LOGOS: A Computational Framework for Neuroinformatics Research*

Michael Stiber[1,2,3], Gwen A. Jacobs[1,3], Deborah Swanberg[4]
[1]Dept. of Molec. & Cell Biol., Univ. of Calif., Berkeley, CA 94720-3200
[2]Comp. Sci. Dept., Montana State Univ., Bozeman, MT 59717
[3]Center for Comp. Biol., Montana State University, Bozeman, MT 59717-3505
[4]Dept. of Comp. Sci. and Eng.-0407, Univ. of Calif. San Diego, La Jolla, CA 92093-0407
stiber@acm.org
gwen@nervana.montana.edu
deborah@vision.ucsd.edu

Abstract

Neuroinformatics presents a great challenge to the computer science community. Quantities of data currently range up to multiple-petabyte levels. The data itself are diverse, including scalar, vector (from 1 to 4 dimensions), volumetric (up to 4 dimensional spatio-temporal), topological, and symbolic, structured knowledge. Spatial scales range from Angstroms to meters, while temporal scales go from microseconds to decades. Base data vary greatly from individual to individual, and results computed can change with improvements in algorithms, data collection techniques, or underlying methods.

We describe a system for managing, sharing, processing, and visualizing such data. Envisioned as a "researcher's associate", it will facilitate collaboration, interface between researchers and data, and perform bookkeeping associated with the complete scientific information life cycle, from collection, analysis, and publication to review of previous results and the start of a new cycle.

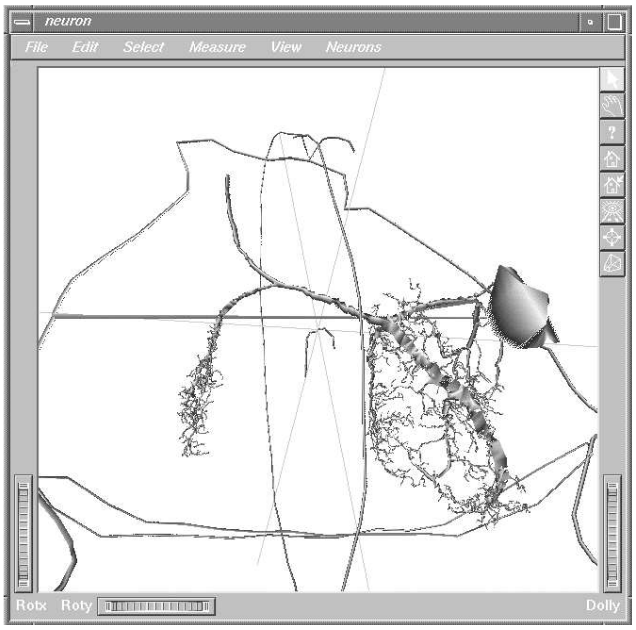

Figure 1. An example rendered neuron. Cell is shown within outline of enclosing ganglion. The cell is represented as a tree (with each branch including diameter data along its length) and an associated "cloud" of output sites (too small to be visible here).

1. Introduction

Neuroscientists study the various anatomical, physiological, and functional components of nervous systems to better understand how the "low-level" activity of individual cells maps to behavior. In this research process, large amounts of complex data are collected, but technology has not yet provided systems which integrate this data to help scientists analyze, visualize, and understand it [4, 10].

There are several aspects of this which are unusual when compared to most other scientific data processing activities. First of all, the base data gathered from experiments are diverse, including anatomical (2D and 3D images, 3D geometries as shown in Fig. 1), physiological (times series, point processes; see example in Fig. 2), molecular (2D and 3D density distributions), and symbolic (functions, behaviors). Additionally, data are gathered from single cells in

*This work was supported by NSF grant number BIR-9507314 to G.A.J.

Figure 2. An example physiological recording, in this case of the simultaneous discharge of two nerve cells (X-axis is time; Y-axis voltage). Cell discharges are often recorded as a sequence of voltages along time. Alternatively, they may be assimilated to point processes by noting only the times of occurrence of the large voltage spikes.

individual animals and can vary greatly from one animal to another. However, researchers usually don't want to ask questions about a particular cell or individual; they want to generalize from the examples they've seen to produce an understanding of how cells and systems function.

We address here three major problems associated with neuroscience data processing, or *neuroinformatics*:

1. The construction of a basic system to support neuroscience data management: the LOGOS system.
2. Providing researchers with the ability to present queries and receive responses in terms of typical cells, relying on an extension called METALOGOS to map these higher-level constructs to operations on data gathered from individual experiments.
3. The use of METALOGOS as a *cells-to-systems interface*, whereby the user can manipulate data pertaining to large numbers of cells which collectively contribute to a particular function.

To make 2 and 3 above more concrete, consider the following example [5]. Crickets have elongated sensory organs, called *cerci*, which project behind them. These cerci are covered with hairs which serve as transducers for air motion. Transduced signals are carried by sensory neurons to an abdominal *ganglion*: a collection of nerve cells and inter-neuronal connections. The many sensory neurons each transmit their messages to a number of second-level neurons, or *interneurons*; each interneuron receiving input from many sensory cells.

