The IP\textsuperscript{2}L Framework
(Independent Parallel Particle Layer)
Version 1.0 \textsuperscript{1}
User’s Reference Manual

Andreas Adelmann, Yves Ineichen

Abstract

IP\textsuperscript{2}L (Independent Parallel Particle Layer) is an object-oriented framework for particle based applications in computational science requiring high-performance parallel computers. One of IP\textsuperscript{2}L ’s most attractive features is its high performance on both single-processor and distributed-memory multicomputers machines. As future releases of the library will also support shared-memory multicomputers, IP\textsuperscript{2}L ’s authors have had to think very carefully about how to obtain the best possible performance across a wide range of applications on different architectures. IP\textsuperscript{2}L is a library of C++ classes designed to represent common abstractions in applications where particles, fields and operators like FFT’s are needed.

Application programmers use and derive from these classes, which present a data-parallel programming interface at the highest abstraction layer.

Lower, to the user (programmer) hidden implementation layers encapsulate distribution and communication of data among processors. The supported platforms are: Linux based Beowulf clusters, Cray XT5/6, SGI Ultrix and the IBM Blue Gene series.

The main goals of the IP\textsuperscript{2}L framework includes:

\textsuperscript{1}Release Date: July 3, 2012
• Portability across serial, distributed, and parallel architectures with no change to source code
• Development of reusable, cross-problem-domain components to enable rapid application development
• High efficiency for kernels and components relevant to scientific simulation
• Framework design and development driven by applications from a diverse set of scientific problem domains
• Shorter time from problem inception to working parallel simulations

IP²L is currently in development the version here is the first "developer release". IP²L is inspired and partially based on POOMA r1 designed and implemented by scientists working at the Los Alamos National Laboratory’s Advanced Computing Laboratory.

The report is organized as follow: in chapter 1 an introduction based on examples including installation instructions is presented. Chapter 2 and 3 are describing support classes followed by an discussion on 3D parallel FFT’s in IP²L. The next two chapters explaining the use of particles and fields. Appendix A - F describing design and implementation details of the most important classes in the framework.
# Contents

1 Introduction 7
1.1 Example 1 Laplace solver using Jacobi iteration 7
1.2 Example 2 Power Spectrum 8
1.3 Example 3 Particle in Cell Code (PIC) 9
   1.3.1 The ChargedParticles Class 10
   1.3.2 The main 10
   1.3.3 initFields 12
   1.3.4 myUpdate 12
   1.3.5 gather 13
1.4 Example 4 A Particle Particle - Particle Mesh (P3M) Solver 13
1.5 Installation 16
   1.5.1 Building IP2L 16
   1.5.2 Used Compilers and Libraries 16
1.6 Acknowledgements 18
   1.6.1 Citation 18

2 Framework Setup 19
2.1 Initialising IP2L 19
2.2 Utility Classes in IP2L 19
   2.2.1 Inform Class 19
   2.2.2 Timer Class 21
   2.2.3 Memory Footprint Class 22

3 FFT 23
   3.0.1 Improving FFT Performance 25

4 Particles 27
   4.1 Basic Particle Object Characteristics 27
   4.2 Defining a User-Specified Particle Class 28
      4.2.1 Selecting a Layout: ParticleLayout and Derived Classes 29
      4.2.2 The ParticleUniformLayout Class 29
List of Tables

1.1 Supported Architectures and needed Libraries . . . . . . . . . . . 17
List of Figures

1.1 Decomposition of the Green function. .......................... 14
1.2 Error and time scaling of the \( P^3M \) solver for different interaction radii. Measured for a uniform spherical distribution of \( 10^4 \) particles on 4 processors. .......................... 15
Chapter 1

Introduction

One of IP^2L’s most attractive features is its high performance on both single-processor and distributed-memory multiprocessor machines. As future releases of the library will also support shared-memory machines.

The heart of the problem IP^2L’s authors face is that while data-parallel programming is a natural way to express many scientific and numerical algorithms, straightforward implementations of it do exactly the wrong thing on modern architectures, whose performance depends critically on the re-use of data loaded into cache. If a program evaluates A+B+C for three arrays A, B, and C by adding A to B, then adding C to that calculation’s result, performance suffers both because of the overhead of executing two loops instead of one, but also (and more importantly) because every value in the temporary array that stores the result of A+B has to be accessed twice: once to write it, and once to read it back in. As soon as this array is too large to fit into cache, the program’s performance will drop dramatically.

1.1 Example 1 Laplace solver using Jacobi iteration

Code Listing

```c
#include "Ippl.h"
int main(int argc, char *argv[])
{
    Ippl ippl(argc,argv);
    Inform msg(argv[0]);
    const unsigned N=8;
    const unsigned Dim=2;

    Index IGLOBAL(N); // Specify the global domain
    Index JGLOBAL(N);

    Index I(1, N-1); // Specify the interior domain
    Index J(1, N-1);
    FieldLayout<Dim> layout(IGLOBAL,JGLOBAL);
}"
```
GuardCellSizes<Dim> gc(1);

typedef UniformCartesian<Dim> Mesh;

Field<double,Dim,Mesh> A(layout,gc);
Field<double,Dim,Mesh> b(layout,gc);

assign(A,0.0); // Assign initial conditions
assign(b,0.0);

b[N/2][N/2] = -1.0; // put a spike on the RHS

double fact = 0.25;

// Iterate 200 times
for (int i=0; i<200; ++i) {
    assign(A[I][J],fact*(A[I+1][J] +
                        A[I-1][J] +
                        A[I][J+1] +
                        A[I][J-1] - b[I][J]));
}

msg << A << endl;
return 0;

The syntax is very similar to that of Fortran 90: a single assignment fills an entire array with a scalar value, subscripts express ranges as well as single points, and so on. In fact, the combination of C++ and IP2L provides so many of the features of Fortran 90 that one might well ask whether it wouldn’t better to just use the latter language straight up. One answer comes down to economics. While the various flavors of Fortran are still used in scientific computing, Fortran’s user base is shrinking, particularly in comparison to C++. Networking, graphics, database access, and operating system interfaces are available in C++ programmers long before they’re available in Fortran (if they become available at all). What’s more, support tools such as debuggers and memory inspectors are primarily targeted at C++ developers, as are hundreds of books, journal articles, and web sites.

Another answer is that the abstraction facilities of C++ are much more powerful that those in Fortran. While Fortran 90 supports an attractive array syntax for floating point arrays one could not, for example, efficiently extend this high level syntax to arrays of vectors or tensors. Until recently, Fortran has had two powerful arguments in its favor: legacy applications, and performance. However, the importance of the former is diminishing as the invention of new algorithms force programmers to rewrite old codes, while the invention of techniques such as expression templates has made it possible for C++ programs to match, or exceed, the performance of highly-optimized Fortran 77.

1.2 Example 2 Power Spectrum

A sinussoidal field $\rho(i,j,k) = a_1 \sin(k_1 \frac{2\pi}{n_x} i) + a_5 \sin(k_5 \frac{2\pi}{n_z} i), i = 1 \ldots n_x, j = 1 \ldots n_y, k = 1 \ldots n_z$ with $n_x, n_y$ and $n_z$ denoting the grid size is generated and the
power spectrum calculated. This examples shows how to initialise fields, compute
discrete complex-complex FFT and compute the resulting powerspectrum.

Assume a real density field is defined like

```c
typedef Field<double,Dim,Mesh_t,Center_t> Field_t;
Field_t rho;
```

we then can immediately initialize the field according to the above formula

```c
assign(rho[I][J][K], a1*sin(2.0*pi/nr_m[0]*k1*I) +
       a5*sin(2.0*pi/nr_m[0]*k5*I));
```

Normalizing to \( \max(\rho) \leq 1.0 \) with

```c
rho /= max(rho)
```

we then assume to have defined a complex field ”fC” and a complex-complex
FFT.

```c
fC = rho;
fft->transform("forward", fC);
```

Here we used the in place version of the FFT to obtain \( \rho \) in Fourier space. Now
we can compute the power spectrum:

```c
pwrSpec = real(fC*conj(fC));
```

and calculate the 1D pwr-spectrum (in x direction) by integrating over y and z:

```c
NDIndex<3> elem;
for (int i=lDomain[0].min(); i<=(lDomain[0].max()-1)/2; ++i) {
   elem[0]=Index(i,i);
   for (int j=lDomain[1].min(); j<=(lDomain[1].max()-1)/2; ++j) {
      elem[1]=Index(j,j);
      for (int k=lDomain[2].min(); k<=(lDomain[2].max()-1)/2; ++k) {
         elem[2]=Index(k,k);
         f1D[i] += pwrSpec.localElement(elem);
      }
   }
}
```

The power spectra of the local domain is stored in \( f1 \). We have to update all other
node so that each node has the full power spectrum by:

```c
reduce(&f1[0],&f1[0]+f1_lenght,OpAddAssign());
```

assuming the non local part of \( f1 \) is initialized with zero.

### 1.3 Example 3 Particle in Cell Code (PIC)

This example discusses how to write a 3D Particle in Cell Code (PIC). The complete
source file can be found at `IPPL_ROOT/test/particles`. The this presentation
details are omitted, only the structure and important issues are highlighted.
1.3.1 The ChargedParticles Class

The base class ParticleBase is augmented with attributes such as charge to mass ratio \( qm \), the vector momenta \( \mathbf{P} \) and the vector holding the electric field \( \mathbf{E} \).

**Code Listing**

```cpp
ChargedParticles(PL* pl, Vector_t hr, Vector_t rmin,
    Vector_t rmax, e_dim_tag decomp[Dim]) :
    ParticleBase<PL>(pl),
    hr_m(hr),
    rmin_m(rmin),
    rmax_m(rmax),
    fieldNotInitialized_m(true)
{
    this->addAttribute(qm);
    this->addAttribute(P);
    this->addAttribute(E);
    for(int i=0; i < 2*Dim; i++) {
        this->getBConds()[i] = ParticlePeriodicBCond;
        bc_m[i] = new PeriodicFace<double,Dim,Mesh_t,Center_t>(i);
        vbc_m[i] = new PeriodicFace<Vector_t,Dim,Mesh_t,Center_t>(i);
    }
    for(int i=0; i<Dim; i++)
        decomp_m[i]=decomp[i];
}
```

The arrays bc_m and vbc_m holding the boundary conditions for particles and fields. In decomp_m the domain decomposition is stored.

1.3.2 The main

**Code Listing**

```cpp
int main(int argc, char *argv[]) {
    Ippl ippl(argc, argv);
    Inform msg(argv[0]);

    Vektor<int,Dim> nr(atoi(argv[1]),atoi(argv[2]),atoi(argv[3]));
    const unsigned int totalP = atoi(argv[4]);
    const int nt = atoi(argv[5]);

    e_dim_tag decomp[Dim];
    int serialDim = 2;
    Mesh_t *mesh;
    FieldLayout_t *FL;
    ChargedParticles<playout_t> *partBunch;

    NDIndex<Dim> domain;
    for(int d=0; d<Dim; d++) {
        domain[d] = domain[d] = Index(nr[d] + 1);
        decomp[d] = (d == serialDim) ? SERIAL : PARALLEL;
    }
```

In the first part of main, the discrete computational domain (domain) and the domain decomposition (decomp) is constructed. We have choose a 2D domain.
decomposition with \( z \) serial i.e. not parallelized.

**Code Listing**

```cpp
class Mesh_t
{ public:
    Mesh_t(const Domain& domain);
};

class FieldLayout_t
{ public:
    FieldLayout_t(const Mesh_t* mesh, const int* decomp);
};

class playout_t
{ public:
    playout_t(const FieldLayout_t* FL, const Mesh_t* mesh);
};

Vector_t hr(1.0);
Vector_t rmin(0.0);
Vector_t rmax(nr);

class ChargedParticles<playout_t>
{ public:
    ChargedParticles<playout_t>(const playout_t* PL, const Vector_t& hr, const Vector_t& rmin, const Vector_t& rmax, const int* decomp);
};
```

Here we construct the mesh the field layout (FL), describing how the fields are distributed and finally the particle layout PL. The latter is the used as a template argument to construct the particle container. For this example the mesh size is set to unity and the computational domain is the given by the number of mesh points defined in \( nr \).

**Code Listing**

```cpp
unsigned long int nloc = totalP / Ippl::getNodes();

partBunch->create(nloc);
for (unsigned long int i = 0; i< nloc; i++) {
    for (int d = 0; d<Dim; d++)
        partBunch->R[i](d) = IpplRandom() * nr[d];
}
partBunch->qm = 1.0/totalP;
partBunch->myUpdate();
partBunch->initFields();
```

Now each node created \( nloc \) particles and initialized the coordinates randomly in the computational domain. A fixed charge to mass ration is assigned. The `myUpdate()` moves all particles to their node defined by the domain decomposition and initialized the fields. In the last call the fields gets initialized with the sinusoidal electric field.
The last part of main consists of a simple integration scheme to advance the particles. The call `gather` interpolates the electric field at the particle position from the nearby grid points by a second order cloud in cell (CIC) interpolation scheme.

### 1.3.3 initFields

#### Code Listing

```c
void initFields() {
    NDIndex<Dim> domain = getFieldLayout().getDomain();
    for(int i=0; i<Dim; i++)
        nr_m[i] = domain[i].length();
    int nx = nr_m[0]; int ny = nr_m[1]; int nz = nr_m[2];
    double phi0 = 0.1*nx;
    Index I(nx), J(ny), K(nz);
    assign(EFD_m[I][J][K](0), -2.0*pi*phi0/nx * cos(2.0*pi*(I+0.5)/nx) * cos(4.0*pi*(J+0.5)/ny) * cos(pi*(K+0.5)/nz));
    assign(EFD_m[I][J][K](1), ...);
    assign(EFD_m[I][J][K](2), ...);
    assign(EFDMag_m[I][J][K], EFD_m[I][J][K](0) * EFD_m[I][J][K](0) + EFD_m[I][J][K](1) * EFD_m[I][J][K](1) + EFD_m[I][J][K](2) * EFD_m[I][J][K](2));
}
```

### 1.3.4 myUpdate

#### Code Listing

```c
void myUpdate() {
    if(fieldNotInitialized_m) {
        fieldNotInitialized_m=false;
        getMesh().set_meshSpacing(&(hr_m[0]));
    }
```
getMesh().set_origin(rmin_m);
EFD_m.initialize(getMesh(), getFieldLayout(), GuardCellSizes<Dim>(1), vbc_m);
EFDMag_m.initialize(getMesh(), getFieldLayout(), GuardCellSizes<Dim>(1), bc_m);
}
	his->update();

1.3.5 gather

Code Listing

```cpp
void gather() {
    IntCIC myinterp;
    E.gather(EFD_m, this->R, myinterp);
}
```

1.4 Example 4 A Particle Particle - Particle Mesh (P^3M) Solver

Particle Particle - Particle Mesh solvers take the close range interaction of particles into account by combining a mesh solver as seen in the previous section with a quadratic Particle Particle computation for particles that are closer than a given interaction radius $r_i$ (see [http://en.wikipedia.org/wiki/P3M](http://en.wikipedia.org/wiki/P3M), [http://arxiv.org/abs/astro-ph/9805096](http://arxiv.org/abs/astro-ph/9805096) and [http://arxiv.org/abs/astro-ph/0512030](http://arxiv.org/abs/astro-ph/0512030)). To be able to combine these two solutions the Greens functions of the PIC solver and the particle-particle interaction have to be modified such that they add up to the desired Green function:

$$G(x) = G_{pp}(x) + G_{mesh}(x).$$ (1.1)

In case of the Green function $G(x) = \frac{1}{||x||^2}$ this can be achieved by setting

$$G_{mesh}(x) = \begin{cases} G(x), & \text{if } ||x|| > r_i \\ -\frac{3||x||^2}{r_i^4} + \frac{4||x||}{r_i^3}, & \text{else} \end{cases}$$ (1.2)

and

$$G_{pp}(x) = G(x) - G_{mesh}(x)$$ (1.3)

see also Figure 1.1.
Therefore the Green function from the PIC example has to be replaced by the following modified Green function:

```cpp
template<>
struct SpecializedGreensFunction<3>
{
    template<class T, class FT, class FT2>
    static void calculate(Vektor<T, 3> &hrsq, FT &grn, FT2 *grnI, double R) {
        grn = where(lt(R*R, grn), 1./sqrt(grn),
                     ((grn*sqrt(grn))/R-2*grn)/(R*R*R) + 2/R);
        grn[0][0][0] = grn[0][0][1];
    }
};
```

The short range interaction is handled by IPPL’s pairbuilding mechanism which applies the Green function to each pair of particles whose distance is below the interaction radius (see 4.3.1). For this purpose ChargedParticles contains
Error $= \left( \Sigma (x_{\text{analytical}} - x)^2 \right)^{0.5} / n$

Figure 1.2: Error and time scaling of the $P^3M$ solver for different interaction radii. Measured for a uniform spherical distribution of $10^4$ particles on 4 processors.

the following member function:

**Code Listing**

```cpp
void calculatePairForces(double interaction_radius)
{
    HashPairBuilder< ChargedParticles<playout_t> > HPB(*this);
    // apply the field to each pair, the -1 is the field constant
    HPB.for_each(RadiusCondition<double, Dim>(interaction_radius),
                  ApplyField<double>{-1,interaction_radius});
}
```

Which calls `ApplyField` for each pair of particles that fulfills the condition $||x_1 - x_2|| = ||x|| < r_i$. 

