hat's perverted practice of naming version 1.5.2 as `python` and version 2.x as `python2`. 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.

7.10.2 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.

7.10.3 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; Pyvox does not currently contain any such programs but might in the future.

7.11 Release Checklist

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

1. Build and regression test on the current test platforms. Fix or explain away any bugs.

2. Update the `NEWS` and `README` files as needed. Commit them to the repository.

3. Choose the new version number and update the `pyvox.ver` file.

4. Regenerate the files `doc/pyvox.pdf` and `./configure` if necessary and commit to the repository.

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 `pyvox-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. `pyvox-nn.nn`.

7. Tar and gzip, in the form `pyvox-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 `/img/devel/distrib`.

10. Add to the BBL website and update the web page.

11. Email announcements as appropriate.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[2] David M. Beazley. *Python Essential Reference*. New Riders, second edition, 2001.

[3] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. 1995.

[4] Samuel P. Harbison and Guy L. Steele, Jr. *C: A Reference Manual*. Fifth edition, 2002.

[5] Donald Lewine. *POSIX Programmer's Guide: Writing Portable UNIX Programs*. 1991.

[6] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Second edition, 1992.