1 The SCIRun Computational Steering Software System

Steven G. Parker[1]
David M. Weinstein[2]
Christopher R. Johnson[3][4]

ABSTRACT
We present the design, implementation and application of SCIRun, a scientific programming environment that allows the interactive construction, debugging, and steering of large-scale scientific computations. Using this "computational workbench," a scientist can design and modify simulations interactively via a dataflow programming model. SCIRun enables scientists to design and modify model geometry, interactively change simulation parameters and boundary conditions, and interactively visualize geometric models and simulation results. We discuss the ubiquitous roles SCIRun plays as a computational tool (e.g. resource manager, thread scheduler, development environment), and how we have applied an object oriented design (implemented in C++) to the scientific computing process. Finally, we demonstrate the application of SCIRun to large scale problems in computational medicine.

1.1 Introduction

1.1.1 Visual Computing and Interactive Steering

In recent years, the scientific computing community has experienced an explosive growth in both the possible size and the possible complexity of numeric computations. One of the significant benefits of this increased computing power is the ability to perform complex three-dimensional simulations. However, such simulations present new challenges for computational scientists. How does one effectively analyze and visualize complex 3D data? How does one solve the problems of working with very large datasets often

[1]Email: `sparker@cs.utah.edu`.
[2]Email: `dweinste@cs.utah.edu`.
[3]Email: `crj@cs.utah.edu`.
[4]Department of Computer Science, University of Utah, Salt Lake City, Utah, 84112, US.

consisting of tens to hundreds of gigabytes? How does one provide tools that address these computational problems while serving the needs of scientific users?

Scientific visualization clearly plays a central role in the analysis of data generated by scientific simulations. Unfortunately, though visualization may in itself be more computationally intensive than the original simulation, it is often performed only as a mystical post-processing step after a large-scale computational batch job is run. For this reason, errors invalidating the results of the entire simulation may be discovered only during post-processing. What is more, the decoupling of simulation and visualization presents serious scientific obstacles to the researcher. A visualization package may provide only a limited data analysis capability and may be poorly matched to the underlying physical models used in the simulation code. As a result, the researcher may expend significant effort trying to use a data analysis package only to walk away frustrated.

In 1987, the Visualization in Scientific Computing (ViSC) workshop reported [2]:

> Scientists not only want to analyze data that results from super-computations; they also want to interpret what is happening to the data during super-computations. Researchers want to *steer* calculations in close-to-real-time; they want to be able to change parameters, resolution or representation, and see the effects. They want to drive the scientific discovery process; they want to *interact* with their data.
>
> The most common mode of visualization today at national supercomputer centers is batch. *Batch processing* defines a sequential process: compute, generate images and plots, and then record on paper, videotape or film.
>
> *Interactive visual computing* is a process whereby scientists communicate with data by manipulating its visual representation during processing. The more sophisticated process of *navigation* allows scientists to *steer*, or dynamically modify computations while they are occurring. These processes are invaluable tools for scientific discovery.

Although these thoughts were reported close to ten years ago, they express a very simple and still current idea: scientists want more interaction than is currently present in most simulation codes. While the scientific computing community is still trying to find better ways to address these needs, we feel that the problems encountered by computational scientists encompass a wider range of issues, including but not restricted to scientific visualization. Our efforts, therefore, include a diverse range of techniques, including, among others, the use of scripting languages, existing software, visual dataflow programming, and a sophisticated system designed exclusively for computational steering. In this chapter, we focus on the latter, the SCIRun[5][1] computational steering software system.

SCIRun is a scientific programming environment that allows the interactive construction, debugging and steering of large-scale scientific computations [3]. SCIRun can be envisioned as a "computational workbench," in which a scientist can design and modify simulations interactively via a dataflow programming model. SCIRun enables scientists to modify geometric models and interactively change numerical parameters and boundary conditions, as well as to modify the level of mesh adaptation needed for an accurate numerical solution. As opposed to the typical "off-line" simulation mode - in which the scientist manually sets input parameters, computes results, visualizes the results via a separate visualization package, then starts again at the beginning - SCIRun "closes the loop" and allows interactive steering of the design, computation, and visualization phases of a simulation.

The dataflow programming paradigm has proven useful in many applications. In the scientific community, it has been successfully applied in several scientific visualization packages, including AVS from Advanced Visual Systems Inc., and Iris Explorer from SGI. We have extended the use of the dataflow programming model into the computational pieces of the simulation. To make the dataflow programming paradigm applicable to large scientific problems, we have identified ways to avoid the excessive memory use inherent in standard dataflow implementations, and we have implemented fine-grained dataflow in order to further promote computational efficiency.

1.1.2 An Iterative Environment for Scientific Computing

Currently, the typical process of constructing a computational model consists of the following steps:

1. Create and/or modify a discretized geometric model;
2. Create and/or modify initial conditions and/or boundary conditions;
3. Compute numerical approximations to the governing equation(s), storing results on disk;
4. Visualize and/or analyze results using a separate visualization package;
5. Make appropriate changes to the model; and

[5]SCIRun is pronounced "ski-run" and derives its name from the Scientific Computing and Imaging (SCI) research group which is pronounced "ski" as in "ski Utah."

6. Repeat.

The "art" of obtaining valuable results from a model has up until now required a scientist to execute this process time and time again. Changes made to the model, input parameters, or computational processes are typically made using rudimentary tools (text editors being the most common). Although the experienced scientist will instill some degree of automation, the process is still time consuming and inefficient. Ideally, scientists and engineers would be provided with a system in which all these computational components were linked, so that all aspects of the modeling and simulation process could be controlled graphically within the context of a single application program. While this would be the preferred *modus operandi* for most computational scientists, it is not the current standard of scientific computing because the creation of such a program is a difficult task.

Difficulties in creating such a program arise from the need to integrate a wide range of disparate computing disciplines (such as user interface technology, 3D graphics, parallel computing, programming languages, and numerical analysis) with a wide range of equally disparate application disciplines (such as medicine, meteorology, fluid dynamics, geology, physics, and chemistry). Our approach to overcoming these difficulties is to separate the components of the problem. SCIRun's dataflow model employs "modules" that can be tailored for each application or computing discipline. Although this method is proving successful at partitioning many of the complexities, we have found that some complexities remain, such as the burdens of parallel computing and user interfaces. Much work goes into simplifying the programming interfaces to these features so that they will be used, rather than ignored, by module implementors.

1.1.3 Steering

The primary purpose of SCIRun is to enable the user to interactively control scientific simulations while the computation is in progress [4, 5]. This control allows the user to vary boundary conditions, model geometries, or various computational parameters during simulation. Currently, many debugging systems provide this capability in a very raw, low-level form. SCIRun is designed to provide high-level control over parameters in an efficient and intuitive way, through graphical user interfaces and scientific visualization. These methods permit the scientist or engineer to "close the loop" and use the visualization to steer phases of the computation.

The ability to steer a large scale simulation provides many advantages to the scientific programmer. As changes in parameters become more instantaneous, the cause-effect relationships within the simulation become more evident, allowing the scientist to develop more intuition about the effect of problem parameters, to detect program bugs, to develop insight into the operation of an algorithm, or to deepen an understanding of the physics of the problem(s) being studied.

The scientific investigation process relies heavily on answers to a range of "What if?" questions. Computational steering allows these questions to be answered more efficiently and therefore to guide the investigation as it occurs.

1.2 Requirements of SCIRun as a Computational Steering System

Initially we designed SCIRun to solve specific problems in Computational Medicine [6, 7, 8, 9], but we have made extensive efforts to make SCIRun applicable in other computational science and engineering problem domains. In attacking the specific problems, we found that there were a wide range of disparate demands placed on such a system. Each of these demands reveals a different facet of what we call SCIRun.

1.2.1 SCIRun the Operating System

In a sophisticated simulation, each of the individual components (modeling, mesh generation, nonlinear/linear solvers, visualization, etc.) typically consumes a large amount of memory and CPU resources. When all of these pieces are connected into a single program, the potential computational load is enormous. In order to use the resources effectively, SCIRun adopts a role similar to an operating system in managing these resources. SCIRun manages scheduling and prioritization of threads, mapping of threads to processors, inter-thread communication, thread stack growth, memory allocation policies, and memory exception signals (such as segmentation violations).

1.2.2 SCIRun the Scientific Library

SCIRun uses a visual programming interface to allow the scientist to construct simulations through powerful computational components. While the visual programming environment is the central focus of SCIRun, it requires a powerful set of computational tools. In the first stage of SCIRun, we have concentrated on integrating the computational components that we have used to solve our own computational problems. We have recently expanded focus and are now in the process of integrating popular libraries and tools, such as *Diffpack* [10, 11], *SparseLib++* [12], and *PETSc* [13, 14] into the SCIRun environment.

1.2.3 SCIRun the Development Environment

Perhaps the most powerful facet of SCIRun is the ability to use it in the development phases of a simulation. SCIRun augments the development environment by providing convenient access to a powerful set of computational components. However, these components could never be comprehensive, so SCIRun also provides an environment whereby new modules can be developed efficiently. If a module triggers a segmentation violation, bus error or failed assertion, SCIRun stops the module at the point of error, thus allowing the developer to attach a debugger to the program at the point of failure. This avoids the frustrating experience of trying to reproduce these errors in the debugger. In addition, SCIRun provides simple instrumentation of module performance (CPU times printed out interactively), feedback execution states (waiting for data, percent completed, etc.), and visualization of memory usage. SCIRun employs dynamic shared libraries to allow the user to recompile only a specific module without the expense of a complete re-link. Another SCIRun window contains an interactive prompt which gives the user access to a Tcl shell that can be used to interactively query and change parameters in the simulation.

1.2.4 Requirements of the Application

SCIRun is not magic – it is simply a powerful, expressive environment for constructing steerable applications, either from existing applications or starting from the ground-up. The application programmer must assume the responsibility of breaking up an application into suitable components. In practice, this modularization is already present inside most codes, since "modular programming" has been preached by software engineers as a sensible programming style for years.

More importantly, it is the responsibility of the application programmer to ensure that parameter changes make sense with regard to the underlying physics of the problem. In a CFD simulation, for example, it is not physically possible for a boundary to move within a single timestep without a dramatic impact on the flow. The application programmer may be better off allowing the user to apply forces to a boundary that would move the boundary in a physically coherent manner. Alternatively, the user could be warned that moving a boundary in a non-physical manner would cause gross errors in the transient solution.

1.3 Components of SCIRun

In order to implement the requirements described above, we have broken down SCIRun into a layered set of libraries. These libraries are organized as shown in Figure 1.1.

