Loci : A Tutorial

May 31, 2019

Contents

Chapter 1

An Introduction to Loci

Loci is a both a C++ library and a programming framework specifically designed for developing computational simulations of physical fields, such as computational fluid dynamics. One advantage the framework provides is automatic parallelization. Once an application is described within the Loci framework, the application can be executed in parallel without change, even though the description within the framework included no explicit parallel directives. A particular advantage of the programming framework is that it provides a formal framework for the development of simulation knowledge-bases using *logic-relational* abstractions. While the approach will probably be alien to most who begin to use it, the programming model is extremely powerful and worth the patience required to adjust to a new way of thinking about programming.

Several major components comprise the Loci framework and these include facilities for managing sets of entities, containers that can associate values with entities, a database for managing user provided facts, a database for managing user provided rules, and finally, a query system that can generate programs that satisfy specific user requests. Most of these facilities are provided as a C++ library. In addition to this library, Loci provides a preprocessor program that translates high level descriptions into C++ code. The main purpose of this preprocessor is to automate the more mundane aspects of the C++ interface and is not actually needed to develop Loci programs. This tutorial will mainly focus on using the loci preprocessor. Loci programs are provided with files using the `.loci` suffix to indicate these are files that contain Loci preprocessor directives.

Chapter 2

Basic Concepts

2.1 Notation used in this document

In this document we use the `typewriter` font to distinguish actual *Loci* programming keywords, classes, and data-structures.

2.2 Compiling *Loci* Programs

The most direct way to compile *Loci* programs is to use the Makefile template provided in this tutorial. It is usually as simple as including the `Loci.conf` file that comes as part of your *Loci* installation. See appendix A for an example makefile or refer to example Makefiles in the tutorial directory.

2.3 *Loci* Initialization

Before any of the main *Loci* functionality can be used (that is the components that follow this section), *Loci* must be initialized. *Loci* has an initialize function that must be called before executing *Loci* functionality and a finalize method that must be called just before exiting the program. Note, that the include file `#include <Loci.h>` includes all commonly used components of the *Loci* framework, including definitions of the initialization routines. For example, see below:

```
#include <Loci.h>

int main(int argc, char *argv[]) {
   // Initialize Loci
   Loci::Init(&argc, &argv) ;

   // ...
   // Loci Program
   // ...

   // Before exiting, call finalize to let Loci clean up.
   Loci::Finalize() ;
   return 0 ;
}
```

2.4 Entities, Sets, and Sequences

Probably the most fundamental concept of *Loci* is that of entities. In *Loci*, computations are represented by associating values with entities. Although entities can be considered in rather abstract terms, in *Loci* we often will often interchange the meaning of entity with the integer identifier that is used to label a given entity. Thus we may talk of entity 1 when we are really referring to the entity labeled 1. Note, that while the entity itself is immutable, its label may change in the course of executing a *Loci* program, and in particular when *Loci* schedules parallel programs. Generally the user is unaware of this fact, but it can become important in a few cases that will be mentioned in later examples.

As important as the concept of entity is the concept of entity collections. Generally, it is useful to consider groups of entities that have similar attributes. In *Loci* we have two provided types for representing sets of entities: 1) the `interval` and 2) the `entitySet`. For example, if we wish to represent the entities labeled from 1 to 100 we would use the *Loci* type `interval(1,100)`. On the other hand, the `entitySet` can be used to represent arbitrary collections of entities. Once we have described a collection of entities using the `entitySet` class we can also create new sets of entities using unions, intersections, and other useful set operations.

The `entitySet` class provides true set semantics. That is, ordering of insertion is not preserved and there is no duplication. Either an entity is in the set or it is not. If we need to preserve the order of entities for looping or other control then we use the `sequence` class. The `sequence` class provides operations for concatenation and reversal and can be thought of as a list of entity labels. It should be noted that users generally don't create sequences in *Loci*, but rather the scheduler generates sequence of entities for computations. However, if there is ever a need to keep track of a particular ordering of entities, then sequences are the data-structure that accomplishes this task.

The following program segment (included with the tutorial programs) provides examples of how to create and use sets and sequence of entities in *Loci*.

```
// Example program illustrating how to manipulate Entity, entitySet,
// and sequence data structures in Loci

// Every Loci program includes the Loci header file.
#include <Loci.h>

// includes for standard C++ functions
#include <iostream>
using std::cout ;
using std::endl ;
#include <vector>
using std::vector ;

int main(int argc,char *argv[])
{

  Loci::Init(&argc,&argv) ;

  ////////////////////////////////////////////////////////////////////////////
  // The Entity class
  ////////////////////////////////////////////////////////////////////////////
  // An Entity is labeled by an integer.
  // The collections of entities discussed below are represented by
  // collections of integers.
  Entity e10 = Entity(10) ;  // First, the entity labeled by integer 10
  cout << "e10 = " << e10 << endl ;

  ////////////////////////////////////////////////////////////////////////////
  // The interval (A collection of consecutively labeled entities)
  ////////////////////////////////////////////////////////////////////////////
  // For example, onedigit is an interval of entity labels consisting of
  // only one decimal digit.
  interval onedigit = interval(0,9) ;
  cout << "onedigit = " << "[0,9]" << endl ;

  ////////////////////////////////////////////////////////////////////////////
  // Class entitySet provides facilities for general sets of entities.
  ////////////////////////////////////////////////////////////////////////////
  // Initialization
  // An entitySet can be initialized to an interval.
  entitySet A = onedigit ;
  entitySet B = interval(14,100) ;
  entitySet C = interval(5,15) ;
  entitySet F = interval(10,20) ;
  cout << "A = " << A << endl ;
  cout << "B = " << B << endl ;
  cout << "C = " << C << endl ;
  cout << "F = " << F << endl ;
```

```
////////////////////////////////////////////////////////////////////////////
// For efficiency, an entitySet is stored as an ordered set;
// more precisely, as an ordered set of ordered intervals.
// The class for an ordered set of ordered intervals is
// intervalSet.  Lower-level methods and operators take intervalSet
// and interval arguments.

////////////////////////////////////////////////////////////////////////////
// Adjunction
////////////////////////////////////////////////////////////////////////////
// We can also add an individual entity to any existing entitySet using the
// += operator, for example we can include the entity e10 in set B:
B += e10 ;

//  A = ([0,9])
//  B = ([10,10][14,100])
//  C = ([5,15])
cout << "A = " << A << endl ;
cout << "B = " << B << endl ;
cout << "C = " << C << endl ;
cout << "F = " << F << endl ;

/////////////////////////////////////////////////////////////////////////////
// num_intervals
/////////////////////////////////////////////////////////////////////////////
// The num_intervals() method returns the number of intervals in the
// internal representation of an entitySet as an intervalSet.
cout << "B = " << B << endl ;
// B.num_intervals() = 2
cout << "B.num_intervals() = " << B.num_intervals() << endl ;

////////////////////////////////////////////////////////////////////////////
// Neither order nor duplication matters to an entitySet.
// For example
entitySet E = B + C ;
//  E = B union C =  ([5,100])
// [gives the set ([5,100]) without duplicating 14 and 15]
cout << "E = B union C =  " << E << endl ;

////////////////////////////////////////////////////////////////////////////
// Distinguished constants
////////////////////////////////////////////////////////////////////////////
// EMPTY is a distinguished constant for the empty set.
cout << "EMPTY = " << EMPTY << endl ;


// UNIVERSE_MAX and UNIVERSE_MIN are distinguished constants for the
// largest and smallest integers that may be used to label entities.
cout << "UNIVERSE_MAX = " << Loci::UNIVERSE_MAX << endl ;
cout << "UNIVERSE_MIN = " << Loci::UNIVERSE_MIN << endl ;
```

```
// Another useful derived constant is ~EMPTY or not EMPTY, the '~' operator
// provides the set complement with respect to the universal set, therefore
// ~EMPTY is the universal set (Set of all possible entities).

cout << "Universal Set = " << ~EMPTY << endl ;

///////////////////////////////////////////////////////////////////////////
// Set membership
///////////////////////////////////////////////////////////////////////////
// Method inSet tests whether an entity is an element of an
// entitySet.
// The argument to inSet is the integer label of the entity.
cout << "A.inSet(9) = " ;
if ( A.inSet(9) )
    cout << "TRUE." ;
cout << endl ;
cout << "A.inSet(10) = " ;
if ( ! A.inSet(10) )
    cout << "FALSE." ;
cout << endl ;

///////////////////////////////////////////////////////////////////////////
// Set operations
///////////////////////////////////////////////////////////////////////////
// entitySet supports union, intersection, relative complement,
// and complement relative
// to the set of permitted identifiers.
//
// For example, entitySet D becomes the union of A and B
// That is D = ([0-9],[14-100])
entitySet D = A + B ;
//  D = ([0,10][14,100])
cout << "D = " << D << endl ;
// A & C  gives the intersection of A and C (the interval [5-9])
cout << "A intersect C = " << (A & C) << endl ;
// A - C gives the set difference (A take away C) (the interval [0-4])
cout << "relative complement of A in C = " << (A - C) << endl ;
cout << "A intersect (B union F) = " << (A & (B + F)) << endl ;
cout << "A union (B intersect F) = " << (A + (B & F)) << endl ;

///////////////////////////////////////////////////////////////////////////
// Explicit membership list
// We can also create an entitySet from an arbitrary list
// of entity identifiers.
// For example,
int values[] = {10,15,12,1,14,16,17} ;
entitySet vset = create_entitySet(values,values+7) ;
// vset = ([1,1][10,10][12,12][14,17])
cout << "vset = " << vset << endl ;
```

```
///////////////////////////////////////////////////////////////////////////
// Creating an entitySet from a vector of integers.
// create_entitySet works with std::vector and begin() and end().
// (See a standard C++ book for more details on using vector<> and other
// STL containers.)
vector<int> vec ;
for(int i=10;i>0;--i)
  vec.push_back(i) ;
entitySet vecset = create_entitySet(vec.begin(),vec.end()) ;
// vecset = ([1,10])
cout << "vecset = " <<vecset << endl ;

///////////////////////////////////////////////////////////////////////////
// Iteration over an entitySet
// We can also iterate (loop) over an entitySet in a fashion similar to
// how we iterate over standard C++ containers.  For example, to iterate
// over all of the entities in vset we would write a loop as follows.
// First we create an iterator ei for entity sets.
entitySet::const_iterator ei ;
// Then we use the iterator to loop over a given entitySet using the
// begin() and end() methods.  For example to loop over the entities
// contained in vset we write:
cout << "looping over vset:" ;
for(ei = vset.begin(); ei != vset.end(); ++ei)
  cout << " " << *ei ;
cout << endl ;
// above outputs:
// looping over vset: 1 10 12 14 15 16 17

///////////////////////////////////////////////////////////////////////////
// Min, Max, size, set membership
// Other useful methods include Min() and Max() which can be used to find
// the largest and smallest integer labels of entities contained in a given
// entitySet.  For example, vset.Max() == 17 and vset.Min() == 1

// vset.Min() == 1, vset.Max() == 17
cout << "vset.Min() = " << vset.Min() << endl ;
cout << "vset.Max() = " << vset.Max() << endl ;

// Similarly, we can check the number of entities contained within an
// entitySet by using the size() method.  For example

// vset.size() == 7
cout << "vset contains " << vset.size() << " entities" << endl ;

// We can also check to see if a particular entity label is in a
// entitySet using the inSet() method.  For example
if(A.inSet(5))
  cout << "entity labeled 5 is in entitySet A" << endl ;
```

```
//////////////////////////////////////////////////////////////////////////////
// Equal, less_than, greater_than, Union (interval),
// Union (entitySet), Intersection (interval),
// Intersection (entitySet), [absolute] Complement,
// Print, Input
//
// The entitySet.Equal method tests whether the sets are equal.
if ( A.Equal((A & C) + (A - C)) )
  cout << "A = ((A & C) + (A - C))." << endl ;
else
  cout << "A != ((A & C) + (A - C))." << endl ;

//  The entitySet.less_than and greater_than methods
//  provide a linear ordering of entitySets.
//  The linear ordering is lexical.
if ( A.less_than (A & C) )
  cout << "A&C <= A." << endl ;


if ( A.less_than (A) )
  cout << "A <= A." << endl ;


entitySet G = A - C ;
if ( G.greater_than (A) )
  cout << "A >= A - C." << endl ;

// Methods for union and intersection, for intervals and for
// intervalSets.  (Remember that an entitySet is currently
// implemented as an intervalSet.)
// Note that these methods modify the object to which they belong.
cout << "G = " << G << "." << endl ;
G.Union (interval(13,32)) ;
cout << "G Union [13,32] = " << G << "." << endl ;
cout << "G = " << G << "." << endl ;
G.Union (D) ;
cout << "G Union D = " << G << "." << endl ;
cout << "G = " << G << "." << endl ;
G.Intersection (interval(13,32)) ;
cout << "G Intersection [13,32] = " << G << "." << endl ;
cout << "G = " << G << "." << endl ;
G.Intersection (D) ;
cout << "G Intersection D = " << G << "." << endl ;
cout << "G = " << G << "." << endl ;

// Method for absolute complement of an entitySet.
// Note that the complement replaces the given set.
// G = ([14,32]).
G.Complement() ;
```

```
  // G = ([#,13][33,#]).
  // The first occurrence of "#" stands for Loci::UNIVERSE_MIN;
  // the second occurrence of "#" stands for Loci::UNIVERSE_MAX.
  cout << "Complement of G = " << G << endl ;
  cout << "G = " << G << endl ;

  ///////////////////////////////////////////////////////////////////////////
  // Sequences
  //
  // Sequences provide a way of storing ordered lists of entities.
  // Usually, users don't need to worry about creating sequences
  // directly in Loci, but instead Loci creates sequences to describe
  // the order in which calculations will be carried out.  However,
  // for completeness we will give some examples here of how to create
  // sequences, and of how to iterate over sequences.

  // We can create an arbitrary sequence from a list of integers
  // in a fashion similar to the create_entitySet function.  For
  // example:
  int listvals[] = {10,15,12,1,14,16,17} ;

  sequence vseq = create_sequence(listvals,listvals+7) ;

  // vseq = ([10,10][15,15][12,12][1,1][14,14][16,17])
  cout << "vseq = " << vseq << endl ;

  // Note that we can also create a sequence from an entitySet, however
  // in this case the sequence will contain the contents of the entitySet
  // in increasing order.  For example
  sequence Aseq = A ;

  // Aseq = ([0-9])
  cout << "Aseq = " << Aseq << endl ;

  // Also we can append to sequences, for example:
  sequence Cseq = Aseq + vseq;

  // Cseq = ([0,10][15,15][12,12][1,1][14,14][16,17])
  cout << "Cseq = " << Cseq << endl ;

  // Similarly we can reverse the order of a sequence using the Reverse()
  // method.  For example
  Cseq.Reverse() ;

  // Cseq = ([17,16][14,14][1,1][12,12][15,15][10,0])
  cout << "reversed Cseq = " << Cseq << endl ;

  Loci::Finalize() ;
  return 0 ;
}
```

2.5 *Loci* Containers

In *Loci*, containers are entity based. That is, a container provides an association between entities and values. There are two basic types of containers: stores and parameters. Stores are used to associate values with entities, while parameters are used to associate a value with a sent of entities. For example, in a simulation each of the nodes of a mesh will have a position vector, and this will be represented in *Loci* using a store container. However, the time-step of a simulation is a single value that is shared by all of the simulation entities and this will be represented using a parameter in *Loci*. The `store` is a templated container that can be used to contain arbitrary types. For example:

```
// We create a store of floats
store<float> x ;
// We create a store of float vectors, this is OK also
store<std::vector<float> > particles ;
```

Once we create the store container we can use the `allocate()` method to allocate values over some set of entities. For example, to allocate the above containers over 100 entities we would use code such as:

```
 // allocate stores x and particles
 entitySet alloc_set = interval(1,100) ;
 x.allocate(alloc_set) ;
 particles.allocate(alloc_set) ;
```

After allocating the container, we can access the values of the container using the array operator. In this sense a store looks like an array with array bounds that are general sets. For example, if we want to initialize the contents of our containers we might use code that iterates over the allocated set such as the following:

```
// initialize the container to the value zero
entitySet::const_iterator ei ;
for( ei = alloc_set.begin(); ei != alloc_set.end(); ++ei) {
  x[*ei] = 0 ;
  particles[*ei].push_back(0) ; // Calling vector method push_back()
}
```

Note, we can also query the domain, or defining set of entities, for any container by using the `domain()` method. For example

```
// write out all of store x to cout
entitySet xdom = x.domain() ;
for( ei = xdom.begin(); ei != xdom.end(); ++ei)
  cout << "x["<<*ei<<"]=" << x[*ei] << endl ;
```

For parameters *Loci* provides the `param` templated class. This class can also be used to hold arbitrary types. It associates a given value with a collection of entities. By default this collection of entities is the universal set, but there are methods for limiting this to any subset of entities. We create a param similar to the store type with code such as:

```
// Create the timestep
param<float> timestep ;
```

We then can assign a value to the parameter using the dereferencing `*` operator. For example:

```
// Set the timestep to 1ms
*timestep = 1e-3 ;
```

Similarly we can assign a value for a subset of entities (such as a boundary entities) using the parameters facility as well. By default the param associates a value with all possible entities, but this association can be changed using the `set_entitySet()` member function. For example:

```
param<real> Twall ; // Create wall temperature
Twall = 300 ;
// Constraint Twall to only apply to boundary entities (as given)
entitySet wallBoundary = interval(1000,1500) ;
Twall.set_entitySet(wallBoundary) ;
```

2.6 *Loci* Relations

In addition to containers, *Loci* provides ways of describing relationships between entities. The simplest of these relationships is the constraint. The constraint simply identifies a grouping of entities and is used to assign attributes to entities. For example, a boundary condition may be specified by placing those entities in the boundary in a boundary constraint. For example, suppose entities labeled $1, 2$, and 3 are at an inflow boundary, we might construct such a structure by creating a constraint and assigning these entities to the constraint. For example:

```
// set inflow constraint
constraint inflow ;
*inflow = entitySet(interval(1,3)) ;
```

Alternatively, constraints might be used to enable and disable some feature in the solver. In this setting a constraint may either contain the empty set, or may contain all possible entities. For example, we might use a constraint to select between having viscous terms or not using the following type of setup:

```
constraint viscous ;
*viscous = EMPTY ; // default to not empty
if(mu_set) // If viscosity set, then enable viscous terms
  *viscous = ~EMPTY ; // Constraint set to contain all entities
```

Note that in this setup we use the tilde to complement the `EMPTY` set to achieve the result of identifying every entity. This set will contain all entities not in the empty set (e.g. everything).

