Thoughts on Computational Toolsets for Neuroimaging

Robert W Cox, PhD — rwcox@mcw.edu

1. Introduction and Issues

The purpose of this document is to provide some framework for discussion of the future path that functional neuroimaging software development should take. My own experience in this area started about 6 years ago, with the creation of the FMRI analysis package *AFNI*. I am not trying to write a master plan, but rather am laying out my opinions on the structures that need to be defined and developed.

One underlying assumption for my thoughts is the belief that most neuroimage analysis software will be created in non-commercial environments. Reasons for this include the relatively poor* market for the software, the rapidly changing needs of the users, and the need for openness in scientific work. These programs are intended for scientific data analysis, not for potential clinical applications of FMRI or other methodologies. Scientists will always want to know precisely how the processing methods work, will want to try different methods, and will need to analyze larger, more complex data sets to answer larger, more complex questions.

*In size and in money

The major themes of this document are a collection of *-ilities* that a collection of software tools for neuroimaging should possess:

- [**i**] *Flexibility* ≡ can be used in ways not clearly foreseen by the designers.
- [**ii**] *Extensibility* ≡ is 'easy' to add new features that fit into the toolset.
- [**iii**] *Interoperability* ≡ can combine different tools without major pain.
- [**iv**] *Survivability* ≡ has long term utility.
- [**v**] *Portability* ≡ design/implementation is not bound to ephemeral platforms.
- [**vi**] *Usability* ≡ basic operations of each tool are 'easy' to access.
- [**vii**] *Scalability* ≡ has a path to deal with large and intricate data collections.

These themes are not all independent, and no doubt a few more could be tacked onto the list.

The major issues facing us are the obstacles to achieving these goals:

- [**a**] *Data Interchange* ≡ data elements, domains, formats, input mechanisms.
- [**b**] *Execution Interchange* ≡ how to get different program units to work together?
- [**c**] *Monolithic Systems* ≡ many tools in one package, not cleanly separated.
- [**d**] *Hidden Information* ≡ poor documentation on usage, algorithms, implementation.
- [**e**] *Poor Organization* ≡ lack of generality and modularity in design.
- [**f**] *Clumsy Implementation* ≈ many tools are written by graduate students.

To my mind, the most urgent issues are [a] (§2 [p.2] and §3 [p.5]) and [b] (§4 [p.12] and §5 [p.13]). I will spend much of the rest of this rumination on these points; in §6 [p.16] I will summarize my personal judgments. The other issues listed above must be addressed in the design of the data and execution models.

> Web addresses of various things that I mention along the way can be found at the end of this document in §7 [p.16]. Examples from *AFNI* are displayed in smaller type, as in this paragraph.

2. The Content of Our Data

There are more subtleties to FMRI data than are apparent from the outside. Not all of the structures I will describe are used by major neuroimaging software tools at present, but the potential for applying them must not be foreclosed. Issues that I will address in this section include:

[**a**] Atomic data types—what kind of data can be stored inside an object?
[**b**] Data conformation—over what kinds of regions can data be defined?
[**c**] Data assembly—how can multi-object collections be specified and assembled?

§3 [p.5] will address the issue of how these data structures are to be stored. §4 [p.12] will discuss how data can be passed between program units.

2.a. Atomic Data Types

This refers to the kinds of information that can be stored at each node in a dataset. At present, most tools deal with simple numbers as the atomic data types (e.g., 16 bit shorts, 32 bit floats). Allowing for more complex data elements will necessary, both to hold new types of imaging information and to hold the results from complex processing techniques.

One type of data that must be supported is *categorical* data, which means each value is from a discrete set, usually with no concept of arithmetic or ordering on the elements. Another way to think of this is as attaching a qualitative label to each node. **Applications**: **[i]** Brain parcellation and region-of-interest (ROI) selection, in which each node is categorized in some way (automatically or manually). **[ii]** Acquisition of subject responses, which often have a categorical component (e.g., the left response button was pressed). (Not only brain image data needs to be stored, but also many forms of auxiliary data; a unified data hierarchy should be used for everything.) Each value would be represented by an integer, which would also point to a descriptive string. In this way, one could have a string-valued 3D volume.

Another important extension is the support of vector- and matrix-valued datasets, where a vector or a matrix is stored at each point. **Applications**: **[i]** Storing data from multi-image acquisition methods—the GREASE sequence at MCW can gather up to 4 64×64 images in a single shot (about 150 ms); each image has a different mix of T_2 and T_2^* weighting. **[ii]** Storing color datasets as RGB-valued vectors—these will be useful for including atlas data. **[iii]** Storing the results of time series modeling in each voxel: the parameter estimates form a natural vector value, and their covariances form a natural matrix value. **[iv]** Storing the results of diffusion-tensor imaging, which has shown promise in the mapping of cerebral white matter tracts.

Since vectors and matrices are realized by collections of numbers, it would seem that the multiple 3D volume (sub-brick) structure already present in *AFNI* datasets can support these data types. This is true; however, at present, there is no infrastructure for manipulating vector- and matrix-valued datasets, so it would have to be done on an *ad hoc* basis for each new analysis tool. I am proposing that this infrastructure be defined, so that creating, manipulating, and visualizing a vector-valued dataset is a well-documented and routine set of operations. This is the importance of supporting structured atomic data types.