15
1.5 Installation

IP²L uses the *cmake* build philosophy. The following UNIX environment variables must be set:

IPPL_ROOT

defining where the IP²L source is installed, and

IPPL_PREFIX

where you install the library.

1.5.1 Building IP²L

```
cd $IPPL_PREFIX
CXX=mpicxx F77=gfortran cmake -DCMAKE_VERBOSE_MAKEFILE=OFF
-DCMAKE_INSTALL_PREFIX=$IPPL_PREFIX
make install
```

1.5.2 Used Compilers and Libraries

The supported operating systems and libraries are listed in Table 1.1.
Table 1.1: Supported Architectures and needed Libraries

<table>
<thead>
<tr>
<th>Operating System</th>
<th>HDF5</th>
<th>H5hut</th>
<th>Compiler</th>
<th>Open MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (SL) 2.6.18</td>
<td>hdf5-1.8.8</td>
<td>V 0.99</td>
<td>GNU 4.6.3, icc12.x</td>
<td>1.4.2</td>
</tr>
<tr>
<td>Cray XTx</td>
<td>hdf5-1.8.8</td>
<td>V 0.99</td>
<td>GNU 4.6.3</td>
<td>-</td>
</tr>
<tr>
<td>IBM BG/P&amp;Q</td>
<td>hdf5-1.8.8</td>
<td>V 0.99</td>
<td>GNU 4.6.3</td>
<td>-</td>
</tr>
</tbody>
</table>
1.6 Acknowledgements

The contributions of various individuals and groups are acknowledged in the relevant chapters, however a few individuals have or had considerable influence on the development, Julian Cummings, Yves Ineichen and Jakob Progsch. Misprints and obscurity are almost inevitable in a document of this size. Comments and active contributions from readers are therefore most welcome. They may also be sent to andreas.adelmann@psi.ch.

1.6.1 Citation

Please cite IP\textsuperscript{2}L in the following way:

\begin{verbatim}
@techreport{ippl-User-Guide,  
title = "(The IPPL (Independent Parallel Particle Layer) Framework )",  
author = "A. Adelmann",  
institution = "Paul Scherrer Institut",  
number = "PSI-PR-09-05",  
year = 2009}
\end{verbatim}
Chapter 2

Framework Setup

2.1 Initialising IP²L

IP²L is initialized by passing argc and argv to Ippl() constructor or by creating an instance of Ippl::Options, configuring it, and then passing that options object to Ippl::initialize(). After the Ippl() constructor call MPI (or any other parallel subsystem) is proper initialized. With Ippl::getNodes() or Ippl::myNode() you can for example gather information how many compute nodes/cores are available and on which of the nodes you are running.

Code Listing

```cpp
#include "Ippl.h"
int main(int argc, char *argv[])
{
    Ippl ippl(argc, argv);
    ....
}
```

2.2 Utility Classes in IP²L

IP²L provides, and uses internally, a number of useful utility classes which you may find helpful when developing new applications.

2.2.1 Inform Class

The Inform class is used to print messages to the console or to a file. It has an interface which is very similar to the iostream classes in C++, and it is mostly used in those situation where you might print a message to cout or cerr. An Inform object is created with a prefix string, which is then appended to all lines of output from the Inform object. Inform essentially takes in data to be printed, formats it
for printing just as an ostream object would, but also appends the prefix message
to all lines of output. Most important Inform will also indicate which node printed
the message when running in parallel.

**Constructing New Inform Objects**

The constructor for Inform has the form

```
Inform(char *prefix = 0, int node = 0)
```

where prefix is a string to prepend to all output lines, and node indicates on what
node the Inform object should actually print out the information it is given. Notice
that both of these arguments have default values; if no arguments are used when
creating a new Inform object, no prefix will be used, if only one argument is
given, then node default to 0, which means this Inform object will only print out
messages on node 0.

```
Inform blankmsg;
blankmsg << "Some text." << endl;
```

This Inform object will print the text it is given to standard out. The final " endl"
is a special manipulator object, which signals the Inform object to print out the
message it has been given. It will automatically append an endline to the message
if it does not already have one at the end. It is important to use endl with an Inform
object if it is not ever used, the Inform object will never print out its accumulated
text.

```
Inform testmsg("mytest");
testmsg << "More text. argc = " << argc << endl;
```

Here, the prefix is given, if this is used when running in serial, the output will look
like:

```
mytest> More text. argc = 1
```

or, if you use this when there is more than one processor in use, the prefix will
also include the node number in curly brackets:

```
mytest{0}> More text. argc = 1
```

On all other nodes than node 0, when this Inform object is used, it will not print
out the message.

```
Inform testmsg("testall", INFORM_ALL_NODES);
```

This example is similar to the previous example, except the second argument ex-
plicitly specifies which node to print on. This can be a number from 0 .... (num
nodes - 1), or, as in this example, it can be INFORM_ALL_NODES which indicates
the message should be printed on ALL the nodes instead of just one. You can also
change the node on which an Inform object will print after it has been created by
using the setprintNode(int) method of Inform.
Predefined Inform Objects

Creating new Inform objects for printing messages is useful in contexts where you would like a unique prefix to indicate where the message originated, say in a specific class method. However, the IP²L framework provides a set of predefined Inform instances which may be used to quickly generate output message or to make sure all messages have a common prefix. These Inform objects are static members of the IPPL class, which is used to initialize the framework. The predefined instances are:

```cpp
IPPL::Info = new Inform("IPPL");
IPPL::Warn = new Inform("Warning");
IPPL::Error = new Inform("Error", INFORM_ALL_NODES);
```

These three instances are used to print generally informative messages, warning messages, and error messages. Info and Warn only print on node 0 by default; Error will print on all nodes. You may use these to print messages in your own application:

```cpp
+IPPL::Info << "An informative message." << endl;
```

Notice that here that Info was first dereferenced, since it actually is a pointer to an Inform object. A better (and recommended) way to use these predefined instances is to use a macro which is defined for each instance. The macros to use are INFORMMSG, WARNMSG, and ERRORMSG; an example of their use is:

```cpp
WARNMSG("This is a warning: value = " << warnvalue << endl);
```

The argument to the macro is then given to the associated Inform object for printing.

2.2.2 Timer Class

Timer is used to perform simple timings within a program for use in, e.g., benchmarking. It tracks real (clock) time elapsed, user time, and system time. It acts essentially as a stopwatch: initially it is stopped, and YOU tell it to stop and start with method calls. The Timer constructor takes no arguments; you create a new Timer object, and use the following methods:

```cpp
//Start the clock running. Time only accumulates in the Timer when it is running.
void start()
void stop()    //Stop the clock. The clock may be started again later.
void clear()   //Resets the accumulated time to zero
float clock_time()  //Reports the accumulated "wall clock" time in seconds.
float user_time()   //Reports the accumulated user CPU time in seconds.
float system_time() //Reports the accumulated system CPU time in seconds.
float cpu_time()    //Reports user_time() + system_time()
```
Example how to use the timer class:

Code Listing

```cpp
IpplTimings::TimerRef selfFieldTimer_m;  // definition
selfFieldTimer_m = IpplTimings::getTimer("computeSelfField");

selfFieldTimer_m.start();
/* compute something */
selfFieldTimer_m.stop();

IpplTimings::print();
```

Note on Cray XT3/4 only wall clock is reported.

### 2.2.3 Memory Footprint Class

This class allows the application to query the amount of available memory. This feature will be available in V1.0.1.
Chapter 3

FFT

The FFT class provides an interface for performing parallel Fourier transforms of various types on IP^2L Field objects. FFT is templated on the type of transform to perform (CCTransform, RCTransform, or SineTransform); the dimensionality Dim of the fields to be transformed, and the floating-point precision type (either float or double). It is capable of transforming along all dimensions of a Field or only specified dimensions, and it handles all of the data transposes required to make the Fourier transforms efficient automatically. The FFT constructor arguments vary slightly depending upon which type of transform you wish to perform. Generally speaking, you provide an NDIndex object or objects which contain the domains of the input and/or output Fields for the Fourier transform, an optional array of bools of length Dim indicating which dimensions are to be transformed (default is all dimensions), and an optional bool indicating whether or not to compress the intermediate Fields needed to perform data transposes when they are not in use. The default value of this optional argument is false, but the user can set this argument to true if it is necessary to conserve memory. For a complex–to–complex Fourier transform, the input and output fields are of the same element type and are the same size, so only one domain argument is needed. So in the simple case of transforming all dimensions of a Field of type complex<double>, we would construct the FFT object with the code

```cpp
FFT<CCTransform, Dim, double> ccfft(domain);
```

where domain is an NDIndex<Dim> describing the domain of complex Fields to be transformed with the FFT object. A real–to–complex Fourier transform takes a field of real numbers and returns a field of complex numbers (or vice-versa for an inverse complex–to–real transform), so we require separate domain arguments describing each Field in the FFT constructor. From the theory of Fourier mode analysis, we know that a Fourier transform of N real numbers will produce N/2 + 1 unique complex modes, with modes 0 and N/2 being purely
real. Some FFT routines take advantage of the fact that if you pack together the real parts of modes 0 and \(N/2\) as one complex number, you can store all the resulting mode information in the same space as required for the input (i.e., \(N\) real numbers or \(N/2\) complex numbers). Such a technique tends to cause confusion in multidimensional real–to–complex FFTs, since mode data must then be separated out afterwards. So we choose a format in which the \(N/2 + 1\) complex modes are stored separately as complex numbers. Thus, when a real–to–complex transform is performed on a Field of doubles, the resulting Field of type complex<double> will have an extent one greater than half the length of the input field along the first dimension to be transformed and the same length along all other dimensions. This conformance of domains is checked by the FFT constructor. We might construct an FFT object for real–to–complex transforms with the line

\[
\text{FFT<RCTransform,Dim,double> rcfft(rdomain,cdomain,tdim);}
\]

where rdomain and cdomain are the conforming domains for the real and complex fields and tdim is an array of bools indicating whether or not to transform each dimension. Note that we assume the axes of the field are to be transformed along in the order indicated by the domain arguments for a forward FFT and in the reverse order for an inverse FFT. Each Index object inside the provided domain should refer to a particular axis of the input Field, and these axes are transformed along in order. A sine transform is a special type of Fourier transform in which only the sine (odd) modes are retained. This transform has a field of real numbers for both its input and output, and its effect is to keep only that portion of the data which exhibits odd parity (i.e., vanishes at the endpoints of the interval). Typically, one wishes to enforce odd parity along one or more dimensions of a field, and then perform a standard real-to-complex transform along remaining dimensions. Hence, we require that the user provide two arrays of bools in the constructor: the first to indicate along which dimensions to perform a sine transform, and the second to indicate all of the transform dimensions (both sine transforms and standard FFTs). For example,

\[
\text{FFT<SineTransform,Dim,double> sinefft(rdomain,cdomain,sinedim,tdim);} 
\]

constructs an FFT object for doing sine transforms along the dimensions indicated by sinedim and a standard real-to-complex FFT over the other dimensions included in tdim. Alternatively, such transforms could be achieved in two steps, doing the sine transforms and the standard FFTs separately. In this case, we might construct our sine transform FFT object with the code

\[
\text{FFT<SineTransform,Dim,double> sinefft2(rdomain,sinedim);} 
\]

and then construct a second FFT object for handling the real-to-complex transform. Note that a sine transform FFT object which is doing only sine trans-
forms requires only a single domain argument describing the real input and output Fields in its constructor.

Once the appropriate FFT object has been constructed, a Fourier transform of data is invoked using the transform member function. The normal arguments to this function are an integer value of +1 or -1 to indicate the sign of the exponential used in the transform (i.e., the direction of the transform, forward or inverse), and the input and output Fields. For this "two-field" form of the transform function, there is also an optional argument of type bool, which indicates whether or not the input Field is considered to be constant by the transform function. The default value of this optional argument is false, which allows the transform routine to attempt to use the input Field as temporary storage and avoid doing an additional data transpose. You should set the value of this argument to true if you must preserve the contents of the input Field for later use. We would use our previously constructed FFT object for real–to–complex transforms to perform a forward FFT in the following manner:

```plaintext
cfft.transform(+1,realField,complexField);
```

The results of the transform are automatically normalized such that a forward transform followed by an inverse transform returns the original data. For convenience, the FFT class has a member function `setDirectionName` which allows you to associate a character string with each of the transform directions +1 and -1. You might choose to refer to these directions as "xtok" and "ktox", for example.

In the case of a complex-to-complex FFT or a pure sine transform; the input and output fields are the same size and of the same type. In these instances, we offer the option of performing the transform "in place"; that is, using just one Field argument for both the input and output. For example, we could perform an inverse complex-to-complex FFT with the code

```plaintext
cfft.transform(-1,complexField);
```

### 3.0.1 Improving FFT Performance

Some improvement in performance of the transform method may be obtained by careful selection of the axis ordering of input and output Fields. In order to perform a parallel FFT along a particular dimension, the FFT object will first reorder the axes so that the first axis is the one to be transformed. It does this by assigning the field data into a new Field with a domain in which the order of the original Index objects has been permuted. This new Field, which is maintained internally by the FFT class, has a data layout that is serial along this first dimension and parallel along all other dimensions. With this layout, each processor can independently perform FFTs along the serial axis for each of the one-dimensional
strips of data it owns. To subsequently transform along another dimension, the FFT object must again transpose the data so that the next dimension to be transformed is now the first dimension and is serial. These data transposes can be fairly costly to perform. We can eliminate at least one data transpose if the output \texttt{Field} supplied by the user has the same layout characteristics needed for the final transform (or, in the case of an “in place” transform, if the input \texttt{Field} matches the layout needed for either the first or last transform), and has no guard cell layers. For instance, let us assume we have a three-dimensional \texttt{Field} of complex numbers and we want to transform all dimensions. If the Index objects \texttt{I}, \texttt{J}, and \texttt{K} describe the first, second, and third axes of our \texttt{Field} domain, we could perform a forward FFT with the line

\begin{verbatim}
ccfft.transform(+1,complexField1);
\end{verbatim}

If the first dimension of \texttt{complexField} is serial, the transform method will skip the first data transpose because the input data is already distributed appropriately for transforming along the first dimension. Similarly, if we were to call an inverse transform with this same \texttt{Field}, it would transform the axes in reverse order, and we would be able to skip the final data transpose. Alternatively, we might choose to do this FFT using separate input and output \texttt{Field}s:

\begin{verbatim}
ccfft.transform(+1,complexField1,complexField2);
\end{verbatim}

In this case, the final optional argument to the "two-field" transform function defaults to false, meaning that \texttt{complexField1} is not considered constant and may be used in place of a temporary \texttt{Field} to avoid the first data transpose. In addition, the output \texttt{Field} can be used in place of the final temporary \texttt{Field} if it has the proper layout. If \texttt{complexField2} has its axes reordered so that its first axis is the final axis to be transformed (e.g., \texttt{K}, then \texttt{I}, then \texttt{J}) and that first axis is serial, then we can skip the final data transpose. This choice of data layout results in a slightly faster parallel FFT, and it is often convenient if all you need to do is transform the data, do a brief computation with the transformed data, and then invert the transform.

