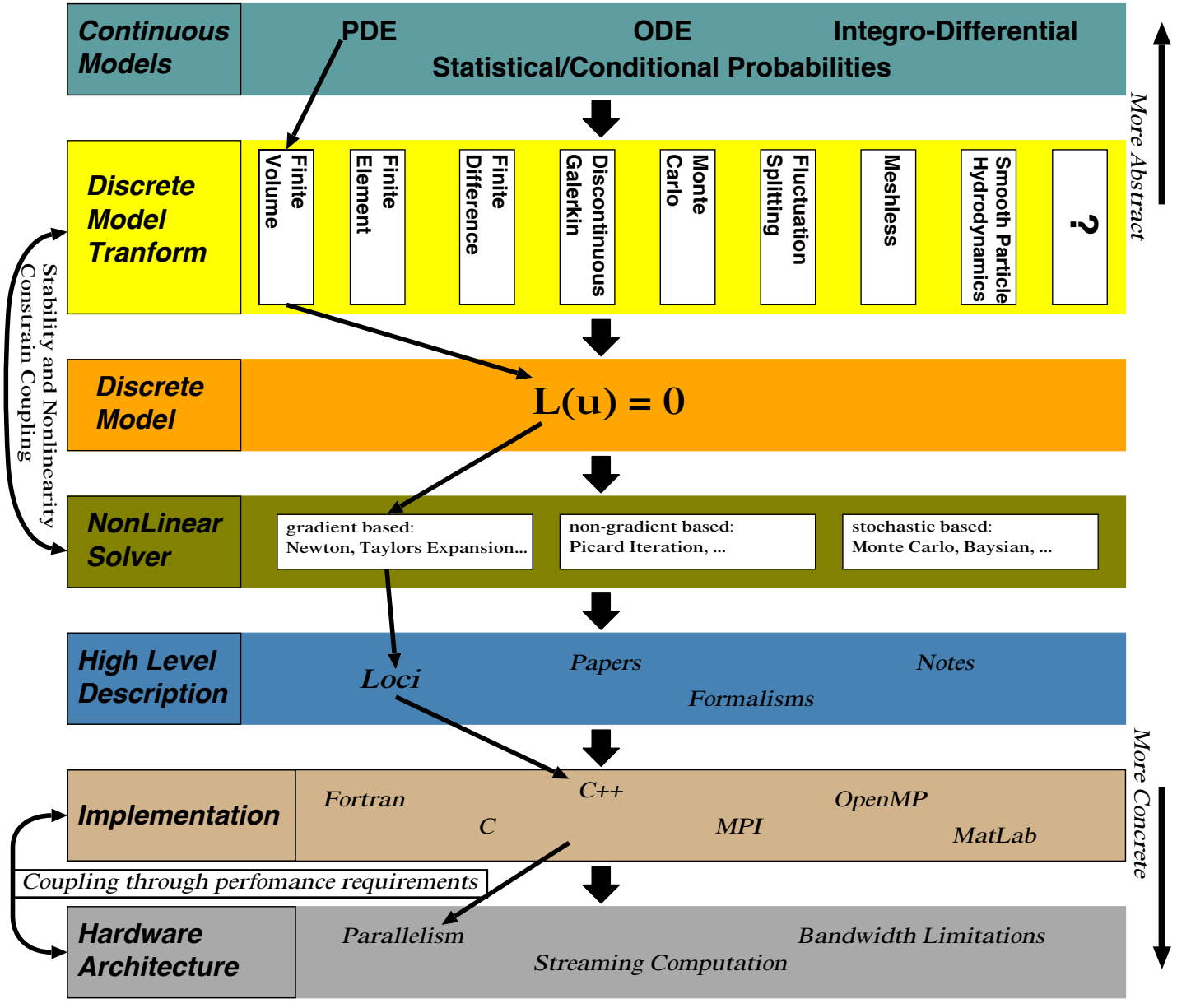
A Tutorial for *Loci*

Edward Luke

January 8, 2023

What is *Loci*?

- *Loci* was originally developed in 1999 as part of National Science Foundation funding supporting the development of advanced multidisciplinary simulation software.
- *Loci* is a sophisticated auto-parallelizing framework that simplifies the task of constructing complex simulation software.
- *Loci* is free software available under the Lesser GNU Public License.
- The *Loci* paradigm is domain specific but powerful and able to capture a wide range of numerical application software.