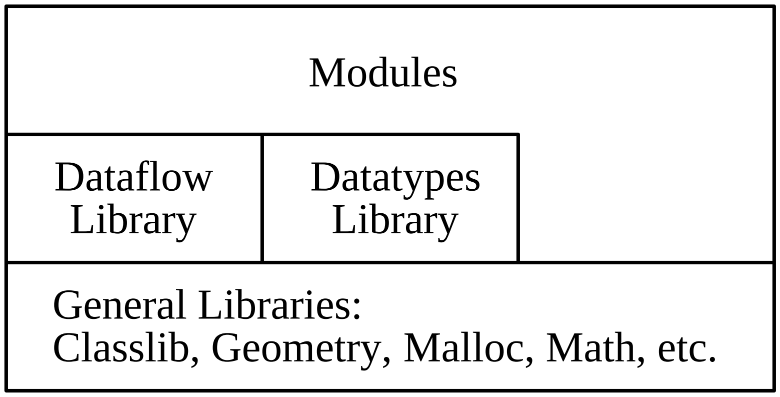


FIGURE 1.1. SCIRun library organization.

SCIRun uses an object oriented design; however, it should be stressed that we have paid careful attention to avoid over-using the object oriented paradigm to a point that efficiency suffers.

In implementing the SCIRun kernel and modules, we leverage off of a powerful toolbox of C++ classes that have been tuned for scientific computing and operation in a multi-threaded environment. We discuss these classes below, starting with the lowest level library and proceeding to more complex libraries. We describe each of the toolbox components here, starting with the lowest layer. In discussing higher layers, we describe how features of the lower layers are leveraged to facilitate implementation.

1.3.1 `Malloc, operator new:` *libMalloc*

We have encountered several problems with the implementations of `malloc/free` and `new/delete` that are available on current Unix systems. Difficulties with the current implementations of `malloc` and `new` include:

1. They are not robust against erroneous behavior. This is particularly confusing when the user's program crashes in `malloc`, while the actual error resulted from freeing a bad pointer in a previous call. A multithreaded environment further exacerbates this problem, allowing errors in one thread to cause another thread to crash.
2. They are not thread-safe (reentrant) on many systems. This is typically the case on systems without a native implementation of threads. Accessing `malloc` and `free` in such an environment can cause frequent non-deterministic crashes.
3. They do not reveal statistics about their operation.
4. They do not return memory to the operating system when it is no longer being used.

5. They are very slow when allocating and deallocating large numbers of small objects.
6. They have a large percentage of memory overhead for small objects.

Of course, the goal would be to resolve all of these problems, but we find that many of the requirements conflict. For example, it is difficult to have bullet-proof behavior against errors without incurring additional overhead, even for small objects.

The implementation of libMalloc centers around the **Pool** class. **Pool** defines a constructor and destructor, as well as the methods `alloc, free, realloc, get_stats` and `audit` as shown below.

```
class Pool {
   Pool();
   ~Pool();
   Mutex lock;
   void* alloc(size_t size, char* ctag, int itag);
   void free(void* p);
   void* realloc(void* p, size_t size);
   void audit();
   void get_stats(size_t statbuf[18]);
   int nbins();
   void get_bin_stats(int bin, size_t statbuf[6]);
   ...
};
```

Pool represents a pool of memory. At startup, there is a single pool, `default_pool`, from which requests from `malloc` and `new` are granted. The implementations of `malloc` and the `new` operator simply call the `alloc` method of the default pool. Subsequently, the `free` and operator `delete` methods call the `free` method of the default pool. The default `malloc` and operator `new` provide generic information as the two tags for the allocation, but there are alternate interfaces that automatically provide the file and line number for these tags.

The `alloc` method uses three slightly different memory allocation algorithms for small, medium and large objects. Based on heuristics from current applications, small objects are those less than 512 bytes, medium objects range from 513 bytes-64k bytes, and large objects are those over 64k bytes. These ranges are configurable at compile time.

Small and medium objects both use an algorithm based on bins. A bin contains a list of free objects. When free space is requested, `alloc` figures out which bin contains objects of the appropriate size, and the first one from the list is removed. Sentinels are placed at the beginning and at the end of the actual allocation. Small and medium bins differ in how the bins are refilled when they become empty. Small bins use an aggressive fill scheme, where 64k worth of objects are placed in the bin's free list in order to minimize the number of refills. Medium objects, on the other hand, use a less aggressive scheme - objects are allocated from a larger pool one at a time. Large objects are allocated with independent `mmap` calls to the operating system. This allows the large objects to be returned to the operating system when they are no longer needed. In order to avoid releasing and re-requesting memory, these large chunks are returned to the operating system (unmapped) in a lazy fashion. It is possible for this policy to fragment the address space of the program, but in practice this has not been a problem, and will never be a problem for 64 bit programs.

The algorithms for the three different allocation ranges are based on the philosophy that bigger objects can afford to use more CPU cycles in trying to be efficient, since large objects will be allocated less frequently and used for a longer period of time. It is also more valuable to minimize waste for large objects than for small allocations.

In order to make the pool thread safe, each of the methods acquires the mutex before accessing or modifying any data in the Pool, and releases the mutex when these operations are complete. The `alloc` and `release` methods attempt to minimize the time that the pool is locked by performing most operations (tag/header manipulation, verification, etc.) without holding the lock.

This implementation resolves all of the problems that we described above, except for items five and six. The memory overhead (item six) is approximately the same as current implementations, and the time overhead for small objects (item five) is considerably smaller, but still too large. In the next section, we will see a mechanism that may be layered on top of libMalloc to resolve these problems.

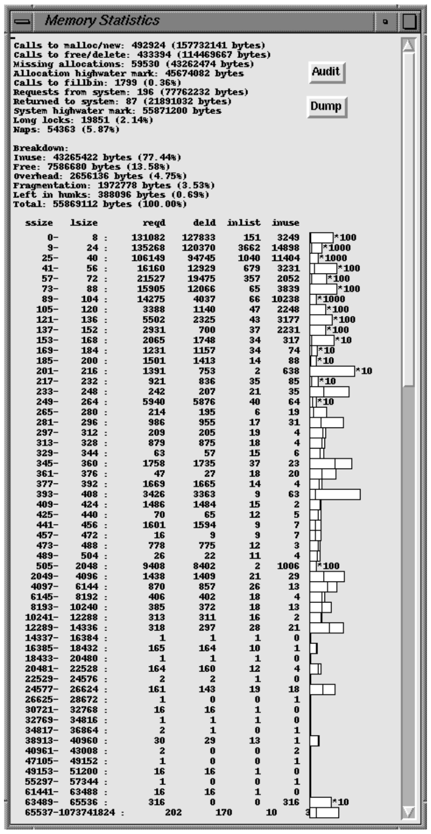
This memory allocator can also reveal statistics about its operation. Figure 1.2 shows these statistics displayed by a running program.

1.3.2 The Multitask Library: libMultitask

SCIRun derives much of its flexibility from its internal use of threads [15]. Threads allow multiple concurrent execution paths in a single program. SCIRun uses threads to facilitate parallel execution, to allow user interaction while computation is in progress, and to allow the system to change variables without interrupting a simulation. However, standards for implementing threads are only starting to appear, and the standards that are appearing are, thus far, cumbersome. libMultitask is a layer that provides a simple, clean C++ interface to threads and provides abstraction from the actual standard used to implement them.

Tasks

The Multitask library provides a class **Task**, which encapsulates a thread. The **Task** constructor requires a name for the **Task** and a priority. A new



```
Calls to malloc/new: 492924 (157732141 bytes)
Calls to free/delete: 433394 (114469667 bytes)
Missing allocations: 59530 (43262474 bytes)
Allocation highwater mark: 45674082 bytes
Calls to fillbin: 1799 (0.36%)
Requests from system: 196 (77762232 bytes)
Returned to system: 87 (21891032 bytes)
System highwater mark: 55871200 bytes
Long locks: 19851 (2.14%)
Naps: 54363 (5.87%)

Breakdown:
Inuse: 43265422 bytes (77.44%)
Free: 7586680 bytes (13.58%)
Overhead: 2656136 bytes (4.75%)
Fragmentation: 1972778 bytes (3.53%)
Left in hunks: 388096 bytes (0.69%)
Total: 55869112 bytes (100.00%)

  ssize   lsize        reqd     deld  inlist  inuse
     0-       8 :    131082   127833     151   3249  *100
     9-      24 :    135268   120370    3662  14898  *1000
    25-      40 :    106149    94745    1040  11404  *1000
    41-      56 :     16160    12929     679   3231  *100
    57-      72 :     21527    19475     357   2052  *100
    73-      88 :     15905    12066      65   3839  *100
    89-     104 :     14275     4037      66  10238  *1000
   105-     120 :      3388     1140      47   2248  *100
   121-     136 :      5502     2325      43   3177  *100
   137-     152 :      2931      700      37   2231  *100
   153-     168 :      2065     1748      34    317  *10
   169-     184 :      1231     1157      34     74  *10
   185-     200 :      1501     1413      14     88  *10
   201-     216 :      1391      753       2    638  *10
   217-     232 :       921      836      35     85  *10
   233-     248 :       242      207      21     35
   249-     264 :      5940     5876      40     64  *10
   265-     280 :       214      195       6     19
   281-     296 :       986      955      17     31
   297-     312 :       209      205      19      4
   313-     328 :       879      875      18      4
   329-     344 :        63       57      15      6
   345-     360 :      1758     1735      37     23
   361-     376 :        47       27      18     20
   377-     392 :      1669     1665      14      4
   393-     408 :      3426     3363       9     63
   409-     424 :      1486     1484      15      2
   425-     440 :        70       65      12      5
   441-     456 :      1601     1594       9      7
   457-     472 :        16        9       9      7
   473-     488 :       778      775      12      3
   489-     504 :        26       22      11      4
   505-    2048 :      9408     8402       2   1006  *100
  2049-    4096 :      1438     1409      21     29
  4097-    6144 :       870      857      26     13
  6145-    8192 :       406      402      18      4
  8193-   10240 :       385      372      18     13
 10241-   12288 :       313      311      16      2
 12289-   14336 :       318      297      28     21
 14337-   16384 :         1        1       1      0
 16385-   18432 :       165      164      10      1
 18433-   20480 :         1        1       1      0
 20481-   22528 :       164      160      12      4
 22529-   24576 :         2        2       1      0
 24577-   26624 :       161      143      19     18
 26625-   28672 :         1        0       0      1
 30721-   32768 :        16       16       1      0
 32769-   34816 :         1        1       1      0
 34817-   36864 :         2        1       0      1
 38913-   40960 :        30       29      13      1
 40961-   43008 :         2        0       0      2
 47105-   49152 :         1        0       0      1
 49153-   51200 :        16       16       1      0
 55297-   57344 :         1        0       0      1
 61441-   63488 :        16       16       1      0
 63489-   65536 :       316        0       0    316  *10
 65537-1073741824 :       202      170      10      3
```

FIGURE 1.2. Statistics of the custom allocator, showing bytes allocated and freed, high water marks, and spinlock statistics. Some of these statistics are also displayed for each bin. To the right of each bin a small graph shows the objects in the freelist and the objects in use for each range of sizes.

thread is created when the creator calls the `activate` method of the **Task** class, which will cause the `body` method to be started in a separate thread. `Activate` will return immediately and will not wait for `body` to complete - thus triggering concurrent execution in the program, similar to a `fork()` operation. However, all of the threads (**Tasks**) will share access to a common heap - unlike the traditional `fork()` function. **Task** is an abstract base class because it does not actually provide a `body` function. Other classes inherit from **Task**, providing a `body` function to do the actual work of the thread. The thread continues until the `body` function returns, or until `Task::exit` is called.