Constraints can identify a group of entities are related, however it cannot provide a relationship from one entity to another (e.g. how do faces relate to cells, what nodes make up a face, etc). We use a `Map` container to describe these types of relationships. The `Map` is used to relate any given entity with another. The map is analogous to a `store` that contains entities, and as such can be allocated, and assigned values similar to a generalized array much like the `store` container. For example, if we wanted to create a `Map` for all entities to the left of an entity in a number line we might write code such as:

```
  entitySet nodes = interval(0,10) ; // A number line from 0 to 10
  entitySet left = (nodes >> 1) & nodes ; // Shift set to get nodes that
                                          // have a left side
  // Create Maping from current node to the node to the left
  Map leftNode ;
  leftNode.allocate(left) ; // Allocate over all entities that have a left side
  // Assign left node by looping over left nodes and nodes at the same time
  entitySet::const_iterator ni = nodes.begin() ;
  for(entitySet::const_iterator li=left.begin();li!=left.end();++li,++ni)
    leftNode[*li] = *ni ; // node *ni is to the left of node *li
```

Loci also includes other types of map containers such as `mapVec` and `multiMap` which provide mechanisms for having multiple entity associations. They will be discussed later in the tutorial as we get to more advanced topics.

2.7 Databases within *Loci*

One of the facilities that *Loci* provides is the management of fact and rule databases. The fact database provides a repository for the containers described earlier. Once a container is put in the fact database, it can be retrieved at a later time by its assigned name. For example, if we wanted to store the `leftNode` map computed in the previous section into a *Loci* fact database we would use the `create_fact()` method. For example:

```
  fact_db facts ; // Create the fact database

  // Insert leftNode into the fact database
  facts.create_fact("leftNode",leftNode) ;
```

The fact can later be retrieved from the fact database using the `get_fact()` method. For example, we can pass the fact database into a function and then use it to get a copy of the `leftNode` map with code such as:

```
void worker(fact_db &facts) {
  Map leftNode ;  // Create Map container
  leftNode = facts.get_fact("leftNode") ;
  // ... code using leftNode follows
}
```

In addition to the fact database, *Loci* provides a facility for storing rules that describe computations that can generate new facts. We will describe rules more in the following section. Usually the user doesn't directly manipulate the rule database, but rather tells *Loci* when to install rules into the rule database that the has been identified by the *Loci* framework. In general, the user creates the rule database and fills it with rules that were registered with the system automatically before `main()` executes. As in the following example:

```
  rule_db rdb ; // create the rule database called rdb
  rdb.add_rules(global_rule_list) ; // Add all registered rules to the database
```

Both the fact database and the rule database are fundamental to programming using the *Loci* framework. The fact database describes what you know about a problem, the rule database describes what you can derive. Both of these components are used to make *Loci* queries using the `Loci::makeQuery` command. For example to query for the computed temperature one would implement code such as:

```
 // Query Loci for fact derived fact 'temperature'
  if(!Loci::makeQuery(rdb,facts,"temperature")) {
    cerr << "query failed!" << endl ;
  }
```

In most *Loci* programs, the user queries for the generic variable "solution" which just indicates that final solution to the problem. For most time dependent problems, the most interesting information is the intermediate values obtained in the time evolution, not the final value.

2.8 *Loci* Helper Classes

Loci also provides a few helper classes that are often useful in numerical computations. One is the `Array` template class which provides a mechanism for creating Arrays as first class objects that are appropriate for using as classes used in templated containers. (*Never use C arrays in templated containers. Their semantics are different from other C++ objects and may break templated code in unexpected ways.*) In addition to the `Array` template class, classes for three and two dimensional vectors are also provided. See the following example code to see how to use these helper classes.

```
#include <Loci.h>

#include <iostream>
using std::cout ;
using std::endl ;

int main(int argc, char *argv[])
{

  Loci::Init(&argc,&argv) ;

  // Loci provides a helper class for creating Arrays as first class objects
  // (Loci containers must contain first class objects, so if you want a
  // store to contain an array, use this class.
  // The array is a template class parameterized on the type and size of the
  // array
  Array<double,5> A ;  // Analogous to double A[5]

  // Otherwise Arrays act much like C arrays.  The one exception is that
  // unlike C arrays, if you pass these arrays to a function they will be
  // passed by value not by reference!

  for(int i=0;i<5;++i)
    A[i] = double(i) ;

  // Also Assignment works on the entire array
  Array<double,5> B = A ;

  // Other tricks are also added (mostly for compatibility with Loci reduction
  // rules to be covered later.

  B += A ;  // Does element by element addition of array A to array B
  // Operators +,-,*, and / are similarly overloaded

  // Also Array supports STL iterator style access, e.g.
  double total = 0;
  for(Array<double,5>::const_iterator ii=B.begin();ii!=B.end();++ii)
    total += *ii ;

  cout << "total = " << total << endl ;

  // 3d vectors are supported as types, templated by the type of x,y, and z
  vector3d<double> v1(0.0,1.0,2.0),v2(1.0,2.0,0.0),v3(2.0,1.0,0.0) ;

  // 3d vectors overload all of the expected algebraic operations in the usual
  // way including scalar vector multiply.  For example, the vector found by
  // "averaging" the vectors v1, v2, and v3 is computed as follows.
  vector3d<double> avg = (v1+v2+v3)/3.0 ;

  // Also 3dvectors support cross and dot products
  vector3d<double> cross_product = cross(v1,v2) ;
  // Also dot and cross products can be nested...
  double           dotcross = dot(cross(v1,v2),v3) ;
  // L2 norm is also provided
  double           areav1xv2 = norm(cross_product) ;

  // Of course interaction with scalar values acts as expected.
  // Normalize cross product
  cross_product /= areav1xv2 ;

  cout << "average vector = " << avg << endl ;
  cout << "cross_product = " << cross_product << endl ;
  cout << "areav1xv2 = " << areav1xv2 << endl ;
  cout << "dotcross = " << dotcross << endl ;

  // 2d vectors are supported as types, templated by the type of x and y
  vector2d<double> v4(1.0,1.0),v5(-1.0,1.0),v6(2.0,3.0) ;

  // 2d vectors also support cross and dot products similar to 3d vectors
  // with the exception that cross products in 2d space are scalars (the z
  // component of a 3d cross product).
  double twodcross = cross(v4,v5) ;
  double twoddot   = dot(v5,v6) ;

  // Of course all operators are overloaded as in the 3d vector case.
  vector2d<double> twodavg = (v4+v5+v6)/3.0 ;

  cout << "twodcross = " << twodcross << endl ;
  cout << "twoddot =   " << twoddot << endl ;
  cout << "twodavg =   " << twodavg << endl ;

  Loci::Finalize() ;
  return 0 ;
}
```

2.9 The options_list class

In many circumstances the user needs to input complex optional information to a solver. These parameters are usually provided as a part of the initial facts that are provided by the user and may express several options to be used by a particular functional part of the system. To help standardize the input of this information *Loci* provides a powerful system for inputting complex and hierarchical information called the `options_list`. This form of input allows named assignment of data, specification of units that data is presented, input of lists and other complex data forms. A common use of the `options_list` is the assignment of boundary conditions in Loci solvers. This section will discuss how to use the `options_list` type to allow for more powerful inputs to your *Loci* programs.

When a parameter type is an `options_list` type, then the input in the ASCII version of the initial facts (the `vars` file) will have a variable that is delimited by less-than and greater-than symbols. For example the boundary conditions are input using an `options_list` type:

```
boundary_conditions:<
 BC_1=specified(Twall=500K),
 BC_2=specified(Twall=300K),
 BC_3=adiabatic, BC_4=adiabatic
>
```

The basic function of the `options_list` is to give a way of specifying multiple parameters. In this case the parameters are the boundary surface names which are assigned to the boundary condition. The boundary conditions are represented by complex data types. For the specified case the boundary conditions are represented by a named function type where the arguments contain a named list of variables (which can be thought of as another nested `options_list`). The general structure of an `options_list` variable will be as follows: First a '¡' character which opens the options block, then a comma separated list of *name* = *argument* where `argument` can be the following:

1. a floating point number
2. a floating point number with units annotation
3. a string value in “quotes”
4. a list of floating point numbers enclosed in ‘[’ and ‘]’
5. a list of name assignments enclosed in ‘[’ and ‘]’
6. a name
7. a name with arguments enclosed in (parenthesis)

The options list can be used in two modes: one where the names of the variable names are limited to a preset list or the default mode where any name can be used on the input. The first mode is enabled by passing a string to the constructor that contains a colon separated list of allowable variable names. The simplest way to use options list for inputs is to simply put the class in a `param` and allow it to be read in with a optional rule. For example:

```
$type param<options_list> boundary_conditions ;

$rule optional(boundary_conditions) {}
```

Once the option list is read in the following member functions can be used to interrogate the values stored in the option list:

1. `optionExists`: This member function will return true if the variable name given in the string argument is defined in the option list.
2. `getOptionNameList`: This member function returns a list of the names stored in the option list using the type `options_list::option_namelist`.
3. `getOptionValueType`: This member function returns the value type associated with the option passed in as the argument. The value type may be `REAL`, `NAME`, `FUNCTION`, `LIST`, `STRING`, `BOOLEAN`, or `UNIT_VALUE`.
4. `getOption`: This member function retrieves the value associated with the named option. The second argument is the returned value and may be the types `bool`, `double`, `string`, or `options_list::arg_list`.
5. `getOptionUnits`: This member function is passed the name, a requested unit, a double variable where the returned value will be stored, or alternatively the last argument may be a `vector3d` for retrieving 3d vector values.

For simple data input the `options_list` type is straightforward to use. Some examples:

```
void extract_data(const options_list &ol,
                  double &alpha,
                  double &temperature,
                  string &name,
                  bool &adiabatic) {
   // for nondimensional data just use getOption
   ol.getOption("alpha",alpha) ;
   // If we want a default value set it and check if the
   // option is available
   temperature = 300 ; // default value
   if(ol.optionExists("temperature")) {
     // for dimensional data specify what units you want
     // to retrieve the data in.  This will be the default
     // units for this input
     ol.getOptionUnits("temperature","kelvin",temperature) ;
   }
   // for other datatypes the input works the same
   ol.getOption("filename",name) ;
   ol.getOption("adiabatic",adiabatic) ;
}
```

For more complex types, for example lists of values which can occur in options lists for cases such as:

```
valueList : < list = [ 1.0, 2.5, 1.0, 5.0 ],
              funclist = funcname(1.0, 2.5, 1.0, 5.0) >
```

In these cases the list of values can be extracted by first extracting the `arg_list` and then looping over the list and extracting the data item associated with each term. For example, a generic function that can extract a list of doubles assigned to a variable would be implemented as:

```
  void getList(const options_list &ol, std::string vname,
               vector<double> &valuelist) {
    if(ol.getOptionValueType(vname) == Loci::LIST) {
      // It is a list so get list
      Loci::options_list::arg_list list ;
      ol.getOption(vname,list) ;
      int size = list.size() ;
      // loop over list and insert into valuelist
      for(int i=0;i<size;++i) {
        if(list[i].type_of() != Loci::REAL) {
          cerr << "improper list for " << vname << endl ;
          Loci::Abort() ;
        }
        double val = 0 ;
        // get each list item with get_value call
        list[i].get_value(val) ;
        valuelist.push_back(val) ;
      }
    }
  }
```

For more powerful use of the option lists it is possible to consider recursive specifications where the contents of a list or an argument list of a function can be converted to an option list and then queried further. For example, in the boundary condition example above the specified boundary condition had arguments in the same form as the options list itself. In this case it is possible to parse this structure by using the `arg_list` to create another options list for the purpose of parsing the structure. For example, one way to parse the options of the specified boundary condition is as follows:

```
   void parseBoundary(const options_list &ol, string bcname) {
     if(ol.optionExists(bcname)) {
       if(ol.getOptionValueType(bcname) == FUNCTION) {
         string bctype ;
         options_list.arg_list fvalues ;
         // get the function and the arguments
         ol.getOption(bcname,bctype,fvalues) ;
         // create options list for function arguments
         options_list fol ;
         fol.Input(fvalues) ;
         if(bctype == "specified") { // now it is a specified bc
           // now check argument options (e.g. specified(Twall=300K))
```

```
if(fol.optionExists("Twall")) {
   double Twall ;
   fol.getOptionUnits("Twall","kelvin",Twall) ;
   // ...
```

Chapter 3

A Simple Example

Before we begin describing how to create rules in the *Loci* framework, first lets consider a simple example problem so we can show how the problem is decomposed into facts and rules. In this case, let us consider the one-dimensional time-dependent diffusion equation as discretized by a finite-volume method. A formal description of this problem is as thus: given the interval $x \in [0,1]$ for a given diffusion constant ν, the one-dimensional time-dependent diffusion equation is described by the equations

$$u_t = \nu u_{xx},\ x \in (0,1), t > 0, \tag{3.1}$$
$$u(x,0) = f(x),\ x \in [0,1], \tag{3.2}$$
$$u_x(0,t) = g(t),\ \text{where } g(0) = f_x(0),\ \text{and} \tag{3.3}$$
$$u(1,t) = h(t),\ \text{where } h(0) = f(1). \tag{3.4}$$

Equations (3.1) through (3.4) formally define the problem to be solved; however, the methodology of solution is left open. A complete specification for finding an analytical solution might be stated as follows: using the Laplace transforms and associated algebraic identities, find the value of the function $u(x,t)$ such that the definitions given in equations (3.1) through (3.4) are satisfied. Notice that this specification contains three distinct parts: 1) a definition of the problem, 2) a collection of transformations, and 3) a goal that must be satisfied. For this case, an analytic solution to the problem may be found for a few specific functions $f(x)$, $g(t)$, and $h(t)$. In general, however, analytical solutions to PDE problems of interest to engineering are either impractical or impossible, due to the complexity of the geometries involved and the non-linearity of the equations themselves. For this reason, approximate numerical methods are often used to solve PDE based problems. However, the basic approach of problem and solution specification through definitions, transformations, and goals applies equally well to numerical solution methods. The question is, how does one formally specify the problem and solution methodology for these numerical methods in this definition-transformation-goal style? Let us explore this question as we derive a finite-volume discretization of the time-dependent diffusion equation.

3.1 A Finite Volume Solution

The first step in any discretization method is to numerically approximate the function $u(x,t)$ as a discretization of the spatial domain (in this case the interval $[0,1]$). For this example, the finite-volume discretization method is chosen. Using this discretization approach, the interval $[0,1]$ is divided into N sub-intervals, as illustrated in figure 3.1. To facilitate describing the discretization process, the N sub-intervals, or cells, are labeled by $c = N+1, \cdots, 2N$, while the interfaces at the boundaries of sub-intervals are labeled $i = 0, \cdots, N$. Note that the typical labeling used for theoretical purposes would include half step labels for the interfaces, while a typical unstructured application code might label both cells and interfaces starting from zero and use context to distinguish between the two cases. However, for the purposes of automating reasoning about these entities of computations it is assumed that these labels are integers and that independent computational sites (in this case, cells and interfaces) are labeled distinctly. The proposed labeling satisfies both of these constraints.

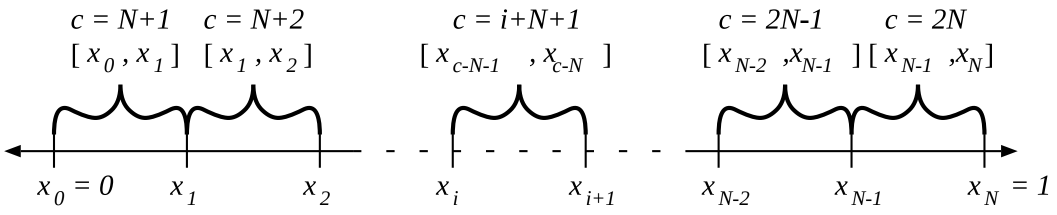

Figure 3.1: A Discretization of the Interval $[0,1]$

As illustrated in figure 3.1, the discretization yields $N+1$ interfaces which have the positions given by

$$x = \{(i, x_i) | i \in [0, \cdots, N], x_i = i/N\}. \tag{3.5}$$

Notice that the variable x in this equation is described by a set of ordered pairs where the first entry is the entity identifier, whereas the second entry is the value bound to that entity. This is a more general abstraction of the array. For example `x[i]` is represented abstractly as $\{x_i | (i, x_i) \in x\}$.