Continuous Models
PDE
ODE
Integro-Differential
Statistical/Conditional Probabilities
More Abstract
Discrete Model Tranform
Finite Volume
Finite Element
Finite Difference
Discontinuous Galerkin
Monte Carlo
Fluctuation Splitting
Meshless
Smooth Particle Hydrodynamics
?
Stability and Nonlinearity Constrain Coupling
Discrete Model
L(u) = 0
NonLinear Solver
gradient based: Newton, Taylors Expansion...
non-gradient based: Picard Iteration, ...
stochastic based: Monte Carlo, Baysian, ...
High Level Description
Loci
Papers
Notes
Formalisms
Implementation
Fortran
C
C++
MPI
OpenMP
MatLab
More Concrete
Coupling through perfomance requirements
Hardware Architecture
Parallelism
Bandwidth Limitations
Streaming Computation

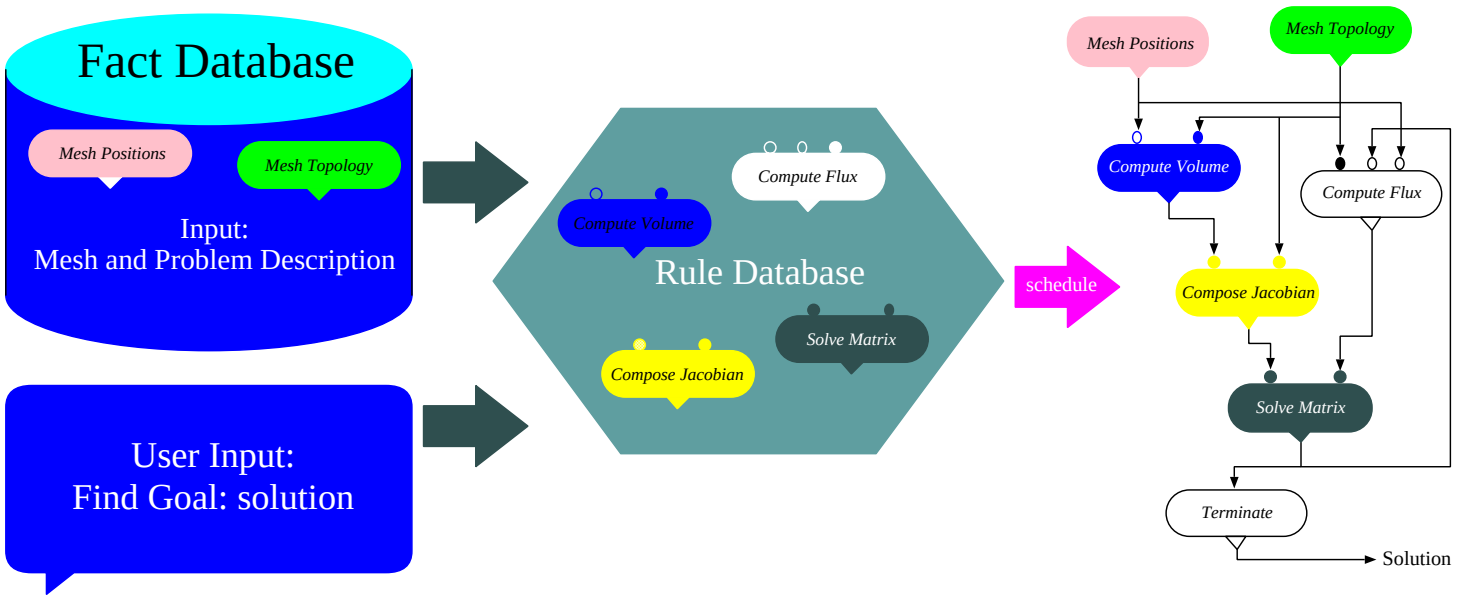
Loci and C++

- The *Loci* framework is built using the C++ language.
- A preprocessor (`lpp`) translates *Loci* code into the native C++ code
- Generally users of *Loci* only need to know a small amount of C++ to be effective
- It is possible to interface to external applications and subroutines such as Fortran, however this is an advanced topic and there can be many limitations particularly when executing in parallel

What is a declarative programming model?

- Most traditional programming models are imperative, that is they are implicitly a list directions that will be performed in a specified order (e.g. How to solve a problem)
- Declarative programming models work by declaring properties of objects without specifying a recipe for solution. (e.g. focusing on describing the components that will solve the problem, but not how they will be used)
- In the declarative approach the assembly of components to solve the problem (the how) is determined by the application of logical inferences from the component specification.
- Getting used to thinking about problems in a declarative way is the main learning curve for *Loci* programming.