Another issue of relevance to the performance of the transform method is the type of routine used to perform the actual one-dimensional FFT. Currently, we provide two options for this. The first is Fortran 77 implementations of FFT routines from the Netlib repository. These are portable and highly optimized routines that we invoke via C++ wrapper functions. The second option (available only on SGI and Cray systems) is native FFT routines from the SGI/Cray Scientific Library. These routines can be substituted for the portable Netlib routines by supplying the option \texttt{USE_SCSL_FFT} to the configure utility before compiling the IP2L library. These native library routines tend to be somewhat faster than the portable Fortran routines, and we plan to offer the ability to use native FFT routines such as FFTW in the future.
Chapter 4

Particles

This section describes the IP$^2$L framework classes which provide the capability to performing particle-based simulations. We first describe how to design and instantiate Particle classes customized to the needs of a specific application, and then discuss the possible operations and expressions in which a particle object may be employed and end with an ready to use example.

4.1 Basic Particle Object Characteristics

The IP$^2$L framework treats Particle classes as containers which store the characteristic data for $N$ individual particles. Each particle has several attributes, such as position, mass, velocity, etc. Looked at in another way, Particle classes store several attribute containers, where each attribute container holds the value of that attribute for all $N$ particles. Particle objects in IP$^2$L may be thought of as shown in the following diagram: ...

There are two particle attributes predefined, namely R (position) and ID a global unique identifier.

The data type of each attribute, the number of attributes, and the names for these attributes are completely customizable, and are specified in the manner described in the following sections. Any number of different Particle classes may be defined and used within a given simulation. Also, the Particle objects may interact with IP$^2$L Field objects or may be used independently. In addition to the attributes, each Particle object uses a specific layout mechanism, which describes the data of the individual particles is spread across the processors in a parallel environment. The IP$^2$L framework provides several different Particle layout classes, any of which may be selected to partition the particle data among processors. The choice of layout depends on the intended use of the Particle object, as discussed later. Once defined and instantiated, Particle objects in
the IP2L framework may be used in many ways, including:

- Operations involving all the particles within a Particle object may be specified using simple expressions, in a manner very similar to that used for Field objects. These expressions may involve any of the attributes of the particles as well as other scalar data, and they may use not only the standard mathematical operators +, -, *, /, etc., but also standard mathematical functions such as \( \cos() \), \( \exp() \), \( \text{mod}() \), etc.

- Alternatively, you may set up explicit loops that perform operations involving the attributes of a single particle or a subset of all the particles.

- Particles may be created or destroyed during a simulation.

- Particle-to-Field and Field-to-Particle operations may be performed (e.g., a particular Particle attribute may be deposited onto a specified Field using a chosen interpolation method).

### 4.2 Defining a User-Specified Particle Class

There is no specific class within the IP2L framework called Particle. Rather, the first step in deploying particles within a IP2L application is to define a user-specified Particle class, which contains the attributes required for each particle, as well as any, specific methods or data the user may need. To do this, the ParticleBase and ParticleAttrib classes are used, along with a selected subclass of the ParticleLayout class. The steps to follow in creating a new Particle class are:

- Based on the type of interactions which the particles have with each other and with external objects such as a Field, select a method of distributing the particles among the nodes in a parallel machine.

- Next, decide what attributes each particle should possess.

- Third, create a subclass of ParticleBase which includes these attributes (specified as instances of the ParticleAttrib class template).

- Finally, instantiate this user-defined subclass of ParticleBase and create and initialize storage for the particles which are to be maintained by this object.

The following sections describe in more detail how to accomplish these steps.
4.2.1 Selecting a Layout: ParticleLayout and Derived Classes

When used in a parallel environment, the IP2L framework partitions the particles in a Particle container among the separate processors and includes tools to spread the work of computing and the results of expressions involving Particle attributes among the processing nodes. There are, however, different ways in which particles may be distributed among the processors, and the method which should be used depends upon how the particles in a Particle object will interact with each other and with Field objects (if at all). The IP2L framework includes different Particle layout mechanisms, which are all derived from the ParticleLayout class. Each Particle object needs its own layout object; that is, you cannot create a layout object and give it to more than one Particle object. The methods typically used to determine how to assign particles to particular nodes are based on analysis of the position (\( R \)) attribute of each particle. Thus, ParticleLayout and its derived classes have two template parameters: the type and the dimensionality of the particle position attribute (this particle position attribute is discussed in more detail later). The following sections describe the particle layout mechanisms currently available in the IP2L framework.

4.2.2 The ParticleUniformLayout Class

The ParticleUniformLayout class maintains an equal number of particles on each node, with no consideration of particle position. ParticleUniformLayout is useful in those cases where particles do not interact with each other but perhaps with some other external agent, so that no consideration need be made about which particles are located near to others. In that case, with this layout maintains an equal balance of memory usage among the processors and requires relatively small amounts of interprocessor communication. If you require the ability to compute an interaction between a particle and its nearest neighbors, this is not the proper layout to use, in that case, the ParticleSpatialLayout class (discussed next) is a better choice. The constructor for ParticleUniformLayout takes no arguments, but does require the two template parameters that specify the type and dimensionality of the particle position attribute. An example of creating a new ParticleUniformLayout instance for a 3D particle simulation is:

```cpp
ParticleUniformLayout<double, 3> uniformlayout();
```

4.2.3 The ParticleSpatialLayout Class

ParticleSpatialLayout, in contrast to ParticleUniformLayout, assigns particles to nodes based upon their spatial location relative to a FieldLayout.
It is useful when the particles will be interacting with other particles in their neighborhood or with a Field object. ParticleSpatialLayout will keep a particle on the same node as that which contains the section of the Field in which the particle is located. If the particle moves to a new position, this layout will reassign it to a new node when necessary. This will maintain locality between the particles and any Field distributed using this FieldLayout. Further more it will help keep particles which are spatially close to each other local to the same processor as well. As with all the layout classes, ParticleSpatialLayout requires the type and dimensionality of the particle position attribute as template parameters. The constructor for ParticleSpatialLayout takes one argument: a pointer to a FieldLayout object that tells the ParticleSpatialLayout how the Field is allocated among the parallel processors, so that the particles may be maintained local to this Field. Note that you do not, need to create a Field instance itself, you only need to give ParticleSpatialLayout a FieldLayout object. An example of creating an instance of this class is as follows:

```cpp
FieldLayout<3> myfieldlayout(Index(l6), Index(16), Index(32));
ParticleSpatialLayout<double,3> myparticlelayout(&myfieldlayout);
```

Note that the dimensionality of the FieldLayout and the ParticleSpatialLayout (in this example, 3) must be the same. You may also create a ParticleSpatialLayout instance without providing a FieldLayout. In this case, particles will remain on the node on which they were created. If at some future time you wish to provide a FieldLayout object to tell the ParticleSpatialLayout where to place the particles, you may do so using the setFieldLayout (FieldLayout<Dim>* ) method of ParticleSpatialLayout. This is useful when reading particles in from an external source and the size of the spatial domain containing the particles is not known until all the particles have been read. The following example demonstrates the use of the capability:

```cpp
ParticleSpatialLayout<double,3> myparticleLayout;
// calculate the size of the domain required to contain all the particles
// create a new FieldLayout object based on these calculations
FieldLayout<3> myfieldlayout(Index(minx, maxx), Index (miny, maxy), Index(minz,maxz);
myparticlelayout.setFieldLayout(&myfieldlayout);
```

ParticleSpatialLayout also provides functionality to maintain cached ghost particles from neighboring nodes which might be required for particle-particle interaction. A caching policy can be defined using the fourth template parameter of ParticleSpatialLayout:

```cpp
typedef UniformCartesian<Dim, double> Mesh_t;
typedef ParticleSpatialLayout<double, Dim, Mesh_t, BoxParticleCachingPolicy<double, Dim, Mesh_t> > playout_t;
```
The available caching policies are: NoParticleCachingPolicy, BoxParticleCachingPolicy, and CellParticleCachingPolicy. With NoParticleCachingPolicy there is no caching whatsoever. BoxParticleCachingPolicy extends the interface of ParticleSpatialLayout by two functions void setCacheDimension(int d, T length) and void setAllCacheDimensions(T length) which are used to set the size of the cached region around the local domain in units of space. CellParticleCachingPolicy extends the interface of ParticleSpatialLayout by two functions void setCacheCellRange(int d, int length) and void setCacheCellRanges(int d, int length) which are used to set the size of the cached region around the local domain in units of grid cells of the mesh. BoxParticleCachingPolicy is the default policy.

The caching can be enabled or disabled by calling the enableCaching() or disableCaching() member functions of ParticleSpatialLayout. Caching is disabled by default.

4.2.4 Selecting Particle Attributes: The ParticleAttrib Class

ParticleAttrib is a class template that represents a single attribute of the particles in a Particle object. Each ParticleAttrib contains the data for that attribute for all the particles. Within a user-defined Particle class, you declare an instance of ParticleAttrib for each attribute the particles will possess and assigns to it an arbitrary name. ParticleAttrib requires one template parameter, the type of the data for the attribute. As an example, the statement:

```
ParticleAttrib<
```

density;
```
double
```

declares an instance of ParticleAttrib named 'density', which will store a quantity of type double for all the particles of the Particle class that contains this data member.

4.2.5 Specifying a User-Defined Particle Class: The ParticleBase Class

ParticleBase is the class that all user-defined Particle classes must specify as their base class. It stores the list of attributes for the particles (which are maintained as instances of ParticleAttrib) and a selected parallel layout mechanism. In addition to providing all the capabilities for performing operations on the particles and their attributes, ParticleBase also defines two specific attributes which all user-defined Particle classes inherit:
ParticleAttrib<Vektor<T,Dim>> R;
ParticleAttrib<unsigned> ID;

The first attribute, R, represents the position of each particle. Each position is stored as a Vektor<T, Dim>, which is a IP^2^L data type representing a dim-dimensional vector with elements of type T. The second attribute, ID, stores a unique unsigned integer value for each particle. The values are not guaranteed to be in any particular order, but they are guaranteed to be unique for each particle. ParticleBase has one template parameter, the layout class to be used to assign particles to processors (e.g., ParticleSpatialLayout). The data type and dimensionality of the particle position attribute (R) will be the same as those used to create the specific ParticleLayout derived class. Each ParticleBase contains one instance of the chosen layout class. There are two constructors for ParticleBase: a default constructor that creates a new instance of the layout class using the layout’s default constructor, and a constructor which takes a pointer to an instance of the layout class. The second version of the ParticleBase constructor is useful when the desired layout class requires arguments to its constructor (e.g., ParticleSpatialLayout, which may be give in a FieldLayout pointer).

Using ParticleBase, ParticleAttrib, and a selected class derived from ParticleLayout, you can create a user-defined Particle class using the following code template:
class Bunch : public ParticleBase< ParticleSpatialLayout<double,3> > 
{

public:
  // Attributes for this particle class (besides position and ID).
  ParticleAttrib<double> qm;  // q/m ratio
  ParticleAttribs Vektor<double,2> > vel;  // velocity

  // constructor
  Bunch(Layout_t *L) : ParticleBase<Layout_t>(L) {
    addAttribute(qm);
    addAttribute(vel);
  }
};

Let us describe this example in detail by discussing the important lines in the order of use.

Line 1: You may select whatever name is appropriate for the specialized Particle class, but it must be derived from ParticleBase.

In this case, we explicitly specify the type of layout to use (ParticleSpatialLayout), with particle position attribute type and dimensionality template parameters of double and 3, respectively. Alternatively, Bunch may have been declared as a class template itself and may have passed on the layout template parameters to ParticleBase. In that case, the first line would instead look like

template <class PLayout>
class Bunch : public ParticleBase<PLayout>

Lines 5-6: Here is where the attributes for the particles in the Particle object are declared. They may be given any name other than R or ID. Instead of stating the type and dimensionality of this attribute specifically, you may also use one of the following typedefs and constants defined in ParticleBase:

- Dim - the dimensionality of the particle position attribute (in this example, 3)
- Position_t - the type of data used to store the position attribute components (here, this type is double)
- Layout_t - a synonym for the specified layout class
- ParticlePos_t - a typedef for the particle position attribute; it is shorthand for ParticleAttrib< Vektor<Position_t,Dim> >

and could have been used to specify the attribute vel in the above example as ParticlePos_t vel;

- ParticleIndex_t - a typedef for the particle global ID attribute; it is short for ParticleAttrib<unsigned>
The constructor for this user-defined class must initialize ParticleBase with a pointer to an instance of the selected layout class.

In this example, the layout class is ParticleSpatialLayout, but using one of the typedefs listed above, we can abbreviate this as Layout_t. Note that we only define one constructor here, omitting the default constructor. This is done because ParticleSpatialLayout (which we have hard-coded as the layout for this user-defined Particle class) requires an argument to its constructor, and this can only be provided if we use a constructor for our Particle class as shown here. A new instance of this class would be declared in an application as follows:

```cpp
Bunch myBunch (new ParticleSpatialLayout<double,3>(myFieldLayout));
```

where myFieldLayout was a FieldLayout object created previously. The only action that is required in the constructor for the derived class is to inform the base class of the declared attributes, using the addAttribute(.) method of ParticleBase, which registers the specified ParticleAttrib instance with the parent class ParticleBase. The order in which attributes are registered is not important.

### 4.2.6 Example Particle Classes: The Genparticle and GenArrayParticle Classes

The IP2L framework provides two classes which are examples of Particle classes derived from ParticleBase: Genparticle and GenArrayParticle. They may be used as samples from which to build new classes, or they may be used to quickly include particle capabilities in an application. Genparticle is a Particle class with three attributes: R and ID inherited from ParticleBase, and an attribute named data with elements of an arbitrary type. Genparticle has two template parameters: the type of particle layout to use and the type T of attribute data. It has two constructors just as ParticleBase does: the default and one taking a layout pointer. An example of instantiating a GenParticle object is shown below.

```cpp
GenParticle<ParticleUniformLayout<float,3>,UserDefinedType> GP();
```

GenArrayParticle is almost identical to GenParticle, the difference being that GenArrayparticle contains not just one but an array of attributes data [0 ... N-1] of a specified type. The number of elements in the attribute array, N, is given as a third template parameter. The following example shows a GenArrayParticle being created with 5 floats stored in the data array for each particle:

```cpp
GenArrayParticle<ParticleSpatialLayout<double,3>,float,5> GAP{
  new ParticleSpatialLayout<double,3>(myFieldLayout)};
```
It is important to note that the array data in GenArrayParticle contains a set of particle attributes of the same type. In situations where it is necessary to have a variety of particle attribute types, you may use the Genparticle class with the type of data being specified as a user-defined struct containing the various attributes needed.