Task also provides static functions to return the number of processors available on the system, to start-up multiple threads for a function, and to cause all threads to exit.

Intertask Communication

The Multitask library also provides a number of convenient synchronization primitives for these tasks to communicate with each other – Inter-Task Communication (ITC). ITC primitives are:

- **Mutex** provides `lock, try_lock` and `unlock` methods.
- **Semaphore** is a counting semaphore, providing `down, try_down` and `up` methods.
- **Barrier** provides a single `wait(int nwait)` method to allow a group of threads to stop executing at the barrier until all `nwait` threads arrive.
- **ConditionVariable** provides `wait(Mutex& lock), cond_signal` and `cond_broadcast` methods.
- **CrowdMonitor** a multiple-reader, singler-writer access control primitive, provides `read_lock, read_trylock, read_unlock, write_lock, write_trylock` and `write_unlock methods`.
- **Mailbox** a fixed-length, thread-safe FIFO (First-In, First-Out communication pipe), allows multiple senders and multiple receivers. This is a template class that provides `send, try_send, receive` and `try_receive` methods. The mailbox allows multiple threads to send tokens to the mailbox and an arbitrary number of threads to receive tokens from the mailbox. These tokens are typically pointers to a message structure. Using this primitive, one can implement threads which behave like a small server.
- Other structures, including **AsyncReply** which provides a single pigeon hole rendezvous point, and classes to perform reduction operations.

The Task and ITC methods have been implemented in four different environments: SGI IRIX (using `sproc` and `us*` primitives), and with Posix threads (aka pthreads), with Solaris threads, and with `setjmp/longjmp`.

1.3.3 Generic Tools: libClasslib

This is a collection of various tools that are valuable in constructing SCIRun's kernel and computational modules. Some of these data structures overlap those available in the Standard Template Library (STL)[16], but our implementation predates common acceptance of STL. In implementing these, we have not tried to make extravagant general-use interfaces. Rather, we have designed our tools to be simple, easy to use, and efficient. We have also designed these interfaces to perform the operations that we require, avoiding the temptation to over-engineer them. Data structures that we have implemented include an unbounded queue, a bounded stack, a dynamic hashtable, and a dynamic array class. These structures use templates to make them usable as containers for any type.

TrivialAllocator

Another useful tool is the **TrivialAllocator** class. This class is designed to increase the efficiency of the `new/delete` operator for small objects that are allocated and deleted frequently. The **TrivialAllocator** simply keeps a list of free objects that are ready to be used. Using the **TrivialAllocator** for a particular class simply requires redefining the operator `new` and operator `delete` methods to use the `alloc` and `free` methods of the **TrivialAllocator**. Using `Small_alloc.alloc` is significantly more efficient than using the general operator `new`, because most of the time it simply returns the first item off of the free list. Objects are allocated in groups, using the second parameter to the constructor as the number of objects to be allocated at a time. `Small_alloc.free` always just puts the object back on the free list. The free list is accessed in a last-in/first-out manner to maximize cache locality. Since these will be used in a multithreaded environment, `alloc` and `free` both require the acquisition and release of a mutex. However, this is a separate mutex from the global allocator, so it will not be subject to the same contention.

This tool allows us to work around the per-object overhead and allocation time required for small, high-use objects. However, it does so at the expense of the overrun detection and consistency checks that our implementation of `new/delete` provides. A future implementation will provide a mechanism by which trivial allocators can be disabled through an environment variable - reducing run time performance, but allowing the consistency checks to be made.

Handles and LockingHandles

Handles are a "smart pointer" mechanism for automatically maintaining reference counts in objects [17]. SCIRun uses these to facilitate the sharing of large data structures between modules. The last **Handle** to "let go" is responsible for deleting the object. Reference counting provides a simple form of garbage collection, without the high overhead and unpredictability associated with a full garbage collection system. The largest weakness is that reference counting can fail to destroy objects which contain circular references (including circular lists, self references, and back pointers). However, it does provide the advantage that objects are destroyed immediately when the last handle releases the object. This feature is essential for large scale scientific computing where the memory resources held by such a handle need to be carefully controlled.

A **handle** contains a single data member `rep`, which contains a pointer to the actual representation. It also defines constructors, a destructor and accessor methods that increment and decrement a reference count in the object. The objects used in a handle must provide a member called `ref_cnt`, which is initialized to zero at construction time. In addition, objects which support the `detach` operation must support a `clone` method which will duplicate the object. Since template syntax is sometimes rather clumsy, it is often convenient to typedef a "smart pointer" type for different object types, such as:

typedef LockingHandle<Object> `ObjectHandle;`

A **LockingHandle** is similar to **Handle**, but with each object also providing a mutex that is locked under all of the `ref_cnt` operations. For more details, see Appendix A.

Timers

Two very convenient classes are the **CPUTimer** and the **WallClockTimer**. These provide a stopwatch interface to acquire time information. Methods include `start, stop, clear`, and `time`. The `time` method simply returns the total accumulated time between `start` and `stop` pairs. Typically, these timers access the Unix timer functions appropriate for the operating system, but they can also utilize a high resolution user-space mappable timer on systems that support it.

Persistent Objects

Modern scientific computing codes use modern data structures that are generally more complex than a large array of numbers. We needed a facility to save complex data structures on disk. In order to solve this problem, we employed the idea of "persistent objects," where complex data structures can be flattened into a data file and then reconstructed into their original form. Since C++ does not support persistent objects automatically, we designed a set of utilities to make these input and output routines in a simple manner.

There are three different levels on which the persistent I/O routines operate:

1. **Primitive:** There is an overloaded global function, `Pio`, for each of the basic types (float, double, int, etc.), which can both read and write these primitive types.

2. **Simple Structure:** For basic structures without class derivations, the user can overload a global `Pio` function to serialize these objects. Usually, this function just calls `Pio` on each of the individual data elements.

3. **Complex Structure:** For a complicated class hierarchy, the user writes a virtual `io` function which emits a class name and version

number, then the `io` method for the superclass, and then calls other `Pio` functions on the individual components.

The basic stream I/O routines support both binary files and somewhat humanly readable text files. Binary input and output uses the Sun Microsystem's XDR library for portable binary I/O, which allows us to move files between different architectures without the need to convert to ASCII text. Binary files are sometimes smaller, and are always a factor of 3-5 faster in reading and writing. In addition, we support versioning of objects so that we can change the data structures in the code without requiring the conversion of all old data files.

To illustrate the simplicity of reading and writing objects in this manner, consider an example of a 3D tetrahedral mesh:

```
class Mesh {
    Array1<Node*> nodes;
    Array1<Element*> elems;
    ...
};

#define MESH_VERSION 1

void  Mesh::io(Piostream& stream)
{
    stream.begin_class("Mesh", MESH_VERSION);  // identify type & version
    Pio(stream, nodes);        // read/write the nodes
    Pio(stream, elems);        // read/write the elements
    stream.end_class();
}
```

The **Array1** classes know how to read and write themselves:

```
template<class T>
void Pio(Piostream& stream, Array1<int>& array)
{
    stream.begin_class("Array1", ARRAY1_VERSION);  // id the type & version
    int size=array._size;    // grab the size of the array...
    Pio(stream, size);    // ...and write it
    if(stream.reading())    // if we're reading...
        array.resize(size);    // ...allocate space for objects
    for(int i=0;i<size;i++)
        Pio(stream, array.objs[i]);    // read/write all of the objects
    stream.end_class();
}
```

and the **Node/Element** classes know how to read and write themselves:

```
void Pio(Piostream& stream, Node*& node)
{
    stream.begin_cheap_delim();
    if(stream.reading()) // if we're reading...
        node=new Node(); // ...allocate a new node
    Pio(stream, node.pt); // read/write the node's location
    stream.end_cheap_delim();
}

void Pio(Piostream& stream, Element* elem)
{
    if(stream.reading()) // if we're reading...
        elem=new Element(); // ...allocate a new element
    stream.begin_cheap_delim();
    Pio(stream, elem->n[0]); // read/write all of the
    Pio(stream, elem->n[1]); //    indices for the four
    Pio(stream, elem->n[2]); //    nodes composing the
    Pio(stream, elem->n[3]); //    tetrahedral element
    stream.end_cheap_delim();
}
```

It is important to remember that these small functions will both read and write the mesh for both binary files and text files. This feature virtually eliminates the potential for incompatibilities between the reading code and the writing code. However, the real power comes when we emit a scalar field based on these meshes:

```
class ScalarFieldUG : public ScalarField {
    MeshHandle mesh;
    Array1<double> data;
    ...;
};

void ScalarFieldUG::io( Piostream& stream)
{
    stream.begin_class("ScalarFieldUG", SCALARFIELDUG_VERSION);
    ScalarField::io(stream); // This serializes the base class
    Pio(stream, mesh); // read/write the mesh
    Pio(stream, data); // read/write the data
}
```

Then, we could emit multiple scalar fields:

```
    Pio(stream, field1); // read/write field1
    Pio(stream, field2); // read/write field2
```

```
   ...
```

In this example, the fields might share a common mesh. In this case, the mesh would be written into the file only once. We have omitted many of the details of how this is implemented internally, but the `Pio` routines do not need to be concerned with this mechanism – it is handled internally by the **Piostream**.

We have found that this mechanism is a very powerful way to implement I/O for complex data structures. The versioning system allows us to use datafiles that were written years ago without converting them. Using the same code for reading and writing drastically reduces the number of errors in the input/output routines. In addition, this mechanism has increased the utility of binary files by avoiding the need to write a separate I/O function.

1.3.4 Geometry Library

libGeometry provides **Point**, **Vector**, **Plane**, **Ray**, **Transform**, and **BoundingBox** classes for convenient computation of 3D geometry. The addition, subtraction, and multiplication operators have all been implemented to allow these components to be used in a convenient fashion.