Declarative Programming in Loci



Loci programming preliminaries

- Most *Loci* programs will include “Loci.h”
- All Loci programs will need to be initialized and finalized
- When using MPI, calls to MPI initialization/finalization are not needed

Loci Initialization Code

```
#include <Loci.h>

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

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

   // Call finalize for Loci clean up.
   Loci::Finalize() ;
   return 0 ;
}
```

Entities, Sets, and Sequences

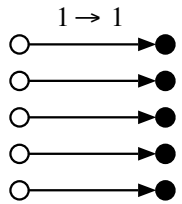
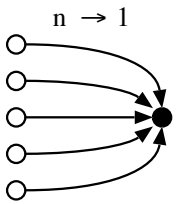
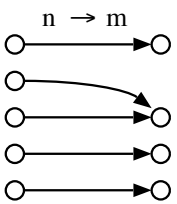
- `Entities` are an important concept in *Loci*
- `Entities` are what gives an object an identity in *Loci*
- All entities have a unique identifier and can be used to see which object we are addressing
- Groups of entities can be represented efficiently as a set called an `entitySet`
- A group of consecutively numbered entities can be represented in a compact form as an `interval`
- A ordered sequence of entities is called a `sequence`

Entity Sets

- `entitySet` stores sets in a compressed form as a sorted sequence of non-overlapping intervals
- Thus the set $A = 1, 2, 3, 5, 6, 7, 8, 9, 10, 100$ will be represented in an `entitySet` as `([1,3],[5,10],[100,100])`.
- The UNIVERSAL set which includes all possible entities is represented with the special notation `([#,#])`.
- The EMPTY set is represented as `()`
- Set operations supported are Union (`+`), Intersection (`&`), Difference (`-`), Complement (`~`) *(show EXAMPLE)*

Loci Containers

- *Loci* provides methods for associating values with entities or associating entities with other entities.
- The data types that perform this association are called containers.
- Four main types of containers types include `store`, `parameter`, `map`, and `constraint`

store relates indices to values	**parameter** relates many indices to a single value	**index map** relates indices to indices	**constraint** specifies a subset of indices
			

Container Themes

- `store<T>` : associates value type `T` with entities
- `storeVec<T>`: associates n value types T with entities
- `multiStore<T>`: associates variable number of types with entities
- `Map`: associates 1 entity per entity
- `mapVec<n>`: associates n entities per entity
- `multiMap` : associates variable number of entities per entity

Container Examples

```
// We create a store of floats
store<float> x ;
// We create a store of std::vectors
store<std::vector<float> > particles ;
// allocate stores x and particles
entitySet alloc_set = interval(1,100) ;
x.allocate(alloc_set) ;
particles.allocate(alloc_set) ;
// initialize the container to the value zero
for( int i=0;i<101;++i) {
  x[i] = 0 ;
  particles[i].push_back(0) ;
}
```

Parameter 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) ;
```

Constraint Example