Another application of structured atomic data types would allow more complicated statistical estimates to be stored at each point, including confidence intervals, and allowing basic distributional statistics to have voxel-specific distributional parameters. **Applications**:

[**i**] The bootstrap method is a computationally intensive resampling technique for estimating the goodness-of-fit of a model, making fewer assumptions than are usually needed in statistical analyses. The results can't be expressed as being drawn from some classical distribution, and are usually reported using confidence intervals. [**ii**] t-statistics computed with unpooled variance estimates have the number of degrees of freedom estimated in each voxel, so a brick-wide value is not applicable. The same remark applies to correlation coefficient estimates (β distributed) when the voxel data are correlated in time.

> *AFNI* now supports the statistical interpretation of the values stored in a dataset; for example, the values in a sub-brick can be t-statistics, with a brick-wide number of degrees of freedom; *AFNI* can then convert the t-statistic into the equivalent p-value. (Nine families of distributions are supported—those contained in the CDF library from U Texas.)

2.b. Data Conformation

An important capability is to generalize the types of regions over which a dataset can be defined. This is needed to store and display results from selected sub-regions, some examples of which are given below. Abstractly, a dataset is a mapping from some domain (region) into some set of possible values. The basic domain types should include

- [**i**] Rectangular 'volumes' defined in 1–3 dimensions; **Applications**: Time series, planar and volume images.
- [**ii**] Non-rectangular 'volumes'; **Application**: To store extracted subsets of volume data, such as used-defined regions-of-interests, clusters of active voxels, and atlas-defined brain structures.
- [**iii**] 1D curves and 2D curved surfaces, as subsets of 3D space; **Applications**: (1D) To mark and store data from specific curves in the brain, such as the crown of a gyrus; (2D) To represent the cortical surface, manually or automatically extracted from volumetric data.
- [**iv**] Discrete point sets within 3D space; **Applications**: To store data gathered at arbitrary points, such as EEG time series, or data subsets gathered by some criteria.
- [**v**] Non-continuous categorical 'dimensions' (see below for applications).

In addition, domains could be defined as tensor products of any basic domain type with a rectangular volume (type [i]) or a categorical dimension (type [v]). (A point in a tensor product of two basic domains is uniquely specified by picking out a point in the first basic domain, and then picking out a point in the second basic domain. For example, the domain of a 3D+time dataset is the tensor product of a 3D volume with a 1D time interval; each point is given by its 3D coordinates (x, y, z) and its 1D coordinate (t).) At each sample point in the product domain the system should be able to store any of the data types described in §2.a. **Applications**: [**i**] The tensor product of a 2D surface with a 1D interval will allow the creation of time-dependent datasets defined over the cortical surface. [**ii**] The tensor product of a 3D volume with a 2D volume will allow the creation of datasets from time-dependent 3D imaging spectroscopy—the first 3 dimensions will be the image coordinates, the second 2 dimensions will be spectroscopic frequency (Hz or ppm) and time. [**iii**] Categorical dimensions are intended to allow the stacking up of multiple basic values at each dataset point; for example, the result of a time series fit could be a vector of estimated parameters stacked up with their estimated covariance matrix.

It is an open question as to whether topologically more complex data domains (e.g., tori, spheres) are required. The circular domain may be useful for some purposes.

Domain objects will be a key component of any future data format. In many functional imaging applications, the principal desired result is a map of active regions, or a segmentation of the anatomy. Such results will be directly representable as domain objects. The applications listed above show the importance of being able to operate on and produce geometrical structures as data, or as components of data.

The first 3 domain types are conceptually continuous, but must be realized discretely on the computer.

> In *AFNI* (which only supports domain type [i]), a 3D dataset can be interpolated on demand to a new resolution; in this respect, *AFNI* datasets are considered to be defined over a continuous rectilinear domain rather than just being an array of voxel values at forever fixed locations. This capability originally came about because it was necessary to overlay functional images onto anatomical images that were gathered on incommensurate grids, and to transform both types of data to the Talairach coordinate system for intersubject comparisons.

The philosophy that the data is defined over an underlying continuum should be continued. This will allow mapping from 3D volumes onto 2D surfaces (restriction), and vice-versa (prolongation).

> For linear dimensions, *AFNI* currently supports only regularly spaced grids (with a possible spatially dependent offset for the time-axis in 3D+time datasets, to allow for the uneven times at which slices might be gathered).

Nonuniform grids should be allowed in future processing systems, at least for one dimensional domains. **Applications**: **[i]** An uneven image acquisition interval, gated to the cardiac cycle, has been used to suppress much of the physiological 'noise' in FMRI of the brain stem. **[ii]** Many event-related FMRI experiments use event timing that is not commensurable with TR.

Irregular volume domains (type [ii]) can be implemented in several ways. One straightforward approach will be to 'shrink-wrap' a rectangular volume about the desired domain, and then mark as unused the points outside the desired non-rectangular region. In this way, the codes for implementing irregular domains will be the same as for rectangular domains, with a little postprocessing to allow for the unused points.