The obvious question to ask here is: what computation does the cercal sensory system *as a whole* perform? It is not usually feasible, however, to perform an experiment to answer this question directly. Instead, a researcher could perform the following experiments:

1. Stain a single neuron in an individual cricket with a chemical dye, so that its entire structure can be seen easily. Digitize its three-dimensional geometry. As shown in Fig. 1, a cell is represented as: a tree with each branch having particular diameters at each node, and a cloud of small spheres, or *varicosities*, which correspond to connection points between that cell's output and other cells' inputs. This could be done for a large number of sensory cells and interneurons
2. Insert an electrode into a cricket, and record the physiological responses (similar to that shown in Fig. 2) of sensory cells to air currents having different direction and magnitude. This would allow one to compute a *tuning curve* for each cell: a mapping from the space of wind velocity to neuron output intensity. For sensory cells, this map is typically a simple function.
3. Make physiological recordings of interneuron responses to air motion.

Experiments 1 and 2 can be used to build a database for the first stages of the cercal system [11]. However, it is likely that the results of experiment 3 would be difficult to interpret initially, since each interneuron receives input from many sensory cells (and thus the map from wind velocity to response is complex and difficult to either summarize meaningfully or provide constancies from one animal to another). Instead, one must first determine what cellular aspects are conserved across different animals: in this case, the directional sensitivity of corresponding hairs and the region of the ganglion to which each hair's sensory neuron projects.

Based on this, the summary display shown in Fig. 3 can be produced. A number of sensory cells were used to generate this diagram. For each, the distribution of varicosities was used to produce an estimate of what its connection

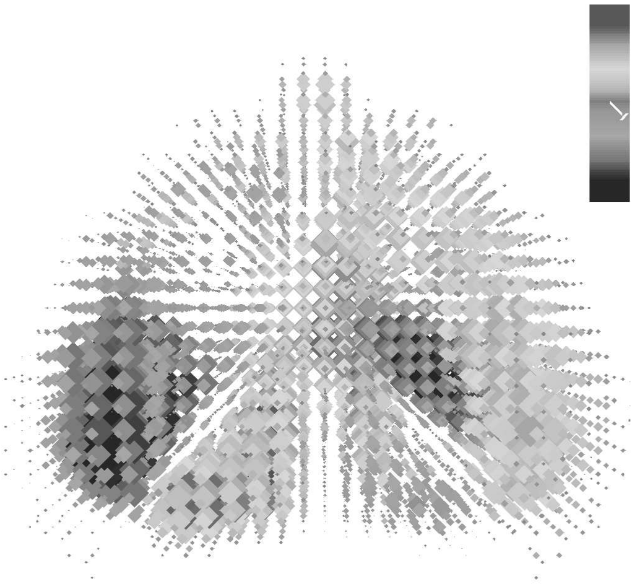

Figure 3. System-level view computed from a number of individual cells and animals. This display shows sign and magnitude of response to air motion to the right rear encoded by shade of grey (original was in color) and diamond size.

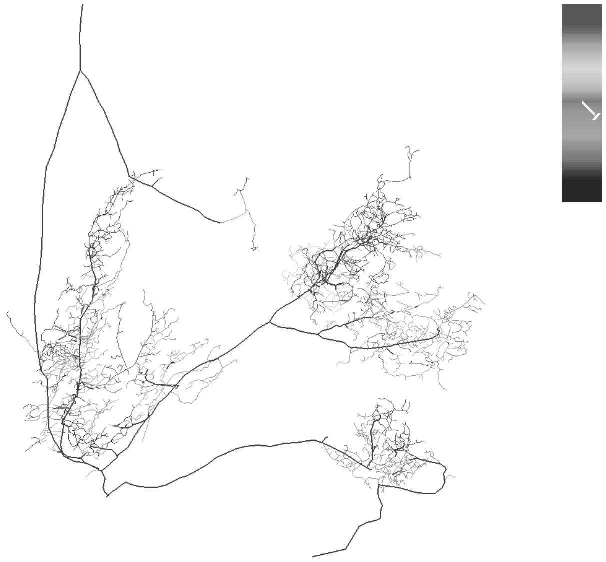

Figure 4. System response in Fig. 3 mapped onto rendering of an interneuron (greyscale rendering of a color original).

strength to another cell would be at each point in the volume of the ganglion. If an interneuron has an input region in a sub-volume densely filled with sensory cell outputs, we would expect it to receive strong signals from that sensory cell. Contrastingly, an input in a sparse sub-volume would yield little input.

This density function was then used as a weight for multiplication with the sensory cell's wind velocity tuning function. The total system response for a number of sensory neurons can be computed as the sum of the individual cells' tuning functions, weighted by their densities at each point within the ganglion. Using this 3D map of net velocity tuning, one can compute the overall system response to a puff of air of a particular speed and direction, which is shown in the figure as grey level (from a color original).

We can take this one step further, by showing how this system-level response maps to the input of a particular interneuron. Fig. 4 shows the response in Fig. 3 mapped onto an interneuron's structure. All branches of the interneuron smaller than some cutoff diameter were assumed to receive input from sensory cells. For each point on the surface of these branches, the net system response computed for Fig. 3 was taken as the input that would be received [11], indicated by shade of grey in Fig. 4 (better displayed in the color original). Once this is accomplished, one could then proceed to compare these inputs with the physiological recordings made of the interneuron output to help determine the computation it performs.