In addition, this discretization yields N intervals, or cells, which are represented by the mappings between cells and interfaces by way of the following relationships

$$\begin{aligned} il &= \{(c,l) | c \in [N+1, \cdots, 2N], l = c - N - 1\}, \\ ir &= \{(c,r) | c \in [N+1, \cdots, 2N], r = c - N\}. \end{aligned} \tag{3.6}$$

The mappings il and ir provide mappings from every cell to their left and right interfaces. The domain of ir and il is $[N+1, \cdots, 2N]$, or the cells in the discretization, while the ranges are $\text{ran}(ir) = [0, \cdots, N-1]$ and $\text{ran}(il) = [1, \cdots, N]$. This mapping is used to conveniently describe subscripts, *i.e.* $x_{c-N} = ir \rightarrow x$, where the composition operator, $\rightarrow$, defines the application of the mapping, as in

$$il \rightarrow x = \{(c, x_l) | (c,l) \in il, (l, x_l) \in x\}. \tag{3.7}$$

Using this notation, it is possible to conveniently describe cell based calculations. For example, a generic description of each cell center is given by

$$x_c = (ir \rightarrow x + il \rightarrow x)/2. \tag{3.8}$$

Note that the definition of x provided by equation (3.8) is only applicable to cells since only cells are in the domain of maps ir and il; however, this does not prevent the definition of x for other entities (for example, interfaces) via other rules.

The mappings il and ir are used to describe the first step of the finite-volume discretization, where integration of equation (3.1) over each cell produces the equation

$$\frac{d}{dt}\int_{il\to x}^{ir\to x} u dx = \int_{il\to x}^{ir\to x} \nu u_{xx} dx = \nu(ir \to u_x - il \to u_x). \tag{3.9}$$

Equation (3.9) is an exact equation, which can be integrated numerically to obtain a numerical solution algorithm. For example, when a second order mid-point rule is used to evaluate the spatial integrations, these equations become:

$$\frac{d}{dt}u = \frac{\nu}{L}\left[ir \to u_x - il \to u_x\right], \tag{3.10}$$

where

$$L = \int_{il\to x}^{ir\to x} dx = ir \to x - il \to x. \tag{3.11}$$

Equation (3.10) describes the numerical method for the spatial integrations, but it is not complete. The gradient term, u_x, located at the interfaces has not been defined as a numerical approximation. The most straightforward approximation for u_x is a central difference formula using the values at the cell centers at either side of the interface. In order to perform this calculation it will be convenient to have a mapping from interfaces to cells similar to the development of il and ir. These mappings are defined by the relations

$$\begin{array}{rcl} cl & = & \{(i,l)|i \in [1,\cdots,N], l = i+N\}, \\ cr & = & \{(i,r)|i \in [0,\cdots,N-1], r = i+N+1\}. \end{array} \tag{3.12}$$

Using the definitions of cl and cr of (3.12), a numerical approximation to the gradient can be given as

$$u_x = \frac{cr \to u - cl \to u}{cr \to x_c - cl \to x_c}. \tag{3.13}$$

Notice that this equation uses the x-coordinate at the cell centers that is computed by equation (3.8). In addition, since this rule uses both maps cr and cl, it only defines u_x on the intersection of the domains of cr and cl, given by $[1,\cdots,N-1]$. By this reasoning, equation (3.13) only provides gradients at the internal faces of the domain. The gradient at the boundary faces is provided by the boundary conditions given in equations (3.3) and (3.4). The question is, how do these boundary conditions specify u_x at the boundaries without specifying u_x everywhere in the domain? Obviously additional information must be provided that constrains the application of boundary condition gradients only to the boundary interfaces. A solution to this problem can be found with the observation that the boundary interfaces have the distinction that either cl is defined or cr is defined, but not both. Using this fact, the rules for calculating the boundary gradients can be given by

$$\begin{aligned} u_x &= g(t), \text{constraint}\{\neg\text{dom}(cl) \wedge \text{dom}(cr)\}, \\ u &= h(t), \text{constraint}\{\text{dom}(cl) \wedge \neg\text{dom}(cr)\}. \end{aligned} \tag{3.14}$$

Here the constraint term added to the rule indicates a constraint on the application of the rule. In this case it constrains the application of the boundary conditions to the appropriate boundary faces.

At this point we have not selected the time integration method. For this simple example we will use the explicit first order Euler time integration method which is expressed using the following notation:

$$\frac{(u^{n+1} - u^n)}{\Delta t} = R(u^n), \tag{3.15}$$

where

$$R(u) = \nu \frac{ir \to u_x - il \to u_x}{L}. \tag{3.16}$$

Rearranging we arrive at the following time integration scheme:

$$u^{n+1} = u^n + \Delta t R(u^n) \tag{3.17}$$

At this point, the computation of u^{n+1} from u^n is completely specified. However, before any such iteration can begin, an initial value, or $u^{n=0}$, must be given. To be consistent with the finite-volume formulation, the derivation of the initial conditions begins with the integral form of equation (3.2), given by

$$\int_{il\to x}^{ir\to x} u^{n=0} dx = \int_{il\to x}^{ir\to x} f(x) dx. \tag{3.18}$$

Using a midpoint rule to numerically integrate this equation one obtains the rule

$$u^{n=0} = f(x), \text{constraint}\{(il, ir) \to x\}. \tag{3.19}$$

For this rule, the constraint is used to indicate that although the coordinates of the interfaces cancel in the derivation, their existence is predicated by the integration. In other words, the derivation assumed a cell perspective that includes left and right interface positions.

3.2 On Problem Specification

For an analytic solution method, equations (3.1) through (3.4) are sufficient to define the problem at hand. For numerical solution methods, additional definitions are required, due to the fact that these are inexact methods. For example, there are often trade-offs between discretization and accuracy that require additional specification. In addition, since discretization for complex geometries (grid generation) is not a completely automatic process, the discretization becomes part of the problem definition for numerical solution methods. For the example diffusion problem already introduced, the definition of the numerical problem consists of spatially independent information such as the diffusion constant ν, the initial condition function $f(x)$, the numerical time step Δt, and a representation of the discretization of space. The discretization of space is given by a set of positions, (3.5), and the collection of mappings given in (3.6) and (3.12). Table 3.1 summarizes these formal definitions for the example diffusion problem.

Table 3.1: A Summary of Definitions for the Example Diffusion Problem

fact	meaning
ν	given diffusion constant
$f(x)$	given initial condition
$g(t)$	given left bc
$h(t)$	given right bc
Δt	given time-step
x	$\{(i, x_i) \mid i \in [0, \cdots, N], x_i = i/N\}$
il	$\{(c, l) \mid c \in [N+1, \cdots, 2N], l = c - N - 1\}$
ir	$\{(c, r) \mid c \in [N+1, \cdots, 2N], r = c - N\}$
cl	$\{(i, l) \mid i \in [1, \cdots, N], l = i + N\}$
cr	$\{(i, r) \mid i \in [0, \cdots, N-1], r = i + N + 1\}$

3.3 On Specification of Process

Given the definition of the problem, the process of solving the problem is dictated by a prescribed set of transformations. For example, consider equation (3.8) as an example of a transformation that transforms x located at il and ir into a cell x_c. To simplify discussions of the structure of the calculations, the transformation rules are represented by a rule signature that is denoted by a list of targets of the transformation delineated from the sources of the transformation by the left arrow symbol, '$\leftarrow$'. Thus the cell center position calculation is represented by the rule signature $x_c \leftarrow (ir, il) \to x$. This rule signature represents the augmentation of the set of ordered pairs defined in equation (3.5) with the additional set given as

$$x_c \leftarrow \{(c, x_c) \mid x_c = (x_l + x_r)/2, (l, x_l) \in x, (r, x_r) \in x, (c, l) \in il, (c, r) \in ir\}. \tag{3.20}$$

For the moment, the augmentation of x with this set can be considered as a set union operation, with the caveat that it will become more complex once issues of specification consistency are considered. Given this notation, the specification of the finite-volume scheme derived in this section can be summarized by eight rules given in table 3.2.

Table 3.2: A Summary of Rules Describing the Solution of the Example Diffusion Problem.

Rule	Rule Signature	Equation
Rule 1	$x_c \leftarrow (ir, il) \to x$	(3.8)
Rule 2	$L \leftarrow (ir, il) \to x$	(3.11)
Rule 3	$u_x \leftarrow (cr, cl) \to (u, x_c)$	(3.13)
Rule 4	$u_x \leftarrow h, t, \text{constraint}\{\text{dom}(cl) \wedge \neg\text{dom}(cr)\}$	(3.14)
Rule 5	$u_x \leftarrow g, t, \text{constraint}\{\text{dom}(cr) \wedge \neg\text{dom}(cl)\}$	(3.14)
Rule 6	$R \leftarrow \nu, L, (ir, il) \to u_x$	(3.16)
Rule 7	$u^{n+1} \leftarrow u^n, R^n, \Delta t$	(3.17)
Rule 8	$u^{n=0} \leftarrow f, x_c, \text{constraint}\{(il, ir) \to x\}$	(3.19)

3.4 On Interpreting the Problem Specification

The eight rules described in the preceding section form the basis of creating a *Loci* program. Before we consider how to do this, lets first consider how the facts given in table 3.1 and the computational elements described in table 3.2 can be composed to create a time dependent simulation. Note, that the final values of interest are the values for $u^n, n = 0, 1, \cdots$. It is clear to see that rules 7 and 8 describe how to accomplish this computation. However, more information is needed to complete the operation, specifically note that we need the residual R evaluated at iteration level n, denoted by R^n. However, also note that the residual rule given by rule 6 does not have a time notation (no superscript n). This rule is said to be described at stationary time, that is the relationship is established as an invariant to iteration. That is, R is defined as the same relationship among variables regardless of iteration identifier. In this case, we know that we will need to compute R for every time-step because it is a function of u which itself is dependent on the iteration. Usually when converting a paper description of an algorithm a program implementation we need to make these determinations. In *Loci*, this is automatically performed by time promotion deductions. That is, *Loci* will automatically determine that the residual and gradients of u will need to be recomputed each time-step, while x_c and L will not. To accomplish this *Loci* will invoke either rule promotions or variable promotions, that is that if we have a rule $x_c \leftarrow (ir, il) \rightarrow x$ then *Loci* may either promote the rule to the iteration (e.g $x_c^n \leftarrow (ir^n, il^n) \rightarrow x^n$, or promote the variable to the iteration (e.g. $x_c^n \leftarrow x_c$), as needed to satisfy the computation. As a policy, *Loci* only schedules computations at an iteration as required, all other computations are performed once and reused through iterations using variable promotion.

3.5 Creating the Fact Database

Before we begin performing computations we have to describe how to express the information in table 3.1 as a *Loci* fact database (`fact_db`). The code we are about to describe is included in the `1D-Diffusion` directory in the *Loci* tutorial. The facts described in this table are essentially the result of a one dimensional grid generation process that is handled by the `generate_grid` function as described below:

```
// Generate a 1d grid over the number line from [0,1] consiting of N segments
// The resulting grid is installed in the fact_db facts
void generate_grid(fact_db &facts, int N) {
```

The first step to generating this discretization of the interval $[0, 1]$ is to allocate space for the nodes and cells that will comprise the final grid. We ask the fact database to generate unique entity numbers for these entities using the `get_allocation()` method:

```
  // Allocate the nodes and cells of the grid
  entitySet nodes = facts.get_allocation(N+1) ;
  entitySet cells = facts.get_allocation(N) ;
```

Now that we have these allocations we can first create the fact “x”. This fact use the `store` container as it is an association of floating point values with entities. This container is first allocated over the prescribed entities. The values are assigned by giving the lowest numbered entity the value of $x = 0$, and then adding a delta x to subsequent values. The resulting values are then placed in the fact database using the `create_fact` method, as shown below:

```
// setup x coordinates for nodes
store<float> x ;
x.allocate(nodes) ;
float dx = 1./float(N) ; // Uniform delta x
entitySet::const_iterator ni ;
float xtmp = 0 ;
// iterate over nodes and assign positions by adding dx to
// preceeding x value
for(ni=nodes.begin();ni!=nodes.end();++ni) {
  x[*ni] = xtmp ;
  xtmp += dx ;
}
// Add node positions to facts
facts.create_fact("x",x) ;
```

Now we need to develop the connectivity relations between cells and nodes as described by `cl` (cell left), `cr` (cell right), `il` (interface left), and `ir` (interface right). Note that while all cells will have a value for `il` and `ir`, only the interior nodes will simultaneously have values for `cl` and `cr`. To compute the nodes that will be used for these maps, we use the shifting operator of the entity set which adds or subtracts values from all entity identifiers, as in:

```
// Find the nodes that are on the left and right side of cells
// by shifting the allocated numberings to the left or right by one
entitySet left_nodes = (nodes >> 1) & nodes ;
entitySet right_nodes = (nodes << 1) & nodes ;
```

We now have the sets needed to allocate these containers. Note that since we are describing a relationship between various entities, the container that we use is a `Map`. A Map can be thought of as a `store` that contains entity identifiers instead of values. The allocation is performed as follows:

```
// Allocate maps for the left cell and right cell of a node
Map cl,cr,il,ir ;
cl.allocate(left_nodes) ;
cr.allocate(right_nodes) ;
il.allocate(cells) ;
ir.allocate(cells) ;
```

Now we are able to create these `Maps`. We use a technique of finding a correspondence between left nodes and cells or right nodes and cells to establish this relationship. In addition, we build both `cl` and `ir` at the same time recognizing that one map is simply the transpose of the other as illustrated below:

```
  entitySet::const_iterator ci ;
  // Assign left nodes to cells in consecutive order
  ci = cells.begin() ;
  for(ni=left_nodes.begin();ni!=left_nodes.end();++ni,++ci) {
    cl[*ni] = *ci ;
    ir[*ci] = *ni ;
  }
  // Assign right nodes to cells in consecutive order
  ci = cells.begin() ;
  for(ni=right_nodes.begin();ni!=right_nodes.end();++ni,++ci) {
    cr[*ni] = *ci ;
    il[*ci] = *ni ;
  }
```

We now install these relations in the fact database:

```
  facts.create_fact("cl",cl) ;
  facts.create_fact("cr",cr) ;
  facts.create_fact("il",il) ;
  facts.create_fact("ir",ir) ;
```

Now we must compute the entities that are on the boundaries. We do this by examining the domain (entities for which a container has a definition) of the `cl` and `cr` maps, similar to the previous specifications. There are two boundaries in this one dimensional problem, the left boundary and the right boundary. They are represented by `constraint` containers which are special data types used in *Loci* to give sets of entities special attributes. The boundary condition constraints are specified below:

```
  // Identify boundary conditions
  constraint left_boundary ;
  constraint right_boundary ;
  *right_boundary = cl.domain() - cr.domain() ;
  *left_boundary = cr.domain() - cl.domain() ;

  facts.create_fact("left_boundary",left_boundary) ;
  facts.create_fact("right_boundary",right_boundary) ;
```

Finally, for some computations it is useful to identify the geometric cells in the problem (for example if ghost cells were employed.) Here we can easily create such an identification using a constraint as in:

```
  constraint geom_cells ;
  *geom_cells = cells ;
  facts.create_fact("geom_cells",geom_cells) ;
```

Now we have created a *Loci* implementation of the facts described in table 3.1. To complete the *Loci* implementation we will need to describe the rules as well. This is discussed next.

3.6 Creating the rule database

To capture what is specified in the table 3.2 we will create a database of rules. To simplify this process we use the *Loci* preprocessor that will convert rule specifications into standard C++ code. A program that has a `.loci` postfix will automatically be compiled with the *Loci* preprocessor if you use one of the provided example makefiles. Any valid C++ source file is also a valid *Loci* preprocessor file, that is we are free to mix standard C++ code and *Loci* specific directives in this file. The *Loci* preprocessor becomes activated through the use of the "`$`" symbol as will be described in the following paragraphs.

In the beginning you will want to tell the *Loci* preprocessor what the types of the *Loci* variables that you will be working with. For example, the *Loci* preprocessor will need to know what the types of the facts that were created earlier to describe the one-dimensional mesh. We can describe the types to the preprocessor using the `$type` keyword. This first argument after this keyword is the variable name, while the second argument is the type. The statement ends with a semicolon. For example, the type definitions for the initial fact database are given by:

```
// Setup Types for initial facts
$type il Map ; // interface left
$type ir Map ; // interface right
$type cl Map ; // cell left
$type cr Map ; // cell right
$type x store<float> ; // node positions
```

Generally we would like to share types between separate files to simplify compilation and source code management. It is possible to do this with the *Loci* preprocessor. For example, the above definition can be put into a header file, say the file `"mesh.lh"` and then included into the *Loci* program using the keyword `$include`. NOTE: this must be `$include` not the C preprocessor directive `#include`! For example, to include the above type information file use the line:

```
$include "mesh.lh"
```

3.6.1 Specifying User Tunable Inputs

Before we begin specifying the *Loci* rules as described in table 3.2, lets discuss feature of *Loci* that allow the user to easily change some of the facts used to describe the simulation. In this case, the user may want to change the viscosity, the number of cells in the discretization, or the number of time-steps to simulate. All of these parameters can be given default values using special *Loci* rules, and then the user can change these default values by providing a special "vars" file. For example, we can specify the default values for these parameters with the following lines of code:

```
// Input parameters
$type N param<int> ;                // How many nodes
$type nu param<float> ;             // diffusion coefficient

$rule default(N) {
  $N=50 ;
}

$rule default(nu) {
  $nu = 1.0 ;
}
```

In these lines of code we can see our first *Loci* rules. The rules are preceded with `$type` specifiers so that *Loci* will know the types of the variables. Then the `$rule` specifier tells the *Loci* preprocessor that a rule is about to be defined. Immediately after the `$rule` specifier is the rule type that specifies the type of the rule. The rule signature is provided in the parenthesized region, while the braces enclose the code actually executed when the rule is required. The `default` rule type is specifically to provide a mechanism for users to change a given value in the fact database, but also to provide a default value if the user does not specify one. Note the use of the "`$` " to identify the *Loci* variables in the computational part that is enclosed in braces. We also note that we provide the `optional` rule type to specify values that the user can provide, but for which no default value will be provided if the user provides no value. In this case, the facts will simply not contain this attribute.