The [iv]$^{\text{th}}$ and [v]$^{\text{th}}$ domain types are discontinuous—there is no unambiguous concept of changing resolution and interpolating between sample points in these cases. Type (iv) is intended to support applications such as EEG, where time series data is gathered at an irregular set of points in 3D. In this example, it makes sense to propagate the data into other parts of 3D space; however, the methods required are application-specific.

A systematic hierarchy of data classes should be created to package up data values and data domains in a unified way. This will make further development and extension of the data objects needed for functional brain imaging much simpler and much more powerful than it is now.

2.c. Data Assembly—Collections of Imaging Datasets

As larger collections of subjects and tasks are gathered and analyzed together, the need for a systematic way to collect image datasets for analysis is growing.

> We are facing this difficulty in *AFNI*, which has the ability to do 2- and 3-way ANOVA (voxel-by-voxel inference). The number of 3D datasets in any given analysis often exceeds 100, which must all be specified individually on the command line or in the script file. At present, each investigator deals with this by coding information about the data in the filename, and then uses the filenames to select which datasets go into a particular analysis. This *ad hoc* solution is only weakly tenable, and things will only get worse.

My first thought was to allow the user to include descriptor strings in a dataset that could then be used for selection; effectively, allow something like "*all 3D activation datasets with subject fields* `HANDEDNESS=RIGHT` *and* `SEX=MALE`" to specify which datasets are used for a specific purpose. After reflection, I realized that this is nothing more than distributing a database of information about datasets into the dataset headers.

As a site progresses from small FMRI studies to large ones, the scientists generally discover the necessity of keeping track of their experiments and subjects with a database. The effect of requiring header fields that describe the experiment would be to duplicate partially the experiments database.

One alternative is to develop the ability to assemble collections of datasets by interrogating an external database. The database should return pointers to datasets that meet user-specified criteria. The database will not actually contain the volumetric (etc.) data.

In any case, after the datasets are collected, they must be packaged up. I see no reason why this packaging should be different from the other packaging I'm talking about in this document. That is, a collection of datasets would be described as a dataset itself, whose 'atomic' values are datasets. For most purposes, the domain of the collective dataset would be described with categorical dimensions, although it is possible to conceive of uses for the continuous domain types (e.g., results from experiments where some task parameter is varied continuously).

3. Data Interoperability—Defining a 'File' Format

Some people have argued that it is impossible to get the neuroimaging community to agree on a data file format. I disagree; I believe that if the top 3–4 sites and 2–3 platforms agree on a format, if it is well documented, if it is not too difficult to use, and if basic tools are created to manipulate this format, then most other sites will fall into place. Furthermore, I believe that this is a prerequisite for any scheme to unite software tools from multiple institutions.

My main point is that data transfer between tools will almost inevitably require the development of a master data format, and this will be both the biggest problem and the biggest achievement (if it is achieved). Standardizing methods for invocation of tools (at least of certain classes), while not trivial, will be a less difficult task, although it will provide the final sizzle that makes cooking the bacon so pleasant.

Consider the example of Mosaic, the first widely distributed point-and-click graphical Web browser, and the direct ancestor of Netscape and Internet Explorer. This was actually a minor achievement built on top of the standards for image formats (GIF and JPEG), text formatting (HTML, a subset of SGML), data transfer (HTTP, built atop TCP/IP), and

method invocation (CGI)—all of which existed before. Mosaic glued it all together and made it fun, but neuroimaging software developers aren't yet at the final gluing stage—we need to assemble a few more standard pieces before we can apply the glue to get the sizzle.* Without standards, even Internet applications don't talk to each other; as an example, consider the current state of streaming video, with 3 incompatible proprietary formats.

*Don't ask me to cook

Issues that I will address in this section include:

[**a**] Prescribing content—is it reasonable to 'make' people use a data format?
[**b**] 'Must have' tools—what capabilities will make this format the winner?
[**c**] Format design—how to be easy to use *and* extensible?
[**d**] Basic operations—what is needed to get off the ground?

The word *File* is in quotes in the header for this section since data collections do not need to be accessed from disk files. One useful alternative input source is an Internet connection (a stream of data bytes), perhaps using the HTTP mechanism.

3.a. Prescribing Content?

The objection to developing a format for data exchange is the potential perception that we would be prescribing the allowed content of data sets. But this problem will arise with any conversion tools that are applied to allow data to flow from one analysis package to another. Not all formats and types of data described in §2 [p.2] will be supported—especially at first—so anyone whose great new method would benefit from an unsupported feature will essentially be having his content edited from the start. A specific example:

> *AFNI* contains a plugin to estimate the time delay of the hemodynamic response in each voxel. The result takes into account the different time origin of each slice. *AFNI* allows each slice to have a separate temporal offset, and these can be arbitrary (e.g., uniformly spaced in time interleaved slices are not mandatory). This extra auxiliary information is stored with each time-dependent dataset, and is used in the time delay estimation module.
>
> Suppose that someone wants to use *AIR* to register the volume time series, and then use this *AFNI* module to do the time delay estimation. *AIR* will not get or deal with or preserve the slice time offset information—why should it? This means that the data conversion tool must preserve this auxiliary data across the invocation of *AIR*, then reattach it to the volumes returned by *AIR* before sending the image time series to *AFNI* . This would be a complex undertaking. On the other hand, if the time offset for each slice is *not* handed to *AFNI*, then the time delay estimates will be wrong.