We have chosen to separate the concept of a **Point** from the concept of a **Vector** [18]. For the sake of efficiency, these are both specialized for 3 dimensions, with an `x`, `y`, and `z` component. A **Point** differs from a **Vector** only in the operations that can be performed on it. A **Point** indicates position in 3D space, and a **Vector** indicates direction. A **Point** subtracted from another **Point** produces a **Vector**, a **Vector** added to another **Vector** produces a **Vector**, and so forth. A cross product is defined only for **Vectors**, since they do not make geometric sense for **Points**. This has proven to be a useful way to help the programmer reason about geometric transformations and to write correct code for geometric manipulations.

1.3.5 The Math Library

libMath provides a somewhat eclectic collection of various core numerical functions, such as `Min`, `Max`, and `Abs`, which have all been overloaded for various numerical types. libMath also contains several core linear algebra loops, such as dot products, vector-vector multiply, etc. These loops have all been highly tuned to take maximum advantage of various architectures.

1.4 The Datatypes Library

The libraries described above are the building blocks from which the SCIRun kernel and SCIRun modules are created. The next layer consists of the Datatypes library. These datatypes are the data structures that get passed from one SCIRun module to another. Reflecting our work in Finite Element Analysis, the Datatypes library has four main parts:

1. **Meshes:** Unstructured tetrahedral meshes.
2. **Fields:** Scalar and Vector fields representing scalar and vector valued functions of space. These are currently defined on regular grids or with per-element or per-node values on an unstructured mesh.
3. **Surfaces:** Triangular mesh and parametric surface boundaries, optionally tagged with boundary condition information.
4. **Matrices:** Dense, Compressed row storage (symmetric and non-symmetric), Tridiagonal, and other matrix formats.

The Datatypes library is a powerful set of data structures for scientific computing, but it also provides a powerful method of extending SCIRun. A typical dataflow oriented program can be extended by adding new modules to implement new algorithms. However, SCIRun can be extended by extending the abstract interfaces in the Datatypes library to operate on other data formats. For example, one could make a new **Field** datatype which implements the **ScalarField** interface for some type of spectral method. Then most downstream modules would be able to operate on the data without modifying those modules and without converting and duplicating the data. Note that we are careful to say "most of the time." There are times when we violate these abstract interfaces for efficiency purposes. There will be an example of this shown when we discuss the **IsoSurface** module.

1.4.1 The Mesh Class

The **Mesh** class is not an abstract interface but is meant to be a powerful class for operating on unstructured grids. A **Mesh** consists of a set of **Nodes** and a set of **Elements**. A **Node** contains its **Point** in 3D space, and an **Element** contains a pointer to four different **Nodes**. This data completely specifies the mesh, but we also maintain other information for making mesh operations efficient. The full version of the **Node** data structure also contains a list of the elements to which it is connected. The **Element** data structure contains the face neighbors - the elements which neighbor its four faces. In addition, the **Element** data structure can optionally contain the element basis function and element volume. The element basis function is a large amount of data, increasing the size of the element data structure from 40 bytes to 176 bytes. This can cause memory limitations for large problems, but it avoids repeated recalculation of these basis functions and so increases the speed of finite element matrix assembly, element interpolation, and many other operations. The user may select at compile time whether or not to maintain the element basis functions.

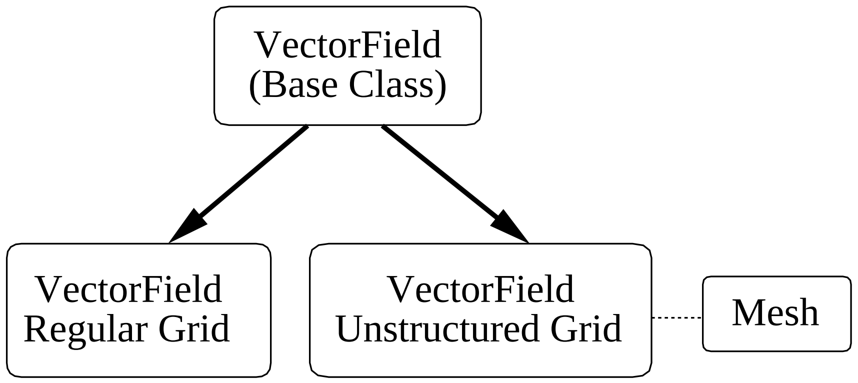


FIGURE 1.3. VectorField class hierarchy showing the how the unstructured grid and regular grid versions are derived from a common base class. The dotted line to the **Mesh** shows that the Unstructured Grid version contains a 3D tetrahedral **Mesh**. In addition, the field contains a Vector value for each node or element in the mesh.

We could have chosen to make meshes of arbitrary dimension, but we opted for the simplicity and efficiency afforded by hard coding the dimension to three.

1.4.2 Fields Data Structures

The fields data structures provide two base classes: **ScalarField** and **VectorField**. We chose to separate these two types in order to clarify which operations make sense on which type. **ScalarField** provides interfaces for interpolating the value at a point (`interpolate`), determining the minimum and maximum scalar values (`minmax`), and for determining the spatial bounds of the field (`get_bounds`). There are currently two different implementations of the **ScalarField**: **ScalarFieldRG**, which defines a scalar field using a regularly sampled grid, and **ScalarFieldUG**, which contains a **Mesh**, and values for each node or element. **ScalarFieldRG** and **ScalarFieldUG** implement the `interpolate`, `minmax`, and `get_bounds` methods described above. Most modules use this abstract interface to access **ScalarField** data.

The **VectorField** class is similar to the **ScalarField** class, but does not provide a `get_minmax` interface. Figure 1.3 shows the class structure for the **VectorField** classes. The **ScalarField** classes are structured similarly.

1.4.3 Surfaces Data Structures

There is another class hierarchy that describes a surface in three-dimensional space. For our work, the most important surface is a triangulated surface defined as a collection of 3D points and the corresponding triangles that connect them. These surfaces can be tagged with boundary condition information for later integration with finite element problems. A few other surfaces are provided such as cylinders, spheres, and points (a degenerate surface).

1.4.4 The Matrix Class

Matrices: We implement a base class called **Matrix**, which defines `multiply` and `transpose multiply` abstract methods. Instances of this class may be implemented with *SparseLib++* [12], or other sparse matrix packages. SCIRun also implements a compressed-row storage matrix which has been highly tuned for the architectures which we use most often.

The discussion of the **SolveMatrix** module will display how these abstract interfaces are used to implement the conjugate gradient algorithm without regard for the actual layout of the matrix. Other operations, such as matrix-matrix multiply, are better implemented with code that checks the actual type of the matrices and uses the most efficient multiplication algorithm and resultant matrix format.

1.5 Dataflow

To this point, we have described a powerful set of computational elements that may be composed by a variety of means. SCIRun composes computational algorithms with these data elements using a dataflow style "boxes and wires" approach. An example of a dataflow network is shown in Figure 1.4.

1. A module, drawn as a box in the network, represents an algorithm or operation. A set of input ports and a set of output ports define its external parameters.

2. A port provides a connecting point for routing data to different stages of the computation. Ports are typed: each datatype has a different color, and datatypes cannot be mixed. In SCIRun, ports can be added to and removed from a module dynamically. Input ports are represented on the top of the module icon, and output ports are on the bottom. Output ports can cache datasets to avoid recomputation. The user can select which datasets should be cached and which should not.

3. A datatype represents a concept behind the numbers. Datatypes are quantities such as scalar fields or matrices, but are not the specific representations, such as regular grids, unstructured grids, banded matrices, sparse matrices, etc.

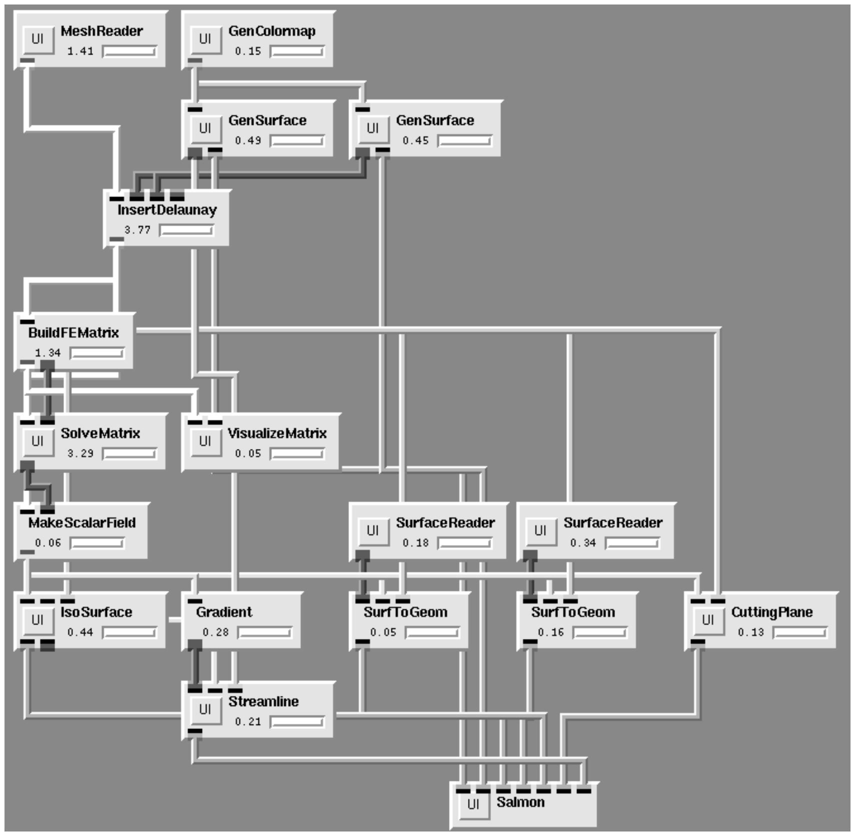


FIGURE 1.4. An example of a fairly complex dataflow network, showing the modules (the boxes), the connections (the wires between them), and the input/output ports (the points on the modules that the wires connect). On a color monitor, the colors of the ports and connections indicate the type of data that flows through them.

4. A connection connects two modules together: the output port of one module is connected to the input port of another module. These connections control where the data is sent to be computed. Output ports can be connected to multiple input ports, but input ports accept only a single connection.

5. A network consists of a set of modules and the connections between them. This represents a complete dataflow "program."

The dataflow library is responsible for deciding which modules need to be executed and when. A module is typically re-executed when the user changes a module parameter, when new data is available at an input port, or when data is required at its output port.