There are additional design goals for LOGOS which, while perhaps not as novel as those discussed above, are no less necessary. These include:

- The logical and physical views should make explicit distinctions between raw and (possibly various levels of) processed data [2, 7].
- Human understanding of any field changes over time; data capture and analysis changes, too. The schemas which underlie data storage should be evolvable [3].
- Scientific computing occurs within a heterogeneous hardware and software environment; a scientific data management system should accommodate this [8].
- A domain-specific user interface which accommodates differing levels of user sophistication (including those who write their own applications) should be used [2].
- Because this is a *scientific* data management system, capabilities such as maintenance of audit trails, error tracking, and data security are essential [10].

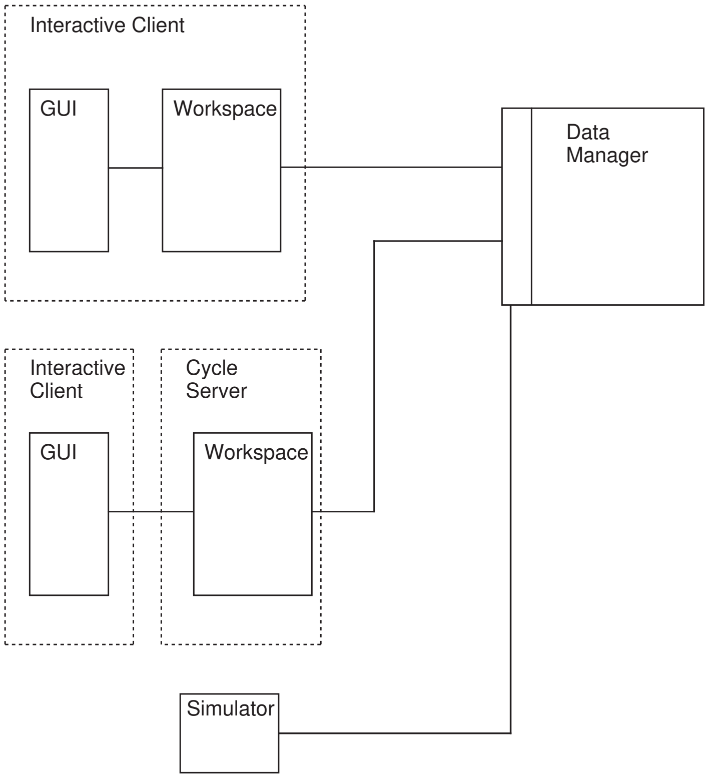

Figure 5. LOGOS **system architecture, showing support for distribution of data management, computation, and user interface tasks.**

2. LOGOS Architecture

The overall LOGOS system architecture is presented in Fig. 5. Its basic distributed nature is dictated by the desire to support existing hardware and software whenever feasible. It is therefore divided into three main modules: a *data manager* (DM) as a persistent data store and query server (implemented using ObjectStore, an off-the-shelf object-oriented database management system), a *workspace* (WS) for performing data analysis and other computationally intensive tasks (the results of which might later become part of the data manager's store), and a *user interface* (UI).

Many researchers make daily use of Macintosh and Intel-based machines, and it was decided early-on to support two user interfaces: one hosted on a high-performance Unix workstation and another implemented for less expensive hardware. However, use of a less expensive machine on one's desktop might not mean lack of access to more powerful hardware elsewhere. A separate workspace process, which can be hosted on some cycle server, minimizes the penalty paid. If one has access to a graphics workstation, then a higher-performance user interface — currently based on Open Inventor [12] — would be available, and there would be the option of running the workspace on the same workstation or another machine. Current LOGOS development is performed on Silicon Graphics workstations.

There is one additional way that some users might want to access the system. Instead of using the user interface provided with LOGOS, one might want to stick to some existing software, for example a neural simulation system like GENESIS [1] or a mathematical analysis package like MATLAB. Appropriate access methods can be provided so that a user of GENESIS, for example, could connect to the data manager to retrieve experimental data to serve as the basis for simulations.

The neuroinformatics problem is complex, and it is unrealistic to suppose that any initial system design will solve all of the issues of user interface design, visualization, data management, use of domain knowledge, etc. LOGOS was designed to be a framework for research into these issues, and thus its division into these major subsystems maximizes our ability to isolate each of these areas. Object-oriented design and implementation using C++ has been used to increase our ability to encapsulate independent system functions. Of course, there is necessarily a correspondence between WS and UI data objects, since WS data must be viewed by the user. However, this architecture does allow isolation of decisions of *how* the user views and interacts

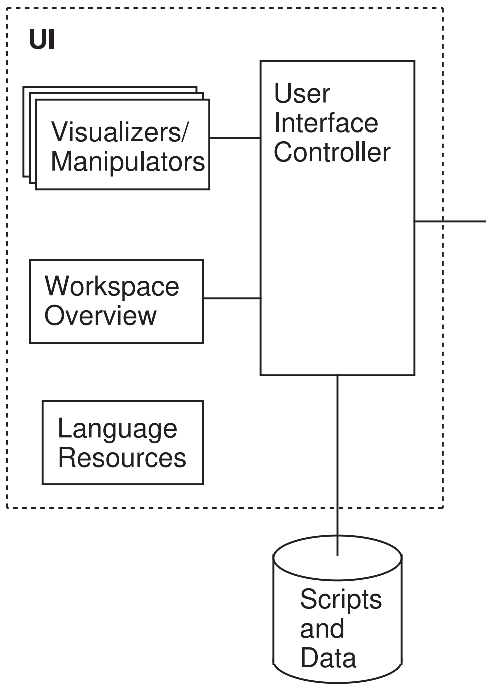

Figure 6. User interface modules.

with the data from the operations the WS performs on them.