4.2.7 Using Particle Classes in an Application

After a specific Particle class has been defined and created in a IP2L application, you may create and initialize new particles, delete unwanted particles, and perform computations involving these particles. This section describes how to accomplish these tasks.

4.2.8 Creating New Particles

When a Particle object is created, it is initially empty. Storage for new particles is allocated using the create (unsigned) method of ParticleBase. For example, if a Particle object bunch has been created already, the statement

\[
bunch.create(100);
\]

will allocate storage for 100 new particles. All the attributes for the particles in the Particle object will have this new storage allocated. The data is uninitialized, except for the global ID attribute; you must assign the proper values to the position and any other attributes that have been defined. The new storage is appended to the end of any existing storage.

ParticleBase includes two methods that allow you to query how many particles exist. The function getTotalNum() will return the total number of particles being stored among all the processors; the function getLocalNum() will return the number of particles just on the local node. Although the new storage space is allocated on the local processor on which the call to create was executed, the Particle class will not officially add the particles to its local count (and will not tell any other processors it has created these new particles) until you call the update() method of ParticleBase. Thus, a call to getLocalNum() will report the same number just before and just after the call to create. The storage does exist after create is called, but only after the update method (which is discussed in more detail in a later section) has been called will all the processors have correct information on their local and total particle counts.
4.2.9 Initializing Attribute Data

After calling create to allocate new storage, you must initialize the data. This should be done after calling create and before calling update for the Particle object. After the data is initialized, the update routine will properly distribute the particles to their correct node based on the layout mechanism chosen for that Particle object and possibly the positions of the particles as set during their initialization. The following example shows one way to initialize the data for newly created particles when running on a single-processor machine. (This example will be modified in the following section for the case of running in parallel.)

Code Listing

```cpp
// create and initialize data for an instance of Bunch
Bunch myBunch(new Bunch::Layout_t(myFieldLayout));
int currLocalPtcls = myPtcls.getLocalNum();
myBunch.create(100);
for (int i = 0; i < 100; i++) {
    myBunch.R[currLocalPtcls + i] = Vektor<double, 3>(0.0, 1.0, 0.0);
    myBunch.vel[currLocalPtcls + i] Vektor<double, 3>(1.0, 1.0, 1.0);
}
myBunch.update();
```

In this example, 100 new particles are created, and the R and vel attributes are initialized to Vektor quantities. Notice that each attribute is accessed simply by specifying it as a data member of the myBunch object. After create was called, even though the 100 particles were not added to the Particle object’s count of local particles, the storage was allocated and it was possible to assign values to the new elements in the attribute storage (accessed simply using the [] indexing operator). Finally, calling update added the new storage to the count, of particles stored in myBunch. Further calls to getLocalNum and getTotalNum would report the proper values.

4.2.10 Initializing Attribute Data on Parallel Architectures

The code shown in the previous example has one problem when used on parallel architectures: the call to create is performed on each processor, so if there were P processors a total of 100*P particles would be created. This may be the desired behavior, if so, the previous example is sufficient. However, if you are reading data on particle positions and other attributes from a file or some other source, you may wish to create particles on a single processor, and then distribute the data to the proper nodes. To do this, you need to call create and assign initial data on only one node but call update on all the processors. The singleInitNode() method of ParticleBase will return a boolean value indicating whether the local processor should be used to create and initialize new particles in this way. The following example demonstrates how to use this method for initializing par-
// create and initialize data for an instance of Bunch
Bunch myBunch(new Bunch::Layout_t(myFieldLayout));
int currLocalPtcls = myPtcls.getLocalNum();
if (myBunch.singleInitNode()) {
    myBunch.create(100);
    for (int i = 0; i < 100; i++) {
        myBunch.R[currLocalPtcls + i] = Vektor<double, 3>(0.0, 1.0, 0.0);
        myBunch.vel[currLocalPtcls + i] Vektor<double, 3>(1.0, 1.0, 1.0);
    }
}
myBunch.update();

4.2.11 Deleting Particles

Particles may also be deleted during a simulation. The method destroy(unsigned M, unsigned I) of ParticleBase will delete M particles, starting with the Ith particle. The index I here refers to the local particle index, not the global ID value. Thus I = 0 means delete particles starting with the first one on the local processor.

Unlike the situation when creating new particles, the storage locations for the deleted particles will not be removed from attribute data storage until update is called. Instead, the requests to delete particles are cached until the update phase, at which time all the deletions are performed. You are allowed to issue multiple delete requests between updates. For example, if there are 100 particles on a local node, and you request to delete particles 0 to 10 and then request to delete particles 60 to 70, nothing will change in the attribute storage until you call update, and no change will occur to the local and total particle counts until update() is complete.

4.2.12 Updating Particles: The update() Method

The update() method of ParticleBase is responsible for making sure that all processors have correct information about how many particles exist and where they are located in a parallel machine. As mentioned previously, update must be called by all processors after a sequence of particle creation or deletion operations. The update method is also responsible for maintaining a proper assignment of particles to processors, based on the particular ParticleLayout class used to create the ParticleBase object. Typically, this layout mechanism depends on the position of particles, so when particles change their position, they may need to be reassigned to a new processor to maintain the proper layout. In this case, the update method should be called whenever a computation is complete which alters the attributes (e.g. position) that a layout depends upon. The following
short example demonstrates using update in conjunction with some operation that alters the x-coordinate of a set of particles.

Code Listing

```c
// do some computation involving myBunch for several time steps
while (computation_done == false) {
    // for each particle, add some constant to the x coordinate
    myBunch.R(0) += 0.11
    // update the Particle object; this may move particles between nodes
    myBunch.update();
    // determine if the computation is done, etc.
}
```

### 4.2.13 Using Particle Attributes in Expressions

Computations involving particle attributes may be performed in many ways. Data-parallel expressions that involve all particles of a given Particle object may be used, or specific loops may be written that employ attribute iterators or nearest-neighbor pairlist iterators.

**Attribute Expressions**

Just as with the Field class, you may perform data-parallel operations on particle attributes using a simple expression syntax, which the IP2L framework will translate into efficient inlined code. These operations will be performed for every particle. The expressions may include any of the attributes in a Particle object as well as scalar values, may use mathematical operators such as +, -, *, / etc., and may call standard mathematical functions such as $\cos()$, $\exp()$, $\text{mod}()$, etc. for an attribute value of each particle. Some examples are shown below.

```c
double dt = 2.0;
myBunch.R += myBunch.vel* dt;
myBunch.vel = 1. - log (1. + myBunch.R * myBunch.R) ;
myBunch.update();
```

Attribute expressions will perform their operations on all the particles in the Particle object, including any new particles allocated via a call to create, even before update has been called. This fact is useful when initializing the attributes for newly created particles (e.g., to set the init value for some scalar quantity to zero). Generally, however, unless you are performing an initialization of new particles, you should avoid using particle expressions of this type after calls to create or destroy and before a call to update.

Some attributes, such as Vektors or Tenzors, have multiple components, and you may wish to involve only the Nth component of the attribute in an expression. To do so, use the () operator to select the Nth component of that attribute. For instance, using myBunch from the previous example, you can change just the x-coordinate of the particle position attribute R as follows:
myBunch.R(0) = myBunch.R(1) - \cos(myBunch.R(2));

For 2D or 3D quantities, use two or three indices. For example, if rho is a 3x3 Tenzor attribute of myBunch, you can do the following:
myBunch.rho(0,0) = -(myBunch.rho(0,1) + myBunch.rho(0,2));

Attribute expressions may also use the where operator in much the same way as for Field expressions. The first argument to where is some expression that results in a boolean value for each particle. The second and third arguments are expressions that will be evaluated for a particle if the first argument is true or false, respectively, for that particle. For example,
myBunch.vel = where(myBunch.R(0) > 0.0, -2.0 * myBunch.vel, myBunch.vel)

changes the value of the vel attribute in myBunch when the x-coordinate of the particle position is positive.

4.2.14 Particle Iterator Loop

You also have the capability of performing operations on specific particles using iterators or standard indexing operations. The ParticleAttrib containers in a Particle class may be used just as regular STL containers. The begin() and end() methods of the ParticleAttrib class will return an iterator pointing to the first element and just past the last element, respectively, of the attribute. These iterators may be used in an explicit loop just as if they were pointers into the attribute array.

ParticleAttrib<unsigned>::iterator idptr, idend = myBunch.ID.end();
for (idptr = myBunch.ID.begin(); idptr != idend; ++idptr)
  cout << "Particle ID value: " << *idptr << endl;

Iterators are available for all ParticleAttribs. As an alternative, you may simply use the [] operator to access the attribute data of the Nth particle on a node, treating ParticleAttrib as a regular array of data.

int nptcls = myBunch.getLocalNum();
for(int i=0; i < nptcls ++i) {
  cout << "Particle ID value: " << myPtcls.ID[i] << endl;
}

4.3 Nearest-Neighbor Interactions (Jakob)

4.3.1 Particle - Particle Interactions

Efficient particle - particle interactions can be achieved by use of PairBuilder objects. The basic usage is as follows:
This will call PairFunctor for each pair of particles that fulfills the PairCondition. There are three PairBuilders available: HashPairBuilder, BasicPairBuilder and SortingPairBuilder. HashPairBuilder should be used in most cases since it has the best time complexity. There are three predefined PairConditions. TrueCondition is always true and can be used to iterate over all pairs, RadiusCondition is true for each pair that is closer than a given interaction radius and BoxCondition is true for each pair where one particle lies inside a bounding box around the other particle. The following code shows how to iterate over all pairs within a given interaction radius:

```cpp
struct PairFunctor{
    void operator()(std::size_t i, std::size_t j, Bunch<ParticleLayout_t> &P) const {
        //some interaction involving particles i and j
    }
};
```

HashPairBuilder< Bunch<ParticleLayout_t> > HPB(myBunch);
HPB.for_each(RadiusCondition<double, Dim>(interaction_radius), PairFunctor());

To correctly generate all pairs in a multi process simulation a caching strategy has to be chosen so each process also has the required ghost particles. To achieve this for the example given one would call

```cpp
PL->setAllCacheDimensions(interaction_radius);
PL->enableCaching();
```

with PL being the ParticleSpatialLayout of the bunch.

It is also possible to write custom pair conditions. These have to provide a `operator()` that takes two vectors and returns a bool, when the functor is used, it is passed two vectors that represent the particle positions. Additionally pair conditions have to provide a `getRange` function that takes an integer as input and returns the “radius” along that dimensions for which the pair condition can be true. In other words: for two particle positions `a` and `b` for which the pair condition returns true, the condition `|a[i] - b[i]| <= getRange(i)` must hold.

### 4.3.2 Particle-Field Interactions

Many particle-based simulation methods, including ”particle-in-cell” (PIC) simulations, rely on the ability of particles to interact with field quantities. For instance, in particle-based accelerator (plasma) simulations, you typically track the motions of charged plasma particles in a combination of externally applied and self-generated electromagnetic fields. In a IP2L application, such fields might be
stored as Field objects of type Vektor existing on a pre-defined mesh. Particles moving through this mesh must be able to “gather” the current value of a Field to their exact positions. Additionally, in order to compute the values of self-generated fields, the particles must be able to ”scatter” the value of an attribute onto nearby mesh points, producing a Field. These gather/scatter operations are done using a set of IP²L interpolation methods.

IP²L provides a hierarchy of interpolation classes, each derived from the base class Interpolate and each containing the basic gather and scatter functions. The gather method allows you to gather one or more specified Fields into an equal number of ParticleAttribs. Similarly, scatter will accumulate one or more ParticleAttribs on to an equal number of Field objects. An example of how to scatter the particle density to a Field is shown below.

```cpp
InterpolateNGP<Dim> myInterpolater{myBunch}; // create NGP interpolater
Field<double,Dim> ptcl_density{myfieldlayout}; // create density field
myInterpolator.scatter(myBunch.density,ptcl_density); // do scatter
```

The various classes derived from Interpolate implement these gather and scatter methods using different well-known interpolation schemes, such as nearest grid point (NGP), linear interpolation, and the subtracted-dipole scheme (SUDS). You may use these provided classes as a template for deriving new classes from Interpolate that implement other interpolation schemes of interest.

In case of the CIC Interpolation and non-cyclic boundary condition, care has to be taken to not place particles in the outer half of boundary cells. Otherwise values will be scattered out of the grid and be irretrievable.
Chapter 5

Using the Field and Related Classes

This section introduces the interface of the Field class and related classes. We describe how to instantiate Field objects, use Index objects to perform index operations, perform expression operations with overloaded operators, apply boundary conditions, use the where construct for conditionals, invoke reduction operations, and use mathematical functions.

5.1 Field Object Instantiation

5.1.1 Field Template Parameters

The Field class is parametrized on 4 template parameters: type T, dimensionality Dim, mesh type Mesh, and centering Centering.

Field<class T, unsigned Dim, class Mesh=UniformCartesian<Dim,MFLOAT=double>,
class Centering=Mesh::DefaultCentering>

The T parameter represents the type of data that can be stored inside of a Field. Currently, the Field class supports the intrinsic types bool, int, float, double. One may use any user-defined type or class as the template parameter; however, one must also add traits to the framework to implement the desired data-parallel promotion properties so that Field operations work. The framework includes

Vektor<Dim, T>, Tenzor<Dim, T>, SymTenzor<Dim, T>

classes\(^1\), which are (mathematical) vectors, tensors, and symmetric tensors whose elements are of type T. Traits are implemented in these classes so that they may

\(^1\)The strange spellings avoid conflicts with other classes such as the STL vector class.
serve as elements of fully-functional Field objects. The Dim parameter represents the dimensionality of the Field that is being constructed. This must correspond to the Dim parameters in all other objects used to construct the Field. The Mesh parameter represents the mesh on which the field is discretized. IP²L pre-define two appropriate classes (Cartesian and UniformCartesian) to use for this parameter, one of which serves as the default value of the Field “Mesh” template parameter: UniformCartesian<Dim, double>. Refer to the IP²L User Reference for details on the UniformCartesian class; basically, it represents a Cartesian mesh with uniform grid spacings. The Cartesian class represents a cartesian mesh with nonuniform grid spacings. NB.: the type parameter MFLOAT for Cartesian represents only the data type used to store internal information like mesh spacing values; if double satisfies the user, he need not specify it.

The Centering parameter represents the centering of the field on its mesh. IP²L pre-defines Cell and Vert classes to represent cell and vertex centering, and has implementations of appropriate mechanisms for Cartesian and other classes which use them. IP²L also predefines a CartesianCentering class to represent more general centerings—combinations of vertex and cell centering direction-by-direction and component-by-component for Fields with multicomponent element types such as Vektor. Finally, IP²L predefines a wrapper class CommonCartesianCenterings with typedefs several common special cases to represent face and edge centerings, for example, refer to the IP²L User Reference for details.

### 5.1.2 Invoking the Field Constructor

There are six steps in the general construction of a Field:

1. Construct Index objects, one for each dimension of the Field. The Index objects describe the desired index domain along the axis.

2. Construct an NDIndex object with the dimensions of the Field. A single NDIndex object contains N Index objects, and fully describes the total index domain.

3. Populate the NDIndex with the Index objects created in step 1.

4. Construct a FieldLayout object with the NDIndex object. The FieldLayout object will control how the data of a specified Field object will be partitioned among physical nodes in a parallel environment.
5. If desired, construct BConds and GuardCellSizes objects for specifying boundary conditions and guard-cell layers, respectively. If unspecified, these default to no-op and zero.