The only practical way I can see for an interoperability conversion toolset to deal with such issues is to define its own format for storing neuroimaging data—a format that can encompass the range of data domains, data types, and auxiliary information discussed in §2 [p.2], and a format that is extensible to include new types of information and operations. Then the conversion toolset can take what is needed out of its own file to create files that a particular tool can read, then take the files that this tool produces and fit them back into its own format.

However, if such a format is developed, why shouldn't toolkits use it directly? If the format doesn't allow certain kinds of content, then it will not be able to deal with toolkits that need that type of information. This is an argument for creating an extensible format, not an argument for eschewing the idea for a standard format.

3.b. What Tools Will Win the Hearts and Minds of the Neuroimagers?

Before I describe the requirements of a neuroimaging data format, I will digress into a vision of some tools that will help the adoption of a standard format:

- [**i**] The ability to easily assemble and disassemble structured data files into simpler components (e.g., volumetric time series ↔ individual image slices).
- [**ii**] Easy to use tools for visualizing complex datasets and making publishable figures.
- [**iii**] A Web browser interface that will allow researchers to 'publish' their results to the Web in the form of viewer-controlled images, renderings, and animations.

3.c. Format Design

Datasets are defined by the atomic values stored at each node, and by the domains into which the nodes are gathered. Atomic data types must be equipped with basic operations on individual values, and domains must be equipped with operations that allow basic collections of atomic values to be extracted.

At the same time, neuroimaging datasets come with lots of auxiliary information:

> *AFNI* dataset headers contain a log of the processing commands that created them, can contain arbitrary text notes added by the user, can contain a set of matrices/vectors describing their geometric relationship to 'parent' datasets (i.e., the transformation to Talairach coordinates), etc. In *AFNI*, all such information is stored in the form of *attributes*, which are named arrays of `int`s, `float`s, or strings. Each attribute is stored in the header file with its name, its element count, and an ASCII representation of its elements. When a dataset is read, the attributes are all read in. Those that are needed for dataset construction are probed to set up the data arrays, etc. Those that are not needed are ignored. In this way, a new processing module can add a new attribute to dataset headers to store some new type of auxiliary information; the presence of this new attribute will not interfere with the usual dataset operations in any way.

The *AFNI* dataset format is not flexible enough for the many needs outlined in §2 [p.2]. Instead, I propose the development of a format based on XML, possibly building on the Caltech XSIL project. XML is a language for describing and storing hierarchical data files. The files can be constrained in structure using a DTD file, or can be free to contain any set of nested elements. XML does not provide processing itself, but there are a number of codes available for parsing XML documents. (XSIL is an XML DTD and a set of Java routines for processing XML files.)

XML: eXtensible Markup Language

DTD: Document Type Definition

XSIL: eXtensible Scientific Interchange Language

URL: Uniform Resource Locator

XML elements need to be designed to contain each of the data types described in §2 [p.2]. Housekeeping auxiliary elements (e.g., like *AFNI*'s processing log) need to be defined as well. The actual voxel data need not be stored in an XML file. Instead, the file can contain URLs indicating where the data can be found. (Binary data cannot directly be stored in the XML format, but can be encoded into text using the base64 scheme, which takes up about 35% more space and approximately doubles the time to read a large file and store it in memory.)

An example of what a simple XML header file for a 3D volume might look like is

```
<?xml version="1.0" encoding="UTF-8"?>
<RECT3D n1="256" n2="256" n3="124" datum="short"
        byteorder="MSBfirst" src="elvis.spgr">
  <!- The HISTORY and NOTE elements below are optional ->
  <HISTORY>
    <NOTE date="04 Jan 2001" user="scan@signa">
       Acquired at Froedtert Memorial Lutheran Hospital</NOTE>
    <NOTE date="07 Jan 2001" user="rwcox@manwe.biophysics.mcw.edu">
       3dBlur -input=buddy -output=elvis -fwhm=5.25</NOTE>
  </HISTORY>
</RECT3D>
```

Note the nested structure of elements, each of which starts and ends with a labeled tag. This structure is what makes XML relatively easy to parse and validate for syntactic correctness.

VRML: Virtual Reality Markup Language

To the extent possible, we should build on relevant pre-existing standards. As an example, the VRML standard can be used as a way of describing 1D and 2D curved domains embedded in 3D space.

Development of such a format for storing data is not enough. There is also the issue of getting the data into a program. This subject will be addressed in §4 [p.12]; at this point, I will say that a parallel development of data structures for the major programming languages (C/C++, Java?, ...) should be carried out. There will be no requirement that developers use these data definitions to store the data read from a dataset file, but these definitions will be available to make programming life a little easier.