2.1. User Interface

Fig. 6 shows that the user interface has four components: one or more *viewers/manipulators*, which handle examination and interactive modification of data, a *workspace overview* (WSO), which provides views of metadata (information about workspace data and function objects), *language resources* supporting multi-lingual capabilities, and the *user interface controller* (UIC), which sequences the operations of the UI components and their communication with the workspace and provides a file system interface for command scripts and data import/export. This provides isolation of data visualization to the viewers/manipulators (currently implemented using Open Inventor) and metadata UI issues to the WSO. The UIC provides a "generic" connection to the workspace.

Upon system startup, the UIC requests that the user log in to the system. This allows for security in access to data and logging of who performed what operations. The UIC then connects to the WS and request login verification. Assuming the login is accepted, the UIC then presents a list of data types that it can accept (constrained by the capabilities of its viewer) and the WS replies with a list of information about data objects it contains and functions that it can provide. The UIC displays a WSO, which shows the systemwide function and data object information and allows the user to select data objects and operations to be performed on them. Commands from the WSO flow to the UIC, which may dispatch them to the WS or create one or more viewers or manipulators.

A viewer is a means for rendering data, typically graphically, and allowing a limited set of local and WS operations to be performed on the rendering or data, respectively. A viewer produces no new data items to add to the workspace. A viewer provides a set of local operations which modify the appearance of the rendering. A viewer also allows the user to request WS performance of certain operations on the data corresponding to a particular rendering, which would pass via the UIC to the WS. The viewer extends the WSO command dispatching capability by allowing user specification of some subpart or single element of a data object as the input for a WS function.

A manipulator is a viewer that also offers interactive functions over its data. A manipulator can produce new, derived data, which pass via the UIC to the WS. An example is the interactive alignment of two neurons, which can produce an alignment data object for storage in the WS (see section 3.4 for a discussion of the alignment process).

2.2. Workspace

The workspace shown in Fig. 7 has four major components: a *workspace controller* (WC) which manages communication with the UI and sequences the activities of the other WS modules, *processing plug-ins* which represent a set of operations over the data (see section 3.2), a *Working memory* (WM) that provides a temporary data repository, and a *query manager* (QM) which assembles queries for the data manager and transfers data between the DM and WM.

When a UI requests a connection to the WS, the WC first performs login verification. Assuming login is successful, the WC retains information about the current user for the duration of the session, and uses this to tag newly imported base data and newly computed derived data. The WC then negotiates a connection protocol with the UI, this consisting primarily of the UI notifying the WC of the types of data it is capable of displaying (this guaranteed to be a subset of the data the WC is capable of sending) and the WC providing a list of data contained in WM and functions provided by the processing plugins and QM. Subsequently, the WC is responsible not only for receipt and dispatch of computation requests from the UI, but also updating the UI on changes in the contents of WM (the result of the computation requests) and converting data in the WM into types displayable by the UI.

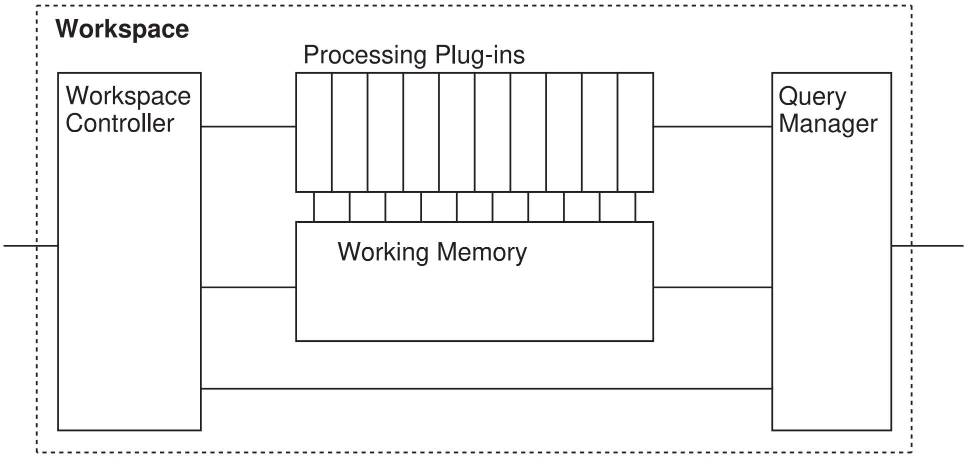

Figure 7. Workspace block diagram.

3. Data, Functor, and Metadata Models

Data in LOGOS can be considered as existing in a discretized, 2D space, with one axis being physical location within the system (UI, WS, DM) and the other being the *category* of information. Three categories exist: *data*, including base (experimental) and derived, *functors* or function objects encapsulating WS processing plugins, DM operations, and UI manipulator capabilities, and *metadata* about data and functors.

Data may move from one physical location to another. When they do, they may change form (data structures, methods) and/or content (e.g., pruning down of WS data to visualizable attributes before transmittal to the UI). Creation of data or functor objects implies creation of corresponding metadata, and movement of the former implies the previous movement of the latter. Thus, the metadata can contain information (or *handles*) necessary for referring to data between subsystems.