6. Finally, construct a Field with the FieldLayout, BConds, and GuardCellSizes object as arguments to the constructor. This target Field must be parametrized as described in Section 5.1.1. The Dim template parameter must match the one for the FieldLayout and other objects involved, or you will get a compiler error.

For the cases of a 1, 2, or 3 dimensional Field, you may omit steps 2 and 3; instead directly pass the one, two, or three Index objects as arguments to the FieldLayout() constructor. The Dim template parameter must match the number of Index objects passed or you will get a compiler error.

The following code segment demonstrates the construction of a single two dimensional Field of double’s using the six-step method described above:

**Code Listing**

```cpp
unsigned Dim = 2;
int Nx = 100, Ny = 50;
Index I(Nx), J(Ny); // Step 1
NDIndex<Dim> domain; // Step 2
domain[0] = I; // Step 3
domain[1] = J; // Step 3
FieldLayout<Dim> layout(domain); // Step 4
Field<double, Dim> A(layout); // Step 5
```

The following three examples show the construction of a 3 dimensional Field without the intermediate NDIndex construction:

```cpp
Index I(100), J(5), K(25);
FieldLayout<3> layout(I,J,K);
Field<double, 3> A(layout);
```

You may also construct Field via copy constructor, wherein a Field is copied into another Field. This results in an element-by-element copy of the data:

```cpp
// assuming we have constructed a 2D Field of doubles in A
A = 2.0;
Field<double, Dim> B(A);
// B now contains the values 2.0 everywhere
```

### 5.2 The Index Class

The Index class represents a strided range of indices, and it is used to define the index extent of Field objects on construction and to reference subranges within Field’s in expressions. The constructor for Index takes one, two or
three int arguments. In the case of three arguments, these represent the base index value, the bounding index value and the stride. The two and one-argument cases are simplifications, with the one-argument case being qualitatively different; in particular,

Index I(8);

instantiates an Index object representing the range of integers from 0 to 7 inclusive, with implied stride 1. The two-argument

Index J(2,8);

instantiates an Index object representing the range of integers [2, 8], with implied stride 1. The three-argument

Index K(1,8,2);

instantiates an Index object representing the range of integers [1, 8], with stride 2; that is the ordered set \{1, 3, 5, 7\}.

Note that the single argument in the one-argument case defines the number of elements, rather than the bound. This means that Index J(8), which represents [0, 7], is different than Index J(0,8) and Index J(0,8,1), which both mean [0, 8].

As illustrated in Section 5.1.2, you use Index’s in constructing the FieldLayout object which goes into the Field constructor. The sizes of the Index’s used to construct the FieldLayout determine the size of the Field in each dimension; here size means the number of integers in the range represented by the Index. For example, the following code segment instantiates a 3-dimensional Field A having size 5 in the first dimension, size 9 in the second dimension, and size 4 in the third dimension:

Code Listing

```c
unsigned Dim = 3;
int N = 5;
int Ny = 9;
int Nz = 4;
Index I(Nx), J(Ny), K(Nz);
FieldLayout<Dim> layout(I,J,K);
Field<double, Dim> A(layout);
```

You can also use Index objects for initializing Field elements with integer ranges of values. This and more typical use of Index object in conjunction with Field objects is discussed in Section 5.7.2.

Finally, IP^2L defines various operators on Index objects, mostly used to represent finite-different stencil operations on Field’s, as described in Section 5.7.2. If

Index I(8);

is an Index object representing [0, 7], then the expression
I - 1
represents the range of the same length offset by −1, or [−1, 6]. Similarly, the expression
I + 1
represents [2, 8].

5.3 The NDIndex Class

The NDIndex class is primarily a container which holds N Index objects. It is templated on the spatial dimension N, and the constructor takes N Index arguments. For example:

```cpp
Index I(5), J(9), K(4);
NDIndex<3> Domain(I, J, K);
```

An NDIndex object appears as an array of Index objects; you may access the Index object for and dimension using the [] operator. For example:

```cpp
Index tmpJ = Domain[1];
```

5.4 The FieldLayout Class

FieldLayout is the class responsible for determining where the data in a Field object is located. It is templated on the number of indices for the Field; when constructing a new FieldLayout object, you must tell it what is the index range for each dimension (or axis). A single NDIndex object may be used as the argument to a new FieldLayout instance:

```cpp
Index I(5), J(9), K(4);
NDIndex<3> Domain(I, J, K);
FieldLayout<3> Layout(Domain);
```

Or, possibly more conveniently, you may just specify the N Index objects to the constructor of the FieldLayout directly, without explicitly creating an NDIndex object:

```cpp
Index I(5), J(9), K(4);
FieldLayout<3> Layout(I, J, K);
```

5.4.1 Specifying Serial or Parallel Layout

By default, a FieldLayout object will partition all N dimensions in a parallel fashion. For example a 2D Field with indices running from 0...5 in each dimension, created with a FieldLayout specified as follows:
Index I(6), J(6);
FieldLayout<3> Layout(I, J);

will have both the I and J indices partitioned across the nodes in parallel. This
would lead to a layout something like that shown in the following figure, if there
are four nodes:

TODO: BILD

Unless you tell it otherwise, FieldLayout will attempt to distribute the data
among the processors by subdividing each dimension in turn until it has the proper
number of subregions. Those axes which are considered for subdivision are the
parallel axes, which means that a given node will only contain Field data for a
subset of the indices along that dimension. You can, however, tell FieldLayout
which axes to subdivide, and which to maintain as serial. Serial axes are not ever
partitioned by FieldLayout. You must have at least on parallel dimension in a
given FieldLayout; by default, all axes are parallel.

To specify serial axes, you provide additional arguments to the FieldLayout
constructor, using the keywords SERIAL or PARALLEL. If you create a new
FieldLayout by just specifying N Index objects, then you may provide up to
N more arguments to the constructor to set the corresponding dimension’s layout
method. For example, we may change the earlier example of a 2D Field to have
the second dimension use a serial layout as follows:

Index I(6), J(6);
FieldLayout<3> Layout(I, J, PARALLEL, SERIAL);

In this case, the data would be partitioned into four subregions like the following
(the horizontal direction is the first dimension, the vertical direction the second)

TODO: BILD

If an NDIndex object is used to create the FieldLayout instead of several
Index objects, to change the default layout style you must instead provide an ar-
day of keywords (of type e_dim_tag) specifying the layout for the N dimensions.
For example:

Code Listing

Index I(5), J(9), K(4);
NDIndex<3> Domain(I, J, K);
e_dim_tag ParallelMethod[2];
ParallelMethod[0] = PARALLEL;
ParallelMethod[1] = SERIAL;
ParallelMethod[2] = SERIAL;
FieldLayout<3> Layout(Domain, ParallelMethod);

5.5 Boundary Condition Classes

One of the great frustrations in using data parallel objects is the proper represen-
tation of boundary conditions. Most data parallel environments (such as HPF or
CMFortran) require that one perform special operations to observe periodic or reflected behaviour at the boundary. This requirement obscures the original clarity of the index notation. You construct Field objects within the IP2L framework using BConds boundary condition object which defines the behaviour of Field and Field indexing operations at the boundaries. This makes the same, clear indexing notation do the right thing under a variety of imposed boundary conditions.

5.5.1 Available Boundary Conditions
IP2L pre-defines classes to represent 6 different forms of boundary conditions:

1. Periodic boundary condition: PeriodicFace
2. Positive reflecting boundary condition: PosReflectFace
3. Negative reflecting boundary condition: NegReflectFace
4. Constant boundary condition: ConstantFace
5. Zero boundary condition (special case of constant): ZeroFace
6. Linear extrapolation boundary condition: ExtrapolateFace

They represent boundary conditions for a single dimension of a (possibly) multidimensional field, for one “side” (or face) of the mesh along that dimension. That is, you must specify two boundary condition objects for each dimension of the Field – one for each face of the mesh along that dimension. These classes are parametrized on the same four template parameters as Field (see Section 5.1.1); the defaults for Mesh and Centering are UniformCartesian and Cell. As a further refinement, you may specify boundary conditions for individual components of multicomponent Field elements such as Vektor.

5.5.2 Using Boundary Conditions With Fields
The BConds class is a container for the individual specialized boundary conditions; this is the argument passed to the Field constructor. A BConds object acts very much like an array of boundary conditions: when first created, the BConds object is empty, and you add new boundary condition objects to it by treating it as a vector and assigning to its elements. The basic procedure is to construct a BConds object, then construct new or use existing boundary condition objects (from the list above) to fill it, as illustrated in this example:

Code Listing
unsigned Dim = 2;
Index I(4), J(4);
BConds<

Field Layout<Dim> layout(I, J);
Field <double, Dim> A(layout, GuardCellSizes<Dim>({1}, bc);

Again, the individual face boundary condition objects (in this example, PeriodicFace) perform their task for only a single face of the mesh. In this way, there may be different types of boundary conditions in different dimensions. The face boundary-condition constructors take an unsigned argument designating the face according to the following numbering convention: The integers 0 and 1 apply to the boundaries of the first coordinate direction where 0 represents the negative face and 1 represents the positive face. The integers 2 and 3 apply to the boundaries of the second coordinate direction where 2 represents the negative face and 3 represents the positive face. This pairing of integers and domains continues into higher dimensions. The constructors also take optional second and third unsigned parameters to specify a single Field element component rather than all of them. Refer to the IP3L User Reference for more details on these classes, and a detailed discussion of how the various boundary conditions affect Field operations.

5.5.3 Default Boundary Condition

If a Field is constructed with no BConds object specified, the default is for that Field to have NO boundary conditions. In that case, the boundary conditions container within the Field is empty. It is possible to add additional boundary conditions for a specific face to a Field after it has been constructed; to do so, retrieve the boundary condition container from the Field using the method Field::getBConds(), and then add new face-specific boundary conditions to the returned BConds container object as shown in the previous example.

5.6 The GuardCellSizes Class

A GuardCellSizes class, an optional argument to the Field constructor, represents the maximum separation (in elements) of Field elements which will be combined in Field expressions. Typically, this reflects the order of finite differencing in stencil operations. The primary reason for guard cells is parallelism – a Field domain-decomposed into multiple subdomains with data from adjacent subdomains, so that the stencil operations have all required data locally. The GuardCellSizes class is parameterized on the unsigned value Dim, which
represents the number of dimensions of the `Field` object. This `Dim` value must match the corresponding parameter value of the `Field` object.

The constructor for `GuardCellSizes` takes either one or two arguments, which are either `unsigned` or `unsigned*`. The one-argument forms specify the same number of guard layers for all dimensions; the two-argument forms specify different numbers for the right and left faces; the unsigned forms specify the same number of layers for all dimensions; and the `unsigned*` forms specify different number of layers for the different dimensions:

```
GuardCellSizes(unsigned s);  // Same no. left&right, same for all directions
GuardCellSizes(unsigned *s); // Same no. left&right, value for each direction
// Diff. left&right, same for all directions
GuardCellSizes(unsigned l, unsigned r); // Diff. left&right, value for each direction
GuardCellSizes(unsigned *l, unsigned *r);
```

Section 5.7.2, “Using Index Objects with Field’s”, shows examples using `Field` indexing to implement stencil operations. It discusses the numbers of guard layers required by one of the examples.

5.7 Operations on Field Objects

5.7.1 Assignment

A single line of code which contains an assignment operator and a `Field` on the left hand side of the assignment operator is called a `Field` expression. Many different terms may appear on the right-hand side of a `Field` expression (or as the second argument in an `assign()` call as described below). These include scalars, `Index`'s, `Field`'s and `IndexingField`'s. Currently because of the lack of template member functions in C++ compilers, you must use the `assign()` function rather than the `operator=`:

```
assign(Lhs, Rhs);
```

where `Lhs` and `Rhs` are `Field` expressions. When template member functions become available, you will simply write:

```
Lhs = Rhs
```

Refer to the IP2L Users Reference for more details, and examples showing where you may use `operator=` and where you must use `assign()`. The following are examples of legal assignments:

```
unsigned Dim = 2, int N = 100;
Index I(N), J(N);
FieldLayout<Dim> layout(I, J);
Field<double, Dim> A(layout), B(layout), C(layout);
```
\[ A = 2.0; \]
\[ \text{assign}(A, 2.0 + B); \]
\[ \text{assign}(B, A + 2.0); \]
\[ \text{assign}(B[I][J], 3.0 + B[I][J]); \]
\[ \text{assign}(A[I][J], I + A[I][J]/C[I][J]); \]

For cases where more than one term exists on the right hand side of an assignment, the \text{assign()} call must be made. Any combination of scalars, Field's, IndexingField's (indexed Field objects; see Section 5.7.2), and Index's can be put as the second argument of the \text{assign()} call. The only requirement in combining terms is that the appearance of an Index object anywhere inside of an expression requires that all the Field objects contained in the expression must be indexed. It is not possible to combine Field's and IndexingField's in a single expression. Nor is it possible to combine Field's and Index objects in a single expression.

Another intermediate solution to accommodate the lack of member function templates (and therefore the ability to use the \text{operator=} member function with more than one term on the right hand side of an expression) is the utilization of the accumulation operators (since this does not require member function templates). Thus, instead of writing
\[ \text{assign}(A, 4.0 + B); \]
one could write
\[ A = 0.0; \]
\[ A += 4.0 + B; \]

This technique can be used with any of the accumulation operators (+, -, *, /).

### 5.7.2 Using Index Objects with Field's

The Field object works intimately with Index objects to perform a wide variety of operations. Use the Index object to specify the access pattern into a data parallel Field object. Do this by using Index objects inside the brackets following a Field object as follows:
\[ A[I][J] = B[I][J]; \]

A Field object followed by brackets containing Index objects is called an IndexingField, because IP\(^2\)L internally uses an IndexingField class as the return value for the Field::operator[].

You can use Index objects to initialize a Field with integer range data – that is, assign to a strided range of Field elements the values of a strided range of integers multiplied by the element type. This only works of multiplication by an int is defined for the Field element type, which it is for the intrinsic types \{int, float, double, bool\} and the IP\(^2\)L pre-defined Field element.
classes \{Vektor, Tenzor, SymTenzor\}. For multidimensional Field's, the range of values is replicated along the other dimensions. For example, given a Field that is size 8 in its first dimension and size 4 in its second dimension, the code segment
assign(A[I][J], I);
produces the following values in the Field A:

```
  0 1 2 3 4 5 6 7
  0 1 2 3 4 5 6 7
  0 1 2 3 4 5 6 7
  0 1 2 3 4 5 6 7
```

(Here, as in subsequent figures like this displaying Field values, the positive direction of the first coordinate is from left to right and the positive direction for the second coordinate is from top to bottom.) Likewise, an assignment of the form
assign(A[I][J], J);
produces

```
  0 0 0 0 0 0 0 0
  1 1 1 1 1 1 1 1
  2 2 2 2 2 2 2 2
  3 3 3 3 3 3 3 3
```

The Index objects used to access ranges of values in a Field object do not have to be the same Index’s used in constructing the FieldLayout object used to construct the Field. You can use Index objects of smaller size to access a subrange of the Field. For example, if we wanted to have an 8 by 8 Field with zeros everywhere except for a 4 by 4 subregion in the center, the following code segment would accomplish this goal:

**Code Listing**

```cpp
unsigned Dim = 2;
Index I(8), J(8);
Index I2(2, 5), J2(2, 5);
FieldLayout<Dim> layout(I, J);
```
Field<double, Dim> A(layout);
A = 0.0;
A[1][2] = 1.0;

This would produce the following values in the Field A:

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0
0 0 1 1 1 1 0 0
0 0 1 1 1 1 0 0
0 0 1 1 1 1 0 0
```

The lower-limiting case range for an Index object is, of course, a single element. For this case you can just use an integer constant or variable; the following assigns a single element of the Field A:

```
Index I(4), J(4);
FieldLayout<2> layout(I, J);
Field<double, 2> A(layout);
A = 0.0;
A[1][1] = 1.0;
```

The resultant Field A contains the values:

```
0 0 0 0
0 1 0 0
0 0 0 0
0 0 0 0
```

The typical use for indexing is stencil operations, using Index expressions adding or subtracting integer constants to represent the finite differences. This
amounts to global data transformation upon a Field through the use of Index operations. For example, if a 4 by 4 Field named A is initialized as follows:

```cpp
unsigned Dim = 2;
int N = 4;
Index I(N), J(N);
FieldLayout<Dim> layout(I, J);
Field<double, Dim> A(layout, GuardCellSizes<Dim>(1));
assign(A[I][J], I + 1);
```

then the values in the Field A will be:

```
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

Now, let’s form another Field, B, and assign to it the value of an IndexingField of A which represents an indexed operation:

```cpp
Field<double, Dim> B(layout);
assign(B[I][J], A[I+1][J]);
```

Here we see that the Field A has been indexed with something other than a plain Index object. Rather, it has been indexed by an index expression. The Index objects have been overloaded to allow addition and subtraction by integers to produce other Index objects. The framework recognizes this operation as requesting that all the data in A be shifted to the left (along the first dimension in the negative direction) by 1 position. The Field B contains the values:

```
2 3 4 0
2 3 4 0
2 3 4 0
2 3 4 0
```

Indexing operations which access data beyond a Field boundary set the target positions to zero. For the reminder of this section, we shall assume this
zero valued boundary condition (which is the default condition when no boundary condition is specified). A variety of boundary conditions can be set on each boundary of a Field and are discussed in detail in the next section.