3.d. Basic Operations on Data 'Files'

Besides providing a definition for how data should be stored external to a program, some basic utilities for manipulating such files need to be provided. These include

- [**i**] Data display—tools for looking at a neuroimage data file.
- [**ii**] Data integrity—tools for checking a neuroimage data file.
- [**iii**] Data operations—manipulations on atomic data types.
- [**iv**] Data extraction—what types of 'chunks' of data can be extracted from an object?
- [**v**] Data insertion—what types of chunks of data can an object accept?
- [**vi**] Data conversion—to what other types of objects can a given object be converted?
- [**vii**] Data selection—how can subsets of an object be specified for extraction?

I envision these as standalone programs which are basically wrappers for executable modules that can be incorporated into other programs. These utilities will be useful in themselves, providing a way of producing standard-formatted output files from standard-formatted inputs. In addition, they could be incorporated into scripts or other execution paradigms (§4 [p.12]) to assemble more complex capabilities.

3.d.i. *Data Display*

Viewers differ from other processing tools only in that they do not usually produce output that goes back into the dataset file format, but simply consume data and produce images formatted for viewing on the screen. Each dataset domain type and atomic data type should have a basic image generation utility. Each utility will take as input a set of visualization parameters (dependent on the domain type), data extraction parameters, and produce as output an image file in some standard format (e.g., PPM, PNG, JPEG). Display of the image will be a separate system-dependent issue.

3.d.ii. *Data Integrity Checking*

A utility is needed to check a dataset file for integrity. These checks would include consistency of the XML document, and existence of external files and the presence of valid data in them. Since datasets might contain references to other datasets (§2.c), it will also be necessary to check collections of files for consistency.

3.d.iii. *Data Operations*

Each atomic data type will generally come with an underlying set of operations: numerical scalars support arithmetic; vectors support addition, scalar multiplication, dot products, and projection; matrices support more operations than can be listed here; and statistical types support various inferential and conversion (e.g., t-into-z scores) operations. Utilities should be written to carry out these operations on datasets; for example, a routine to compute the product of a matrix-valued and a vector-valued dataset, producing a new vector-valued dataset. I see these utilities written as wrappers for functions that take as input collections of atomic objects and produce as output new collections of objects. The ability to operate on many atomic objects at once is important for the sake of efficiency—in principal, everything could be written in terms of operations on single objects (e.g., a single matrix-vector multiply), but in practice, this would be very inefficient when millions of the same operation need to be executed.

A new atomic data type may require a special interpolation algorithm to preserve its structure when it is resampled from one data domain to another. For example, standard interpolation methods applied to each component of a unit vector will not usually result in a unit vector. This data type might be used to store the results of principal direction analysis of a diffusion tensor image series.

3.d.iv. *Data Extraction*

A powerful set of functions for extracting data from datasets is necessary for two reasons. First, the user needs to be able to visualize pieces of his data in several different ways; for example, as 2D slices along an arbitrary cut plane, and as 1D or 2D graphs along arbitrary cut lines or planes. Second, a programmer needs to be able to get the pieces of a dataset that are most convenient for his new tools; for example, a routine to display a projection of the data would most naturally operate on 1D subsets of the data.

There are two different meanings to extracting values from a dataset: collecting atomic values from subsets of the dataset's domain, and getting individual numbers (or other values) out of atomic values when these are themselves structured types.

At the most basic level of subset extraction, it must be possible to extract a single value from a dataset, given a description of where it is to come from (i.e., its coordinates, or its array

index). In principle, this is enough: it would be possible to build all data processing on the basis of extracting one voxel value at a time. Such procedures would be very inefficient, and since large-scale data processing is one of our major themes, more efficient data processing models must be implemented.

> In *AFNI*, there are routines for extracting 3D volumes and 2D slices at any orientation (at a fixed time), and 1D time series (at a fixed location) from 3D+time dataset. Which routine is used depends on the processing 'quantum'—the basic chunk of data that the algorithm requires, or that the programmer finds convenient to deal with. For example, the *AFNI* plugin that computes the Fourier time-to-frequency transform of a 3D+time dataset uses the 'extract one 1D time series' model, since no intervoxel computations are needed. *AFNI* supports only a limited number of different dataset quanta for its basic rectangular datasets.

Data subsets need to be able to be extracted on geometric criteria. From a volumetric dataset (types [i] and [ii]), it should be possible to extract nD subsets along any cut direction, for $n = 1, 2, 3$. From a curved dataset (type [iii]), it should be possible to extract subsets that pass through certain types of 3D regions (e.g., the intersection of the surface with a plane or volume). It should also be possible to extract data from one dataset based on geometrical criteria from another dataset. (The data selection methods described in §3.d.vi could be implemented by producing domains that map out the desired data; the application of these intermediate domains to a given dataset is an example of extracting data from one dataset based on criteria from another.) These abilities will make it possible for the programmer to deal with the quantum of data that he needs, and make it possible for the user to display the data that he believes is relevant.

Extraction of data subsets will usually be done by making copies (perhaps interpolated on-the-fly to a new resolution). Wrapper utility programs will write new datasets with the extracted data values and the appropriate domain description.