When the user moves a user interface component, such as a 2D slider or a 3D widget, the module sends a message to the scheduler requesting execution. The scheduler analyzes the dataflow graph to determine what other modules will also need to be executed. The dataflow scheduler uses the following algorithm to determine which modules need to be executed:

```
execute_list = modules that requested execution;
resend_list = empty;
foreach module in the execute_list {
  foreach module connected to an output {
    if the connected module is not in the execute list,
      add it
  }
  foreach module connected to an input {
    if the connected output port has a cached dataset,
      add it to the resend list
    else add it to the execute_list
  }
}
cull all ports from the resend_list whose modules appear
      in the execute list
send resend messages to the ports in the resend list
send execute messages to the modules in the execute list
```

Note that all of the modules are executed at the same time. The modules actually communicate directly with each other using a threadsafe FIFO for the dataset handoffs. Each thread will wait for data to appear on the input ports. A module does not have to gather data from the inputs in order, and it may interleave computation with receiving the data. However, in order to satisfy the requirements for determinacy in a dataflow program, the module is not able to request datasets in a first-come, first served or any other non-deterministic order. It is important to note that modules do not send actual data, but rather a handle to the data (see Appendix A. for a discussion of handles).

Sending a dataset to the `matrix_out` port looks like this:

```
MatrixHandle matrix = new DenseMatrix(...);
.. build the matrix ..
matrix_out->send(matrix);
```

Receiving a dataset from the `matrix_in` port looks like:

```
MatrixHandle matrix;
if(!matrix_in->receive(matrix))
     return; // returns false if the dataset is not
             // available - this is usually due to
             // an error upstream
```

```
  .. do cool stuff with the matrix ..
```

For each input port, the module must do exactly one receive, and for each output port, the module must do exactly one send. If the module wishes to send multiple datasets (as intermediate results, or for feedback loops), it can use a special method called `send_multi`, which also arranges for the modules downstream to be executed again.

The `send/receive` pairs shown above are a simple atomic protocol where the entire dataset is transferred at once. However, it often makes more sense to use "fine grain" dataflow where appropriate. Fine-grain protocols are tuned to the specific layout of the data, such as for receiving slabs of a regular grid for isosurfacing or scanlines for image processing. In such cases the module receives scanlines and sends scanlines for each scanline in the image, but it semantically sends or receives the full dataset exactly once.

We chose to implement a centralized scheduler in order to avoid redundant module re-execution in a branching situation. Since we leave the central scheduler out of the loop for individual dataset handoffs, it does not become a bottleneck in the execution of the program.

The Dataflow library also contains a base class from which all modules are derived. This class contains the data structures that are required to implement the dataflow structures; it also contains various utility functions, such as `update_progress`, a function that the module writer can call periodically to update the graph on the module icon that indicates the approximate percentage of work the module has completed.

1.6 Steering in a Dataflow System

All of the pieces described above have been designed to support steering of large scale scientific simulations. SCIRun uses three different methods to implement steering in this dataflow oriented system:

1. Direct lightweight parameter changes. The **SolveMatrix** module allows the user to change the target error even while the module is executing. The parameter change does not pass a new token through the dataflow network, but simply changes the internal state of the **SolveMatrix** module, effectively changing the definition of the operator rather than triggering a new dataflow event.

2. Cancellation. When parameters are changed, the module can choose to cancel the current operation. For example, if boundary conditions are changed, it may make sense to cancel the computation in order to focus on the new solution. This makes the most sense when solving elliptic problems, since the solution does not depend on any previous solution.

3. Feedback loops in the dataflow program. For a time varying problem, the program usually goes through a time stepping loop with several major operations inside. The boundary conditions are integrated in one or more of these operations. If this loop is implemented in the dataflow system, then the user can make changes in those operators which will be integrated on the next trip through the loop.

These three methods provide the mechanisms whereby computational parameters can be changed during the execution of the program.

1.7 Modules

We have thus far described a handy set of tools and a dataflow system for imposing control structure. However, the real value in SCIRun comes from how these components are leveraged in actual algorithms, or in SCIRun - actual modules. This section covers how a module is constructed and then describes some of the modules in SCIRun, showing how they use the features that we have described above.

1.7.1 Writing a Module

The process of writing a new module involves writing a new C++ class. The constructor for this class creates the input and output ports for the module and defines parameters which the user interface may control. A single virtual function, `execute`, is overloaded to perform the actual work of the module. The `execute` function typically receives data on the input ports, performs some computation and then sends data on the output ports. Other callback functions can provide input from user interface components and 3D widgets. Adding a user interface to a module involves writing a small Tcl script which sets up the components of the interface.

Existing code may be integrated into SCIRun by writing a small wrapper module which passes data into and out of an existing C, C++ or Fortran program. The wrapper module may be required to translate data structures before passing them to the existing code and before sending them down the pipe. In order to avoid this translation, existing code can also be incorporated by extending the class hierarchy that flows through the dataflow network. For example, instead of translating to a specific **ScalarField** class the module could send a new subclass of the **ScalarField** that would provide `interpolate` and other methods for use by downstream modules.

SCIRun was originally designed as an environment for developing new simulations and computational components. We are currently working on ways to more automatically incorporating existing packages into the SCIRun visual programming environment.

1.7.2 FEM and Matrix Modules

The **BuildFEMatrix** module takes a tetrahedral mesh and builds a stiffness matrix for a finite element approximation. The current version of this module approximates the generalized 3D Poisson's equation (or its homogeneous counterpart, Laplace's equation) in the discretized physical domain Ω:

$$\nabla \cdot \sigma \nabla u = f \tag{1.1}$$

where f is a vector of sources and σ is 3×3 tensor corresponding to the materials within the domain. The scalar field u is subject to the boundary conditions:

$$u = u_0 \text{ on } ?_1 \text{ and } \sigma \nabla u \cdot \mathbf{n} = 0 \text{ on } ?_2 \tag{1.2}$$

The mesh elements have been tagged with a corresponding tensor property and boundaries have been tagged with the appropriate boundary condition. The module makes two passes - first to build the sparse structure of the matrix using a compressed row storage scheme, and second to fill in the resulting sparse matrix. Building the matrix is done in parallel in order to take full advantage of multiple processors. The module also builds the right hand side (load) vector. Users who wish to use other governing equations can extend this module to build the stiffness and load matrices appropriately.

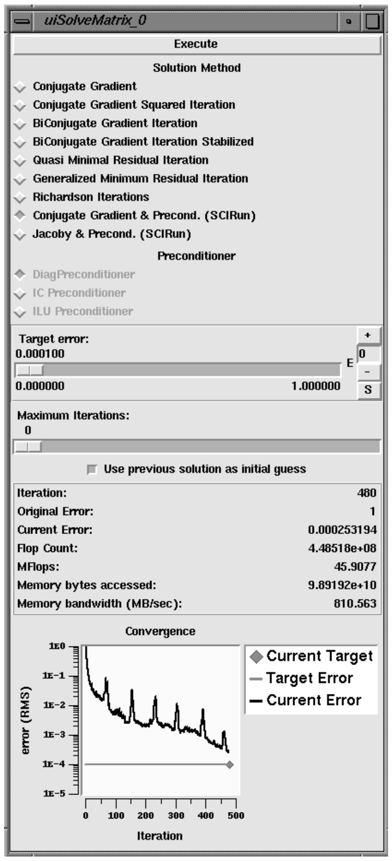
Usually, these matrices are passed into the **SolveMatrix** module, which uses direct or iterative schemes to find the solution, x to the matrix equation:

$$Ax = b \tag{1.3}$$

The **SolveMatrix** module graphs the convergence of the residual as the algorithm iterates. The algorithm stops when the residual is less than some target level (i.e., the solution has converged). The user may move the target residual up or down while the solution is still in progress, thus steering the computation in a simple manner. Figure 1.5 shows the interface for this module in operation.

SolveMatrix uses the `mult` and `mult_transpose` virtual methods in the **Matrix** base class to perform each iteration. As a result, the user could implement a new matrix type which **SolveMatrix** could use without even recompiling the module. This new matrix type does not even need to explicitly store the matrix - it could build the matrix dynamically or implicitly.

SolveMatrix also has an option to use the previous solution as the initial guess for the iterative solver. When small changes are made in the boundary conditions (a common case in an engineering design scenario), the system converges rapidly. This is one instance where an integrated system can actually be more efficient than implementing separate programs for each of the phases.



```
Target error:
0.000100
0.000000                         1.000000
Maximum Iterations:
0
Use previous solution as initial guess
Iteration:                               480
Original Error:                            1
Current Error:                   0.000253194
Flop Count:                      4.48518e+08
MFlops:                              45.9077
Memory bytes accessed:           9.89192e+10
Memory bandwidth (MB/sec):           810.563
```

FIGURE 1.5. The user interface for the **SolveMatrix** module, showing the iterative methods available, the graph of the convergence and the target residual, which may be changed while the computation is in progress.

The **VisualizeMatrix** module draws a figure representing the non-zeros in a matrix. Non-zero entries are draw with small red dots, and zero entries are left black. This gives the user a quick representation of the sparse structure of the matrix, as shown in Figure 1.6. A magnifying glass can reveal the actual numbers in a portion of the matrix by clicking the mouse in the desired region. **VisualizeMatrix** uses two additional virtual functions in the **Matrix** base class: `getRowNonZeros`, which returns the nonzeros in a particular row, and `get`, which returns the number in a specific row and column of the matrix.

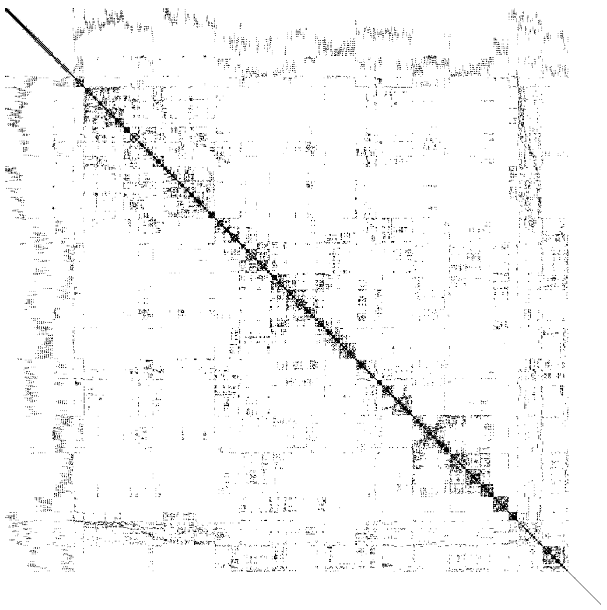


FIGURE 1.6. Visualization of the sparse structure of a matrix. The black dots represent the non-zeros in a matrix with approximately 10,000 rows and columns.

1.7.3 Readers and Writers

The **Reader** and **Writer** modules are very straightforward. They simply call the Persistent object `io` routines which were described above in Classlib (1.3.3). The readers read in a text or binary persistent object file and send the resulting object downstream, and the writers receive an object and then write it out to disk. Since the support for these modules has been provided by the lower layers, these modules do nothing more than provide user interfaces for the filenames. These modules are automatically generated, but provide hooks for adding user-defined reader/writer functions.