```
// set inflow constraint
constraint inflow ;
*inflow = entitySet(interval(1,3)) ;
constraint viscous ;
*viscous = EMPTY ; // default not set
if(mu_set) // if viscous set to
  *viscous = ~EMPTY ; // UNIVERSE
```

The Fact Database

- The fact database is a repository for containers in the *Loci* Framework
- The fact database is used to define the initial facts that define the problem setup
- It is defined by the `fact_db` data type in *Loci*
- Containers are added to the `fact_db` using the `create_fact` member function
- Containers can be retrieved from the `fact_db` using the `get_fact` member function.
- The fact database does a “shallow copy” of the containers. E.g. the reference to the container (called a `storeRep`) is what is actually stored.

What Are Rules?

- Rules are ways to express how one set of facts can be transformed into another set of facts
- Rules come in several forms:
  - `default` Default rules are used to define parameters that can be redefined in the vars file (text version of the fact database)
  - `optional` Optional rules tell *Loci* about the type of data that may be placed in the vars file. Since they do not have a default value their existence implies entry in the vars file.
  - `pointwise` A point by point application entity by entity
  - `singleton` Used to perform computations on the single values of parameters
  - `unit` and `apply` are used to form reductions

The Rule Database

- In *Loci* rules are used to define transformations from on set of values to another.
- Users develop applications in *Loci* by defining transformation rules (Much more on this later!)
- The rule database is used to create combined sets of rules that you wish to use to solve your problem.
- When rules are created in Loci they are automatically added to a list of rules to be processed. This list is called the `global_rule_list`.
- Rules can be added to the rule database by using the `add_rules` member function. Typically this will look like `rdb.add_rules(global_rule_list) ;`

The Query

- In *Loci* applications are developed through making queries to the fact database using a prescribed set of rules.
- The application that is created as a result of the query depends on the data provide in the fact database (also called the *extensive* facts), the provided transformations, and the query.
- The schedule is generated through a process of generating derived facts (*intensive* facts)
- A schedule is generated by using the *makeQuery* call:

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

Loci Helper Classes

- Loci provides helper classes that can simplify program development
- Helper classes include:
  - `Array<T,n>`
    Provides proper semantics for arrays suitable for storing in Loci containers. *Do not put C++ arrays in containers!*
  - `vector3d<T>`
    Provides operators for addition, scalar multiplication, dot and cross products
  - `vector2d<T>`
    Provides operators for addition, scalar multiplication, dot and cross products
- *(Go through example)*

The `options_list` class

- For many solvers inputs may be complex and hierarchical.
- The `options_list` class is provided to help standardize the input of this sort of data.
- It is used in most *Loci* solvers for inputting boundary condition data.
- The general form is a list of assignments of values to named terms called options.
- In general the value assigned to a name may be a real number, a real number with units, a double, a string, a name, a list, or a function.
- Lists or functions may be viewed as a nested options list making the input method very powerful.

`options_list` member functions

- `optionExists`: Returns a true value of the provided name is in the list of attributes
- `getOptionNameList`: Returns a list of attributes that have definitions
- `getOptionValueType`: Returns the type of the data that was assigned to the attribute. This may be `REAL`, `NAME`, `FUNCTION`, `LIST`, `STRING`, `BOOLEAN`, or `UNIT_VALUE`.
- `getOption`: Returns the value associated with the attribute. The second argument is the returned value and may be the types `bool`, `double`, `string`, or `options_list::arg_list`.
- `getOptionUnits`: This returns a double value in the requested units.

A Simple Example: 1-D diffusion

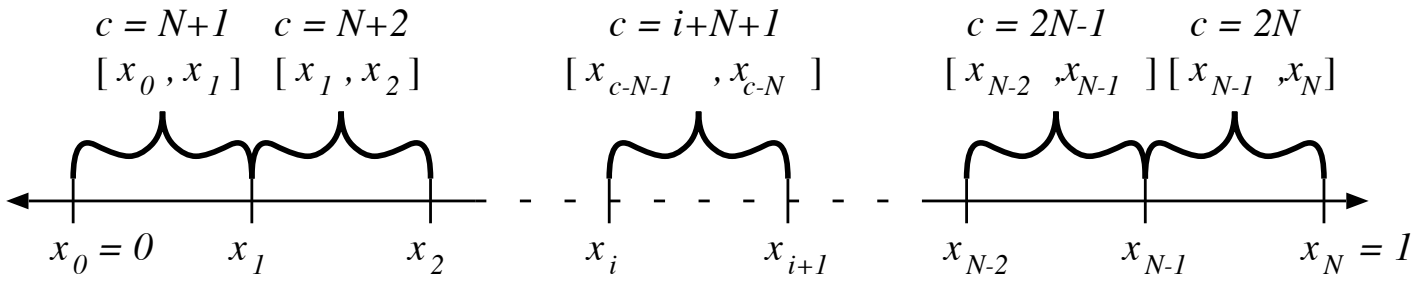
Consider the finite volume method solution to this simple one dimensional diffusion equation:

$$
\begin{aligned}
u_t &= \nu u_{xx}, \ x \in (0,1), t > 0, \\
u(x,0) &= f(x), \ x \in [0,1], \\
u_x(0,t) &= g(t), \ \text{where } g(0) = f_x(0), \ \text{and} \\
u(1,t) &= h(t), \ \text{where } h(0) = f(1).
\end{aligned}
$$

Finite Volume Discretization

- Divide the interval $[0, 1]$ into $N - 1$ cells by defining N nodes such that $x = \{(i, x_i) | i \in [0, \cdots, N], x_i = i/N\}$
- Cells are defined by their interfaces to the left and right:

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



Indirection Operators

- To implement the finite volume scheme we will need to be able to access values at interfaces, this will be done through composition
- For example, to access the left and right nodes of a given cell we could compose the interface maps with the x coordinates with the composition operator:

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

- Using this operator we can now define other attributes that will be needed to perform numerical integration such as the cell center:

$$x_c = (ir \to x + il \to x)/2.$$

Numerical Integration

- Using a midpoint rule to numerically integrated in space and a first order explicit Euler integration for time, the numerical solution to the 1-D diffusion equation can be written as:

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

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

Summary of Definitions for 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	$\{(i, r) \mid i \in [0, \cdots, N-1], r = i + N + 1\}$

Summary of Transformation Rules

Rule	Rule Signature	Equation
Rule 1	$x_c \leftarrow (ir, il) \rightarrow x$	(3.8)
Rule 2	$L \leftarrow (ir, il) \rightarrow x$	(3.11)
Rule 3	$u_x \leftarrow (cr, cl) \rightarrow (u, x_c)$	(3.13)
Rule 4	$u_x \leftarrow h, t, \text{constraint}\{\text{dom}(cl) \wedge \neg\text{dom}(cr)\}$	(3.14)
Rule 5	$u_x \leftarrow g, t, \text{constraint}\{\text{dom}(cr) \wedge \neg\text{dom}(cl)\}$	(3.15)
Rule 6	$R \leftarrow \nu, L, (ir, il) \rightarrow u_x$	(3.16)
Rule 7	$u^{n+1} \leftarrow u^n, R^n, \Delta t$	(3.17)
Rule 8	$u^{n=0} \leftarrow f, x_c, \text{constraint}\{(il, ir) \rightarrow x\}$	(3.19)

Setting up the fact database

- First we create the maps and install them in the fact database (this is the 1-D mesh)
  Go through example online
- Then we can setup default parameters as a *Loci* program:

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

$rule default(N) {  $N=50 ;}
$rule default(nu) {  $nu = 1.0 ;}
```