In *AFNI*, the user/programmer can choose between 4 different 3D interpolation schemes when data is resampled to a new resolution. For some types of data (e.g., categorical data such as tissue type labels) nothing but nearest neighbor resampling makes any sense. For data types that support addition and scalar multiplication, more sophisticated interpolants such as linear, cubic, and sinc will be provided. Some types of data may require special purpose interpolation; for example, if the value is a vector given tissue-type fractions in a voxel (gray-matter, white-matter, and CSF), then cubic and sinc interpolation applied component-by-component can produce unrealistic negative values. For this reason, atomic data type extensions may need to implement their own interpolation methods, or to restrict the use of the default methods supplied by the system.

Getting individual numbers from structured types is the other level of data extraction. For example, if a dataset's atomic type is a vector, the user may wish to make an image of the first component, or of the magnitude; if the atomic type is 'diffusion tensor', an image formed from the ratio of largest to smallest eigenvalues would be useful for visualizing the microscopic tissue anisotropy. The extraction operations allowed on each dataset atomic type must be implemented efficiently, which means that the routines doing the work must be prepared to operate on arrays of data values, not just a single value at a time. In this way, a large number of structured data values can be extracted from a dataset at once using the methods described above, and then sub-values can be extracted from them *en masse*.

3.d.v. *Data Insertion*

Converse to data extraction is the issue of data insertion. This arises when constructing a new dataset (e.g., the average of some input datasets), or extending an existing one (e.g., adding a new point in time).

> A useful feature in the *AFNI* plugin utility library is a routine that applies a user-supplied function to each individual voxel time series in a 3D+time dataset. The input to the user function is the time series; the output is a single number that gets stored at the corresponding voxel in a new 3D dataset. It is thus very easy to create a new plugin that processes datasets in these 1D (time) quanta, since the *AFNI* utility takes care of all the work of creating the dataset and filling it with the results, one voxel at a time.

Data insertion utilities will extend this idea by allowing a new dataset to be 'grown' in various ways: not just one voxel at a time, but in 1D, 2D, and 3D chunks as well. Combined with the extraction routines described above, this will make it easy to add new tools to the system with a minimum of overhead—the system will deal with the overhead of memory management and dataset creation in pieces.

3.d.vi. *Data Conversion*

Like extraction, conversion can be between dataset domain types, and between dataset atomic value types. **Applications**: **[i]** If the cortical surface is extracted from a high-resolution anatomical dataset, then a dataset defined over a 2D curved domain can be created. A geometrical mapping between the volume domain and the 2D surface domain must be implemented to allow functional values created from 3D datasets to be visualized and processed on the surface. **[ii]** Converting a 3D+time dataset to a 3D dataset by computing the mean over time of every voxel is an example of collapsing a domain along a dimension. Computing the power spectral density (PSD) is an example of converting a 3D+time dataset into a 3D+frequency dataset, which has the same dimensionality but a different domain. **[iii]** Taking a 3D+frequency dataset from MR imaging spectroscopy and converting it into a 3D dataset with a vector of estimated chemical concentrations is an example of collapsing a dataset along 1 dimension and also converting the data values into a new atomic type.

3.d.vii. *Data Selection*

Data selection refers to the methods that specify which data is to be extracted, based on the values in the dataset. A key ability will be a set of tools for the user or programmer to select data subsets for visualization or further processing. The table to the right shows the basic types of selection. Combination of selected data regions

Spatially Selective	Temporally Selective	Description
No	**Yes**	Stimulus condition; this may be quantitative or qualitative (e.g., all times when the stimulus was type A)
No	**Yes**	Relationship of stimulus condition now to other stimulus conditions (e.g., stimulus A after stimulus B)
Yes	No	Dataset value in some range (e.g., percent signal change $> 0.5\%$)
No	**Yes**	Subject responses and/or physiological measurements in some range (e.g., response to a forced choice was incorrect)
Yes	No	Spatial contiguity to other selected voxels (i.e., clustering)
No	**Yes**	Motion or other artifact detected above some threshold

using Boolean operations (i.e., `OR`, `AND`) should also be allowed. I don't see any utility in combining spatially and temporally selective operations (but I could be convinced otherwise).

4. Execution Model

The issue is how different processing modules, utilities, programs, etc., are invoked, with their inputs delivered appropriately and their outputs collected for further analysis, display, or storage. Possible techniques include:

- [**a**] Command line programs—read and write standard-formatted files.
- [**b**] Remotely invocable objects/methods—essentially a collection of subroutines and a protocol for invoking them (possibly across a network).
- [**c**] API—development of a library of subroutines and a standard interface for invoking them.

These solutions can be made to work together. In particular, any useful solution will require development of an API to interface to dataset file I/O (including network transport) and at least some of the basic dataset operations listed in §3.d.

The problem is getting a large set of different components to work together in ways that are not foreseen. Some solutions are very dictatorial, which can make everything plug together nicely at the cost of making all developers adhere to a rigid and complex coding scheme. Solutions that are easier for the developer (e.g., [a]) may be hard to combine efficiently, or at all.