To utilize these `default` and `optional` rule types, the user must specifically read in values from a user provided file. For example, we read in the user provided values from a file called `heat.vars` in the file `main.cc` with the following code:

```
  string varsFile = "heat.vars" ;
  facts.read_vars(varsFile,rdb) ;
```

The user provided vars file consists of a braces delimited list of variable assignments. For example, a typical `heat.vars` file might look something like:

```
{
N: 10
nu: 10.0
}
```

3.6.2 Basic Rule Specification

Now we can begin with the implementation of the rules described in table 3.2 in *Loci*. The first of these rules is the computation of the cell center, denoted by the variable `xc` which is represented in *Loci* by the following code:

```
// Rule 1: compute the cell center from node positions
$rule pointwise(xc<-(il,ir)->x) {
  $xc = .5*($il->$x + $ir->$x) ;
}
```

In this specification, the `$rule` directive tells the loci preprocessor that we are about to describe a *Loci* rule. The keyword `pointwise` that follows indicates that the rule represents a point by point computation. The parenthesis that follow describe the rule signature which documents what the rule will use for inputs and outputs. Note that the comma in the rule signature binds most tightly, so we use parenthesis to group items. For example, `(il,ir)->x` means the same thing as `il->x,ir->x`. Similarly, `il->(x,y)` would mean `il->x` and `il->y`, whereas `il->x,y` would mean `il->x` and `y`. Note, when several variables are listed on both the sides of the "`->`" operator, then all combinations are implied. Thus `(il,ir)->(x,y)` is equivalent to `il->x` and `ir->x` and `il->y` and `ir->y`. The code that follows the signature is the actual implementation of equation (3.8) where the "`$`" is used to identify the use of *Loci* variables. NOTE: It is important that the rule signature match the implementation: if `$il->$x` is in the implementation then `il->x` needs to be in the rule signature. If not, then *Loci* may schedule the rule incorrectly.

3.6.3 Boundary Conditions and Constraints

Rule 3 provides a method for computing interface gradients. However, this rule only provides values for the internal interfaces as these are the only interfaces that have both `cl` and `cr` attributes. Boundary conditions are used to define `ux` at the left and right boundaries. For example, at the left boundary $u_x = h(t)$. If we simply defined a rule to define `ux` in this way, how do we limit the specification to only apply to the left boundary? To do this we use a rule constraint. The rule constraint is a set of entities that constrain the rule application. A constraint applied to the rule makes two assertions: 1) the rule can only be applied for entities in found in the constraint set, and 2) all inputs to the rule must be available for the entire set of entities found in the constraint. Thus, we can use the `left_boundary` constraint that was placed in the fact database to apply this boundary condition as illustrated in the following code:

```
// Neumann boundary condition at left boundary, ux = h(t)
$rule pointwise(ux<-h), constraint(left_boundary) {
  $ux = $h ;
}
```

3.6.4 Specifying Iterating Algorithms

The implementation of rules 2-6 follow a similar procedure as rule 1, and can be found in the tutorial under the file "`heat.loci`". The time-stepping rules 7 and 8 require a bit more discussion. First we need to discuss how we represent the superscripts in this rule notation. For example, how is u^n represented in *Loci* code? The superscript is represented in a brace delimited region that follows the variable name, so that u^n becomes `u{n}` in a *Loci* rule signature and `$u{n}` in the implementation. Also note that the type specifications don't include the superscript, therefore the type statement for this variable will be:

```
$type u store<float> ; // Solution variable
```

Now we can specify rules 7 and 8 in a straightforward manner as follows:

```
// Rule 7: initialization of iteration (build rule)
$rule pointwise(u{n=0}<-xc) {
  $u{n=0} = f($xc) ;
}

// Rule 8: time advance using explicit Euler time integration algorithm
$rule pointwise(u{n+1}<-u{n},dt{n},R{n}) {
  $u{n+1} = $u{n}+$dt{n}*$R{n} ;
}
```

Note that the above to rules play different roles in describing the iteration. Rule 7 is called a "build" rule in *Loci*. It describes how to build the initial values of an iteration. It is distinguished by having an output at with a time specification that includes the "=" operator. Rule 7 is called an "advance" rule in *Loci*. The advance rule describes how to advance a variable to the next iteration. It is characterized by having an output that is at an advanced time level from its inputs. These two rules tell *Loci* how to iterate the variable `u` forward in time, but does not describe how, or when, to end the iteration. For this we need a collapse rule which is a rule that tells *Loci* how to compute a value that results from the iteration. For example, we might have a collapse rule that computes a variable called "`solution`" that represents the solution of the time integration problem. Such a rule would be represented as:

```
$rule pointwise(solution<-u{n}),conditional(simulation_finished{n}) {
  $solution = $u{n} ;
}
```

Notice that this rule has an additional "`conditional`" clause. This clause is used to tell *Loci* when it is OK to terminate or "collapse" the iteration. This termination condition will be provided by another rule as described next. For the moment we will use the criterion that when the iteration variable `n` reaches a predetermined value, the loop will terminate. This is accomplished with the following code:

```
$type max_iteration param<int> ;
$type simulation_finished param<bool> ;

// Condition that determines when iteration is complete
$rule singleton(simulation_finished<-$n,max_iteration) {
   $simulation_finished = ($$n >= $max_iteration) ;
}
```

First notice that the type of `simulation_finished` is a bool parameter. This is a single boolean value shared by all of the iterating entities. The loop will terminate when this value evaluates to *true*. Also notice that this rule is a `singleton` rule. This means that the rule is computing a single value. Singleton rules are used to compute parameters from other parameters, and thus apply to single values rather than collections arrayed over entities. Finally, notice that we can access the value of the iterating superscript by using the "`$n`" variable name. This is a special variable (always typed as an integer parameter) that contains the current iteration number. Also, notice that when using this variable in the implementation, two "`$`" symbols appear. The purpose of this rule should be clear at this point: when `$n` is equal to `max_iteration` then `simulation_finished` is true, otherwise it is false. Thus this specifies when *Loci* should terminate the iteration.

3.6.5 Next Step: Global Reductions

The above specification is nearly complete. However we have left out one important detail and that is the specification of the time-step, Δt. How do we arrive at this time-step size? One approach would be to have the user specify this parameter by providing a `default` rule. This would allow the user to input a specific time-step. However, since the explicit Euler time-stepping algorithm has a stability bound, it might be better if the solver computed a stable time-step. The stability limit for this algorithm is given by the equation:

$$\Delta t = \frac{1}{2}\frac{L^2}{\nu}. \tag{3.21}$$

However, since it is possible for the mesh to be non-uniform in our formulation, we would like to find out what is the smallest possible time-step by applying this equation to all possible cells. We can accomplish this operation by using a global reduction. For a global reduction, the output of the rule is a parameter, indicating that there will be a single value shared by many entities. Reductions in *Loci* are specified using two different rule types: `unit` and `apply`. A reduction is always defined with respect to some operator that 1) has an identity, 2) is associative, and 3) is commutative. In this example we will use the minimum operator that an identity (the largest possible number) and is associative and commutative. The computation of the global stable time-step is given by the following *Loci* code:

```
$type dt param<float> ; // simulation timestep

$rule unit(dt), constraint(UNIVERSE) {
   $dt = std::numeric_limits<float>::max() ; // largest allowble timestep
}

$rule apply(dt<-L,nu)[Loci::Minimum] {
    float local_dt = $L*$L/(2.*$nu) ;  // Stable timestep
    join($dt,local_dt) ; // Set dt = min(dt,local_dt)
}
```

Here we see the unit rule being applied to compute the identity of the operator. In this specific case we use the C++ standard to query the maximum floating point value to assign to `dt`. Notice that we provide a constraint of `UNIVERSE` which indicates that `dt` will be associated with all entities in the simulation. The rule that follows is the apply rule that specifies that the `Loci::Minimum` operator will be used. *Loci* provides the following operators: `Loci::Summation`, `Loci::Product`, `Loci::Maximum`, and `Loci::Minimum`. Other operations can be defined by the user as will be described in later sections. Here the apply rule uses the `join` operator to combine the local stable time-steps with the global time-step. The first argument to the join operator is the variable that is being reduced, while the second argument is the variable that is being combined.

Note, there are some important cautions that should be mentioned here. First, it is important that the unit rule assigns the identity value because it is possible that this rule may be computed and combined multiple times (particularly in the parallel), therefore it is not OK, for example, to have the unit rule assign a partial sum expecting that the value will only be added once. Second, it can be deceptively easy to violate the associative and commutative properties of the operator. For example, consider the following *Loci* code:

```
$rule apply(sum<-terms)[Loci::Summation] {
    if($sum < 1) // Error!  Result depends on order terms are summed!
      join($sum,$terms) ;
}
```

Note, it is not the if statement that is wrong, but rather the conditional on the partially summed result. For example, the following code is OK:

```
$rule apply(sum<-terms)[Loci::Summation] {
    if($terms < 1) // OK, result is independent of summing order
      join($sum,$terms) ;
}
```

3.6.6 The *Loci* Generated Schedule

The one dimensional heat solver that is provided with the tutorial can be used to see how *Loci* will create a program from the rules we have described to create a heat solver. An execution schedule is generated by *Loci* automatically when the user issues a `Loci::makeQuery()` function call. The last argument of this function call is the name of the variable that you desire. By default, we usually query for a variable called "solution" which represents the generic solution of a problem and plays a similar role to `main` in C programs. Since the collapse rule of our Euler time-step algorithm generates the variable "solution", a query for this variable will produce a schedule that solves the time-dependent heat equation. By default, this schedule is not provided to the users of *Loci*, but *Loci* can be instructed to print this schedule in human understandable form. For example, to see the schedule generated by the heat solver, you can execute:

```
./heat --scheduleoutput --nochomp
```

The `--scheduleoutput` instructs *Loci* to write out the generated schedule into a file called ".schedule" (on a single processor). The `--nochomp` instructs *Loci* to not perform the chomping optimization on the program. This will make the schedule a little bit easier to read and understand.

Before we look at the entire program, lets first consider what happens if we query for a variable that is easier to compute, for example the time-step control, `dt`. The heat program contains code for the user to specify a query different from "solution" by using the `-q` flag. Thus we can query for the stable time-step using the command:

```
./heat --scheduleoutput --nochomp -q dt
```

In addition to computing the stable time-step, this command generates a file called `.schedule` that contains the execution schedule:

```
dt<-CONSTRAINT(UNIVERSE) over sequence ([0,20])
L<-(il,ir)->x over sequence ([11,20])
dt<-L,nu over sequence ([11,20])
```

Here we see first that the unit rule for the dt computation is called. The sequence is the set of entities that requested `dt`. Then `L` is computed because this will be needed by the `dt` apply rule that follows. Notice, both of these rules execute over the sequence `[11,20]` which are the entity identifiers for the cells in the problem. Once this is computed, the `dt` is successfully computed and the execution schedule terminates. Now lets see what happens when we query for solution:

```
dt<-CONSTRAINT(UNIVERSE) over sequence ([0,20])
ub<-g,CONSTRAINT(right_boundary) over sequence ([10,10])
xc<-(il,ir)->x over sequence ([11,20])
L<-(il,ir)->x over sequence ([11,20])
dt<-L,nu over sequence ([11,20])
promote:cr{n}<-cr
promote:left_boundary{n}<-left_boundary
promote:h{n}<-h
promote:ub{n}<-ub
promote:cl{n}<-cl
promote:xc{n}<-xc
promote:x{n}<-x
promote:max_iteration{n}<-max_iteration
promote:L{n}<-L
promote:ir{n}<-ir
promote:il{n}<-il
promote:nu{n}<-nu
promote:dt{n}<-dt
ux{n}<-h{n},CONSTRAINT(left_boundary{n}) over sequence ([0,0])
u{n=0}<-xc over sequence ([11,20])
generalize:u{n}<-u{n=0}
Iteration Loop{n} {
    simulation_finished{n}<-$n{n},max_iteration{n} over sequence ([11,20])
    if(simulation_finished{n}) {
      solution<-u{n},CONDITIONAL(simulation_finished{n}) over sequence ([11,20])
    } // if(simulation_finished{n})

    -------------- Exit of Loop{n}
    if(simulation_finished{n}) break ;

    ux{n}<-(cl{n},cr{n})->(u{n},xc{n}) over sequence ([1,9])
    ux{n}<-cl{n}->(u{n},xc{n}),ub{n},x{n} over sequence ([10,10])
    R{n}<-(il{n},ir{n})->ux{n},L{n},nu{n} over sequence ([11,20])
    u{n+1}<-R{n},dt{n},u{n} over sequence ([11,20])
} // {n}
```

While this schedule is similar to the previous schedule, we observe many new features that weren't present in the previous example. First we see that in the beginning several variables are computed that will be used during the time-stepping portion of the algorithm. This is followed by a sequence of variable promotion operations indicated by lines such as:

```
promote:cr{n}<-cr
```

This line indicates that the variable `cr` will be used unchanged during the `n` iteration. In essence it is basically indicating that `cr` and `cr{n}` are the same variable. This will allow rules that have been promoted to the time-stepping iteration to access these variables. We then notice that the initial conditions are then computed followed by a variable generalization given by:

```
generalize:u{n}<-u{n=0}
```

This indicates that for the first iteration `u{n}` is the same as `u{n=0}`. Note that after the first iteration, *Loci* sets `u{n}` to the previous iteration value that was computed for `u{n+1}`.

After the generalize rule, we begin the iteration loop. The first operation performed is to compute the condition for collapsing (terminating) the loop. If this is true, the schedule executes the collapsing code and then exits the loop. Otherwise, the loop continues to compute the next iteration value, `u{n+1}` using the provided rules. First the gradients are computed (with the exception of the Neumann BC that was computed in advance). Then the residual is computed using these gradients. And finally, the Euler time integration rule is used to advance the variable `u` to the next iteration. Once this is complete, *Loci* advances n and begins again.

Notice that everything Loci needed to form this schedule could be inferred from the rules that were provided in the previous sections. Looking at the schedule generated by Loci can be helpful in learning how to use *Loci*. Experiment with changing how rules are implemented, what they input, for example, and examine how this changes the schedule. For example, what happens if the stable time-step computation input the variable `u`? What changes would *Loci* need to make to the schedule?

3.6.7 Local Reductions: An Alternative

The fact database described in this example included two sets of maps. One set of maps provided an association from cells to interfaces (`il,ir`), and another provided an association from interfaces to cells (`cl,cr`). However, these maps are intrinsically related to one another in that `il` is the transpose of `cr` and `ir` is the transpose of `cr`. Since these maps represent a storage overhead, it is reasonable to ask if they are both necessary. In fact, we can eliminate one pair of maps if we reorganize some of our computations so that they shift from being cell centric to being interface centric. Note, that this represents a common trade-off found in unstructured solver algorithms, and *Loci* provides techniques that allows us to develop algorithms that can reduce the amount of connectivity storage required.

For example, consider the computation of the cell centers described by the rule:

```
$rule pointwise(xc<-(il,ir)->x) { $xc = .5*($il->$x + $ir->$x) ; }
```

This rule is cell centric: the context of the rule (where computations occur) is cells, as cells are the entities that have attributes `il` and `ir`. Can we transform this rule so that it uses the face centric maps `cl` and `cr`? Yes, we can. But the resulting computation is somewhat less straightforward. Essentially the computation is transformed whereby we visit all of the interfaces in the line and each interface will add half of its position to each side. When all interfaces add their parts, then the final total will be the cell center. This computation is similar to the global reduction described earlier with one important difference: this computation results in many cell centers, not just one. This many-to-many reduction process is called a local reduction. A local reduction is described in the same way as a global reduction in *Loci*, the only difference is that the result of the local reduction is a store rather than a parameter. Thus, we use a unit/apply combination to describe the computation much like the previous example. Here is an example of three rules that can replace the cell center computation using `cl` and `cr` maps:

```
$rule unit(xc), constraint(geom_cells) {  $xc = 0 ; }
$rule apply(cl->xc <- x)[Loci::Summation] {  join($cl->$xc,.5*$x) ; }
$rule apply(cr->xc <- x)[Loci::Summation] {  join($cr->$xc,.5*$x) ; }
```

First note that we have a unit rule that assigns the value of the cell center to the identity of summation (zero). Also note that the rule is constrained to exist only for geometric cells (we put this constraint `geom_cells` in the fact database, as described earlier). This constraint keeps this rule from defining values for entities that are not cells. Now to compute the cell center we look at each interface and add half its position to both the left and right sides. These three rules are equivalent to Rule 1, but use the maps `cl` and `cr` instead. Note that the map appears in the output of the rule rather than the input, and this is essentially how the map becomes transposed. In fact, as a rule of thumb, usually pointwise rules have mappings in the inputs, while apply rules have mappings in their outputs for exactly this reason. Applying this technique to other rules that use `il` and `ir` it is possible to completely eliminate the need for these maps.