The GuardCellSizes object used to construct Field A in this example must specify at least one guard layer in the 1st dimension, to accommodate the “+1” in the indexing operation. The one used, GuardCellSizes<Dim>(1) allows “+/−1” indexing (as in a width-one stencil), and also allows width-one stencils in the 2nd dimension, because the use of the unsigned argument (the constant, 1) specifies one guard layer both left and right for all directions.

Had we wished to shift the Field A down (along the second dimension in the negative direction) we could have written

assign(B[I][J], A[I][J+1]);

Then the values in the Field B are:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

You can shift in the positive or negative direction on any Index object used to index a Field. For example,

Code Listing

```cpp
unsigned Dim = 2;
int N = 4;
Index I(N), J(N);
FieldLayout<Dim> layout(I, J);
Field<double, Dim> A(layout, GuardCellSizes<Dim>(1));
assign(A[I][J], I + J + 1);
```

will initialize the values in the Field A to:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>
and the operation

```c++
Field<double, Dim> B(layout);
assign(B[I][J], A[I+1][J-2]);
```

will produce a Field $B$ with the values:

```
0 0 0 0
0 0 0 0
3 4 5 0
4 5 6 0
```

### 5.7.3 Overloaded operators

IP2L pre-defines a suite of overloaded operators with the `Field` class. These include the unary $-$ operator; the binary operators, $+$, $-$, $*$, and $/$; and the accumulation operators $+=, -=, *=, /=$. Traits [?] determine the appropriate casts and promotions of mixed types inside Field. For example, a Field of int’s added to a Field of double’s would perform the correct promotion of int to double element by element. As mentioned earlier, the assign operator $=$ does not work for most cases because of the lack of member function templates. In addition, the relational operators are not directly available due to conflicts in the current HP reference STL implementation. This functionality is provided through the explicit inlined binary function calls:

<table>
<thead>
<tr>
<th>binary function relationals</th>
<th>corresponding relation operator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gt(A, B)</code></td>
<td>$A &gt; B$</td>
</tr>
<tr>
<td><code>lt(A, B)</code></td>
<td>$A &lt; B$</td>
</tr>
<tr>
<td><code>ge(A, B)</code></td>
<td>$A &gt;= B$</td>
</tr>
<tr>
<td><code>le(A, B)</code></td>
<td>$A &lt;= B$</td>
</tr>
<tr>
<td><code>eq(A, B)</code></td>
<td>$A == B$</td>
</tr>
<tr>
<td><code>ne(A, B)</code></td>
<td>$A != B$</td>
</tr>
</tbody>
</table>
The return value of these binary relational functions is a conforming Field of bool's. Here is an example using binary functional relationals in an expression:

**Code Listing**

```c
unsigned Dim = 2;
Index I(8), J(4);
FieldLayout<Dim> layout(I, J);
Field<double, Dim> A(layout), B(layout), C(layout);
A = 0.0;
assign(B[I][J], J);
assign(C[I][J], I);
assign(A, lt(B, 2, 0)*C);
```

The resulting Field A contains the values:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5.7.4 The `where()` Function

Data parallel simulations often require element-by-element conditionals. The IP2L framework provides a `where()` function which reduces to the inlined conditional operator?: for each element of the Field objects passed to the `where()` function. The `where()` function takes three Field arguments:

```c
assign(A, where(B, C, D));
```

where the Field B’s a Field of bool’s. The value of C is placed into A everywhere that B is true, and the value of D is placed into A everywhere that B is false. Thus,

**Code Listing**

```c
unsigned Dim = 2;
Index I(4), J(4);
FieldLayout<Dim> layout(I, J);
Field<double, Dim> A(layout), B(layout), C(layout);
assign(B[I][J], I - 1);
C = 1.0;
assign(A[I][J], where( lt(B, C), B, C));
```

leaves the following values in Field A:
Since `where()` returns a `Field`, invocations of `where()` may be used as arguments to `where()`; this allows nested element-by-element conditionals. The following example code

**Code Listing**

```cpp
unsigned Dim = 2;
Index I(4), J(4);
FieldLayout<Dim> layout(I, J);
Field<double, Dim> A(layout), B(layout), C(layout), D(layout);
assign(B[I][J], I - 1);
assign(C[I][J], J - 1);
D = 1.0;
assign(A[I][J], where( lt(B, D), B, where( lt(C, D), C, D)));
```

leaves the following values in Field `A`:

```
-1  0  1  1
-1  0  1  1
-1  0  1  1
-1  0  1  1
```

### 5.7.5 Mathematical Functions on Field’s

As would be expected of any framework for scientific simulation, all the standard mathematical operations are included. The unary functions take a `Field` object and return a `Field` object of the same dimension and size where the unary operation has been performed upon each element of the `Field`. The binary functions take two conforming `Field`’s and apply the function pairwise to each member of the two `Field`’s to produce a new conforming `Field` containing the resultant values. The following functions, mirroring those in math.h, are available in the framework for `Field` operations:

- `acos`, `asine`, `atan`, `cos`, `sin`, `tan`, `cosh`, `sinh`, `tanh`,
- `exp`, `log`, `log10`, `pow`, `sqrt`, `ceil`, `fabs`, `floor`. 

58
For machines which provide them in math.h, IP2L provides the Field version of the Bessel, gamma, and error functions erf, erfc, gamma, j0, j1, y0, y1.

5.7.6 Reduction Operations

IP2L includes several reduction operations with the Field class. These include determining the maximum and minimum elements of a Field, the global sum and product of all the elements in a Field, and determining the location of the minimum and maximum values within a Field (typically called minloc and maxloc).

**WARNING:** Currently, these functions are only implemented on Field objects; they will not work on Field expressions. This means that invocations like min(A+B) and min(2.0*A) are illegal!

The following exampled code, which demonstrates the usage of these operators,

Code Listing
```
unsigned Dim = 2;
Index I(10), J(10);
FieldLayout<Dim> layout(I, J);
Field<double, Dim> A(layout);
assign(A[I][J], I + J);
cout << min(A) << endl;
cout << max(A) << endl;
cout << sum(A) << endl;
cout << prod(A) << endl;
```

produces the following output:

```
0
18
900
0
```

The minloc and maxloc capabilities, rather than being separately named functions, are two-argument forms of the min() and max() functions. The second argument is an NDIndex<Dim> object, which is a multi-dimensional container for Index objects. The minloc and maxloc operations fill an NDIndex<Dim> object with one Index object for each dimension; each Index is of size one, representing a single point. The following code segment demonstrates:

Code Listing
```
unsigned Dim = 2;
Index I(10), J(10);
FieldLayout<Dim> layout(I, J);
Field<double, Dim> A(layout);
assign(A[I][J], cos((I-2)*(I-2) + (J-2)*(J-2)));
NDIndex<Dim> LocMin, LocMax;
min(A, LocMin);
max(A, LocMax);
```
The NDIndex<Dim> objects LocMin and LocMax now contain the position (index location) of the minimum and maximum elements of the Field object A.
Appendix A

Fields

The Field class represents the common computational science abstraction of a continuum (mathematical) field discretized on a mesh, with some centering on that mesh. Used in its simplest, default, way with basic-type elements such as double’s or float’s. Field serves as a multidimensional array class. Similarly, you can use collections of simple or not-so-simple instances of the Field class to represent almost any data structure found in computational science; but this is against the philosophy of object-oriented design, in which you should design classes to represent the physics/numerical-mathematics abstractions of the problem domain, rather than the data structures typically found in computer codes.

A.1 Field Class Definition

The Field class is parameterized on the type $T$ of elements (typically a type like double, Vektor, or Tenzor), dimensionality, mesh type, and centering on the mesh:

```cpp
template<class T, unsigned Dim, class Mesh=Cartesian, class Centering=Mesh::DefaultCentering>
class Field : public FieldBase
```


A.2 Field Constructors

To instantiate a Field we use the following of Field constructors:

Field(FieldLayout<Dim>&);
Field (FieldLayout<Dim>&, const GuardCellSizes<Dim>&);
Field (FieldLayout<Dim>&, const BConds<T,Dim,Mesh,Centering>&);
Field (FieldLayout<Dim>&, const GuardCellSizes<Dim>&,
       const BConds<T,Dim,Mesh,Centering>&);
Field (FieldLayout<Dim>&, const BConds<T,Dim,Mesh,Centering>&,
       const GuardCellSizes<Dim>&);

A.3 Field Member Functions and Member Data

IndexingField<T,Dim,l,Mesh,Centering> operator[] (const Index& idx)
IndexingField<T,Dim,l,Mesh,Centering> operator[] (int i)
const iterator& begin() { return Begin; }
const iterator& end() { return End; }
void fillGuardCells();
const GuardCellSizes<Dim>& getGuardCellSizes() { return Allocated;

// Boundary condition handling.
unsigned leftGuard(unsigned d) { return Allocated.left(d); }
unsigned rightGuard(unsigned d) { return Allocated.right(d); }
const Index&: getIndex(unsigned d) { return Layout˜>get_Domain() [d];
const NDIndex<Dim>& getDomain() {returnLayout˜>get_Domain();}

// Definitions for accessing boundary conditions.
typedef BCondBase<T,Dim,Mesh,Centering> bcond_value;
typedef BConds<T,Dim,Mesh,Centering> bcond_container;
typedef bcond_container::iterator bcond_iterator;
bcond_value& getBCond(int bC);
bcond_container& getBConds() {return *BC;}

A.4 Operations on Field Objects

A.4.1 Assignment

For the special case where there is only one term on the right-hand side of an
assignment, the assignment operator can be utilized. Examples of single term
assignments include:

unsigned Dim = 2, int N = 100;
Index I (N), J (N);
FieldLayout<Dim> layout(I,J) Field<double,Dim> A(layout), B(layout);
A = 2.0;
B = A;

For cases where more than one term exists on the right hand side of an assign-
ment, the assign() call must be made. Any combination of scalars, Field’s,
IndexingField’s, and Index’s can be put as the second argument of the
assign() call. The only requirement in combining terms is that the appearance of an Index object anywhere inside of an expression requires, that all the Field contained in the expression must be indexed. It is not possible to combine Field’s and IndexingField’s in a single expression. Nor is it possible to combine Field’s and Index objects in a single expression. The following examples define legal expressions:

```cpp
unsigned Dim = 2;
int N = 100;
Index I(N), J(N);
FieldLayout<Dim> layout(I,J);
Field<
double, Dim> A(layout), B(layout), C.(layout);
assign(A, 2.0 + B);
assign(B, A + 2.0);
assign(B[I][J], 3.0 + B[I][J]);
assign(A[I][J], I + A[I][J]/C[I][J]);
```

The following are examples of illegal expression:

```cpp
B[I][J] = 3.0 + B[I][J]; // must use assign() with indexed Field’s,
assign(B, 2.0 + B[I][J]); // can’t combine indexed Band non-indexed B
assign(B[I][J], A + 2.0); // can’t combine indexed Band non-indexed A
assign(A[I][J], I + A/C[I][J]); // can’t combine indexed C and non-indexed A
```

A.4.2 Boundary Conditions

IP²L pre-defines classes to represent 7 different forms of boundary conditions:

1. Periodic boundary condition: PeriodicFace
2. Positive reflecting boundary condition: PosReflectFace
3. Negative reflecting boundary condition: NegReflectFace
4. Constant boundary condition: ConstantFace
5. Zero boundary condition (special case of constant): ZeroFace
6. Linear extrapolation boundary condition: ExtrapolateFace
7. none (should not be used)

Let’s examine each boundary condition as applied to the same shift operation. In each case, the first assign() invocation shows how the Field A with the following values:

```
63
```
The results after the second `assign()` invocation show how the boundary conditions on \( A \) affect the calculation. For the first example, consider the case where each boundary of the `Field` object \( A \) has periodic boundary conditions:

```cpp
unsigned Dim = 2;
Index I(4), J(4);
BConds<double,Dim> bc;

bc[0] = new PeriodicFace<double,Dim>(0);
bci[1] = new PeriodicFace<double,Dim>(1);
bci[2] = new PeriodicFace<double,Dim>(2);
be[3] = new PeriodicFace<double,Dim>(3);

FieldLayout<Dim> layout(I,J);
Field<double, Dim> A( layout, GuardCellSizes<Dim> (1) , bc) ;
Field<double, Dim> B(layout);
assign(A[I][J] ,I + J);
assign(B[I][J] ,A[I+1][J+1])
```

This code segment produces the following values in the `Field` \( B \):

```
2 3 4 1
3 4 5 2
4 5 6 3
1 6 3 0
```

In the case of the periodic boundary conditions, we see that the values wrap around the domain of the `Field` and pull values from the opposite side of the `Field` when the indexing operations reference positions outside the domain. Note that the specification of a boundary condition above overrides the default behavior, which places zeroes into positions that attempt to obtain data from outside the domain.

Next we consider the case where positive reflecting boundary conditions are applied to each boundary of the `Field` object \( A \) in the same shift operation:
bc[0] = new PosReflectFace <double,Dim>(0);
b[0] = new PosReflectFace <double,Dim>(1);
bc[2] = new PosReflectFace <double,Dim>(2);
be[3] = new PosReflectFace <double,Dim>(3);

assign(A[I][J] , I + J);
assign(B[I][J] , A[I+1][J+1]);

This code segment produces the following values in Field B:

```
  2  3  4  4
  3  4  5  5
  4  5  6  6
  4  5  6  6
```

In the case of the positive reflecting boundary conditions, we see that the values are simply reflected across the boundary over which the Field indexing operation occur. This boundary condition is meant to represent Neuman boundary conditions in physical systems.