COM: Component Object Model

CORBA: Common Object Request Broker Architecture

XML-RPC: XML Remote Procedure Call

SOAP: Simple Object Access Protocol

IDL: Interface Description Language

4.a. Command Line Programs

Many people will limit themselves to developing this kind of utility. The reason is that it can be straightforward, provided there is support in the development language for reading, writing, and accessing standard-formatted dataset structures. One advantage from the point of view of system integration is that catastrophic failure of a separate program is unlikely to bring down the processing system. This is definitely *not* the case for routines that are linked into a master program. Considering that many programs are and will be written by inexperienced programmers, being insulated from the worst effects of bad code is a strong positive.

Command line programs can be invoked by other programs. To aid in this, the inputs they expect and the outputs they produce must be well-documented. Preferably, this would be done in a formalized way, so that other software can read how to use the programs from a specification file. More on this problem of defining interfaces in the next section ....

4.b. Remotely Invocable Objects/Methods

There is a number of schemes/standards for invoking functions, possibly executing them on remote systems:

- [**i**] COM—a Microsoft-specific solution.
- [**ii**] CORBA—a platform-neutral solution promulgated by an industry consortium.
- [**iii**] XML-RPC and SOAP—more recent and less intricate proposals along the same lines.

The first two are fairly complex, and are used primarily for software executing on homogeneous server farms. COM has not been adopted by Unix vendors,[†] and thus is of little interest to many.[‡] CORBA requires implementation of an Object Request Broker (ORB) program

[†] From fear of Bill Gates, no doubt

[‡] e.g., me

on the server computers. Fitting a processing module into the CORBA framework requires a writing an interface description in the CORBA IDL, and writing a wrapper routine to identify the module to the ORB. ORBs from different vendors don't talk to each other with complete fluency.

XML-RPC and SOAP are protocols that implement scaled-down versions of the concepts in COM and CORBA. They rely on using XML as the data transport mechanism and using HTTP as the inter-system communication protocol. Their advantages are that running a Web server is easier than running an ORB, encoding data using XML is machine- and language-independent, and that the software infrastructure is much smaller and simpler. Their disadvantages are that both protocols are new, not really complete, and untried. In addition, there is little room for sophisticated optimization using XML-RPC or SOAP, unlike with COM or CORBA (which might package up numerous requests to reduce communication overhead, and would be able to use more efficient intra-CPU execution techniques when a method is known to reside on the same computer as the program making the call).

For reasons of simplicity, I believe that XML-RPC or SOAP should be considered for use in neuroimaging software inter-operation. COM is too vendor-specific, and CORBA is too heavyweight to get working at many sites—most academic neuroimaging sites won't be able to get CORBA ORBs working well, in my opinion. For the sake of efficiency, a local transport mechanism might be added to the TCP/IP mechanism (e.g., using files or shared memory).

One thing that is good about CORBA is the IDL. This provides a language-neutral way to specify the data that a method expects to receive and that which it will return. Something like this must be adopted to enable interoperability of many different software tools. Perhaps each program/module would 'publish' a template dataset that would specify what kind of file (or other parameter) is acceptable for each input.

Note that a command line program could be invoked by another program (even remotely) using any of the above protocols, if the proper wrapper is created.

4.c. API—Application Programming Interface

Providing an API will be necessary, since just having a specification for how datasets are stored in files isn't enough. COM and CORBA provide APIs—AKA language-bindings—as well as communication services (XML-RPC and SOAP do not yet have APIs). It is possible to go farther than this, and to provide a rich and well-documented library of functions. Of course, such an project is a major effort. One major difficulty is that using an API requires the programmer to adhere to standards for internal structuring of his data. This will be very hard to coordinate across academic sites—harder than a dataset file storage standard, since that is really a peripheral issue, whereas data structures are at the heart of the programming effort.

5. User Interface

The interface that the user sees is important since it affects how he thinks about his data and tasks, and affects how easily the system is to learn. Alternative interface designs include

[a] A custom GUI for each program/module.

[b] Plugins—a way to add subroutines and user interface elements to a master program.

[**c**] Plugouts—development of a protocol for programs to talk to each other through some I/O channel (e.g., TCP/IP).

[**d**] A processing language, in which datasets and processing tools are given named, and actions are ordered by typing command strings (or composing them from menus).

In addition to dataset information (as outlined in §2 [p.2]), a user interface typically stores extra information about the user interaction. For example:

> *AFNI* maintains the idea of a 'focus point'—all the various image windows can be locked to this point, so that the user is looking at the same location in all views.

A complex program will have many such user state variables. An interactive system must have a way of communicating this type of information among modules, as well as sending dataset information around. When the user changes some variable, modules whose functioning depends on this variable need to be informed. The paradigm that has developed for dealing with this kind of situation is event-driven programming. Modules register the events (usually consequences of user actions) that they need to be informed about with a master function. The master function—the 'event-loop'—invokes each module when the appropriate conditions arise.

To add truly interactive capabilities to a program, the new modules must fit into the event-loop interface/protocol of the parent code. For this to be possible, the internal user state variables of the parent program need to be documented and augmentable. For multiple independent developers to agree on this type of cooperation between program units usually requires some external impetus. This can be provided by a vendor or vendor consortium, or possibly by a large customer or an independent standards body.