Note: it would be tempting to combine these two rules into one adding to both the left and right sides at the same time. However, the boundary faces would be left out in this case, since they don't have both a left and right side.

3.6.8 Getting Sophisticated: Parametric Rules

When developing *Loci* rules, one quickly will notice patterns that are repeated in many different rules. In such cases it would be useful to define a family of rules that can be called upon to act on different variables. For example, it would be useful to define a gradient operator that could apply to different variables, then rather than have variable `ux` and an associated rule to compute this specific gradient, it would be useful to have a way of specifying `grad(u)` instead. We can do this using parametric rules. For this one dimensional heat equation, there are several cases where it would be useful to have a parametric rules to capture these common structures. One example is the integration of fluxes that results in a difference between interfaces. It would be useful to have a cell integral function to perform this common operation. In addition, specifying this as a parametric rule will simplify the process of converting to using the `cl` and `cr` maps.

Parametric rules are defined when parametric variables are used in their outputs. A parametric variable is one that has parameters as defined by a parenthesized list of substitution keys. Thus, for example, if we want to create a cell integration of some variable, we can provide that using the parametric variable `cellIntegrate(X)`. How *Loci* interprets the parametric variables depends on where the variable occurs in the rule signature. If the parametric variable is in the output of a rule, then the rule is parametric and describes a family of rules that can be obtained by performing substitutions of the keys listed in the argument list. However, if the parametric variable is in the input, then it becomes a request for an instantiation of a specific version of the parametric rule that is obtained by substituting the provided arguments for the substitution keys given in the parametric rules. This may seem somewhat confusing, but it is actually very straightforward once you see it in action.

For example, suppose that we wished to create rule that could integrate from the bounding interfaces of a cell as is obtained from flux integrations used in the residual equation. We could create a parametric rule to describe an arbitrary integration such as the following code segment:

```
$type cellIntegrate(X) store<float> ;
$type X store<float> ;

$rule pointwise(cellIntegrate(X)<-(il,ir)->X) {
   $cellIntegrate(X) = $ir->$X - $il->$X ;
}
```

What this rule represents is that for some arbitrary variable `X`, `cellIntegrate` can be computed for a given variable, say `R`, by substituting `X` for `R` in the above rule. Thus once we have the rule above we can compute the residual with the following rule:

```
// The 1d diffusion residue
$rule pointwise(R<-nu,cellIntegrate(ux),L) { $R = $nu*$cellIntegrate(ux)/$L ;}
```

Note, that once we have defined this integral, we can use it in other places which could be described in terms of this form of definite integral. For example the length of a cell can actually be represented as an integral of this form, thus we can also use the parametric rule to compute `L` as follows:

```
// We find the length of an interval by integrating the position x
$rule pointwise(L<-cellIntegrate(x)) { $L = $cellIntegrate(x) ; }
```

Parametric rules behave somewhat like subroutines and share many of the same advantages. One advantage is that because many related functions share the same implementation, it is possible to replace one method of computation with another that is transparent to other parts of the computations. In the same way, the use of the `cellIntegrate` parametric rule will allow us to change fewer parts of the program to transform from a cell centric integration as described earlier, to a face centric integration that uses local reductions. Note that parametric rules are compatible with all types for rule specifications and can be used with unit and apply rules. Thus, we can replace the earlier cellIntegrate formulation with the following set of rules to eliminate the use of the `il` and `ir` mappings, as is demonstrated with the following code:

```
// A general function for integrating over a cell boundary
$rule unit(cellIntegrate(X)),constraint(geom_cells) {
  $cellIntegrate(X) = 0 ;
}
$rule apply(cl->cellIntegrate(X)<-X)[Loci::Summation] {
  join($cl->$cellIntegrate(X),$X) ;
}
$rule apply(cr->cellIntegrate(X)<-X)[Loci::Summation] {
  join($cr->$cellIntegrate(X),-$X) ;
}
```

3.6.9 Iterations and Parametric Rules

It is reasonable to ask if we can build iterative algorithms using parametric rules. The answer is: yes we can. For example, if we could build the explicit Euler time integration using parametric rules, then we could employ this time integration method with other residual equations to build various different types of solvers. Here, let us demonstrate how to build a parametric explicit Euler solver to show how it is done. First we need to define what parameters are needed to define the integration method. We select two parameters, the first is the function to be integrated, and the second is the variable of integration. With such a specification we would only need to query for `EulerIntegrate(R,u)` to perform the integration of the diffusion equation described earlier. Now lets see how we accomplish this goal. First we will need to define the types of the variables we will use:

```
$type EulerIntegrate(X,Y) store<float> ;
$type X store<float> ;
$type Y store<float> ;
$type Y_ic store<float> ;
```

Note the variable `Y_ic` will be used for setting the initial conditions. We do that by defining the build rule as follows:

```
// Initialize the iteration using the initial conditions
$rule pointwise(EulerIntegrate(X,Y){n=0}<-Y_ic) {
  $EulerIntegrate(X,Y){n=0} = $Y_ic ;
}
```

Notice that here the special role that the underscore character plays in parametric rules. That is a variable `YY` will not be substituted as it is different from `Y`. However, when the underscore is present, the substitution occurs on each part between the underscore characters. Thus when a rule asks for the variable `EulerIntegrate(R,u)` the above rule will instantiate the rule with the signature

```
EulerIntegrate(R,u){n=0}<-u_ic
```

Thus, we have a mechanism for providing the initial conditions for a specific variable.

Now that we have initialized the iteration, we need to specify when to terminate the iteration. This is accomplished in the same way as previously described, only now using parametric variables instead:

```
// Collapse iteration when finished
$rule pointwise(EulerIntegrate(X,Y)<-EulerIntegrate(X,Y){n}),
                conditional(eulerTimestepFinished{n}) {
  $EulerIntegrate(X,Y) = $EulerIntegrate(X,Y){n} ;
}

// Condition for terminating the timestepping algorithm
$rule singleton(eulerTimestepFinished<-$n,max_iteration) {
  $eulerTimestepFinished = ($$n >= $max_iteration) ;
}
```

We can now describe how to advance the integrated variable to the next time-step by creating a parametric advance rule as that will input the function to be integrated, `Y`. This is performed as follows:

```
// Advance the timestep to the next value
$rule pointwise(EulerIntegrate(X,Y){n+1}<-EulerIntegrate(X,Y){n},dt{n},X{n}) {
  $EulerIntegrate(X,Y){n+1} = $EulerIntegrate(X,Y){n}+$dt{n}*$X{n} ;
}
```

However, we now have a problem: when we try to use this integrator with the rules we have provided before, then the residual function will eventually need the variable “u”. However, in this parametric form, the variable `EulerIntegrate(R,u)` is playing the same role. How do we make `u` available in the iteration? Since this variable would be the output of the rule, the rule wouldn't be identified as a parametric rule by *Loci* so we have no way of specifying this step. In this case, we have to explicitly inform *Loci* that we intend for the rule to be a parametric rule using the parametric keyword as shown:

```
// Extract integration variable so that the residual function can use it
$rule pointwise(Y<-EulerIntegrate(X,Y)),parametric(EulerIntegrate(X,Y)) {
  $Y = $EulerIntegrate(X,Y) ;
}
```

Now with this rule available, the time integration loop will create a variable `u` that can be used by the residual function, `R`.

Now that we have the parametric rules defined, how do we use them to solve the heat equation described earlier? We only need to specify two rules: 1) we need to define the initial conditions, and 2) we need a rule that will instantiate `EulerIntegrate(R,u)`, as shown here:

```
// Setup the initial conditions
$rule pointwise(u_ic<-xc) {
  $u_ic = initialCondition($xc) ;
}
```

```
// Ask to solve the problem by using the Euler Integration on the function
// residual, integrating the variable u
$rule pointwise(solution<-EulerIntegrate(R,u)) {
  $solution = $EulerIntegrate(R,u) ;
}
```

Now we can see how things get assembled by looking at the resulting schedule:

```
Iteration Loop{n} {
    eulerTimestepFinished{n}<-$n{n},max_iteration{n} over sequence ([11,20])
    if(eulerTimestepFinished{n}) {
      EulerIntegrate(R,u)<-EulerIntegrate(R,u){n},CONDITIONAL(eulerTimestepFinished{n}) over
    } // if(eulerTimestepFinished{n})

    -------------- Exit of Loop{n}
    if(eulerTimestepFinished{n}) break ;

    cellIntegrate(ux){n}<-CONSTRAINT(geom_cells{n}) over sequence ([11,20])
    u{n}<-EulerIntegrate(R,u){n} over sequence ([11,20])
    ux{n}<-(cl{n},cr{n})->(u{n},xc{n}) over sequence ([1,9])
    ux{n}<-cl{n}->(u{n},xc{n}),ub{n},x{n} over sequence ([10,10])
    cr{n}->cellIntegrate(ux){n}<-ux{n} over sequence ([0,9])
    cl{n}->cellIntegrate(ux){n}<-ux{n} over sequence ([1,10])
    R{n}<-L{n},cellIntegrate(ux){n},nu{n} over sequence ([11,20])
    EulerIntegrate(R,u){n+1}<-EulerIntegrate(R,u){n},R{n},dt{n} over sequence ([11,20])
} // {n}
solution<-EulerIntegrate(R,u) over sequence ([11,20])
```

Note, we have provided these advanced examples in the heat solver directory under the file "heat2.loci".

Chapter 4

A Three Dimensional Solver

To get into some more practical applications of Loci, lets consider the development of a three dimensional implicit heat equation solver. For this example we will use the finite-volume module that is included as part of the Loci framework that provides utilities for reading three dimensional unstructured meshes, applying boundary conditions to these meshes, and facilities for computing standard operators and integrations used in finite-volume discretizations. Before we begin, lets state the problem that we wish to solve, namely the time dependent heat equation in three dimensions which is given by the equation

$$\frac{\partial}{\partial t}(\rho e) = \nabla \cdot (k \nabla T), \tag{4.1}$$

where ρ is the material density, e is the heat energy, k is the material conductivity, and T is the temperature. If we assume a constant heat capacity, c_p, then temperature is given by the equation $T = e * c_p$.

This is equation is solved using a finite-volume discretization by integrating the equation above over each mesh cell (using the divergence theorem on the right-hand-side). Thus we can rewrite this equation as

$$\int_{\Omega_c} \frac{\partial}{\partial t}(\rho e)\, dV = \int_{\partial\Omega_c} k \nabla T \cdot dS, \tag{4.2}$$

where Ω_c represents the volume occupied by cell c, and $\partial\Omega_c$ represents its respective surface. We can then derive a second order finite-volume scheme by employing a midpoint numerical integration to arrive at the discrete equation

$$\mathcal{V}_c \frac{d}{dt}(\rho e) = \sum_{f \in faces} \left[\mathcal{A}_f k \left(\nabla T_f \cdot \vec{n}_f\right)\right]. \tag{4.3}$$

To arrive at a final scheme, we must select a method for integrating this equation in time. For this example, we will employ an implicit backward Euler time integration method which is written as

$$\mathcal{V}_c \frac{Q^{n+1} - Q^n}{\Delta t} = R(Q^{n+1}), \tag{4.4}$$

where $Q = \rho e$ and the residual function, $R(Q)$, is given by

$$R = \sum_{f \in faces} \left[\mathcal{A}_f k \left(\nabla T_f \cdot \vec{n}_f\right)\right]. \tag{4.5}$$

We can then linearize the equation by using Taylors theorem to arrive at

$$R(Q^{n+1}) = R(Q^n) + \frac{\partial R(Q)}{\partial Q}\Delta Q + O(\Delta t^2), \tag{4.6}$$

where $\Delta Q = Q^{n+1} - Q^n$. Now we can now rearrange these equations to state an implicit scheme for solving the heat equation as

$$\left[\frac{\mathcal{V}_c}{\Delta t}I - \frac{\partial R(Q)}{\partial Q}\right]\Delta Q = R(Q). \tag{4.7}$$

Thus we can solve this problem by combining equation (4.7) and (4.5). Note that the term in square brackets on the right of equation (4.7) is a matrix, and thus the overall equation forms a linear system that must be solved to determine the change in Q, given by ΔQ.

Now we can begin to describe how to solve this equation using the Loci framework. First we need to construct a fact database that contains a representation of the finite-volume mesh as will be described in the following sections.

4.1 Using the Loci finite-volume module

Loci provides a finite-volume module that provides common facilities for building finite-volume methods. The first step to using this module is to load the rules from the finite-volume module into the rule database. This is accomplished with the following code:

```
rule_db rdb ; // Create the rule database
rdb.add_rules(global_rule_list) ; // Add any user defined rules ;

// Load in the finite-volume module called "fvm"
Loci::load_module("fvm",rdb) ;
```

The finite-volume module provides facilities for computing cell and face centroids, finding gradients, performing integrations, interpolating from cell to nodal values, and solving linear systems. The types for these rules are defined in the Loci include file "FVM.lh" which can be included using the Loci preprocessor by including the following line at the beginning of your `.loci` program file:

```
$include "FVM.lh"
```

In addition to a collection of rules for performing finite volume computations, Loci provides a function for reading in parallel a finite-volume mesh file and creating the appropriate data-structures in the Loci fact database. Before reading in the grid, we read in the user defined facts using the `read_vars` method of the fact database. Once we have done this, we can read in the mesh file with the function called `Loci::setupFVMGrid` which gets passed the filename and the fact database. This subroutine will place the finite-volume data-structures into the provided fact database. For example, we can read the mesh file `heat.xdr` with the following code:

```
// First read in user defined facts
string varsFile = "heat.vars" ;
facts.read_vars(varsFile,rdb) ;

// Next read in the grid file
string file = "heat.xdr"
if(!Loci::setupFVMGrid(facts,file)) {
  cerr << "unable to read grid file '" << file << "'" << endl ;
  Loci::Abort() ;
}
```

At this stage a set of facts have been installed in the fact database (`facts`) that describes the unstructured finite-volume mesh. These facts are summarized in table 4.1 shown below. These include the variable `pos` which contains the 3d vector positions of the nodes, the variable `face2node` which contains the nodes in a right-hand-rule order that form the faces. The variables `cl` and `cr` contain the cells to the left and right side of the face respectively as is illustrated in figure 4.1. Note that the face2node map provides a very specific ordering. Namely, when using the right-hand rule, the normal of the face points away from the left cell and into the right cell. In addition, all boundary faces have a normal pointing out of the boundary which means that the left cell is the cell next to the boundary and the right cell is a "ghost" or fictitious cell which might be used to implement certain boundary conditions (such as periodic boundaries). Another important aspect of the orientation of the `face2node` map is that it relates to a coloring of the cells such that the left cell is a lower numbered "color". Thus, if one follows faces from left to right, it is impossible to end up back where one began. This fact will be used in later sections when matrix assembly is required.

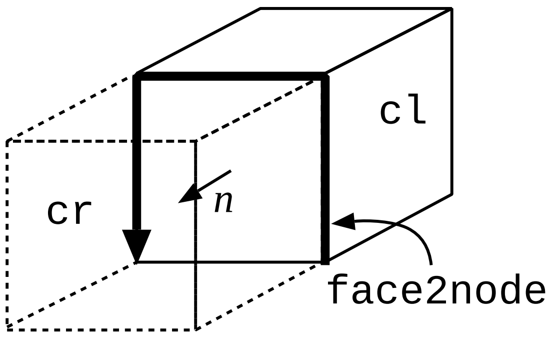

Figure 4.1: The `face2node` map and its relationship to neighboring cells

All boundary faces have a map variable called `ref` that refers to a common entity that represents the boundary surface. The name of each boundary surface is provided in the the `boundary_name` fact. These two variables can be used to assign boundary conditions as will be described in the following section. Finally the constraint `geom_cells` is a set that contains all geometrically defined (*i.e.* physical) cells in the grid, while the constraint `cells` contains all cells including non-physical "ghost" cells created at the boundary.

Table 4.1: A Summary of Facts Describing the 3-D finite-volume mesh.