1.7.4 Visualization Modules

In implementing SCIRun's datatypes, we considered what type of operations SCIRun modules would require. One example of such a consideration was the implementation of the vector and scalar field datatypes. Each of these fields comes in a variety of flavors, corresponding to the internal representation of the data - implicit, explicit, parametric - and the topology of the field - structured (implicit) or unstructured (explicit). But in designing the field datatype, we chose several generic operators to allow module writers to access information from the field without needing to know how the field is internally represented. As discussed above (see section 1.4.2), these operators can query the field for minimum and maximum scalar values or geometric bounds, or for the field's value at an arbitrary point in space. This last operation, retrieving the value of the field at any location, is implemented by an interface called `interpolate`. In the following subsection, we will discuss how we have exploited this generic operator in several of our modules, and we will provide an example of when we chose to side-step this abstraction in order to write a more efficient algorithm based on the specific internal representation of the field.

Interpolation

Many of SCIRun's modules exploit the field interface `interpolate`. The `interpolate` method takes a point as an argument and calculates the value of the field at that specific location. It accomplishes this by determining which element of the field contains the point, and linearly interpolating the values at the element's vertices.

One example of a module that calls the `interpolate` method is the **Streamline** module. The **Streamline** module is used for vector field visualization: by tracing particles advecting through the vector field, the user can examine local flow phenomena around critical points (such as vortices and turbulence) while also gaining a global sense of the field's flow. Figure 1.7 shows an example of the electrical current flow near the heart as computed by the **Streamline** module. We compute the paths of particles through the vector field as discrete line integrals, each corresponding to a streamline. We have several implementations of this integration; one of these is a fourth order Runge-Kutta method. Given a point `p`, we integrate along the path to find the next point along the streamline, `pNew`. The following piece of code accomplishes this:

```
Vector f1, f2, f3, f4;
Point p1, p2, p3, pNew;

vfield->interpolate(p, f1);
p1 = p + (f1 * 0.5);
vfield->interpolate(p1, f2);
p2 = p + (f2 * 0.5);
vfield->interpolate(p2, f3);
p3 = p + f3;
vfield->interpolate(p3, f4);
pNew = p + (F1 + F2 * 2.0 + F3 * 2.0 + F4) / 6.0;
```

The principal idea of this algorithm is that we find the vectors corresponding to particular points near `p` and take a weighted average of these vectors to determine our next location along the streamline. Note that this algorithm will work for any type of vector field that implements the `interpolate` interface. By designing a generic interface to the `interpolate` method, we have abstracted away the details of how interpo-

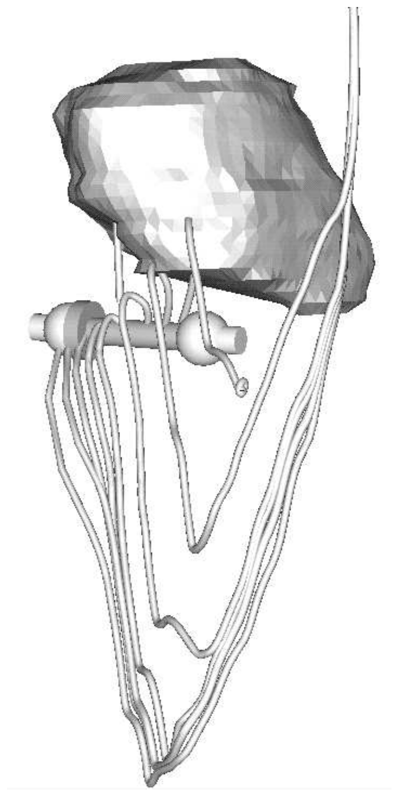


FIGURE 1.7. Visualization of a defibrillator design simulation, showing an electrode, the surface of the heart (epicardium), a 3D widget (rake), and electrical current lines (streamlines). The other electrode in the simulation is obscured by the heart. Much of the current leaving the visible electrode travels away from the heart, due to the high conductivity of blood in a nearby vessel.

lation is implemented for different field types, and consequently the module writer needs not be concerned with what *type* of vector field is generating the streamlines.

Another module that uses the `interpolate` interface is **SurfToGeom**. This module takes a surface, a scalar field, and a colormap as input, and outputs colored geometric primitives that can be passed to a renderer. The module computes these primitives in the following way: it queries the field for scalar values corresponding to points on the input surface; these points are assigned material properties by indexing the resultant scalar values in the color table; and finally, these colored points are grouped into geometric primitives (such as triangles or tri-strips). Here again the **SurfToGeom** module utilizes the `interpolate` interface in order to find the scalar values of the points on the surface.

The user programming the **SurfToGeom** module does not ever have to worry about the type of the incoming scalar field - structured, unstructured, or spectral. As long as the `interpolate` interface is provided, the **SurfToGeom** module will work. Furthermore, as new scalar field types are implemented in the future, they will automatically work with the **SurfToGeom** module as long as the `interpolate` method is implemented, without requiring code to be rewritten or even recompiled.

In contrast to the **Streamline** and **SurfToGeom** modules, the **IsoSurface** extraction module does not use the `interpolate` method to find values at points within the field. While the isosurfacing algorithm could have been implemented this way, we were able to dramatically improve the speed of our algorithm by taking into account (and exploiting) the underlying structure of the data. For example, we have implemented Lorensen and Cline's Marching Cubes algorithm [19] as one option for isosurface extraction. For the case of both structured and unstructured grids, it is much faster to solve the "forward problem" - finding the intersection of the field with each element than it is to solve the "inverse problem" - using various calls to `interpolate` to track down the isovalue points within the field. In this case, it would have been highly inefficient to use the generic `interpolate` operator, so we wrote data-structure dependent code to solve the problem. The **IsoSurface** module lacks the abstraction (and as a result, the cleanliness) of the **Streamline** and **SurfToGeom** modules. As other field types are implemented, we will have to write new isosurfacing code to handle each field type. In this case, there was considerably more to be gained by using data-structure specific information in our implementation than there was to be gained by using the generic accesses permitted by abstracting this information away.

1.7.5 Salmon Module

One of SCIRun's key modules is the graphical viewer called **Salmon**. **Salmon** was named for its ability to spawn multiple views (Roe), and its ability to send messages upstream in the dataflow network. **Salmon** collects the geometric primitives from any number of modules and presents them in a single 3D view. The user can rotate, scale, and translate the objects, as well as manipulate lighting, camera parameters and rendering method, in order to obtain the desired view. Other views can be spawned to separate windows in order to simultaneously display the objects from other viewpoints or with different subsets of the objects.

Geometric primitives are passed from the modules to **Salmon** as a subset of a scenegraph. These scenegraphs are a tree-based display list that define geometric primitives, colors, and other rendering parameters. Drawing the scene involves traversing the graph and emitting OpenGL commands at each node. Scenegraphs can be composed, stored to disk via the persistent object mechanism, and then read back for later display. Parameters in the scenegraph can be changed dynamically by the module using the **Crowd-**

Monitor (multiple reader, single writer) lock described above in libMultiask (1.3.2).

In addition to using the **Salmon** module for visual output, we can also use it for 3D input by allowing the user to interact with specific objects in the scene. These objects, called Widgets[20], allow the user to intuitively augment parameters directly in the 3D scene. For example, in order to provide the starting point for a streamline advection, the user simply drags a **SphereWidget** around in the scene. This interaction is generally more intuitive to a user then typing in numbers or manipulating 2D sliders. This interaction becomes even more powerful when a rake [21, 22] is used to specify a line of starting points for streamline advection.

1.7.6 Other Modules

We have talked about several of the Modules in SCIRun, but there are many more:

- **FEMError** - computes the upper and lower error bounds of a finite element simulation [23, 24].
- **MeshRefiner** - uses the Error fields described above to decide where to add new nodes and remove old nodes in order to refine and derefine the mesh.
- **Gradient** - computes a vector field which is the gradient of the given scalar field.
- **Magnitude** - computes a scalar field which is the magnitude of the given vector field.
- **FFTImage / IFFTImage** - takes the FFT (inverse FFT) of an image, producing another image.
- **FilterImage** - multiplies two images together to perform a filter operation in frequency space.

Even this is not an exhaustive list. Despite the number of modules available, the user will always need something different than what is supplied. In such cases, the user can simply create a new module (by writing a small C++ program). User modules are dynamically linked with SCIRun, thus avoiding the need to relink the SCIRun executable.

1.8 Applications of SCIRun in Computational Medicine

Here we address the application of SCIRun to two bioelectric field problems in medicine: simulation of cardiac defibrillation and simulation of temporal lobe epilepsy.

Every year, a large number of people die suddenly because of abnormalities in their hearts' electrical system (cardiac arrhythmias) and/or from coronary artery disease. While external defibrillation units have been in use for some time, their use is limited because it takes such a short time for a heart attack victim to die from insufficient oxygen to the brain. Lately, research has been initiated to find a practical way of implanting electrodes within the body of patients with recurring and life-threatening arrhythmias to defibrillate a person automatically upon onset of cardiac fibrillation. Because of the complex geometry and inhomogeneous nature of the human thorax and the lack of sophisticated thorax models, most past design work on defibrillation devices has relied on animal studies. We have constructed a large scale model of the human thorax, the Utah Torso Model [6, 7, 25, 26], for simulating both the endogenous fields of the heart and applied current sources (defibrillation devices). Using these computer models, we are also able to simulate the multitude of electrode configurations, electrode sizes, and magnitudes of defibrillation shocks. Figure 1.8 shows the results of such a simulation. Given the large number of possible external and internal electrode sites, magnitudes, and configurations, it is a daunting problem to computationally test and verify various configurations. For each new configuration tested, geometries, mesh discretization levels, and a number of other parameters must be changed.

Excitation currents in the brain produce an electrical field that can be detected as small voltages on the scalp. By measuring changes in the patterns of the scalp's electrical activity, physicians can detect some forms of neurological disorders. Electroencephalograms, EEGs, measure these voltages; however, they provide physicians with only a snapshot of brain activity. These glimpses help doctors spot disorders but are sometimes insufficient for diagnosing them. For the latter, doctors turn to other techniques; in rare cases, they rely on investigative surgery.

Such is the case with some forms of epilepsy. To determine whether a patient who is not responding to medication has an operable form of the disorder, known as epilepsy (most of which occur in the temporal lobe), neurosurgeons use an inverse procedure to identify whether the abnormal electrical activity is highly localized (thus operable) or diffused over much of the brain.