Writing the Rules

Most of the rules translate directly into Loci rules:

```
// Rule 1: compute the cell center from
//         node positions
$rule pointwise(xc<-(il,ir)->x) {
  $xc = .5*($il->$x + $ir->$x) ;
}
// Neuman boundary condition at left boundary,
// ux = h(t)
$rule pointwise(ux<-h), constraint(left_boundary) {
  $ux = $h ;
}
```

Temporal Integration

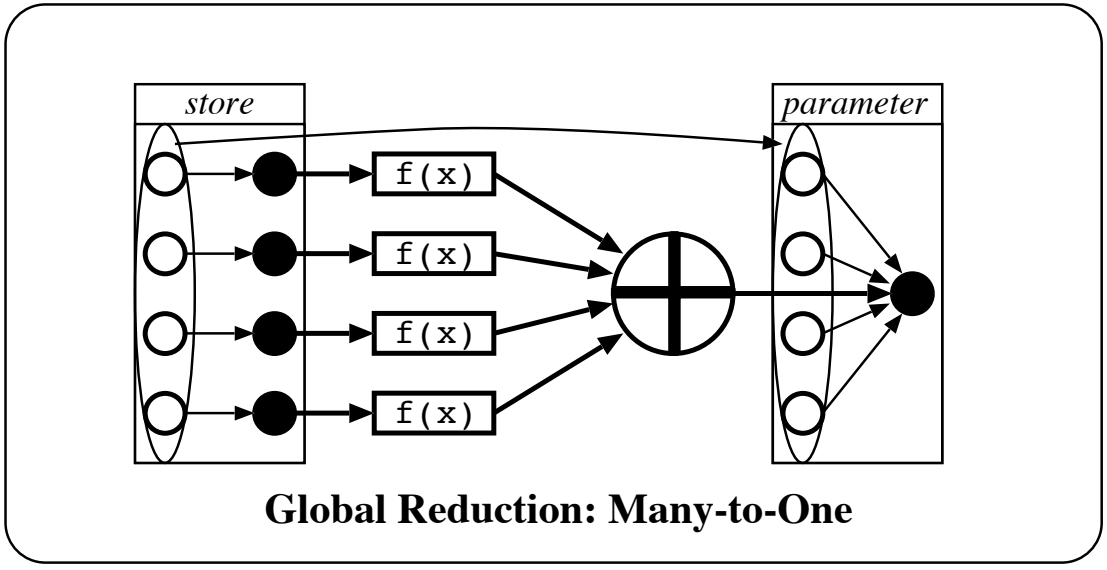
The temporal iteration is then specified

```
// 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} ;
}
```

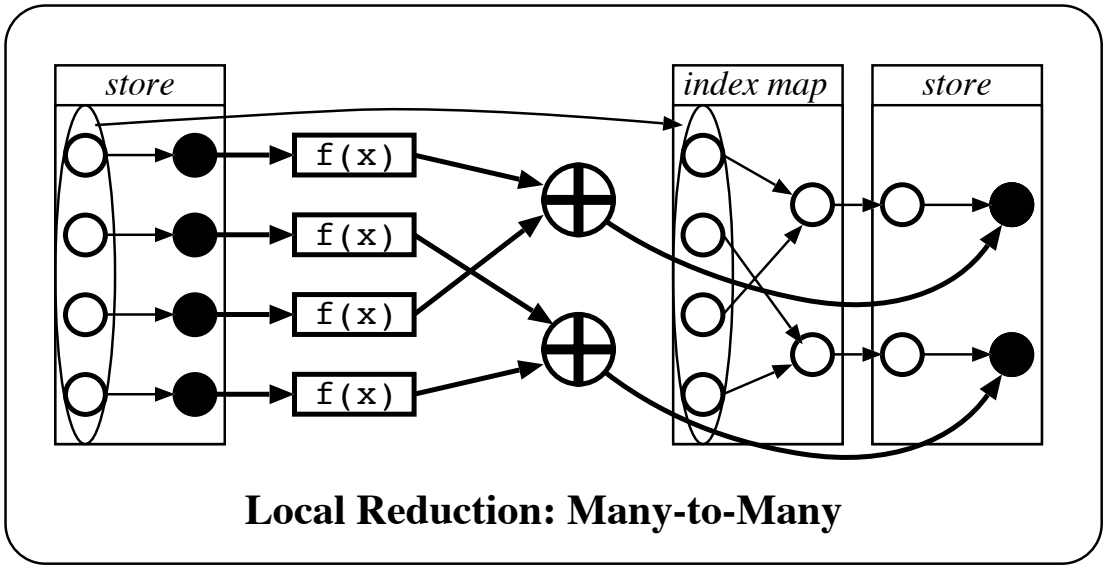
Terminating The Iteration

```
$rule pointwise(solution<-u{n}),
      conditional(simulation_finished{n}) {
  $solution = $u{n} ;
}
$type max_iteration param<int> ;
$type simulation_finished param<bool> ;

// When is  iteration is complete?
$rule singleton(simulation_finished<-
                                $n,max_iteration) {
   $simulation_finished = ($$n >= $max_iteration) ;
}
```