The UI must deal with each category of data differently. Its WSO is a tool for metadata viewing and interaction. At its simplest, it uses metadata to display lists of WS data and available functions, applies constraints about functor argument number and type to provide basic error checking, and passes handles to data and functors to the UIC controller for dispatch to the WS or viewers/manipulators. The viewers and manipulators receive a renderable subset of the contents of WS data objects, their metadata, and a subset of functor metadata, allowing the user to command certain WS operations on the displayed data. The UIC receives metadata and renderable data from the WS and handles for data and functors from the WSO and viewers/manipulators, provides functor metadata for manipulator operations, and routes data to the appropriate destinations.

Within the workspace, all three categories reside in the WM, which is merely a passive, non-persistent store. The WC receives handles to functors and data from the UI, executes functors, receives metadata for any new data created in the WM, and passes this new metadata on to the UI. The WC also, upon UI request, filters WM data to produce visualizable representations (as per the capability set transmitted by the UI at connect time) and transmits them to the UI. When data are created in a UI manipulator, the WC will create metadata for them, store both in WM, and return the new metadata to the UI.

The above use of metadata to describe generic characteristics of both data and functors serves to loosen the coupling among system components, allowing, for example, the addition of functors (and, to some extent, data types) to the WS without modification of the UI.

3.1. Data

All data, whether imported into the system as raw experimental results or derived via (possibly many) operations on other data, are subclassed from an abstract *Data* class. WM operations are all performed on objects of class *Data* (or *Info* — metadata — objects, see section 3.3). Functors, on the other hand, may be specialized to operate on particular subclasses of *Data* or certain of their component parts. For brevity's sake, we present here two examples of the object hierarchies used in LOGOS for storing anatomical data and data derived from anatomical information.

Fig. 8 presents a partial LOGOS class hierarchy for anatomical experimental data gathered from a single animal: a *Preparation*. Such data usually include both the anatomy of one or more *Neuron*s and additional anatomical features not associated with any or these. These additional features, or *Fiducial*s, are used to register coordinate systems between individuals. Fiducials are "landmarks" — relatively invariant across individuals — used to bring the

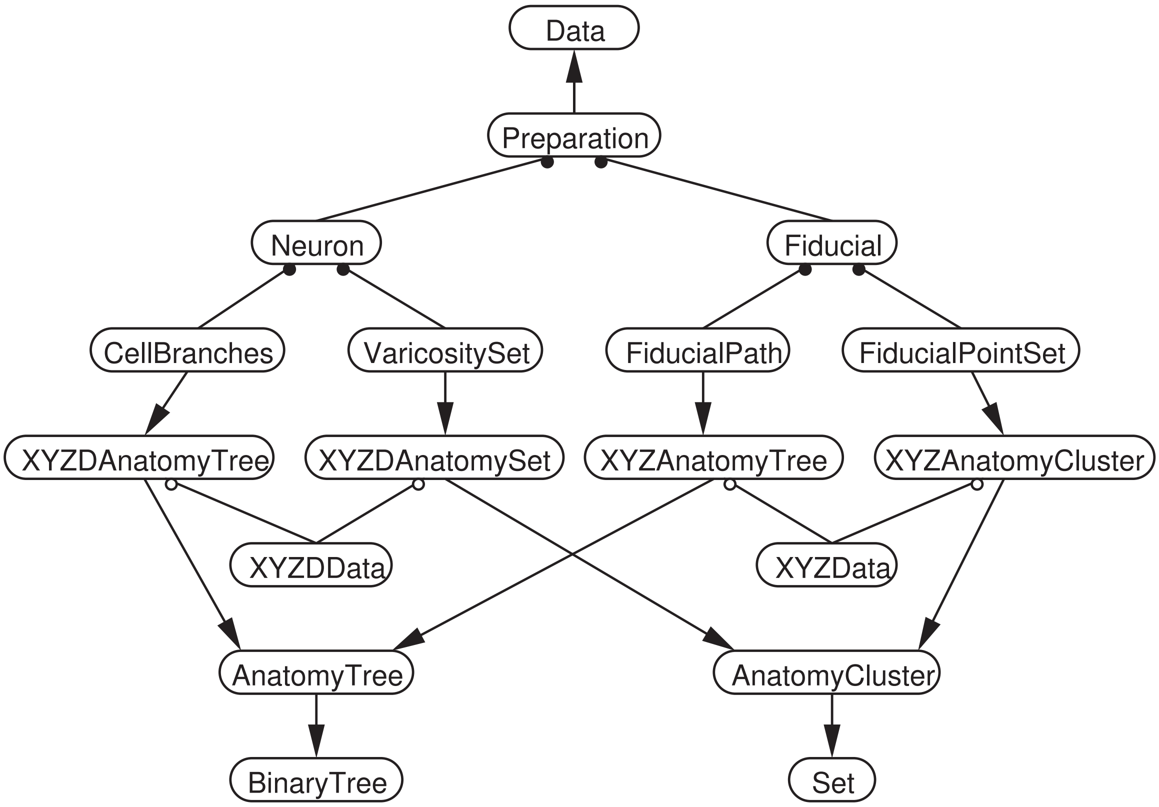

Figure 8. Living preparation class diagram. Arrows indicate parent/child relationship, with arrow direction pointing to superclass. Filled circles indicate "contains" relationship, with circle at container class. Open circles show "uses" relation, with circle at consumer end.

spatial coordinate systems in which multiple preparations' anatomical data are captured into alignment.