Fact	Type	Location	Description
pos	store<vector3d>	nodes	Node Positions
face2node	multiMap	faces	Nodes that form a face
cl	Map	faces	cell left of face
cr	Map	faces	cell right of face
ref	Map	boundary faces	map to referring category
boundary_names	store<string>	boundary categories	boundary category name
geom_cells	constraint	physical cells	set of actual cells
cells	constraint	cells	cells including ghost cells

4.1.1 Setting boundary conditions

When developing solvers for unstructured grids, it is useful for the user to specify information that will be associated with various boundary conditions. For example, in the heat solver we may wish to provide an adiabatic and a specified temperature (Dirichlet) boundary condition. This can be accomplished by using the `boundary_conditions` variable in the vars file. This variable will assign boundary conditions to the various boundary names. For example, the `vars` file input

```
boundary_conditions: <
 BC_1=adiabatic, BC_2=adiabatic, // opposing slice faces
 BC_3=adiabatic, BC_4=adiabatic, // Two symmetry planes
 BC_5=specified(Twall=300K),     // inner surface
 BC_6=specified(Twall=3000K)     // outer surface
>
```

would specify that some of the boundary surfaces, namely those identified as `BC_1`, `BC_2`, `BC_3`, and `BC_4` will be have adiabatic boundary conditions, while two boundary surfaces, `BC_5` and `BC_6` will be given specified temperatures. Loci provides the `setupBoundaryConditions` subroutine for parsing this `vars` file input to create constraints that can be used by Loci rules to apply boundary conditions. This is created by calling the subroutine as follows:

```
   setupBoundaryConditions(facts) ;
```

This routine parses the `boundary_conditions` variable and sets up constraints with the name of the assigned boundary condition with an appended "`_BC`". Thus after this routine is called the constraints `adiabatic_BC` is created that contains all of the faces of `BC_1` through `BC_4`, while the constraint `specified_BC` is created to contain all of the faces of `BC_5` and `BC_6`. In addition, a constraint is created called `Twall_BCoption` that contains the **surface entities** associated with `BC_5` and `BC_6`. These constraints indicate that it will be safe to extract these values from the options list that is created called `BC_options`. Thus, we can extract the value of `Twall` by using a rule such as the following:

```
// Extract Twall from boundary condition options
$rule pointwise(Twall<-BC_options),constraint(Twall_BCoption) {
  $BC_options.getOptionUnits("Twall","kelvin",$Twall) ;
}
```

Once we have extracted the boundary option specified by the user, this value can then be used to enforce the boundary condition. For example, if we wanted to set the face temperature, `temperature_f`, for the specified boundary conditions we would use the following:

```
// Temperature at wall set to specified condition
$rule pointwise(temperature_f<-ref->Twall),constraint(specified_BC) {
  $temperature_f = $ref->$Twall ;
}
```

4.1.2 Creating matrix data-structures

Before we begin to make a query, we will need to setup the matrix data-structures that will be used by the linear system solvers. We do that with the following call:

```
   createLowerUpper(facts) ;
```

This subroutine call will install the multiMaps "upper" and "lower" into the fact database that are created by transposing the "cl" and "cr" maps respectively. These two data-structures represent the upper and lower triangular portions of the matrix as will be explained in the matrix assembly section that follows. We would also mention that many utilities in the finite-volume module make use of these maps, so it would be a good idea to make this call when you plan on using the "fvm" module as we are in this example.

4.2 Computing the residual function

The first step in computing the residual function, $R(Q)$ is to compute the integrated flux through any given face. This is the term inside of the summation of equation (4.5). This equation can easily be evaluated using the rules available through the "fvm" module which can provide the areas, normal vectors, and gradients required. The variable `area` provides both the normal vector (`$area.n`) and the magnitude (`$area.sada` - sada stands for square root of area dot area). The gradient at a face is provided by the parametric variable (`grads_f(X)`, thus the heat flux through a face (`qdot`) is computed using the following Loci rule:

```
// Compute the heat flux through faces
$rule pointwise(qdot<-conductivity,grads_f(temperature),area) {
  $qdot = $area.sada*$conductivity*dot($grads_f(temperature),$area.n) ;
}
```

Note that this will compute fluxes at the face and boundaries. However, if we know the flux a particular kind of boundary face (e.g. adiabatic walls) then we need to provide an alternative computation. This can present a problem for Loci: if there are two ways to compute qdot, which one is correct? Loci will generate an error about conflicting rules when such ties exist. However, we can tell Loci how to resolve such a conflict by using a priority specifier in the signature. For example, if we use adiabatic::qdot in the output of the signature, then Loci will know that this form of qdot is a specialized form that is preferred over the general qdot. Thus we can override the above rule for adiabatic boundaries, setting the flux identically to zero here, using the rule:

```
// Handle Boundary Conditions
// Adiabatic Wall, qdot = 0, grad(temperature) = 0
$rule pointwise(adiabatic::qdot),constraint(adiabatic_BC) {
  $qdot = 0 ;
}
```

Another note about priority specification. Priorities form a hierarchy so that a priority specification overrides any rule without priority specification. And we can then override one priority with another by adding additional names with "::" separators. For example, new::adiabatic::qdot could be used to override adiabatic::qdot. However, specified::qdot would not override adiabatic::qdot because both have the same priority level.

Note that we have computed the fluxes for each face, however the computation described in equation (4.5) is a sum for each cell over all the faces of that given cell. However, we have computed values located at the faces in the mesh, not the cells. How do we perform the cell based sum? For this operation we use a local reduction to reduce the face values to sums over cells. The basic idea is to visit all faces and sum the fluxes to their corresponding left and right cells. As with all reductions in Loci, this operation starts with a unit rule that assigns the initial value of the residual to the identity of our reduction operator. In this case, we will use the summation operator, and so this identity is zero. So we begin the summation with the following unit rule:

```
// Add up contributions from all faces, only define qresidual for geom_cells
$rule unit(qresidual),constraint(geom_cells) {
  $qresidual = 0 ;
}
```

To complete the summation, we must now visit each cell and add its contribution to the left and right sides. For each face we use the join method to add qdot to the left cell residual. Then we do the same for the right side of faces. However, we note that since the face is oriented such that the normal points into the right side, we must change the sign of qdot before adding into the residual on this side. Also note that we could have used the += operator instead of using join, however using join is preferred as it guarantees that we are using the selected operator (in this case Loci::Summation). If you aren't consistent in the use of operators with apply rules, you may get inconsistent results, particularly when running in parallel.

```
// Add to left cell
$rule apply(cl->qresidual<-qdot)[Loci::Summation],
  constraint(cl->geom_cells) {
  join($cl->$qresidual,$qdot) ;
}
```

```
// Add to right cell, note sign change due to normal pointing into cell
$rule apply(cr->qresidual<-qdot)[Loci::Summation],
  constraint(cr->geom_cells) {
  join($cr->$qresidual,-$qdot) ;
}
```

Note that in order to compute the gradient of temperatures as is required in the first rule that computes `qdot`, that we need to also provide values for temperatures at the boundary faces. This is provided through the variable `temperature_f`. Note, that in general if we want to find the gradient of variable `X`, then the variable `X_f` needs to also be defined at the boundary faces. We derive the boundary temperature based on the boundary condition. For an adiabatic wall we just use the cell temperature inside as the wall temperature. Since the temperature of the cell next to the boundary is on the left side of the boundary face (all boundary faces have normals pointing out of the domain), then we simply set the wall temperature to `cl->temperature`, as we do in the following rule:

```
// Compute boundary temperatures
// adiabatic,  dT/dx = 0, so copy temperature from cell to face
$rule pointwise(temperature_f<-cl->temperature),constraint(adiabatic_BC) {
  $temperature_f = $cl->$temperature ;
}
```

For a temperature specified wall, we simply set the boundary face temperature to the user specified value. We can get this value using the `ref` map, provided that a rule to extract this from `BC_options` has already been provided (see previous section on boundary conditions). Thus for the specified temperature boundary condition, we compute the temperature with the following rule:

```
// Temperature Specified Wall
$rule pointwise(temperature_f<-ref->Twall),constraint(specified_BC) {
  $temperature_f = $ref->$Twall ;
}
```

We have now described how to compute the residual. Now we need to assemble the matrix on the left-hand-side of equation (4.7) to complete the solver. This is described in the next section.

4.3 Assembling the matrix

Before describing the assembly of the matrix, lets first begin with the computation of derivatives of the face fluxes with respect to the variables at each side. However, to properly describe the matrix, we will need to have some idea of how the gradient function is implemented. For the purpose of assembling the Jacobian, we will use the following approximation of the flux function, which will be a reasonable approximation for most grids of reasonable quality. Note the flux function is expressed as

$$\dot{q} = \mathcal{A}_f k \left(\nabla T_f \cdot \vec{n}_f\right) \approx \mathcal{A}_f k \left(\frac{T_l - T_r}{(\vec{x}_l - \vec{x}_r)\cdot \vec{n}_f}\right), \tag{4.8}$$

where T_l, x_l, T_r, and x_r are the temperatures and cell centroids of the left and right side of the face respectively. Given this, we can then determine the derivatives of this flux with respect to the left and right conservative variable values. Thus we arrive at the following derivatives:

$$\frac{\partial \dot{q}}{\partial Q_l} = \frac{\partial \dot{q}}{\partial T_l}\frac{\partial T_l}{\partial Q_l} = \frac{\mathcal{A}_f k}{(\vec{x}_l - \vec{x}_r)\cdot \vec{n}_f}\frac{\partial T_l}{\partial Q_l}, \tag{4.9}$$

and

$$\frac{\partial \dot{q}}{\partial Q_r} = \frac{\partial \dot{q}}{\partial T_r}\frac{\partial T_r}{\partial Q_r} = -\frac{\mathcal{A}_f k}{(\vec{x}_l - \vec{x}_r)\cdot \vec{n}_f}\frac{\partial T_r}{\partial Q_r}. \tag{4.10}$$

These derivatives are easily implemented as Loci rules such as the following:

```
// Derivative of flux from left side
$rule pointwise(dqdotdQl<-conductivity,(cl,cr)->cellcenter,area,cl->dTdQ) {
  real distance = dot($cl->$cellcenter-$cr->$cellcenter,$area.n) ;
  $dqdotdQl = $area.sada*$conductivity*$cl->$dTdQ/distance ;
}

// Derivative of flux from right side
$rule pointwise(dqdotdQr<-conductivity,(cl,cr)->cellcenter,area,cr->dTdQ) {
  real distance = dot($cl->$cellcenter-$cr->$cellcenter,$area.n) ;
  $dqdotdQr = -$area.sada*$conductivity*$cr->$dTdQ/distance ;
}
```

Note, however, that these derivatives only apply at the interior faces where `cellcenter` exists on both sides of the face. We need separate derivations for the derivatives of the flux at the boundary faces. here we implement two obvious boundary conditions flux derivatives that follow easily from the above formulations:

```
// derivatives of boundary flux for specified temperature wall
$rule pointwise(dqdotdQl<-conductivity,facecenter,cl->(cellcenter,dTdQ),area),
  constraint(specified_BC) {
  real distance = dot($cl->$cellcenter-$facecenter,$area.n) ;
  $dqdotdQl = $area.sada*$conductivity*$cl->$dTdQ/distance ;
}

// derivative of boundary flux for adiabatic wall (zero)
$rule pointwise(dqdotdQl),constraint(adiabatic_BC) {
  $dqdotdQl = 0 ;
}
```

Before describing how to assemble these derivatives into a system matrix, we should spend a little more time describing data-structures that were created by the call to `setupFVMGrid` and to `createLowerUpper`. While the topology of the face2node map has already been discussed, it should be noted that the selection of the local face coordinates could be considered arbitrarily. However, in the Loci finite-volume tools, we have a specific method of choosing this ordering: specifically the faces are oriented so that the normals point from a lower equation number to a higher equation number. This allows us to associate the matrix non-zero structure directly with the mesh data-structure. Thus, when we invert the `cr` map we get a map that corresponds to the lower triangular part of the system matrix, where the off-diagonal components are stored at the faces. Similarly, the inverse of the `cl` map provides us with the upper triangular part of the system matrix. We can now describe the system matrix using the left and right flux derivatives located at the faces that form the L and U components of the system matrix, supplementing this with a diagonal element, D, stored at the cells. This is illustrated in figure 4.2 which shows the flux Jacobians from the left and right sides (denoted as fjp and fjm respectively) located at the cells, and the D entries associated with the cell. Alternatively, one can look at figure 4.3 to get the matrix assembly view. Here we see that along a row the `lower` map can be used to access the lower non-zero elements, while the corresponding `cl` map can be used to find the corresponding equation with that non-zero entry. A similar procedure is used to describe the upper matrix, only using the `upper` map and $\widehat{\text{cr}}$ to accomplish the same goal. This provides us with a convenient representation of the assembled matrix.

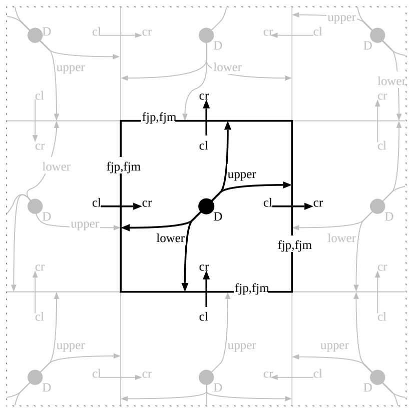

Figure 4.2: Upper and lower maps viewed from the cell

4.3.1 And now the assembly

Now we can describe the procedure for assembling the linear system. First we provide the overall linear system with a name that will be used to collect the different parts of the system. For example, lets call this system the **heat** system of equations. Then we form the linear system by creating the variables **heat_B** for the right-hand-side, and **heat_L**, **heat_U**, and **heat_D** for the system matrix, $A = L + D + U$. First the creation of heat_B is straightforward as it is just the residual function. This is easily implemented as:

```
$rule pointwise(heat_B<-qresidual) {
  $heat_B = $qresidual ;
}
```

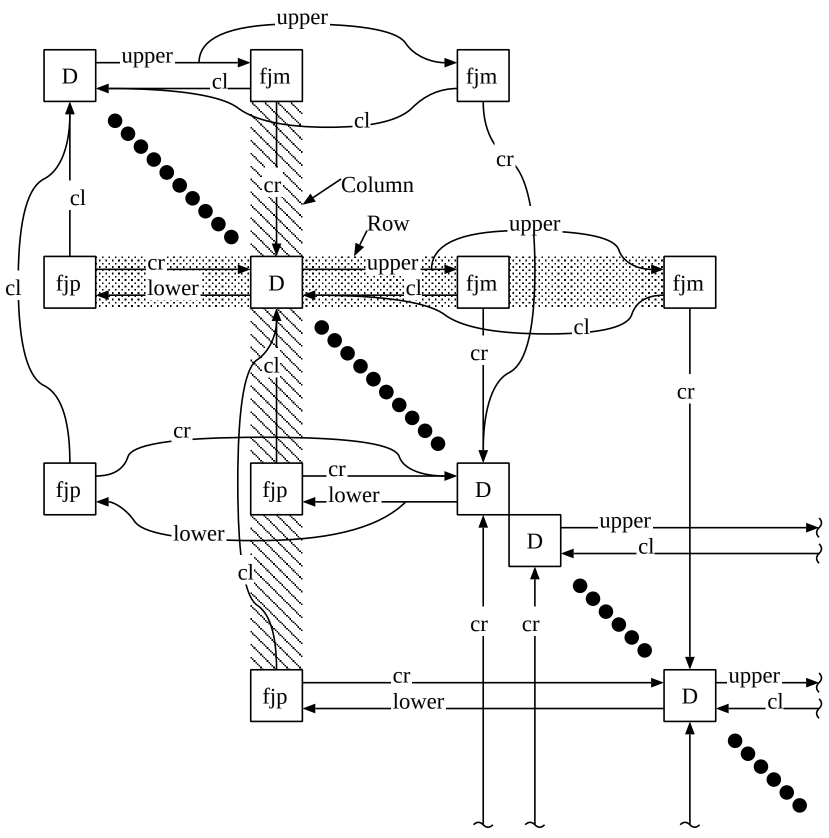

Figure 4.3: Matrix Data-Structure as used by the finite-volume module

The computation of the diagonal term is somewhat more complex. The diagonal term will involve the derivatives of the fluxes (with appropriate sign changes to account for normal vector orientation), plus the term that appears due to the time derivative. We perform this operation with two steps. First we compute the sum of all diagonal contributions from the flux derivatives using a local reduction similar to what was done for the residual evaluation. However, in this case we have to select the derivative side so that it results in a diagonal contribution. This is accomplished with the following code:

```
// To compute the diagonal term, we first must sum the diagonal
// contributions from the flux derivatives.
$type sumDiagonal store<real> ;

// Add up diagonal contributions from flux derivatives
$rule unit(sumDiagonal), constraint(geom_cells) {
  $sumDiagonal = 0 ;
}

// Add contribution from face to left cells
// (e.g. d R(Ql,Qr)/d Ql goes to diagonal of the left cell)
$rule apply(cl->sumDiagonal<-dqdotdQl)[Loci::Summation],
  constraint(cl->geom_cells) {
  join($cl->$sumDiagonal,$dqdotdQl) ;
}

// Add contribution from face to right cells
// (e.g. d R(Ql,Qr)/d Qr goes to diagonal of the right cell)
// Note sign change due to normal pointing into the cell
$rule apply(cr->sumDiagonal<-dqdotdQr)[Loci::Summation],
  constraint(cr->geom_cells) {
  join($cr->$sumDiagonal,-$dqdotdQr) ;
}
```

We can now form the diagonal term of the matrix with the straightforward Loci pointwise rule:

```
$rule pointwise(heat_D<-sumDiagonal,deltaT,vol) {
  $heat_D = $vol/$deltaT - $sumDiagonal ;
}
```

Note, that it would be tempting to put the term `$vol/$deltaT` into the unit rule of *sumDiagonal* to save a rule implementation. However such an approach would be incorrect as Loci expects the unit rule to initialize the variable to the identity of the reduction operator. Since this value would not be the identity, some of the reasoning that Loci uses to prove programs, in particular parallel programs, would be incorrect. Note, that it would be possible to combine this with the reduction by having another apply rule add in the `vol/deltaT` term. However, this would be provided by an additional apply rule, not the unit rule.