Next, consider the case where each boundary of the Field object A has negative reflecting boundary conditions:

bc[0] = new NegReflectFace <double,Dim>(0);
b[1] = new NegReflectFace <double,Dim>(1);
bc[2] = new NegReflectFace <double,Dim>(2);
be[3] = new NegReflectFace <double,Dim>(3);

assign(A[I][J] , I + J);
assign(B[I][J] , A[I+1][J+1]);

This code segment produces the following values in Field B:

```
  2  3  4  -4
  3  4  5  -5
  4  5  6  -6
 -4 -5 -6 -6
```

In the case of the negative reflecting boundary conditions, we see that the values are simply reflected across the boundary over which the Field indexing operation occur and negated. This boundary condition is meant to represent Dirichlet boundary conditions in physical systems.
Next, consider the case where each boundary of the \texttt{Field} object \texttt{A} has constant boundary conditions:

\begin{verbatim}
bc[0] = new ConstantFace <double,Dim>(0,9.0);
b[1] = new ConstantFace <double,Dim>(1,9.0);
b[2] = new ConstantFace <double,Dim>(2,9.0);
b[3] = new ConstantFace <double,Dim>(3,9.0);
assign(A[I][J], I+ J);
assign(B[I][J], A[I+1][J+1]);
\end{verbatim}

Note additional argument to the \texttt{constantFace} constructor. This argument represents the value which is to be fixed on the boundary in that direction. The code segment above produces the following values in \texttt{Field B}:

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
2 & 3 & 4 & 9 \\
\hline
3 & 4 & 5 & 9 \\
\hline
4 & 5 & 6 & 9 \\
\hline
9 & 9 & 9 & 9 \\
\hline
\end{tabular}
\end{center}

In the case of the constant boundary conditions, we see that a fixed value is shifted into the domain across the boundary over which the \texttt{Field} indexing operation occur.

Finally, consider the case where the \texttt{Field} object \texttt{A} has mixed boundary conditions, boundary-by-boundary (i.e., face-by-face):

\begin{verbatim}
be[0] = new PeriodicFace <double,Dim>(0);
be[1] = new PeriodicFace <double,Dim>(1);
be[2] = new PosReflFace <double,Dim>(2);
be[3] = new NegReflFace <double,Dim>(3);
assign(A[I][J], I+ J);
assign(B[I][J], A[I+1][J+1]);
\end{verbatim}

The code segment above produces the following values in \texttt{Field B}:

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
2 & 3 & 4 & 1 \\
\hline
3 & 4 & 5 & 2 \\
\hline
4 & 5 & 6 & 3 \\
\hline
-4 & -5 & -6 & -3 \\
\hline
\end{tabular}
\end{center}
Appendix B

Index Class

The Index class represents a strided range of integer indices (described by base, bound, and stride integer values). You use it to define the size (index extent) per dimension of Field objects on construction and to specify subsets of Field elements along a dimension in Field expressions.

It is important to note that the actual memory address of an Index object is relevant to whether that object may be used interchangeably with another Index object specifying the same index values. In fact, two Index objects specifying the same index values are not interchangeable. This is qualitatively different than the semantics of Fortran 90 array-syntax, for example. You may construct a ”new” Index object with a different name, but which has the same values and is interchangeable with a given Index by setting it equal to the first Index object on construction.

B.1 Index Definition

class Index {
public:
  // Public data member -- iterator class: class iterator
{
public:
  iterator() : Current (0) , Stride(0) {}
  iterator(int current, int stride=1) : Current (current) , Stride(stride) {}
  int operator*();
  iterator operator--(int); // Post decrement iterator& operator--();
  iterator operator++(int); // Post increment iterator& operator++();
  iterator& operator+=(int);
  iterator& operator-=(int);
  iterator operator+(int) const;
  iterator operator-(int) const;
  int operator[](int);
  bool operator==(const iterator&) const;
  bool operator<(const iterator&) const;
  bool operator!=(const iterator&) const;
}
bool operator> (const iterator &); const;
bool operator><(const iterator &); const;
bool operator== (const iterator &); const;

private:
int Stride;
int Current;
);

// Public member functions.
// Constructors:
Index () ; // null range
inline Index(unsigned n); // [0 .. n-1]
inline Index(int f, int l); // [f .. l]
inline Index(int f, int l, int n); // First to Last using Step.

// Destructor
~Index () {}; // Don’t need to do anything.

int id () { return 1 ; }

inline int min() const; // the smallest element.
inline int max() const; // the largest element.
inline int length() const; // the number of elems.
inline int stride() const; // the stride.
inline int first() const; // the first element.
inline int last() const; // the last element.
inline bool empty() const; // is it empty?
inline const Index* getBase() const; // the base index

// Additive operations.
friend inline Index operator+ (const Index&, int);  
friend inline Index operator+(int,const Index&);
friend inline Index operator-(const Index&,
friend inline Index operator*(const Index&,int);
friend inline Index operator*(int,const Index&);
friend inline Index operator/(const Index&,

// Multiplicative operations.
friend inline Index operator*(const Index&);
friend inline Index operator/(const Index&,
friend inline Index operator%(const Index&,
friend inline Index operator^(const Index&,

// intersect with another Index.
Index intersect(const Index&) const;

// Plug the base range of one into another.
Index plugBase(const Index&) const;

// Test to see if two indexes are from the same base.
inline bool sameBase(const Index&) const;

// Test to see if there is any overlap between two indexes.
inline bool touches(const Index& a) const;

// Test to see if one contains another.
inline bool contains(const Index& a) const;

// Split one into two.
inline bool split (Index&, Index&) const;

// iterator begin
iterator begin() { return iterator(First, Stride); }


B.2 Index Constructors

The constructor for Index takes one, two, or three in arguments. In the case of
three arguments, these represent the base index value, the bounding index value,
and the stride. The two and one-argument cases are simplifications.

Index I(8);

instantiates an Index object representing the range of integers from 0 through 7
(i.e. [0,7]) with implied stride 1. The two-argument

Index J(2,8);

instantiates an Index object representing the range of integers [2,8] with implied
stride 1. The three-argument

Index(1,8,2);

instantiates an Index object representing the range of integers [1,8] with stride 2-
.. that is, the ordered set 1, 3, 5, 7.

Note that the single argument in the one-argument case defines the number of
elements, rather than the bound. This means that Index J(8), which repre-
sents [0,7], is different than Index J(0,8) and Index J(0,8,1), which
both mean [0,8].

There is also a special special constructor taking no arguments; this is meant
for use in constructing arrays of Index's.
B.3 Index Member Functions and Member Data

B.3.1 Index iterator

The only public member data is the Index::iterator class. This class has the semantics of an STL random access iterator; the advanced user can use it to iterate over integer index values represented by the containing Index object. The STL semantics means that the class provides increment and decrement operators, increment/decrement by specified integer amounts, deference operator, random-access operator[], and comparison operators. See the class definition in Section B.1 for the full list. The iterator class definition is contained within.

```cpp
Index::iterator begin ()
```

Returns an Index: : iterator positioned at the beginning of the index values represented by the Index object.

```cpp
Index::iterator end ()
```

Returns an Index::iterator positioned beyond the last index value represented by the Index object.

B.3.2 Index Query/Accessor Functions

These functions mostly return values of or values computed from private data members of the Index.

```cpp
int min () const
```

Returns the smallest integer value allowed for the Index object.

```cpp
int max () const
```

Returns the largest integer value allowed for the Index object.

```cpp
int length() const
```

Returns the total number of integer values spanned by the Index object.

```cpp
int stride()const
```

Returns the value of the stride for the Index object.

```cpp
int first() const
```

Returns the integer value of the first element in the Index object.

```cpp
int last () const
```

Returns the integer value of the last element the Index object.

```cpp
bool empty() const
```

70
Returns true/false depending on whether the Index object is empty. Empty (true) means, that it was constructed with the zero-argument constructor.

```cpp
const Index* getBase() const
```

Returns a pointer to the base Index object associated with the Index object. As described at the beginning of this chapter, the memory address of an Index is important, and if two Index's are to be interchangeable they must share the same base address as well as be conforming (have the same base, bound, and stride values).

### B.3.3 Index Arithmetic operator Functions

Here we describe the arithmetic operator functions defined in the Index class to act on Index objects. In the example code in these descriptions, I is an Index object and n is an int.

```cpp
friend Index operator+(const Index&, int)
```

The Index expression I + n invokes this operator. It adds an integer value n to the base and bound values of I, then constructs and returns the resulting Index object using those revised values.

```cpp
friend Index operator+(int, const Index&)
```

Same as the previous, except for the order of the operands. That is, the Index expression n + I invokes this operator.

```cpp
friend Index operator-(const Index&, int)
```

Invoked by I - n subtracts n from the base and bound of I and returns the resulting Index.

```cpp
friend Index operator-(int, const Index&)
```

Invoked by n - I subtracts the base of I from n, multiplies the stride by -1, and returns the resulting Index object.

```cpp
friend Index operator-(const Index&)
```

Negation operator, invoked by -I. This multiplies the base and bound of I by -1 and returns the resulting Index object.

```cpp
friend Index operator*(const Index&, int)
friend Index operator*(int, const Index&)
```

Invoked by I * n and n * I, respectively. These multiply the base, bound, and stride of I by n and return the resulting Index object

```cpp
friend Index operator/(const Index&, int)
```

Invoked by I / n. These divide the base, bound, and stride of I by n and return the resulting Index object (integer division, truncates). Note that the division operator with the operands the other way around (n / I) is not defined.
B.3.4 Index I/O and Message-Passing Functions

These are functions are to write out an \texttt{Index} object, and to pack/unpack and send/receive an \texttt{Index} object as a message between two processes.

\textbf{friend} \texttt{ostream} & \texttt{operator}<<(\texttt{ostream} &, \texttt{const Index} &)

Formatted insertion of the contents of an \texttt{Index} object into the output stream. The interface is the stream I/O \texttt{operator} \	extless{}, invoked by \texttt{os} \textless{} \texttt{I} (\texttt{os} is an ostream object, or a \texttt{IP2L Inform} object).

\texttt{static void} \texttt{findPut} (\texttt{const Index}, \texttt{const Index}, \texttt{const Index}, \texttt{Index}, \texttt{Index})

Need to describe ..... \texttt{Message} & \texttt{putMessage} (\texttt{Message})

Put \texttt{Index} data into a message to send to another node. \texttt{Message} & \texttt{getMessage} (\texttt{Message})

Get \texttt{Index} data out of a message you have received from another node.

B.3.5 Index Comparison Operators

\textbf{bool} \texttt{operator} < \texttt{const Index} \texttt{) const;

A less-than test so we can impose some sort of ordering of two \texttt{Index} objects. The implementation is such that \texttt{I} < \texttt{J} returns true if one of the following is true: the length of \texttt{I} (number of integer index values represented by \texttt{I}) is less than the length of \texttt{J}, the length’s are, equal but the first integer index value in \texttt{I} is less than the first value in \texttt{J}, or (failing either of the first two tests, and given that the length of \texttt{I} is greater than 0) the stride of \texttt{I} is less than the stride of \texttt{J}.

\textbf{bool} \texttt{operator} == \texttt{const Index} \texttt{) const

Test for equality of two \texttt{Index} objects. \texttt{I} == \texttt{J} returns true if the base,bound,stride values of \texttt{I} and \texttt{J} are the same. This does not check that the two \texttt{Index} objects have the same base address.

B.3.6 Index Composition Functions

\texttt{Index intersect} (\texttt{const Index} \texttt{) const

\texttt{I} . \texttt{intersect} (\texttt{J}) returns an \texttt{Index} object containing the intersection of \texttt{I} with \texttt{J} such that it contains all the integer index values contained both in \texttt{I} and \texttt{J} (expressed as the base,bound,stride of the new \texttt{Index} object).

\texttt{Index plugBase} (\texttt{const Index} \texttt{) const

Plug the base range of one into another.
inline bool sameBase(const Index&) const

Test to see if two Index’s are from the same base. Internally, Index contains a (private) Index* which points to the base Index object from which it was constructed. If it is an Index which was constructed explicitly with one of the constructors described in Section xxx then this will be a pointer to itself. If you construct an Index object by arithmetic on an existing, object, the pointer points to, the existing Index. Example:

Index I(10);
Index J(10);
bool t1 = I.sameBase(J); // false
bool t2 = I.sameBase(I+1); // true

inline bool touches(const Index&) const

Test to see if there is any overlap between two Indexes. I.touches(J) returns true if the minimum integer value represented by I is less than or equal to the maximum value represented by J, and the maximum integer value represented by I is greater than or equal to the minimum value represent by J.

inline bool contains(const Index&) const

Test to see if one Index object completely contains another. I.contains(J) returns true if the minimum integer value represented by I is less than or equal to the minimum value represented by J, and the maximum integer value represented by I is greater than or equal to the maximum value represented by J.

inline bool split(Index& l, Index& r) const

Splits one Index object into two. I.split(J,K) divides the set of integers represented by I into two halves, and fills the Index object J with the left half of the set, and the Index object K with the right half of the set.
Appendix C

FieldLayout Class

The FieldLayout class represents the abstraction of a decomposition of a Field object into pieces (subsets of elements). The number of pieces need not be, but typically is, greater than or equal to the number of physical processors or cores used to run the IP^2L program in parallel. This is a decomposition in the sense that the parallel computing literature talks about domain decomposition. The Field object represents a mathematical field discretized on a spatial domain, each piece of it is a subdomain. Equivalent to the actual spatial subdomain is the subset of the index space of the discretization. Currently in IP^2L, these subsets are contiguous, stride-1 subranges of the global (N-dimensional) index space. Other IP^2L mechanisms assign each subdomain to a processor or core. The FieldLayout class provides mechanisms for specifying decomposition, and has two relationships with the Field class: Field uses a FieldLayout reference and invokes its mechanisms to manage the distribution of its data, and FieldLayout maintains a container of pointers to all the Field’s it is used by. When the user requests that a FieldLayout be redistributed, the FieldLayout goes through its list of Field pointers and tells the Field’s to effect the redistribution of their data.