A neuron has a treelike structure, and thus a generic binary tree class is the ancestor of much of its data. Unfortunately, it is often impractical or impossible to capture the cell's anatomy in its entirety, as its branch diameter decreases significantly as distance from the central cell body increases. However, the connections between cells, or *varicosities* are typically large and important enough that they are digitized separately, and modeled as a set of spheres. In both cases, the basic datatype is the 4-tuple of (x, y, z) location and diameter d.

Fiducials, on the other hand, usually have only their locations (without any diameter data) recorded, as they are mostly the outlines of large internal structures (saved as paths) or key feature locations (saved as points).

As was alluded to in the introduction and the previous discussion of aligning coordinate systems, computing general results from the specific examples obtained from particular individuals is challenging. Precise anatomical organization is *not* identical from one animal to the next. However, there are of course certain organizational aspects which are preserved among all members of a particular species. Therefore, though the precise coordinate systems for any two individuals are not the same, the topology of that space is, and in many cases the coordinates of one may be transformed to that of the other by simple operations like translation, rotation, and scaling along individual axes. This transformation is a *3DAlignment*, shown in Fig. 9.

A 3DAlignment is a *DerivedData*, and is computed from two *Preparation*s: a *reference* and an *alignee*. The process involves locating corresponding fiducials belonging to each preparation, and then using them to determine the appropriate transformation. The computation associated with a 3DAlignment may be performed interactively in some cases (see section 3.4).

Another kind of derived data, this time combining information about several neurons for a higher, system-level overview, is the *SynapticDensityField*. A *SynapticDensityField* is computed from the *VaricositySet*s of one or more neurons (for which a single, unifying *3DAlignment* must exist). It is a real-valued function of the (x, y, z) space defined by their unifying *3DAlignment*, with the value at each point being an estimate of the input strength exerted

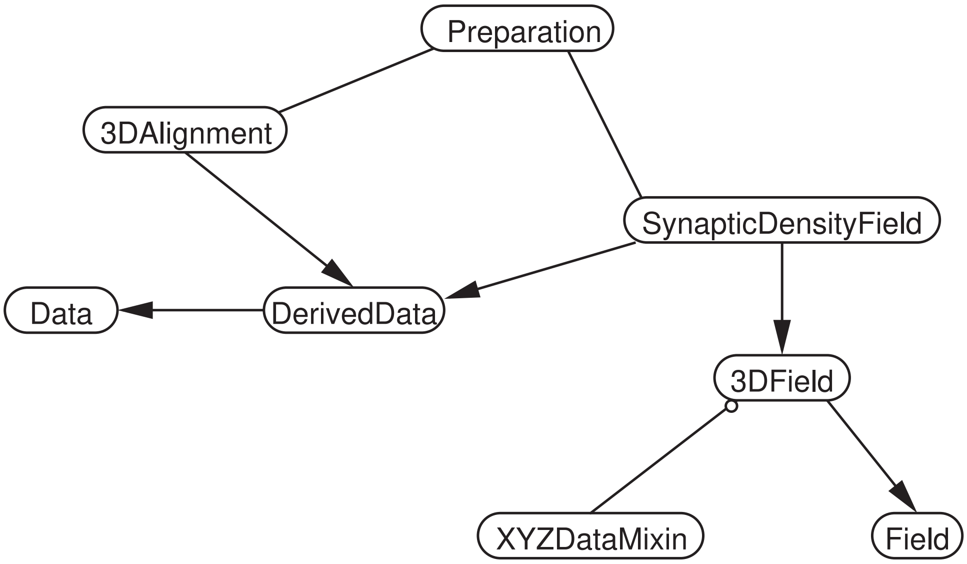

Figure 9. Derived data class diagram. Connectors with arrows and open or filled circles are as in Fig. 8; plain lines indicate the existence of a relation between the two classes.

by the neurons on any other neuron that would receive input at that point. Synaptic density is one way to produce an *average neuron* from specific ones measured. It can also be extended to a vector-valued function, and the result is a complete map of wind velocity within the physical space defined by the sensory neurons' varicosities [6]. Section 3.5 describes how LOGOS functors can be combined to produce a *SynapticDensityField*.

3.2. Functors

The LOGOS functor concept is similar to that provided by the C++ Standard Template Library [9] (STL): it is an object that can be called like a function (an ordinary function, a function pointer, or an object that defines `operator()`). However, unlike the STL, compile-time type checking is not very useful here; we therefore dispense with the STL functor classes and provide one functor class and mechanisms for run-time checking of argument type, number, and order, as well as "documentation data" displayable for the user's benefit (printable function name, short description, etc). For each functor, a corresponding metadata object (*FunctorInfo*) is created with the following information:

Name String containing function name, for user interface.

Description Short documentation string describing the function, for user interface.

Arguments Number of arguments; `unsigned`.

ArgTypes One for each argument; an `enum` that is also used to designate data object type. Note that the actual arguments passed at the eventual function call may include addition information, perhaps taken from associated metadata.

ArgDesignators Some functors take in multiple arguments of the same type, but treat them differently. The user interface cannot be relied upon to provide argument order information, so the role of each argument cannot be implicit in their specification order. A string that documents an argument's role can be associated with any or all (see section 3.4 for an example).

ReturnType An `enum` specifying the type of data object created by the functor. Note that this does not imply that the functions that are eventually called have this return type; they return structures that contain success or failure information plus pointers to data that can be used by the WS to construct the appropriate data object.

Handle A reference to the functor itself, used to invoke it.