Reduction Rules



Global Reduction: Many-to-One



Local Reduction: Many-to-Many

Components of a reduction

- An operator that is associative (and commutative)
- A part that initializes values to the identity of the operator (the unit)
- A part the produces values that will be combined using the operator (the apply)

Stable Timestep Using Global Reductions

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

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

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

Some Pitfalls

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

More Pitfalls

```
$rule unit(sum), constraint(UNIVERSE){
   // Error, not identity of summation!
   $sum = 1.0 ;
}
$rule apply(sum<-terms)[Loci::Summation] {
   join($sum,$terms) ;
}
```

Going to the Implementation

- After assembling the rules and facts we can see what kind of application Loci assembles
- *Loci* provides options that allow you to inspect what it has done. To see what type of program it will generate enter:
  `./heat --scheduleoutput --nochomp`
- *Loci* will also perform different operations depending on what you query. The default query is “solution” but we can also get other schedules by querying other variables:
  `./heat --scheduleoutput --nochomp -q dt`

Run and Inspect Example Code

Loci Reduction Alternative

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

Convert to use cl and cr maps instead:

```
$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) ; }
```

Note: We cannot combine two apply rules into:

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

Parametric Rules

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

// The 1d diffusion residue
$rule pointwise(R<-nu,cellIntegrate(ux),L)
   { $R = $nu*$cellIntegrate(ux)/$L ;}
// We find the length of an interval by integrating the position x
$rule pointwise(L<-cellIntegrate(x)) { $L = $cellIntegrate(x) ; }
```

Parametric unit/apply rules

```
// 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) ;
}
```

Parametric Time Iteration

```
// X is the residual, Y is the independent variable
$type EulerIntegrate(X,Y) store<float> ;
$type X store<float> ;
$type Y store<float> ;
$type Y_ic store<float> ;
// Initialize the iteration using the initial conditions
$rule pointwise(EulerIntegrate(X,Y){n=0}<-Y_ic)
 {  $EulerIntegrate(X,Y){n=0} = $Y_ic ; }

// 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) ; }
```

Parametric Time Iteration

```
// 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} ;
}

// Extract independent variableb for residual function
$rule pointwise(Y<-EulerIntegrate(X,Y)),
     parametric(EulerIntegrate(X,Y)) {
  $Y = $EulerIntegrate(X,Y) ;
}
```

Note the use of the parametric keyword in last rule!

Euler Integration

```
// 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) ;
}
```

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 s
    } // 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])
```

A Three Dimensional Solver

- Next example is an implicit three dimensional heat solver
- Solves the equation $\frac{\partial}{\partial t}(\rho e) = \nabla \cdot (k \nabla T)$
- Using standard FVM methods this becomes the discrete equation:

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

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

The Implicit Formulation

- The residual can be linearized using Taylor's theorem:

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

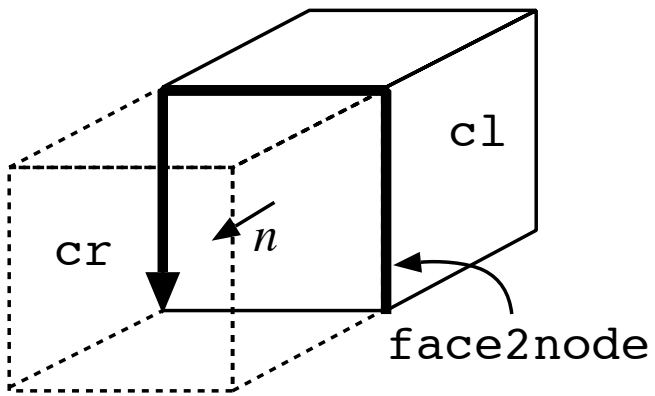
- which can then be used to form the following implicit form:

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

- For this example we will be using the FVM facilities provided for Loci including mesh readers and a module of operators such as gradients.