Using SCIRun, scientists and engineers are able to design internal defibrillation devices and source models for the epileptic foci, place them directly into the computer model, and automatically change parameters (size, shape and number of electrodes) and source terms (position and magnitude of voltage and current sources) as well as the mesh discretization level needed for an accurate finite element solution. Furthermore, engineers can use the interactive visualization capabilities to visually gauge the effectiveness of their designs and simulations in terms of distribution of electrical current flow and density maps of current distribution.

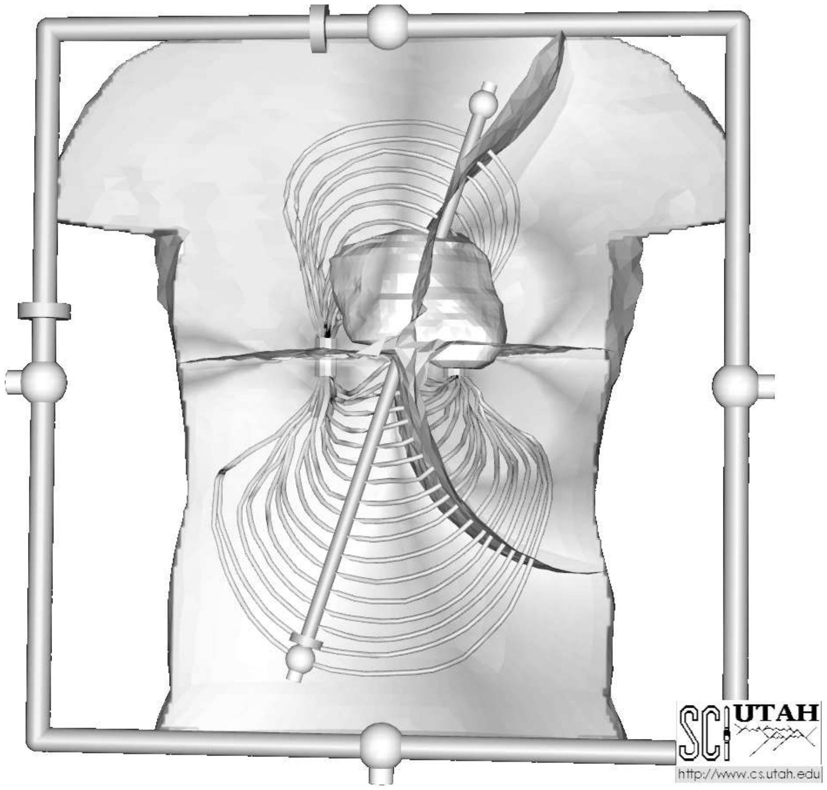


FIGURE 1.8. Visualization of the electrical current lines and an isovoltage surface from a simulation of a cardiac defibrillator design simulation.

Mathematically, these problems are governed by variations of the generalized Poisson's equation for electrical conduction in the physical domain Ω of the thorax or head [27]. Thus one solves (1.1) where $u = \Phi$ are the electrostatic voltages, $f = -I_V$ are the electrical current sources and σ is an electrical conductivity tensor. The boundary conditions in (1.2) are such that $?_1$ is the surface of the internal defibrillator electrodes and $?_2$ is the surface of the torso. $u_0 = \Phi_0$ specifies a Dirichlet boundary of known voltages and $\sigma\nabla\Phi \cdot \mathbf{n}$ represents the current flow normal to the surface of the torso (scalp), which is zero for the insulated boundary of the thorax (or head).

Once the electrostatic potentials are known, one can calculate the current density $\mathbf{J}$ according to:

$$\mathbf{J} = -\sigma\nabla\Phi. \tag{1.4}$$

For the defibrillation problem, electrodes are either implanted internally or applied directly to the chest in order to deliver sufficient electric energy to stop the irregular heart rhythms that signify a fibrillating heart [28, 29].

Mathematically, this can be posed as solving equations (1.1-1.4) with the voltage boundary condition applied on a portion of the torso boundary $\Sigma \subseteq ?_2$ for external defibrillation or from the surface of the defibrillation electrode(s) within the volume of the thorax for internal defibrillation.

Past (and much current) practice regarding the placement of the electrodes for either type of defibrillator has been determined by clinical trial and error. One of our goals is to allow engineers to use SCIRun to assist in determining the optimum electrode placement, size, shape, and strength of shock to terminate fibrillation by solving equations (1.1-1.4) within a detailed model of the human thorax [30, 31, 32, 33, 34].

For the neuroscience problem, the epileptic foci are represented as a set of idealized dipole sources situated in the brain. Using a model of the human skull and brain, the direct EEG problem is posed by solving equations (1.1-1.4) for the voltage and current distribution within the brain and upon the surface of the scalp. For the inverse EEG problem, measured scalp voltages are used as the inputs and equation (1.1) is solved for the source currents I_V. We are currently using SCIRun to investigate the direct and inverse EEG problems as well as using SCIRun as an interactive modeling and visualization tool.

A SCIRun System for Bioelectric Field Problems

A network that can be used to model cardiac defibrillation is shown in Figure 1.4. A similar network is used for a forward solution in the neuroscience application. The network consists of the following modules:

- **SurfaceReader** reads a triangulated surface definition from a file. One of these modules will read the torso boundary (body surface) geometry, and the other will read the epicardium (heart surface) geometry. In the neuroscience application, there may be several surfaces - including the scalp surface, the scalp-skull interface, the skull-cerebrospinal fluid (CSF) interface, the CSF-grey matter interface, and the grey-white matter interface.

- **GenSurface** will generate two cylindrical electrodes for the defibrillation study. Parameters in the interface allow the scientist to control the discretization of the electrodes. In the neuroscience application, these surfaces may represent dipole or other types of source configurations in the brain.

- **SurfToGeom** converts the surface definitions into displayable geometry. A flag in the user interface controls whether or not the geometry is movable. The epicardium and torso boundary should not be moved, since they correspond to physical geometry. The electrode cylinders, on the other hand, must be moved so that various placements can be tested. The **SurfToGeom** module provides 3D widget handles which allow the user to manipulate these surfaces directly. An optional input parameter selection will map scalar field values onto the surface.

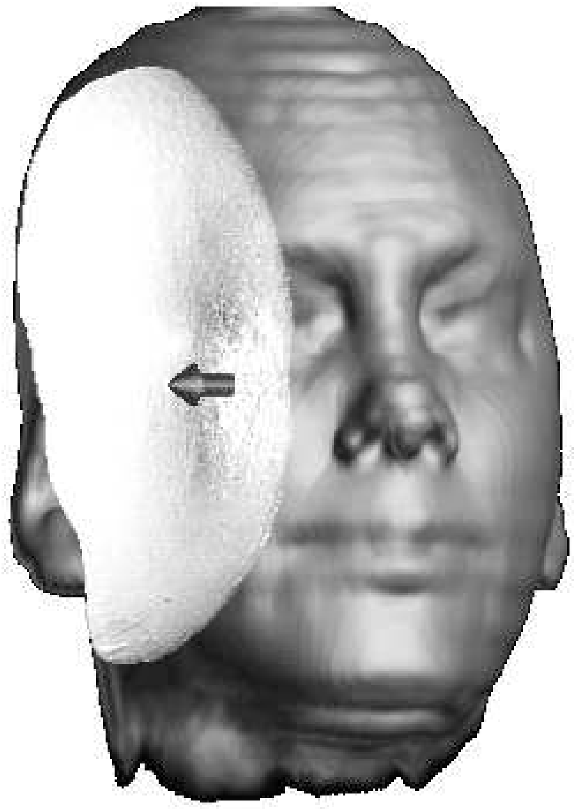


FIGURE 1.9. Visualization of a computational neuroscience simulation, showing an epileptic focus (a dipole source, indicated by the arrow) and an isovoltage surface in the resulting field.

This input is attached to the epicardium and shows the voltages on the surface of the heart.

- **ApplyBC** applies boundary conditions to the various surfaces. The torso and scalp have a zero-flux Neumann boundary. In the defibrillation problem, the two electrode cylinders have Dirichlet boundary conditions corresponding to their respective voltage. In the neuroscience application, there are discrete current sources. The voltage and current sources may be changed interactively.
- **GenerateMesh** discretizes the volume defined on the outside by the torso (scalp) and on the inside by the two voltage source electrodes. This discretization occurs after the application of the boundary conditions so that the mesh generator may optimize the mesh for the particular boundary conditions.

- **BuildFEMatrix** uses the mesh structure, the boundary conditions, and finite element theory to construct a matrix that describes the user-specified configuration. Utilizing controls on the user interface, the user may instruct the module to create a dense matrix, a band diagonal matrix or a compressed sparse-row matrix.

- **SolveMatrix** uses direct or iterative algorithms to find the solution to the matrix equation. For this problem, we use a preconditioned conjugate gradient algorithm for iterative solutions. The scientist controls convergence parameters through the graphical user interface.

- **MakeScalarField** combines the solution of the finite element matrix to the volume mesh generated by **GenerateMesh**. This mesh/solution combination provides a representation of the solution in terms of a scalar field of voltage values.

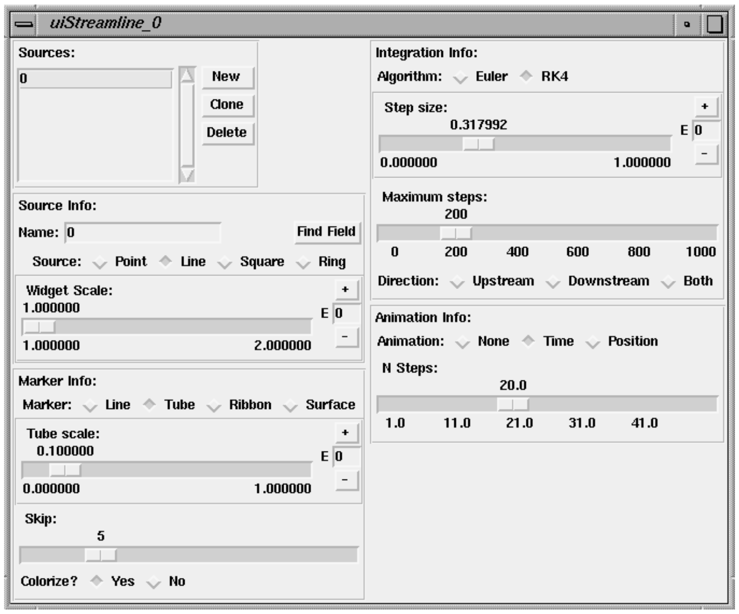
- **IsoSurface** allows interactive extraction of isosurfaces in the voltage field. A small sphere controls the starting point of the isosurface algorithm, and an attached 3D arrow shows the direction of the gradient. The sphere and arrow widget may be moved using the mouse to allow interactive exploration of the voltage field. Dragging on the body of the arrow moves the widget along the line defined by the gradient; dragging on the sphere allows unconstrained movement of the seed point.