3.3. Metadata

There are two types of metadata: that which describes data and that which describes functors. The latter was described in section 3.2; we summarize the former here. Like *FunctorInfo*, *DataInfo* serves the purposes of run-time type checking and input validation and documentation. Unlike functors, however, different data objects of the same class are interchangeable (functors at least have their inherent computational capabilities that define them). We must rely on objects' *DataInfo* to allow us to distinguish among them. *DataInfo* includes:

Name A string containing a creator selected name. For experimental data imported into the system, these are typically selected using some system enabling quick identification of the experiment. For derived data generated by some functor, these may be less useful.

Notes A string containing user-entered notes.

History Functor and argument *handles*, along with date, time, and user. For experimental data, this would indicate import date, time, and creator. For derived data, this would point to the functor that returned it, plus the functor arguments. A complete history can be generated by recursively following the history entry of each data object argument's *DataInfo*.

Tuning An optional field, which for preparations indicates the wind direction that elicits a maximal response.

Handle A reference to the data object itself.

3.4. Interactive Example: The Alignment Process

The process of interactively aligning two preparations' coordinate systems is a simple example of how data and metadata are used by the UI. The interactive alignment process involves displaying two preparations, containing both neuron structure and fiducial landmark data, in a manipulator that provides tools for translation, rotation, and scaling. One preparation — the *reference* — remains locked in place while the other (the *alignee*) is moved, turned, and stretched until the user judges that the two sets of fiducials are matched as closely as practical. The result is a *3DAlignment*.

Like all LOGOS operations, this begins in the WSO, which is displaying lists of WS data objects and functors. The interactive alignment functor resides in the UIC, which creates the following *FunctorInfo* for the WSO:

Name	`''Interactive Alignment''`
Description	`''Use manipulator to manually align two preparations''`
Arguments	`2`
ArgTypes	`arg0: preparation` `arg1: preparation`
ArgDesignators	`arg0: ''reference''` `arg1: ''alignee''`
ReturnType	`3DAlignment`
Handle	reference to the UIC functor

Suppose the user selects two preparations and the "Interactive Alignment" operation. The WSO then checks the number of data objects selected against the "Arguments" field above and the type of each argument against the "ArgTypes" entries. Next, we note that there are two "ArgDesignators", indicating that each argument plays a particular role in the operation. The WSO must then prompt the user to assign a role ("Reference" versus "Alignee") to each of the preparations.

At this point, we are ready to execute the functor, and the WSO passes the functor and argument handles (the argument roles now being implicit in their order) to the UIC. In this case, this is a UIC functor which creates a manipulator, fetches renderable versions of the preparations from the WS, and passes the preparations and their metadata to the manipulator. The user can then interact with the manipulator (or do other operations within LOGOS, as the manipulator is managed as a separate window) until he or she is satisfied with the alignment. At that point, the manipulator returns a new *3DAlignment* object, which is passed to the WS along with its newly-created metadata (which includes a history entry showing it was created by an interactive alignment of the two preparations).

3.5. Abstraction: Computation of Response Maps

A more complex operation that involves WS functors is the computation of the response map shown in Fig. 3. To compute this, we must address the problem of determining a "typical" cell's influence based on our specific experimental data [11]. Ordinarily, this would be accomplished via statistical techniques. However, these are not directly applicable to the tree-like structure of a neuron. Two reasonable assumptions are used to render this problem tractable: that sensory cells influence interneurons only via their varicosities and that the probability of an interneuron receiving input from a typical sensory cell can be computed from the distance its branches are from its varicosities and the varicosities' surface area. Determination of a typical cell's influence is then reduced to:

1. computing a function of 3D space within the ganglion based on the distribution of varicosities in each preparation.

2. combining these multiple functions to form a single, overall influence function (by summation, for example).

The metadata associated with the response map computation is:

Name	`''Compute Response Map''`
Description	`''Estimate overall response to a stimulus''`
Arguments	`2`
ArgTypes	`arg0: preparationList` `arg1: userReal`
ArgDesignators	`arg0: none` `arg1: ''Direction''`
ReturnType	`ResponseMap`
Handle	reference to the functor

The argument to this functor is a list of one or more preparations and a number for which the user is prompted (stimulus wind direction); after the WSO performs its argument checking, the handles are passed to the UIC and then on to the WS. The response map functor is composed of five simpler operations [11]:

1. Find composite *3DAlignment*s which produce a common space for a set of preparations and apply them, producing a set of aligned preparations.

2. For a single *VaricositySet*, compute a *synaptic density field*. This is done by iterating through the varicosities, computing its contribution to the overall field (as a gaussian function of the sphere's surface area), and summing the individual contributions. This produces a *SynapticDensityField* object.

3. For a single *Preparation*, compute its response to wind blowing in a particular direction.

4. For a single *SynapticDensityField*, compute a functional transformation by multiplying the field values by a constant, producing a *ResponseMap*.

5. Sum a set of *ResponseMap*s, producing a new one.

These are sequenced by the WC. One particular operation that is likely to fail is number 1, if no composite alignment can be found (via a minimum spanning tree algorithm) that brings all preparations into the same space. This failure would be reported to the UIC, and the user would need to do additional alignments before trying again. Assuming success, the WC would return metadata for the final *ResponseMap*, and perhaps some of the intermediate objects produced (whether the intermediate objects are temporary or not is an implementation issue to be considered for algorithm efficiency).