To finish the assembly of the linear system we need to describe the upper and lower triangular portions of the matrix. Note that for the description of these terms we need to carefully consider the signs of the resulting system. Recall that the system matrix subtracts the flux derivatives, and we also need to take into consideration the sign changes due to normal vector compatibility with the divergence theorem. Here is the logic: The lower portion of the matrix consists of derivatives of fluxes that were summed into the current cell through it's `cr` map with respect to the cell Q on the left side of this face. Because this was summed with a change of sign due to the normal pointing into the cell, and the subtraction in the system matrix, this term is $-(-(\frac{\partial \dot{q}}{\partial Q_l}))$. The upper off-diagonal terms, by similar argument, will be $-(\frac{\partial \dot{q}}{\partial Q_r})$. The following rules define the off-diagonal parts of the matrix and complete the assembly of the linear system.

```
// Compute matrix lower term from flux derivatives
// Note, we are subtracting del R/del Q in the matrix so there is an extra
// sign change here
$rule pointwise(heat_L<-dqdotdQl) {
  $heat_L = $dqdotdQl;
}

// Compute matrix upper term from flux derivatives
$rule pointwise(heat_U<-dqdotdQr) {
  $heat_U = -$dqdotdQr;
}
```

Now that we have formed the linear system, we can use a parametric rule from the "fvm" module to use the PETSc toolkit to solve the linear system by using the following rule:

```
// Solve linear system described by heat_B, heat_D, heat_L, heat_U
$rule pointwise(deltaQ<-petscScalarSolve(heat)) {
  $deltaQ = $petscScalarSolve(heat) ;
}
```

4.4 Performing the time integration

Now we have nearly all of the components in place to build the implicit time integration method. The completion of this process follows the same lines as the explicit Euler integration provided in the one-dimensional example. We begin this time-stepping method by building the first iteration value for the conservative variable. This is accomplished using the following "build" rule:

```
// Initial Conditions
$rule pointwise(Q{n=0}<-Density,Cp,T_initial) {
  $Q{n=0} = $Density*$Cp*$T_initial ;
}
```

Now that we have $Q^{n=0}$ defined, we can iterate by simply defining how to get from Q^n to Q^{n+1}. This step is trivial using the `deltaQ` computed in the previous section. The "advance" rule that does this is specified as:

```
// Advance the timestep using linear system solution
$rule pointwise(Q{n+1}<-Q{n},deltaQ{n}), constraint(geom_cells) {
  $Q{n+1} = $Q{n}+ $deltaQ{n} ;
}
```

The specification so far gives us the ability to iterate forward in time, but provides no provisions for terminating the iteration. We must provide the conditions of termination and a "collapse" rule that tells Loci what to do in the event of termination. We compute the condition for termination by detecting when a user provided stop iteration has been reached with the following rule:

```
// Determine when we will finish timestepping
$rule singleton(finishTimestep<-$n,stop_iter) {
  $finishTimestep = ($$n > $stop_iter) ;
}
```

We can now specify how to collapse the iteration. In this case, we collapse to the variable `solution` which is the generic query that we usually make for Loci applications. This query plays the same sort of role as `main` in a C program in that it provides us a standard place to start a Loci program. The collapse rule is simply implemented as described below:

```
// Collapse to solution when we are finished iterating
$rule pointwise(solution<-Q{n}),conditional(finishTimestep{n}),
  constraint(geom_cells) {
  $solution = $Q{n} ;
}
```

4.5 Closing the equations

At this stage, the solution to the time dependent heat equation is not quite in place. While we have described how to compute `deltaQ`, the computation of this variable requires the computation of `qdot`, which in turn requires a definition for temperature. So far we have described how the conservative variable $Q = \rho e$ evolves in time, but we have not related this to temperature. We can do this by using the relationship that $e = C_p T$, thus $T = Q/(\rho C_p)$. We can use this relationship to establish the temperature as a function of `Q` with the rule:

```
// Compute temperature from energy
$rule pointwise(temperature<-Q,Density,Cp), constraint(geom_cells) {
  $temperature = $Q/($Density*$Cp) ;
}
```

In addition to the temperature, we also needed the transformational derivative $\frac{\partial T}{\partial Q}$ in order to compute the system matrix. This is provided by the following rule:

```
// Compute transformation derivative from temperature to Q
$rule singleton(dTdQ<-Density,Cp) {
  $dTdQ = 1./($Cp*$Density) ;
}
```

At this point, we will be able to solve the three-dimensional heat equation in a form that is automatically parallelizable.

4.6 Creating plot files

So far we have developed a solver, but even though the solution is time-dependent, we only get to see the final “solution” that is placed in the fact database, when we would like to view the time evolution of the solution. Loci provides a facility whereby rules can be executed as the time-stepping algorithm proceeds. We can do this by creating a variable that computes a special variable called `OUTPUT`. The variable `OUTPUT` is a special variable that is automatically requested in all iteration loops. If the variables that are input to the rule are being evaluated in the iteration, then this rule will be executed. For example, suppose that we wish to display the steady residual each time-step, then we could implement a rule such as:

```
$rule singleton(OUTPUT<-L2Norm(qresidual),$n),
  option(disable_threading) {
  $[Once] { // output only from one thread/processor
    cout << "R" << $$n << ": " << $L2Norm(qresidual) << endl ;
  }
}
```

Lets note several important artifacts about this rule. First note that the signature indicates that the result of the computation is a variable called `OUTPUT` which indicates that the main purpose of this rule will be to have some effect on the outside world. The rule is inputing the variable `L2Norm(qresidual)` which results in the computation of the volume integrated L2-norm of the steady state residual, `qresidual`. The rule also inputs the current time-step `$n` so that it can report it along with the residual output. Also note the conditional on `Loci::MPI_rank` equal to zero. This conditional statement is there to prevent multiple outputs from each processor when running in parallel. Also note the use of the option `disable_threading`. This option tells Loci that this rule is sensitive to the order that computations occur, and to make sure not to do any optimizations that would cause the computation to be called multiple times.

Now we can see how to print simple output to the screen, but how do we get a chance to make a plot of the simulated temperature field? We can perform a similar operation by using Loci’s built in facility to read/write containers (in parallel or serial) to a portable HDF5 file. An example of such a rule follows:

```
$rule pointwise(OUTPUT<-cell2node(temperature),$n,plot_modulo),
  constraint(pos),conditional(doPlot), option(disable_threading),
  prelude {
  int iter = *$$n % *$plot_modulo ;
  ostringstream oss ;
  string varname = "temperature" ;
  oss << "output/" << varname << "_hdf5." << iter ;
  string filename = oss.str() ;

  $[Once] {
    cout << "writing file " << filename << endl ;
  }

  // Create an hdf5 file
  hid_t file_id = Loci::hdf5CreateFile(filename.c_str(),H5F_ACC_TRUNC,
                                       H5P_DEFAULT,H5P_DEFAULT) ;

  // Write the values of the nodal temperatures into the file
  Loci::writeContainer(file_id,varname,$cell2node(temperature).Rep()) ;

  // Close the hdf5 file
  Loci::hdf5CloseFile(file_id) ;
} ;
```

In this rule we output the special variable `OUTPUT` which indicates that this rule will be executed if the inputs are being computed in the given iteration. The rule inputs the parametric variable `cell2node(temperature)` which will interpolate from the cell values of `temperature` and the boundary face values `temperature_f`. We constrain the rule to apply to all entities that have a value for `pos` (which are the nodes in the mesh), and we add the option `disable_threading` which makes sure that all processors will act on this rule in a synchronized and indivisible manner. An important thing to note is the existence of the `prelude` keyword is indicating that we are describing activities that operate on the containers (usually as a prelude to computations), rather than the values contained within the containers. We need this keyword because we are about to write the *containers* to a file. In the prelude, when we access a particular variable, we are not accessing the contents, but rather the container itself. Thus, to access the value of the parameter `plot_modulo` we use the `*` operator (*e.g.* `*$plot_modulo`. In this particular case, we write the file out into the “output” directory with a name that is compatible with using the “extract” post processing utility. We write the container out using the `writeContainer` function, and note that we use the `.Rep()` method to get the abstract representation of the container to pass to this routine. Finally, note that we have placed a conditional execution argument on this rule (conditional execution statements can be placed on collapse rules that end iterations or rules that output the variable `OUTPUT`). This conditional statement is to allow us to only occasionally write out this file. We provide a separate rule to compute the conditional variable (`doPlot`) as implemented below:

```
// Compute when we want to make plot files
$type doPlot param<bool> ;

$rule singleton(doPlot<-$n,plot_freq) {
  $doPlot = (($$n % $plot_freq) == 0) ;
}
```

4.7 Running the solver

The heat solver described can be used to solve the heat equation on general unstructured grids, and can exploit massively parallel distributed memory clusters. The source code to a complete heat solver is provided in the `heat` directory of the tutorial. A test grid and vars file has been included in the directory for your experimentation. To run the example case use the command

```
mpirun -np 1 ./heat test
```

Note you can also look at the schedule that is generated by Loci using the `--scheduleoutput` option. The –nochomp option disables the chomping optimization which may make the schedule easier to understand. For example to see the schedule generated by Loci run

```
mpirun -np 1 ./heat --scheduleoutput --nochomp test
```

The heat solver will dump plot files into the `output` subdirectory. A post-processing tool called `extract` can be used to convert these files into a format that can be visualized by various visualization packages including *ensight*, *fieldview*, and *tecplot*. For example to see the results of the simulation after 10 iterations using ensight, one enters the command:

```
extract -en test 10 temperature
```

This creates a directory `ensight_test` that contains an ensight case file for visualization of the solution results. The argument `temperature` tells extract that you want to extract the nodal plot variable `temperature` that is stored in the `output directory`. Use `extract -help` for more information.

Chapter 5

Using storeVec and storeMat

Loci provides facilities for having vectors and matricies that have a size that is not known until the program executes. This facility would be used, for example, when the number of chemical species is not known until specified by the user, or other applications where the size of inforation stored at each entity could be specified by user input. For these applications we have the containers `storeVec` and `storeMat`. These containers have a method `setVecSize()` that must be called before values are written to the container. Generally this method will be required for pointwise and unit rules that are responsible for creating `storeVec` or `storeMat` containers. Below is an example of a rule that computes a `storeVec`:

```
$type Ivec storeVec<double> ;
$type numBands param<int> ;
$type Omegas param<vector<double> > ;

$rule pointwise(Ivec{n=0}<-numBands,Omegas),constraint(geom_cells),
  prelude {
  $Ivec{n=0}.setVecSize(*$numBands*(*$Omegas).size()) ;
} compute {
  $Ivec{n=0} = mk_Scalar(0) ;
}
```

Note that this code includes a `prelude` block and a `compute` block. The prelude block operates on the containers and is executed before the compute block. The compute block describes, like most Loci rules, what to do for each entity that satisfies the rule signature. In other words, the `compute` block operates on values, while the `prelude` block operates on the containers.

The `storeVec` container creates array of values for each entity. In general you can assign or use the `+=`, `-=`, `=`, or `/=` operators with these arrays. The `mk_Scalar()` function in the above code converts a scalar value to a vector with the same scalar value repeated for each entity. These operators facilitate a straightforward approach to operating on arrays and matricies. However, it is also perfectly acceptable to operate on the individual vector entries using the array operator, (`[]`), such as demonstrated with a rewrite of the above rule shown below:

```
// An equivalent variation of the above code
$rule pointwise(Ivec{n=0}<-numBands,Omegas),constraint(geom_cells),
  prelude {
  $Ivec{n=0}.setVecSize(*$numBands*(*$Omegas).size()) ;
} compute {
  const int vs = $numBands*$Omegas.size() ;
  for(int i=0;i<vs;++i) // Loop over vector and set each entry to zero
    $Ivec{n=0}[i] = 0 ;
}
```

Chapter 6

Debugging Hints

Debugging techniques for Loci programming is both similar to and different from standard programming methods. First you can use debuggers such as `gdb` on Loci programs, although you may want to edit the Makefile to include a line to enable the debugging option and recompile. The line you will add is given as:

```
COPT = -g
```

In addition, after `Loci::Initialize()` you can call the function `set_fpe_abort()` which will cause floating point exceptions to abort the program. In addition, for parallel programs, you can get Loci to automatically create terminal windows with debuggers running on any processes that aborted if you call the program with the following options:

```
heat --display $DISPLAY --debug gdb test
```

In Loci programs, you frequently need to understand why a rule isn't being included in a schedule, or why a schedule is not being formed. There are several techniques that are useful for figuring out what is going on in the Loci schedule. Probably the most useful file in this regard is the files dumped into the `debug/` directory. These files contain descriptions of rules that were removed from the schedule with an explanation of why (e.g. what information was missing that made it impossible to execute that rule). This is often very useful in determining what is going on. In addition, it is usually helpful to examine the schedule files created when using the option `--scheduleoutput`. Usually this is more useful when combined with the `--nochomp` option that disables some optimizations that may make the schedule hard to read. Usually inspections of the schedule are targeted in that we are looking for a particular rule, when it is executed and over what entities.

In addition to the above techniques, there are several ways that we can use the Loci scheduler to help identify problems. The first is to use constraints. If a rule is being used in a schedule, but not over the entities expected, it can be useful to temporarily add a constraint to force the rule to be executed over a particular set of entitites. For example, if a value should be available for all faces, adding `constraint(area)` will force Loci to tell you what variables kept the rule from satisfying the constraint. However, in general it is a good idea to use constraints in a limited fashion, as in the end a constraint limits the way in which a rule might be used. So, once the debugging is finished it is a good idea to remove any unnecessary constraints.

Another approach is to add dummy rules to shortcut part of the computations. For example, in the heat solver if we are wondering why the advance rule isn't being called, we could add rules that we know can be scheduled to fill in part of the code. For example, we could shortcircuit the `deltaQ` computation with the rule:

```
$rule pointwise(deltaQ<-Q) {
// Dummy rule to check for schedule consistency
}
```

If Loci is able to generate a reasonable schedule, you can then change the inputs to the rule to discover where the computations were having trouble. For example changing the rule to the following would check to see that we were able to derive the variable `heat_B`:

```
$rule pointwise(deltaQ<-heat_B) {
// Dummy rule to check for schedule consistency
}
```

Appendix A

Makefile Example

Compiling Loci programs is easy if you copy the make file from the heat example directory in the tutorial and adapt it to your application. All that is needed is to edit `LOCI_BASE` to point to the directory where Loci is installed, let `OBJS` point to a list of the object files that will be compiled into your final executable. Finally, use the `TARGET` variable to set the executable name. The makefile is given as a reference on the following page.

```
# This is the Loci base directory.
LOCI_BASE ?= /Users/lush/Work/Loci/OBJ
# List your object files here
OBJS  =  main.o residual.o matrix.o euler.o varFileInputs.o plot.o
# List the name of your compiled program here
TARGET = heat

############################################################################
# No changes needed below this line

include $(LOCI_BASE)/Loci.conf

default:
$(MAKE) $(TARGET)

$(TARGET): $(OBJS)
$(LD) -o $(TARGET) $(OBJS) $(LOCAL_LIBS) $(LIBS) $(LDFLAGS)


clean:
rm -fr $(OBJS) $(TARGET)

# Junk files that are created while editing and running cases
JUNK = $(wildcard *~) $(wildcard crash_dump.*)  core debug output $(wildcard .schedule*)
# ".cc" files created from .loci files
LOCI_INTERMEDIATE_FILES = $(subst .loci,.cc, $(wildcard *.loci) )

distclean:
rm -fr $(OBJS) $(TARGET) $(JUNK) $(LOCI_INTERMEDIATE_FILES) $(DEPEND_FILES)

DEPEND_FILES=$(subst .o,.d,$(OBJS))

#include automatically generated dependencies
-include $(DEPEND_FILES)
```

Appendix B

The `fvm` Module Services

B.1 Grid Metrics

`cellcenter`	Cell Centroid
`vol`	Cell Volume
`facecenter`	Face Centroid
`area`	Face Area and Normal
`grid_vol`	Total Grid Volume

B.2 Spatial Gradients

`grads(X)`	Scalar Gradient at Cell
`gradv(X)`	Vector (array) Gradient at Cell
`gradv3d(X)`	3-D Vector Gradient at Cell
`grads_f(X)`	Scalar Gradient at Face
`gradv_f(X)`	Vector (array) Gradient at Face
`gradv3d_f(X)`	3-D Vector Gradient at Face

B.3 Face Extrapolations

`lefts(X)`	Scalar Extrapolation to Face Left Side
`rights(X)`	Scalar Extrapolation to Face Right Side
`leftsP(X,M)`	Bounded Scalar Extrapolation to Face Left Side
`rightsP(X,M)`	Bounded Scalar Extrapolation to Face Right Side
`leftvM(X)`	Mixture Extrapolation to Face Left Side
`rightvM(X)`	Mixture Extrapolation to Face Right Side
`leftv3d(X)`	3-D vector Extrapolation to Face Left Side
`rightv3d(X)`	3-D vector Extrapolation to Face Right Side

B.4 Nodal Interpolations

`cell2node(X)`	interpolate scalar to mesh nodes
`cell2node_v(X)`	interpolate vector to mesh nodes
`cell2node_v3d(X)`	interpolate 3-D vector to mesh nodes
`cell2nodeMax(X)`	mesh nodes get maximum neighbor value
`cell2nodeMin(X)`	mesh nodes get minimum neighbor value
`cell2nodeMaxMag(X)`	mesh nodes get maximum magnitude neighbor value
`cell2nodeMaxv3d(X)`	Maximum neighboring 3d value

B.5 Linear System Solvers

`petscScalarSolve(X)`	solve linear system one dof per cell
`petscBlockedSolve(X)`	solve linear system with many dof per cell (double input)
`petscBlockedSSolve(X)`	solve linear system with many dof per cell (float input)

B.6 Basic Norms

`L1Norm(X)`	Volume Integrated L1-norm
`L2Norm(X)`	Volume Integrated L2-norm
`LinfNorm(X)`	Infinity Norm

Appendix C

Datatypes

C.1 Introduction

For the purpose of I/O and communication, datatypes can be considered as abstract representations of the state of an object. Usually this is represented by the memory locations where data is stored. In the Loci framework, we gave to specify how to save and restore the state of new object types explicitly to facilitate interprocessor communication in heterogeneous environments or portable file I/O. This information is provided to Loci using the traits mechanism.