- **Gradient** computes a vector field from the scalar voltage field according to equation (1.4). This yields another form of the solution in terms of electric current density.

- **Streamline** produces vector field lines that reveal the flow of electrical currents within the torso and brain. These field lines are analogous to massless particle traces in fluid flow fields. The streamlines are advected using a 4th order Runge-Kutta technique. The user may choose between a single streamline or a row of streamlines. Adaptation parameters and step sizes are controlled via the 2D user interface, while the positions of the particle sources are controlled with 3D widgets [22].

- **Salmon** provides the underlying structure for viewing geometry and 3D user interaction for both viewpoint control and control of the 3D widgets described above. As the **Streamline** module computes streamlines, or as the **Isosurface** module computes isosurfaces, it will send geometry (lines, triangles or other primitives) to **Salmon**. **Salmon** displays these objects geometry in a rendering window.

Each of these modules is simple enough to be managed easily, but when they are joined together, they accomplish a very complex task. Sample visualizations from this type of network are shown in Figures 1.7 and 1.9.



Sources: 0

Source Info:
Name: 0
Source: Point [✓ Line] Square Ring

Widget Scale: 1.000000 (1.000000 – 2.000000) E 0

Marker Info:
Marker: Line [✓ Tube] Ribbon Surface

Tube scale: 0.100000 (0.000000 – 1.000000) E 0

Skip: 5

Colorize? [✓ Yes] No

Integration Info:
Algorithm: Euler [✓ RK4]

Step size: 0.317992 (0.000000 – 1.000000) E 0

Maximum steps: 200 (0 200 400 600 800 1000)

Direction: Upstream Downstream Both

Animation Info:
Animation: None [✓ Time] Position

N Steps: 20.0 (1.0 11.0 21.0 31.0 41.0)

FIGURE 1.10. Pop-up window for the **Streamline** module.

Pressing the "UI" (user interface) button on any of the modules' icons summons a pop-up window which allows the user to change various parameters for that module. For example, the **Streamline** module's pop-up allows the user to specify integration parameters, display modes, 3D widget types, and other information. As an example user interface, Figure 1.10 shows the pop-up windows for the **Streamline** module.

1.9 Summary

We have presented an overview of the SCIRun software architecture. SCIRun provides support for computational steering by providing powerful libraries and tools at a variety of levels, ranging from operating system level to a high-level visual user interface. SCIRun allows computational components to be efficiently assembled using a dataflow programming paradigm.

The real power of SCIRun comes from its modules. The set of modules can be extended by the user in order to provide custom tools and interfaces to existing programs.

1.10 Future Work

SCIRun is continually evolving. We look forward to the challenges of integrating SCIRun components with other computational paradigms (alternatives to dataflow), and also to the converse: connecting SCIRun's visual programming environment to other computational components. In addition, we are continually trying to stretch SCIRun in new directions by applying it to new applications in a range of disciplines.

In addition to porting SCIRun to new shared memory architectures [35], we have ideas of how to use SCIRun in a distributed environment. SCIRun's role as a miniature operating system would be greatly expanded in such a scenario.

Acknowledgments: This work was supported in part by the National Science Foundation, the National Institutes of Health, the Department of Energy through a Computational Science Graduate Fellowship, and the Utah State Centers of Excellence Program. The authors would like to thank K. Coles and P. Sloan for their helpful comments and suggestions. Furthermore, we appreciate access to facilities that are part of the NSF STC for Computer Graphics and Scientific Visualization.

1.11 References

[1] SCIRun. http://www.cs.utah.edu/~sci/.

[2] T. De Fanti et al. Special issue on visualization in scientific computing. *Computer Graphics*, 21(6), November 1987.

[3] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995. http://www.supercomp.org/sc95/proceedings/499_spar/sc95.htm.

[4] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. *Georgia Institute of Technology Technical Report*, 1994.

[5] J. Vetter, G. Eisenhauer, W. Gu, T. Kindler, K. Schwan, and D. Silva. Opportunities and tools for highly interactive distributed and parallel computing. *Proceedings of the Workshop On Debugging and Tuning for Parallel Computing Systems*, October 1994.

[6] C.R. Johnson, R.S. MacLeod, and P.R. Ershler. A computer model for the study of electrical current flow in the human thorax. *Computers in Biology and Medicine*, 22(3):305–323, 1992.

[7] C.R. Johnson, R.S. MacLeod, and M.A. Matheson. Computational medicine: Bioelectric field problems. *IEEE COMPUTER*, pages 59–67, Oct., 1993.

[8] C.R. Johnson and S.G. Parker. A computational steering model for problems in medicine. In *Supercomputing '94*, pages 540–549. IEEE Press, 1994.

[9] C.R. Johnson and S.G. Parker. Applications in computational medicine using SCIRun: A computational steering programming environment. In *Supercomputer '95*, pages 2–19. Springer-Verlag, 1995.

[10] H.P. Langtangen. Diffpack: Software for partial differential equations. In *Proceedings of the 2nd Annual Conference on Object Oriented Numerics (OON-SKI)*, 1994.

[11] A.M. Bruaset and H.P. Langtangen. A comprehensive set of tools for solving partial differential equations: Diffpack. In M. Daehlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*. Birkhauser, Oslo, 1997.

[12] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Object Oriented Numerics Conference*, pages 214–218, 1994.

[13] W. Gropp and B. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 87–93. IEEE Computer Society Press, 1994.

[14] See the paper on PETSc in this volume.

[15] A. Silberschatz, J. Peterson, and P. Galvin. *Operating Systems Concepts*. Addison-Wesley, California, 1991.

[16] D. Musser and A. Saini. *C++ Programming with the Standard Template Library*. Addison-Wesley, California, 1996.

[17] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, 1991.

[18] T.D. DeRose. A coordinate-free approach to geometric programming. 1989.

[19] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[20] J.T. Purciful. Three-dimensional widgets for scientific visualization and animation. Master's thesis, University of Utah, 1995.

[21] D.B. Conner, S.S. Snibbe, K.P. Herndon, D.C. Robbins, R.C. Zeleznik, and A. van Dam. Three-dimensional widgets. In *Computer Graphics (Proceedings of the 1992 Symposium on Interactive 3D Graphics)*, pages 183–188. ACM, 1992.

[22] K.P. Herndon and T. Meyer. 3D widgets for exploratory scientific visualization. In *Proceedings of UIST '94, ACM Siggraph*, pages 69–70. ACM, November 1994.

[23] F. Yu and C.R. Johnson. An automatic adaptive refinement and derefinement method for elliptic problems. *Proceedings of IMACS*, 3:1555–1557, 1994.

[24] F. Yu, Y. Livnat, and C.R. Johnson. An automatic adaptive refinement and derefinement method for 3D elliptic problems. *Applied Numerical Mathematics*, 1997 (to appear).

[25] R.S. MacLeod, C.R. Johnson, and M.A. Matheson. Visualization tools for computational electrocardiography. In *Visualization in Biomedical Computing*, pages 433–444, 1992.

[26] R.S. MacLeod, C.R. Johnson, and M.A. Matheson. Visualization of cardiac bioelectricity — a case study. In *IEEE Visualization '92*, pages 411–418, 1992.

[27] R. Plonsey and R. C. Barr. *Bioelectricity*. Plenum Press, New York, 1988.

[28] B. Lerman and O. Deale. Relation between transcardiac and transthoracic current during defibrillation in humans. *Circulation Research*, 67:1420–1426, 1990.

[29] M. Mirowski. The automatic implantable cardioverter-defibrillation: An overview. *Journal of the American College of Cardiology*, 6:461–466, 1985.

[30] D. Blilie, J. Fahy, C. Chan, M. Ahmed, and Y. Kim. Efficient solution of three-dimensional finite element models for defibrillation and pacing applications. In *Proceedings of the 13th Annual Conference of the IEEE Engineering in Medicine and Biology Society*, volume 13, pages 772–773, 1991.

[31] S.A. Hutchinson, S. Gao, L. Ai, K.T. Ng, O.C. Deale, P.T. Cahill, and B.B. Lerman. Three-dimensional modeling of electrical defibrillation on a massively parallel computer. In *Computers in Cardiology*, pages 343–346. IEEE Computer Society Press, 1992.

[32] S.A. Hutchinson, K.T. Ng, J.N. Shadid, and A. Nadeem. Electrical defibrillation optimization – an automated, iterative parallel finite-element approach. *IEEE Trans. Biomed. Eng.*, 1995.

[33] X. Min, L. Wang, M. Hill, J. Lee, and R. Mehra. Detailed human thorax fem model for the cardiac defibrillation study. In *Proceedings of the 15th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 838–839, Piscataway, New Jersey, 1993. IEEE Press.

[34] J.A. Schmidt and C.R. Johnson. Defibsim: An interactive defibrillation device design tool. In *IEEE Engineering in Medicine and Biology Society 17th Annual International Conference*. IEEE Press, 1995.

[35] ORIGIN2000 and ONYX2 Technical Report. Silicon Graphics, Revision 6.0, October 4, 1996.

Appendix A. Software Appendix

Appendix A..1 Handles

A **handle** contains a single data member `rep`, which contains a pointer to the actual representation. It also defines ten methods:

1. `Handle()` - initializes `rep` to `NULL`.
2. `Handle(Type* p)` - sets `rep` to `p`, and increments `p->ref_cnt`.
3. `Handle(Handle<Type>& copy)` - copies `rep` from another handle, and increments the reference count.
4. `~Handle()` - decrements `rep->ref_cnt`. If it is now zero, delete the object pointed to by `rep`.
5. `Handle<`**Type**`>& operator = (`**Type**`* p)` - decrements `rep->ref_cnt`, deleting when it reaches zero; sets `rep` to `p` and increments `p->ref_cnt`.
6. `Handle<`**Type**`>& operator = (Handle<Type>& copy)` - decrements `rep->ref_cnt`, deleting if it reaches zero; copies `rep` from `copy` and increments `copy->ref_cnt`.
7. **Type*** `operator -> () const` - this simply returns `rep`, but it allows the handle to be dereferenced using `->` like a normal C++ pointer.
8. **Type*** `get_rep() const` - returns the actual pointer `rep`. This is useful for some code, but is also dangerous since the handle must continue to exist as long as the pointer is being used - otherwise the object could be deleted while it is still being accessed.

9. `bool isNull() const` - returns true if `rep==0`.

10. `void detach()` - if `rep->ref_cnt` is greater than one, then `void detach` clones the object, thus obtaining an exclusive copy of the object. If `rep->ref_cnt` is equal to one, then we already own an exclusive copy of the object. This allows subsequent code to make changes to the referred object without sharing those changes with other handle owners.