Note that these five simpler operations might be utility functions known only to the WS, or they might be functors. The latter capability — composition of functors into more complex operations (as with the STL functor adaptors) — is at the heart of the METALOGOS extension.

4. METALOGOS and the Application of Domain Knowledge

A follow-on to the LOGOS project is METALOGOS, a system which adds domain knowledge so that users may pose system-level queries and receive responses at that level. For example, researchers often desire to think not in terms of particular cells in particular individuals, but rather a particular class of cells. We may know that all sensory cells connected to a particular region of the sensory organ have velocity tuning curves with peak sensitivities for high-speed air motion. Those connected to other regions might have much lower peak sensitivities. A convenient categorization might be "fast" versus "slow", and we might assign a sensory neuron to one category or another based on tuning data, if present or calculable from physiological data, or location of its connection, if only anatomical data is available. Thus, one type of domain knowledge used by METALOGOS is *categorical* or *taxonomic*. Categories may be entered manually, and thus attributed to an individual person, or computed from statistical cluster analysis.

Domain knowledge also includes knowledge of the consequences of various algorithms, for instance how computing a synaptic density field establishes a mapping between single cells in individual animals and either "typical" cells of that type or cell systems. Knowledge about data objects or functors would be stored in their associated metadata, and would start with codifying the information currently contained in strings in a machine-usable format.

The basic flow of processing in METALOGOS is:

1. Accept systems-level query.

2. Use domain knowledge to map systems-level query to a dataflow diagram, starting with queries of existing base and derived data and using available functors to produce a result that is a response to the query. If no such dataflow program can be determined, any partial diagrams should be reported along with failure to the user interface, so the experimenter can either modify the query, modify the dataflow diagram, or add additional domain knowledge.

3. Otherwise, pass dataflow program to command and display interface for execution.

4. Return result to user interface.

METALOGOS is also meant to deal more thoroughly with issues such as attribution of sources for data, etc. This is to some extent an extension of tracking error propagation through calculations; in this case, individual users would be able to assign levels of "trustworthiness" to others, and the system will compute a reasonable estimate of the trustworthiness of its results, based on who originated the knowledge it used. Ideally, METALOGOS would include — either as part of its database or via interface or broker programs — links between data and the publications that use them. This would provide both a mechanism for making the data upon which a publication is based public and the means for other investigators to easily examine the data and algorithms used by others to draw their conclusions, perhaps using one's own data as tests.

5. Discussion

Moving neuroinformatics beyond the stage of merely producing enhanced file systems or electronic anatomical atlases is a difficult task. Successful systems will require architects who are comfortable in both the biological and computer domains, as well as close collaboration with active biology researchers. Though one would not expect to have solutions to the large number of user interface, visualization, database, knowledge representation, etc. problems *a priori*, the systems produced should be useful even in their infancy. This implies a decomposition of the design into components which are as independent from each other as practical. Since any initial data schemas will evolve and grow considerably over the system's life cycle, its initial design should allow this and, as far as possible, foresee possible areas of change. The system should also have at least an element of "the vision thing": an ultimate goal that will improve both in depth and breadth the research process itself.

6. Implementation Status

As of the date of this writing, components of the user interface (viewer/manipulator, WSO) and workspace (WC, WM) have been prototyped. These prototypes are currently being modified from stand-alone, test versions. The remaining UI and WS modules are in the detailed design and coding stages; we expect the UI and WS to be fully functional and integrated together shortly. Data manager implementation awaits acquisition of the ObjectStore ODBMS.

References

[1] J. M. Bower and D. Beeman. *The Book of GENESIS: Exploring Realistic Neural Models*. TELOS, Santa Clara, CA, 1995.

[2] J. C. French, A. K. Jones, and J. L. Pfaltz. Scientific database management. Technical Report 90-21, University of Virginia, Dept. of Computer Science, Aug. 1990.

[3] N. Goodman, S. Rozen, and L. Stein. Building a laboratory information system around a c++-based object-oriented DBMS. In *Proc. 20th VLDB Conf.*, pages 722–9, Santiago, Chile, 1994.

[4] M. Huerta, S. Koslow, and A. Leshner. The human brain project: and international resource. *Trends Neurosci.*, 16(11):436–8, 1993.

[5] G. A. Jacobs. Detection and analysis of air currents by crickets. *BioScience*, 45(11):776–85, 1995.

[6] G. A. Jacobs and F. E. Theunissen. Functional organization of a neural map in the cricket cercal sensory system. *J. Neurosci.*, 1996.

[7] L. Kerschberg, H. Gomaa, D. Menasce, and J. Yoon. Data and information architectures for large-scale distributed data intensive information systems. In *Proc. 8th Int. Conf. Statistical & Scientific Database Management*, pages 226–33, Stockholm, June 1996.

[8] E. Mesrobian, R. Muntz, E. Shek, S. Nittel, and M. LaRouche. OASIS: An open architecture scientific information system. In *Proc. RIDE '96*, pages 107–16, New Orleans, Feb. 1996.

[9] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA, 1996.

[10] Establishing identified neuron databases. Workshop report, National Science Foundation, Arlington, VA, June 1994.

[11] T. W. Troyer, J. E. Levin, and G. A. Jacobs. Construction and analysis of a database representing a neural map. *Microscopy Res. & Tech.*, 29:329–43, 1994.

[12] J. Wernecke. *The Inventor Mentor*. Addison-Wesley, Reading, MA, 1994.