Provided Data Structures

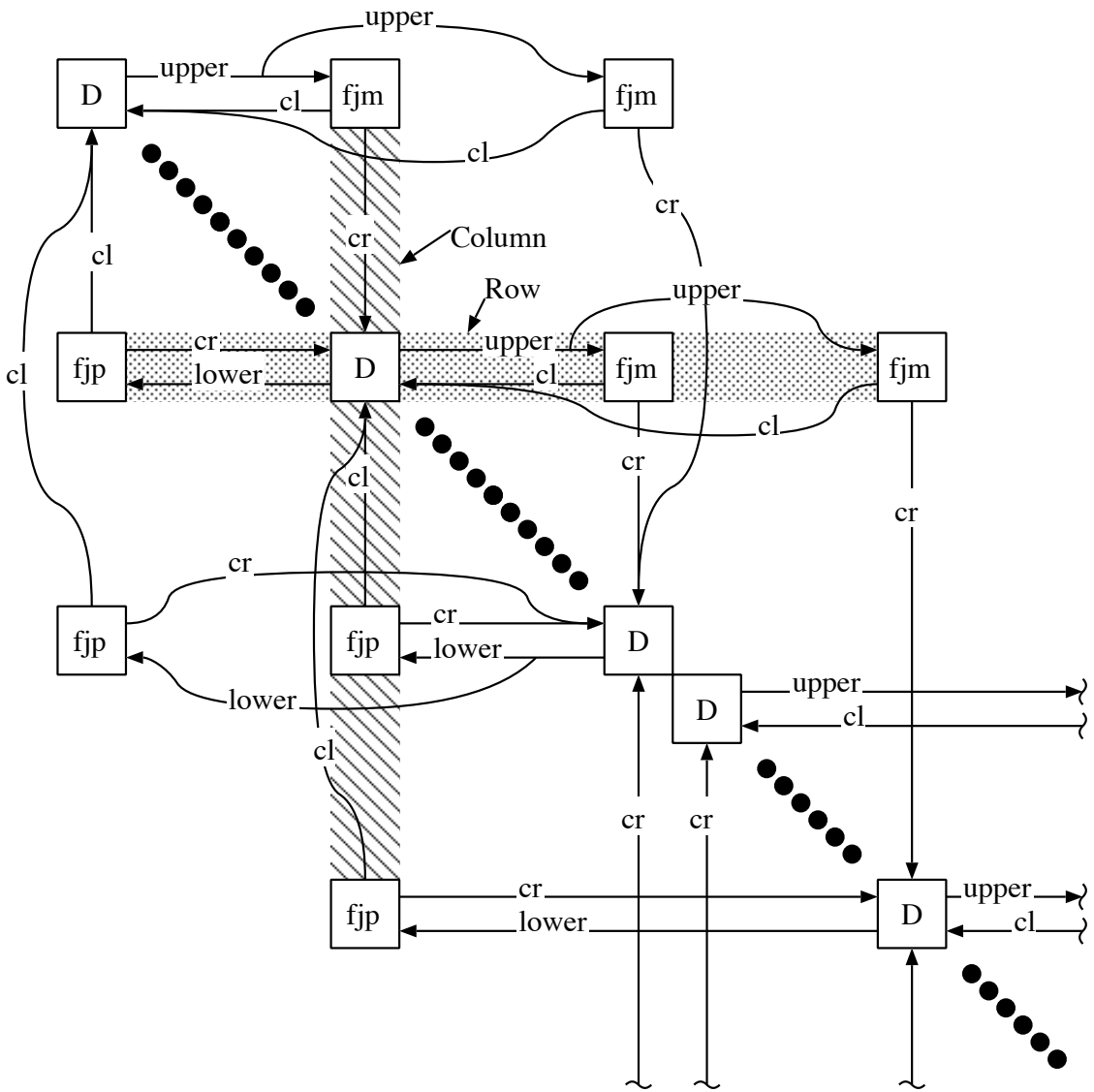
Fact	Type	Location	Description
`pos`	`store<vector3d>`	nodes	Node Positions
`face2node`	`multiMap`	faces	Nodes that form a face
`cl`	`Map`	faces	cell left of face
`cr`	`Map`	faces	cell right of face
`ref`	`Map`	boundary faces	map to referring category
`boundary_names`	`store<string>`	boundary categories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