Batch and Interactive Modes

Some calculations will always be too lengthy to be implemented interactively. Other calculations may be rapid individually, but will have to be executed repeatedly (e.g., blurring a collection of datasets, each of which takes 5 seconds). For these purposes, as well as for documenting the processing trail, the ability to run groups of operations in batch mode is needed. On the other hand, interaction is very useful when learning a system, and when exploring the effects of changing processing parameters or methods.

It is thus desirable to have all processing tools be available in both interactive and batch modes.

> This is currently not the case in *AFNI*. There is a large collection of batch mode programs, which are run from the command line (or script files). Then there is the collection of tools within *AFNI* itself, which are controlled by mouse clicks. For example, to write a transformed functional dataset to disk after the Talairach transformation has been defined on its anatomical 'master' requires selecting it from a menu and then pressing the `Write` button. This takes 5–50 seconds per 3D sub-brick (depending on the CPU speed). Because it was tedious to do a large collection of datasets this way, *AFNI* was extended to allow the selection of multiple datasets for transformation. In essence, this is a hybrid interactive-batch computation, in which the choice of processing is done with a graphical interface, and then the lengthy computations are done without further user input. (Recently, a batch program to do the same thing was finally added, so now it *is* possible to carry out this calculation purely from the command line.)

5.a. Custom GUIs

Some functionality will require development of a custom GUI. Portability and interoperability in such a situation is very hard to ensure. Two possible solutions are

[**i**] Mandating a standard for GUIs (e.g., in Java).

[**ii**] Developing an abstract model for GUIs, with only the actual graphical implementation varying with platform.

Both of these are fraught with potential conflicts between developer groups.

Wrapper GUIs can be developed for command line programs. These are mostly conveniences, since the programs will not interact with the user state variables of a truly user-friendly program.

5.b. Plugins—Tight Integration into a Master Program

As I think of it, a plugin is a module that is added to a master program that can directly access the master's internal data structures and subroutines. This makes it possible for the master and plugins to share information that is user interface oriented; for example:

> An *AFNI* user can draw a 3D ROI on 2D sections in one plugin. As each drawing stroke is completed, the 3D rendering plugin can be notified and update the 3D display of the brain volume with the ROI superimposed. The two plugins have access to the same datasets, and communicate through *AFNI* plugin library routines and data structures.

Such close-knit cooperation requires that the programmer adhere to a strict set of rules so that the master program is not corrupted. If the full advantage of the plugin integration is to be used, the programmer must be fairly advanced and the master program must provide a large number of services to the plugins.

5.c. Plugouts*—Intermodule Communication Protocol

*A word I coined (but the idea isn't mine)

This is a way to provide some of the same user interface integration as plugins, but with a more arms-length approach. A plugout is simply another program running independently, which communicates with the master GUI using some agreed-upon protocol. The data sent down the plugout channel can include user state information as well as datasets. The programs communicating do not have to share the same actual GUI, but must share some common abstractions (e.g., the focus point). Agreeing on these abstractions and on the communications protocol are the points to overcome before this approach can become widespread.

One advantage of this approach is that it is relatively language-neutral. As long as a programming language supports the communication method underlying the plugout protocol, modules written in the language can be integrated into the user interface.

> *AFNI* contains a rudimentary plugout protocol that allows it to exchange the coordinates of the focus point with other applications. The underlying channel is a TCP/IP socket. One plugout application developed at MCW was written in *Matlab*.

5.d. A Processing Language

This concept is really aimed more at the batch mode of operation. The main objection is to the creation of yet another language. It isn't clear that a strong advantage is gained over simpler use of scripting languages (shells, Perl, ...) with command line programs.

6. Judgments

If it's not clear by now, I think that the most important issue is data interoperability. This should be addressed by the crafting of an extensible format as discussed in §3 [p.5]. Design and implementation of some support software should go along with this effort. Not all the structures described in §2 [p.2] need to be developed at once, but the format should be built with them in mind.

To a large extent, a standard neuroimaging dataset format will also solve the issue of executable interoperability for command line programs, especially if a simple XML-RPC or SOAP server program is written to facilitate remote execution.

For closer interoperability, development of a plugout protocol seems like the best solution to me. This will require a detailed planning meeting (or several) between the principal developers, the drafting of clear documentation, and development of support code. Passing data through XML seems like a good idea here, as well. Deciding on the user interface abstraction elements to be passed around will be the hard part.

7. Surfing Targets

Some Web sites to play with:

XML:	`http://www.xml.org/` `http://www.xml.com/pub`
XSIL:	`http://www.cacr.caltech.edu/SDA/xsil/`
XML-RPC:	`http://www.xmlrpc.com/`

SOAP:	`http://www.xml.com/pub/2000/02/09/feature/index.html`
	`http://www.omg.org/xml/hpcwire.html`
	`http://msdn.microsoft.com/msdnmag/issues/0300/soap/soap.asp`

CORBA:	`http://www.omg.org/`
COM:	`http://www.microsoft.com/com/`
Base64:	`http://community.roxen.com/developers/idocs/rfc/rfc1341.html`
VRML:	`http://www.web3d.org/vrml/vrml.htm`
AFNI:	`http://varda.biophysics.mcw.edu/ cox/index.html`