In Loci, the datatype information is encapsulated in `data_schema_traits` template class. A user will need to provide a specialized template for his/her own datatypes before they can be used with Loci containers such as `store`.

C.2 Classification of Datatypes

Datatypes could be classified according to the their relationship between computer memory representation and data which they hold. The basic distinction depends on the way C++ stores the object in memory. If the object is represented in a continuous segment of memory with a fixed layout, then then the memory layout is consistent with the schemas used to send messages or write data to files. If, instead the objects data is scattered through memory or has a size that is determined at run time, then the object must be serialized before it can be sent as a message or placed in file storage. An object whose memory layout is compatible with the contiguous memory schema uses an `IDENTITY_SCHEMA_CONVERTER` for serialization. Other objects must also specify a schema converter. The specifics of this will be specified in the below examples:

```
typedef struct My_Type_t {
    int       iScalar;
    float     fScalar;
    char      cScalar;
    double    dScalar;
} My_Type1;

typedef struct My_Type_t {
    int        iScalar;
    float      fScalar;:
    char       cScalar;
    double    *dScalar;
} My_Type2;
```

For *My_Type1* C/C++ guarantees that the members of a structure are stored in contiguous memory locations with some padding, so the bitwise copy of *My_Type1* will pack the data correctly, whereas for *My_Type2* the address of *dScalar* and not the data will be copied. In the first case, the memory schema and the file schema is handled by an **Identity Function** called the `IDENTITY_SCHEMA_CONVERTER`

In the second case, we need to explicitly convert between memory schema and the file schema by specifying `USER_DEFINED_CONVERTER`.

It should be mentioned here, that pointers are not the only source of difference between these two datatype. STL container like vector, list, queues, maps and virtual classes, all would need user to serialize the state they contains, therefore, they also fall under the category of user defined converter type.

C.3 Predefined Datatypes in Loci

Loci provides some of the frequently used datatypes so that user need not write for themselves.These are

1. **Atomic Datatype** As the name implies, these datatypes are indivisible types. These type are supported by all machines. These are building blocks for all other datatypes. All native C/C++ datatypes are atomic datatypes in Loci. The following table provides all the atomic datatypes support by the Loci.

Loci Datatype	Native C++ Datatype
BOOL	bool
CHAR	signed char
UNSIGNED_CHAR	unsigned char
SHORT	short
UNSIGNED_SHORT	unsigned short
INT	int
UNSIGNED	unsigned
LONG	long
UNSIGNED_LONG	unsigned long
FLOAT	float
DOUBLE	double

2. **Array class** Loci, provides template version of constant size array

```
template <class T, unsigned int n>
class Array {
   public:
   .
   .
   private:
      T  x[n];
};
```

3. **Vector class** Loci, has 2D/3D vesions of vector class(mathematical vectors)

```
template<T>
struct vector3d {
     T x,y,z ;
}

template<T>
struct vector2d {
     T x,y ;
}
```

4. **STL containers parameterized by types using the** `IDENTITY_SCHEMA_CONVERTER`

   All standard STL containers with predefined identity schema types are supported.

   (a) vector
   (b) list
   (c) queue
   (d) set
   (e) map

C.4 Creating your own compound datatypes

Loci, uses `data_schema_traits` template class to determine the datatype of an object. The generic behavior of template class is not suitable for identifying or creating new datatype, so we use **template Specialization** technique to customize or create new datatypes . This `data_schema_traits` class has one static member function `get_type()` in which user specifies the information about new datatype. A general skeleton for creating new datatype look as follows

```
namespace Loci {
   // Skelton for datatype having identity schema
   template <>
   struct data_schema_traits <My_New_Type1 > {
         typedef IDENTITY_CONVERTER Schema_Converter;
         static DatatypeP get_type() {
                CompoundDatatypeP cmpd = CompoundFactory(My_New_Type1());
                LOCI_INSERT_TYPE(cmpd, My_New_Type1, member);
                   .
                   .
                return DatatypeP(cmpd);
         }
   };
}
```

Where **CompoundFactory** is one of the software design pattern for creating a new compound datatype object. Since the allocation of this object is done inside the functions, the question will always arise, who is responsible for deleting the object ? To destroy the objects when they are not needed, we use reference counting and the `CompoundDatatypeP` class stands for *Reference Counting* version of `CompoundDatatype` class.

If your C/C++ structures contains members of predefined Loci datatypes, then it is fairly easy to create corresponding Loci datatype. For example

```
typedef struct My_Compound_Type_t {
       float                       fScalar;
       vector3d<double>            vect3d;
       Array<double,2>             array1d;
       Array<Array<double,2>,4>    array2d;
} My_Compound_Type;

namespace Loci {
 template <> struct data_schema_traits <My_Compound_Type > {
    typedef IDENTITY_CONVERTER Schema_Converter;
    static DatatypeP get_type() {
        CompoundDatatypeP cmpd = CompoundFactory(My_Compound_Type());
        LOCI_INSERT_TYPE(cmpd, My_Compound_Type, fScalar);
        LOCI_INSERT_TYPE(cmpd, My_Compound_Type, vect3d);
        LOCI_INSERT_TYPE(cmpd, My_Compound_Type, array1d);
        LOCI_INSERT_TYPE(cmpd, My_Compound_Type, array2d);
        return DatatypeP(cmpd);
     }
   };
 }
```

C.5 Creating User Defined Datatype

As explained earlier, whenever the memory allocation in any object is not contiguous, it is the users responsibility to serialize the data contained in the objects. The skeleton of user defined schema type will look as follows

```
  // Skelton for datatype having user defined schema
  template <>
  struct data_schema_traits <My_New_Type2 > {
       typedef USER_DEFINED_CONVERTER Schema_Converter;
       typedef char  Converter_Base_Type;
       typedef MyObject_SchemaConverter   Converter_Type;
  };
```

Note : `Converter_Base_Type` *could be any datatype with identity schema*

Following steps must be taken in order to define your own datatype for Loci

1. Specialize the `data_schema_traits` class
   - Specialize `data_schema_traits` template class with your class
   - declare in `data_schema_traits` class

     `typedef USER_DEFINED_CONVERTER Schema_Converter`
   - specify what datatype will be used in conversion. This should be datatype with identity schema defined which means, that we can use any valid Loci atomic or compound datatype.
   - specify the class which has the responsibility of conversion (Serialize class)
2. Specifying Serialize class
   - Specify object reference in the constructor.
   - `getSize()` member function returns the number of atomic datatypes used in this object.(It is not the size of the object in bytes)
   - `getState()` member function gets the state of an object into a contiguous buffer.
   - `setState()` member function sets the state of an object from a contiguous buffer.
3. Overload input/output stream functions. Both atomic and compound datatypes have already been overloaded with input/output streams in Loci.**It is required that these function are overloaded even if the a user doesn't have intention of using them.**

Now we shall give one simple example to show how things work. We define one structure with STL list inside it. Since list may not have contiguous memory, we define it is defined with user defined schema

```
namespace Loci {
//////////////////////////////////////////////////////////////////////////////////
// This is an example of conventional C/C++ structure
//////////////////////////////////////////////////////////////////////////////////

   struct My_Type {
    list<int>  alist;
    friend ostream& operator << (ostream &, const My_Type &);
    friend istream& operator >> (istream &, My_Type &);
};
```

```
//-------------------------------------------------------------------------------/
class My_Type_SchemaConverter;    // Forward Declaration of class
//-------------------------------------------------------------------------------/
// Specialize the data_schema_traits class with "My_Type" class
//-------------------------------------------------------------------------------/

template <>
struct data_schema_traits<My_Type> {
  // This class has user defined schema
  typedef USER_DEFINED_CONVERTER Schema_Converter ;

  // Since list contains "int" we use it directly for our converion
  typedef int Converter_Base_Type ;

  // Here we specify the class used for serialization/deserialization
  // purpose
  typedef My_Type_SchemaConverter Converter_Type ;
};

//-------------------------------------------------------------------------------/
// Define a class which has the responsibity of serialization and deserialization
// of "My_Type" class
//-------------------------------------------------------------------------------/

class My_Type_SchemaConverter {
  // For the schema converter, we always store a reference to the object
  // we are converting schmata for.
  My_Type &RefObj ;
  public:
      explicit My_Type_SchemaConverter(My_Type &new_obj): RefObj(new_obj) {}
      //
      // This member function returns number of elements of type defined
      // in Converter_Base_Type. It is not the size in bytes.
      //
      int getSize() const {
          return RefObj.alist.size() ;
      }

      // Get the state of an object "RefObj" into an array and also size of
      // array. This is a serialization step.
      void getState(int *buf, int &size) {
           size = getSize() ;
           int ii=0;

           list<int> :: const_iterator ci;
           list<int> :: const_iterator begin = RefObj.alist.begin();
           list<int> :: const_iterator end   = RefObj.alist.end();
           for(ci = begin; ci != end; ++ci)
               buf[ii++] = *ci;
      }
      //
      // From a given array, construct the object. This is "Deserialization Step"
      //
      void setState(int *buf, int size) {
           RefObj.alist.clear();
           list<int> :: iterator ci;
           for(int i=0;i<size;++i)
               RefObj.alist.push_back(buf[i]);
      }
};
}
//--------------------------------------------------------------------------------/
```

C.6 Inner Details about Compound Datatype

Compound datatypes are similar to structures in C/C++. These datatypes are a collection of heterogeneous atomic or fixed sized array datatypes. Every member of these datatype has a unique name within the datatype and they occupy non-overlapping memory locations.

In Loci, this datatype is declared in `CompoundType` class. The corresponding counted pointer class is `CompoundDatatypeP`.

A new member can be inserted into the new datatype in either way

- Using member function of CompoundType class

  insert(member_name, offsetof(type, member-designator), member_datatype);

  where `offsetof` is a standard C/C++ function which provides offset of any member (designated by member-designator) in C/C++ structure (designated by type). *member_datatype* could be any valid Loci datatype.

- Using predefined macro

  LOCI_INSERT_TYPE(compound_object, compound_class, insert_member);

  where *compound_object* is the compound datatype for *compound_class* and *insert_member* is the required member of *compound_class* which is inserted into new datatype.

  In order to use the macro *insert_member* must be a first class object and and its own type should be identified by `data_schema_traits` class.

In the following sections we shall gives some examples of creating different Loci datatypes.

C.6.1 Creating compound datatype with only atomic datatypes

The following is a very simple C/C++ structure, which contains only native datatypes. For this structure, we would like to create Loci datatype, which is also described below.

```
typedef struct My_Compound_Type_t {
       int        iScalar;
       float      fScalar;
       char       cScalar;
       double     dScalar;
} My_Compound_Type;


 // data_scheme_traits should be defined in Loci namespace
namespace Loci {
  // Specialize the class with the new class
  template <>
  struct data_schema_traits <My_Compound_Type > {

       // Specify that ideneity Schema will be used for this class
       typedef IDENTITY_CONVERTER Schema_Converter;

       // define the member function
       static DatatypeP get_type() {
             // Create a new product "cmpd" from the factory pattern
             CompoundDatatypeP cmpd = CompoundFactory(My_Compound_Type());
             // Insert a new member into the new compound datatype
             LOCI_INSERT_TYPE(cmpd, My_Compound_Type, iScalar);
             LOCI_INSERT_TYPE(cmpd, My_Compound_Type, fScalar);
             LOCI_INSERT_TYPE(cmpd, My_Compound_Type, cScalar);
             LOCI_INSERT_TYPE(cmpd, My_Compound_Type, dScalar);
            // return pointer to the base class
            return DatatypeP(cmpd);
        }
     };
 }
```

C.6.2 Creating compound datatype with arrays

In the following example, we have inserted a two dimensional array into the structure and define corresponding Loci datatype.

```
namespace Loci {
    typedef struct My_Compound_Type_t {
        int     iScalar;
        float   fScalar;
        double  dScalar;
        Array<double,10>          dArray1D;
        Array<Array<double,3>,5>  dArray2D;
    } My_Compound_Type;

    template<>
    struct data_schema_traits<My_Compound_Type> {
         typedef IDENTITY_CONVERTER Schema_Converter ;
```

```
          static DatatypeP get_type() {
            CompoundDatatypeP ct = CompoundFactory(My_Compound_Type()) ;
            LOCI_INSERT_TYPE(ct,My_Compound_Type, iScalar) ;
            LOCI_INSERT_TYPE(ct,My_Compound_Type, fScalar) ;
            LOCI_INSERT_TYPE(ct,My_Compound_Type, dScalar) ;
            LOCI_INSERT_TYPE(ct,My_Compound_Type, dArray1D) ;
            LOCI_INSERT_TYPE(ct,My_Compound_Type, dArray2D) ;
            return DatatypeP(ct) ;
          }
      } ;
 }
```

C.6.3 Creating compound datatype with nested compound datatypes

In this example, we would demonstrate that any member with valid datatype could be inserted into compound datatype in an hierarchal fashion. There is no restriction on number of levels used to define a datatype.

```
namespace Loci {
   struct Velocity {
     Array<double,3>  comp;
   };
   // Define traits for "Velocity" structure
   template <>
   struct data_schema_traits<Velocity> {
     typedef IDENTITY_CONVERTER Schema_Converter ;
     static DatatypeP get_type() {
       Velocity v;
       return getLociType(v.comp);
     }
   };
   // A structure contains another structure
   struct CellAttrib {
     int      local_id;
     double   density;
     double   pressure;
     Velocity vel;
   };

   // Define traits for "CellAttrib" class. Notice that "vel" is a structure
   // and since its type is already defined, we can insert it similar to other
   // members.
   template <>
   struct data_schema_traits<CellAttrib> {
     typedef IDENTITY_CONVERTER Schema_Converter ;
     static DatatypeP get_type() {
       CompoundDatatypeP  cmpd = CompoundFactory( CellAttrib() );
       LOCI_INSERT_TYPE(cmpd, CellAttrib, local_id );
```

```
      LOCI_INSERT_TYPE(cmpd, CellAttrib, density  );
      LOCI_INSERT_TYPE(cmpd, CellAttrib, pressure );
      LOCI_INSERT_TYPE(cmpd, CellAttrib, vel );
      return DatatypeP(cmpd);
    }
  };
}
```

C.7 Array Datatype

Array datatype consists of homogeneous collection of both compound and atomic datatypes. We can defined array datatypes for standard C/C++ arrays. In order to create array datatype, a user need to provide

- the rank of the array, i.e. number of dimensions

  Note At present, Loci can support arrays with maximum rank of 4. This limitation comes from HDF5 library. If the user wants higher ranked arrays, using **Array class** *is one solution*

- the size of each dimension
- the datatype of each element of the array

In Loci, this datatype is declared in `ArrayType` class. The corresponding counted pointer class is `ArrayDatatypeP`.

- Create 1 D dimensional array datatype

  ```
  dims[0]= 100;
  ArrayType  atype(Loci::DOUBLE, 1, dims);
  ```

- Create 2 D dimensional array datatype

  ```
  dims[0] = 10;
  dims[1] = 20;
  ArrayType  atype(Loci::DOUBLE, 2, dims);
  ```

- Create 3 D dimensional array datatype

  ```
  dims[0] = 10;
  dims[1] = 20;
  dims[2] = 30;
  ArrayType  atype(Loci::DOUBLE, 3, dims);
  ```

- Create 4 D dimensional array datatype

```
     dims[0] = 10;
     dims[1] = 20;
     dims[2] = 30;
     dims[3] = 40;
     ArrayType  atype(Loci::DOUBLE, 4, dims);
```

For example

```
typedef struct My_Compound_Type_t {
    int       iScalar;
    float     fScalar;
    char      cScalar;
    double    dScalar[10][5][2];
}My_Compound_Type;
```

This datatype has multidimensional array, which is not first class objects. We can use `ArrayDatatype` to specify the Loci Datatype as

```
int     rank  = 3;
int     dim[] = {10, 5, 2};
int     sz    = 100*sizeof(double);

My_Compound_Type     type;
CompoundDatatypeP cmpd    = CompoundFactory(My_Compound_Type());
DatatypeP         atom    = getLociType(type.dScalar[0][0][0]);
ArrayDatatypeP    array_t = ArrayFactory(atom, sz, rank, dim);
cmpd->insert("dScalar", offsetof(My_Compound_Type, dScalar), DatatypeP(array_t));
```

This is definitely cumbersome, Instead of using array in this way, if we had used

```
typedef  Array < double, 10 > Array1D;
typedef  Array < Array1D, 5 > Array2D;
typedef  Array < Array2D, 2 > Array3D;
typedef struct My_Compound_Type_t {
   int             iScalar;
   float           fScalar;
   char            cScalar;
   Array3D         dScalar;
} My_Compound_Type;
```

then we can use `LOCI_INSERT_TYPE` macro to insert `dScalar` into new compound datatype.