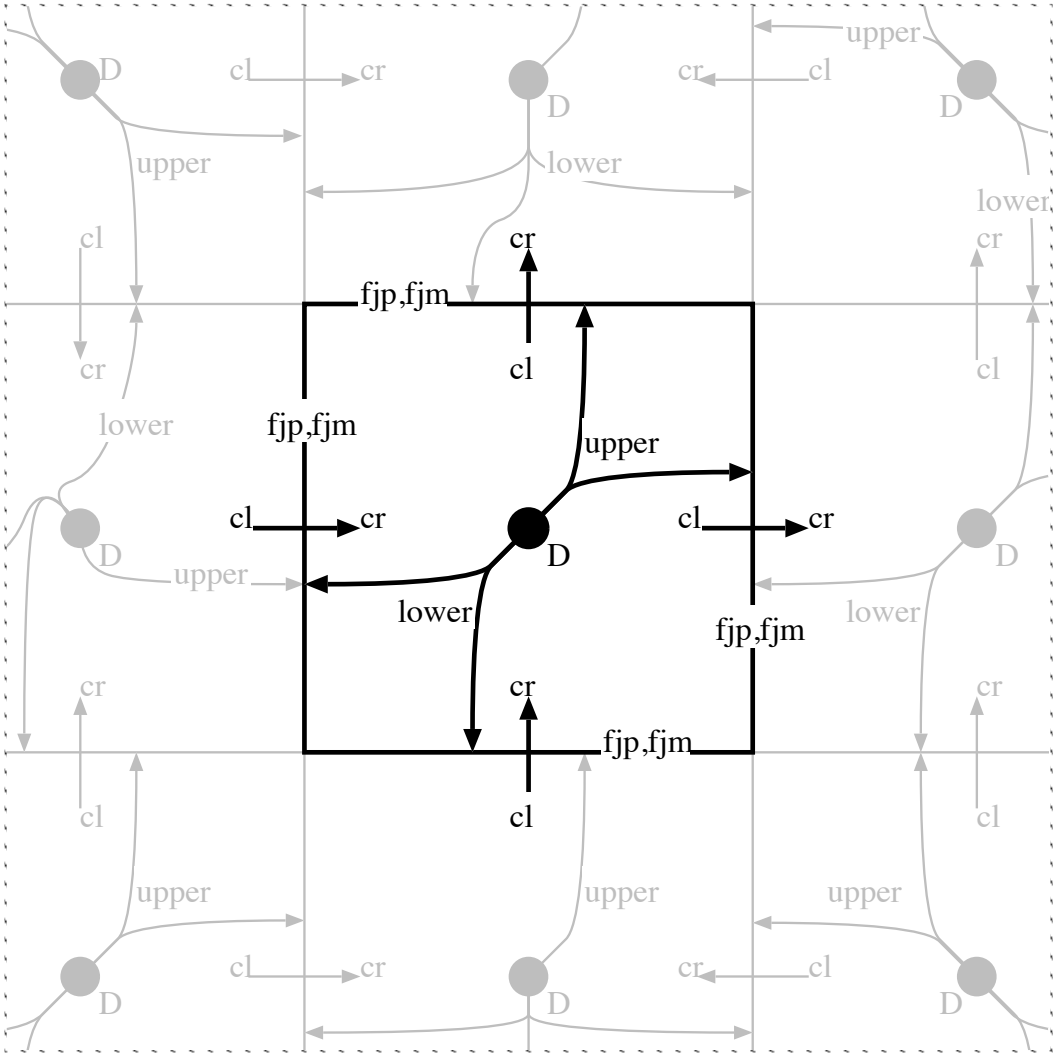
database setup

```
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) ;

// 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() ;
}
// Deconstruct boundary_conditions variable
setupBoundaryConditions(facts) ;
// Setup Matrix
createLowerUpper(facts) ;
```

Matrix Setup



Matrix Cell View



Boundary Condition Rule Setup

```
// Extract Twall from boundary condition options
$rule pointwise(Twall<-BC_options),constraint(Twall_BCoption) {
  $BC_options.getOptionUnits("Twall","kelvin",$Twall) ;
}
// Temperature at wall set to specified condition
$rule pointwise(temperature_f<-ref->Twall),constraint(specified_BC) {
  $temperature_f = $ref->$Twall ;
}
// Handle Boundary Conditions
// Adiabatic Wall, qdot = 0, grad(temperature) = 0
$rule pointwise(adiabatic::qdot),constraint(adiabatic_BC) {
  $qdot = 0 ;
}
```

Residual Evaluation

```
// Compute the heat flux through faces
$rule pointwise(qdot<-conductivity,grads_f(temperature),area) {
  $qdot = $area.sada*$conductivity*dot($grads_f(temperature),$area.n)
}
// Add up contributions from all faces, only define qresidual
$rule unit(qresidual),constraint(geom_cells) {
  $qresidual = 0 ;
}
// 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 to cell
$rule apply(cr->qresidual<-qdot)[Loci::Summation],
  constraint(cr->geom_cells) {
  join($cr->$qresidual,-$qdot) ;
}
```

Residual Evaluation: Missing Part

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

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

Matrix Preliminaries, derivatives

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

and

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

```
// 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 ;
}
```

Matrix Assembly

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

Matrix Assembly

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

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

Time Integration

```
// Initial Conditions
$rule pointwise(Q{n=0}<-Density,Cp,T_initial) {
  $Q{n=0} = $Density*$Cp*$T_initial ;
}
// 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} ;
}
// Determine when we will finish timestepping
$rule singleton(finishTimestep<-$n,stop_iter) {
  $finishTimestep = ($$n > $stop_iter) ;
}
// Collapse to solution when we are finished iterating
$rule pointwise(solution<-Q{n}),conditional(finishTimestep{n}),
  constraint(geom_cells) {
  $solution = $Q{n} ;
}
```

Closing the Equations

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

Running the case

Run the case!