C.1 FieldLayout Definition (Public Interface)

```cpp
// enumeration used to select serial or parallel axes
enum e_dim_tag { SERIAL=0, PARALLEL=1 }

// A base class for FieldLayout that is independent of dimension.
class FieldLayoutBase
{
private:
// Some dummy storage so that it doesn’t confuse purify.
char Dummy;
```
public:

FieldLayoutBase() : Dummy(0) {}
};

// Template for containers.
typedef vmap<Unique::type, my_auto_ptr<Vnode<Dim> > > ac_id_vodes;
typedef DomainMap<NDIndex<Dim>, RefCountedP< Vnode<Dim> >, Touches<Dim>, Contains<Dim>, Split<Dim> > ac_domain_vnodes;
typedef vmap<GuardCellSizes<Dim>, my_auto_ptr<ac_domain_vnodes> > ac_gc_domain_vnodes;
typedef vmap<Unique::type, FieldBase*> ac_id_fields;

// Typedefs for iterators.
typedef ac_id_vodes::iterator iterator_iv;
typedef ac_id_vodes::const_iterator const_iterator_iv;
typedef ac_domain_vnodes::iterator iterator_dv;
typedef ac_domain_vnodes::touch_iterator touch_iterator_dv;
typedef pair<touch_iterator_dv, touch_iterator_dv> touch_range_dv;
typedef ac_id_fields::iterator iterator_if;
typedef ac_id_fields::const_iterator const_iterator_if;
typedef ac_gc_domain_vnodes::iterator iterator_gdv;

public:

// Accessors for the locals by Id.
class FieldLayout public FieldLayoutBase {
public:

// Accessors for the remote vnode containers.
class FieldLayout public FieldLayoutBase {
public:

// Accessors for the remote vnodes themselves.
class FieldLayout public FieldLayoutBase {
public:

// Tell the FieldLayout that a FieldBase has been declared on it
void checkin(FieldBase& f, const GuardCellSizes<Dim>& gc = gc0());

75
// Tell the FieldLayout that a FieldBase is no longer using it.
void checkout(FieldBase& f);

// Compare FieldLayouts to see if they represent the same domain.
bool operator==(const FieldLayout<Dim>& x)
{
    return Domain == x.Domain;
}

// Constructor.
// Default constructor, which should only be used if you are going to
// call 'initialize' soon after (before using in any context)
FieldLayout() { }

// Constructors for 1 ... 6 dimensions
FieldLayout(const Index& il, e_dim_tag pl=PARALLEL, int vnodes=-1);
FieldLayout(const Index& il1, const Index& il2, e_dim_tag pl=PARALLEL, e_dim_tag p2=PARALLEL, int vnodes=-1);
FieldLayout(const Index& il1, const Index& il2, const Index& il3, e_dim_tag pl=PARALLEL, e_dim_tag p2=PARALLEL, e_dim_tag p3=PARALLEL, int vnodes=-1);
FieldLayout(const Index& il1, const Index& il2, const Index& il3, const Index& il4, e_dim_tag pl=PARALLEL, e_dim_tag p2=PARALLEL, e_dim_tag p3=PARALLEL, e_dim_tag p4=PARALLEL, int vnodes=-1);
FieldLayout(const Index& il1, const Index& il2, const Index& il3, const Index& il4, const Index& il5, e_dim_tag pl=PARALLEL, e_dim_tag p2=PARALLEL, e_dim_tag p3=PARALLEL, e_dim_tag p4=PARALLEL, e_dim_tag p5=PARALLEL, e_dim_tag p6=PARALLEL, int vnodes=-1);
FieldLayout(const NDIndex<Dim>& domain, e_dim_tag* p=O, int vnodes=-1);

// Build a FieldLayout given the whole domain and
// begin and end iterators for the set of domains for the local Vnodes.
// It does a collective computation to find the remote Vnodes.
FieldLayout(const NDIndex<Dim>& Domain, NDIndex<Dim>* begin, NDIndex<Dim>* end);

// Initialization functions, for use when the FieldLayout was created using the default constructor.
void initialize(const Index& il1, e_dim_tag pl=PARALLEL, int vnodes=-1);
void initialize(const Index& il1, const Index& il2, e_dim_tag pl=PARALLEL, int vnodes=-1);
void initialize(const Index& il1, const Index& il2, const Index& il3, e_dim_tag pl=PARALLEL, int vnodes=-1);
void initialize(const Index& il1, const Index& il2, const Index& il3, const Index& il4, e_dim_tag pl=PARALLEL, int vnodes=-1);
C.2 FieldLayout Constructors

FieldLayout is parameterized on (unsigned) dimensionality, having the same meaning as the dimensionality template parameter for the Field class. When constructing a FieldLayout object, you must specify the index range for each dimension (or axis). To do this, you can use a single NDIndex object:

```cpp
dindex I(10), J(9), K(4);
NDIndex<3> Domain(I, J, K);
FieldLayout<3> Layout(Domain);
```

For FieldLayout's of \( N \) dimensions up to six, you may instead specify \( N \) Index objects to the constructor of FieldLayout directly, without creating an NDIndex object:

```cpp
Index I(5), J(9), K(4);
FieldLayout<3> Layout(I, J, K);
```
C.2.1 Specifying Serial or Parallel Layout

By default, FieldLayout will attempt to distribute the data among the processors (vnodes, actually) by subdividing each dimension in turn until it has the proper number of subregions. Those axes which are considered for subdivision are the parallel axes, which means that a given node will only contain Field data for a subset of the indices along that dimension. You can, however, tell FieldLayout which axes to subdivide, and which to maintain as serial. Serial axes are not ever partitioned by FieldLayout. You must have at least one parallel dimension in a given FieldLayout; by default, all axes are parallel. To specify things, use the predefined enumeration e_dim_tag, which has values SERIAL and PARALLEL. When you pass a NDIndex<Dim> as the FieldLayout constructor argument, you pass an array of e_dim_tag values, having length Dim. When you construct a FieldLayout with N Index arguments, you may also include up to N more arguments of type e_dim_tag to set the corresponding dimensions layout methods; any omitted dimensions at the end of the list default to PARALLEL. Refer to the first few chapters of this report for appropriate examples.

C.3 FieldLayout Member Functions and Member Data

C.3.1 Access Functions to Containers in FieldLayout

The Index class has several private members whose types are parameterized container classes patterned after STL containers. These have STL-like semantics, including iterators. The container objects themselves are private, but the user has public access to their sizes and iterators over them via Index public member functions. Index provides some typedef’s to make this access easier:

typedef vmap<unique::type, my_auto_ptr<Vnode<Dim> >> ac_id_vnodes;

Type for FieldLayout’s container of pointers to Vnode objects; the IP2L internal Vnode class represents the vnode, or index-space subdomain in this context. The number of subdomains is equal to the number of vnodes, and FieldLayout’s serial/parallel specification determines the extents of the subdomains in Index space.

typedef DomainMap<NDIndex<Dim>, RefCountedP< Vnode<Dim> >, Touches<Dim>, Contains<Dim>, Split<Dim> > ac_domain_vnodes;
typedef vrnap<GuardCellSi’es<Dim>, my_auto_ptr<ac_domain_vnodes> > ac_gc_domain_vnodes;

The first of these typedef’s is only used inside the second.

typedef vmap<Unique::type, FieldBase*> ac_id_fields;

78
Type for FieldLayout’s container of pointers to Field’s using this FieldLayout object, mentioned in the general discussion at the beginning of this chapter.

```cpp
typedef ac_id_vnodes::iterator iterator_iv;
typedef ac_id_vnodes::const_iterator const_iterator_iv;
typedef ac_domain_vnodes::iterator iterator_dv;
typedef ac_domain_vnodes::touch_iterator touch_iterator_dv;
typedef pair<touch_iterator_dv, touch_iterator_dv> touch_range_dv;
typedef ac_id_fields::iterator iterator_if;
typedef ac_iQ_.fields::const_iterator const_iterator_if;
typedef ac_gc_domain_vnodes::iterator iterator_gdv;
```

More to come.....
Appendix D

CenteredFieldLayout Class

The CenteredFieldLayout class inherits from FieldLayout. It represents the same abstraction as FieldLayout, except specialized to a particular type of centering on a particular type of Mesh; it is parameterized on mesh type and centering type. These template parameters have the same meaning as the corresponding parameters for the Field class.

The primary use of CenteredFieldLayout is for guaranteeing correct specification of numbers of elements in Field’s along the various dimensions according to the centering along those dimensions. The Mesh object reference constructor arguments provide for this.

D.1 CenteredFieldLayout Definition (Public Interface)

template<unsigned Dim, class Mesh, class Centering>
class CenteredFieldLayout : public FieldLayout<Dim> {
public:
    //-------------˜--------˜----------------------˜----˜-----------------------
    // Constructors from a mesh object only and parallel/serial specifiers.
    //---------------------˜---------------------˜-------------------------------
    // Constructor for arbitrary dimension with parallel/serial specifier array:
    //This one also works if nothing except mesh is specified:
    CenteredFieldLayout(Mesh& mesh, e_dim_tag *p=0, int vnodes=-1);
    // Constructors for 1 ... 6 dimensions with parallel/serial specifiers:
    CenteredFieldLayout(Mesh& mesh, e_dim_tag p1, int vnodes=-1);
    CenteredFieldLayout(Mesh& mesh, e_dim_tag p1, e_dim_tag p2, int vnodes=-1);
    CenteredFieldLayout(Mesh& mesh, e_dim_tag p1, e_dim_tag p2, e_dim_tag p3, int vnodes=-1);
    CenteredFieldLayout(Mesh& mesh, e_dim_tag p1, e_dim_tag p2, e_dim_tag p3, e_dim_tag p4, int vnodes=-1);
    CenteredFieldLayout(Mesh& mesh, e_dim_tag p1, e_dim_tag p2, e_dim_tag p3, e_dim_tag p4, e_dim_tag p5, int vnodes=-1);
};
D.2 CenteredFieldLayout Constructors

Here is where CenteredFieldLayout differs from FieldLayout. Instead of taking NDIndex& or Index& arguments to specify the numbers of elements along the various dimensions, the constructors take an argument having the type of the Mesh template parameter. This might be, for example, a UniformCartesian object reference. CenteredFieldLayout queries the mesh object for numbers of grid nodes and sets up the right number of elements for subsequent Field objects instantiated to use this CenteredFieldLayout. There are implementations for Cell, Vert, and Cartesian centerings on UniformCartesian and Cartesian meshes. If you are not providing a Mesh object to construct a CenteredFieldLayout, you probably should be just using simple FieldLayout objects instead, though no harm would be done by constructing a CenteredFieldLayout with Index/NDIndex arguments via the inherited constructors from FieldLayout. Refer to Appendix C for more details about FieldLayout. Other than these different constructors, CenteredFieldLayout is the same as FieldLayout.
Appendix E

Meshes

IP^2L predefines classes to represent Cartesian meshes; these typically serve as the Mesh template parameter for Field and other classes parameterized on Mesh type. These classes also provide various mechanism to query mesh geometry ( spacings, cell volumes, etc.) from Mesh objects.

E.1 Mesh Class

The Mesh class is an abstract base class for classes representing computational meshes. Currently, the base class does nothing it does not even provide any virtual functions, because it is difficult to conceive of commonality among potential derived meshes as disparate as unstructured and uniform structured meshes (for example). It does allow writing functions and classes that have Mesh objects as arguments and members; but, so far, that hasn’t been used even within the IP^2L internal implementation. IP^2L predefines the UniformCartesian and Cartesian classes which inherit from Mesh.

E.1.1 Mesh Definition (Public Interface)

```
template< unsigned Dim >
class Mesh
{
}
```

E.2 UniformCartesian Class

The UniformCartesian class represents the abstraction of a uniform-spacing Cartesian mesh discretizing a rectangular region of space. The Mesh has uniform spacing in the sense that the mesh spacings (vertex-vertex distances) along a dimension are the same all along that dimension. The different dimensions
may have different (single) values for mesh spacing. The Cartesian class (Appendix E.3) generalizes this to meshes whose spacings vary cell-by-cell along each dimension. UniformCartesian has mechanisms for returning various kinds of geometrical information from the mesh: cell-cell and vertex-vertex spacings, nearest mesh vertex positions to a given point in space, and others. Many of these; such as a function to return the volume of a particular indexed cell in the mesh, are somewhat redundant, for the uniform case, but make more sense in the nonuniform case (Cartesian); the interfaces of UniformCartesian and Cartesian are meant to be as much alike as possible. UniformCartesian is parameterized on dimensionality Dim, and another parameter MFLOAT. The MFLOAT parameter specifies the elemental type to use in storing and returning mesh geometrical information such as spacings and position coordinates. Generally, this should be a floating-point type, and it defaults to double. Vector values are represented using Vektor<MFLOAT, Dim>.

E.2.1 UniformCartesian Definition (Public Interface)

```cpp
template <unsigned Dim, class MFLOAT=double>
class UniformCartesian : public Mesh<Dim>
{

public:
    // Public member data:
    unsigned gridSizes[Dim]; // Sizes (number of vertices)
    typedef Cell DefaultCentering; // used by Field
    Vektor<MFLOAT, Dim> Dvc[1<<Dim]; // Constants for derivatives

    bool hasSpacingFields;
    BareField<Vektor<MFLOAT, Dim>, Dim>* VertSpacings;
    BareField<Vektor<MFLOAT, Dim>, Dim>* CellSpacings;

    // Public member functions:
    // Constructors
    UniformCartesian() {}; // Default constructor

    // Non-default constructors
    UniformCartesian(NDIndex<Dim>& ndi);
    UniformCartesian(Index& I);
    UniformCartesian(Index& I, Index& J);
    UniformCartesian(Index& I, Index& J, Index& K);

    // These also take a MFLOAT* specifying the mesh spacings:
    UniformCartesian(NDIndex<Dim>& ndi, MFLOAT* delX);
    UniformCartesian(Index& I, MFLOAT* delX);
    UniformCartesian(Index& I, Index& J, MFLOAT* delX);

    // These further take a Vektor<1:FLOAT,Dim>& specifying the origin:
    UniformCartesian(NDIndex<Dim>& ndi, MFLOAT* delX, Vektor<MFLOAT,Dim>& orig);
    UniformCartesian(Index& I, MFLOAT* delX, Vektor<MFLOAT,Dim>& orig);
    UniformCartesian(Index& I, Index& J, MFLOAT* delX, Vektor<MFLOAT,Dim>& orig);
    UniformCartesian(Index& I, Index& J, Index& K, MFLOAT* delX, Vektor<MFLOAT,Dim>& orig);

    ~UniformCartesian() {}; // Destructor

    // Set functions for member data:

    // Function to return the volume of a particular indexed cell in the mesh,
    // for the uniform case, but make more sense in the nonuniform case (Cartesian);
    // the interfaces of UniformCartesian and Cartesian are meant to be as much alike as possible.
```

83
// Create BareField's of vertex and cell spacings allow for specifying layouts via the FieldLayout e_dim_tag and vnodes parameters (these get passed in to construct the FieldLayout used to construct the BareField's).

void storeSpacingFields(); // Default will have default layout

// Special cases for 1-3 dimensions, a la FieldLayout ctors
void storeSpacingFields(e_dim_tag pI, int vnodes=-1);
void storeSpacingFields (e_dim_tag pI, e_dim_tag p2', int vnodes=-1);
void storeSpacingFields (e_dim_tag pI, e_dim_tag p2, e_dim_tag p3, int vnodes=-1);

// Next we have one for arbitrary dimension, a la FieldLayout ctor:
// All the others call this one internally:
void storeSpacingFields(e_dim_tag *p, int vnodes=-1);

// Accessor functions for member data:
// Get the origin of mesh vertex positions:
Vektor<MFLOAT,Dim> get_origin();

// Get the spacings of mesh vertex positions along specified direction:
MFLOAT get_meshSpacing(int d);

// Get the cell volume:
MFLOAT get_volume();

// Formatted output of UniformCartesian object:
void print(ostream&);

// Stream formatted output of UniformCartesian object:
friend ostream& operator<<(ostream&, const UniformCartesian<Dim,MFLOAT>&);

// Other UniformCartesian methods

// Volume of single cell indexed by input NDIndex
MFLOAT getCellVolume(NDIndex<Dim>&);

// Field of volumes of all cells:
Field<MFLOAT, Dim, UniformCartesian<Dim,MFLOAT>, Cell>& getCellVolumeField (Field<MFLOAT, Dim, UniformCartesian<Dim,MFLOAT>, Cell>&);

// Volume of range of cells bounded by vertices specified by input NDIndex:
MFLOAT getVertRangeVolume(NDIndex<Dim>&);

// Volume of range of cells spanned by input NDIndex (index of cells):
MFLOAT getCellRangeVolume(NDIndex<Dim>&);

// Nearest vertex index to (x,y,z)
NDIndex<Dim>& getNearestVertex(Vektor<MFLOAT, Dim>&);

// Nearest vertex index with all vertex coordinates below (x,y,z):
NDIndex<Dim>& getVertexBelow(Vektor<MFLOAT, Dim>&);

// NDIndex for cell in cell-ctrd Field containing the point (x,y,z):
NDIndex<Dim>& getCellContaining(Vektor<MFLOAT, Dim>&);

// (x,y,z) coordinates of indexed vertex:
Vektor<MFLOAT, Dim> getVertexPosition (NDIndex<Dim>&);

// Field of (x,y,z) coordinates of all vertices:
Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Vert>& getVertexPositionField (Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Vert>&);

// Vertex-vertex grid spacing of indexed cell:
Vektor<MFLOAT, Dim> getDeltaVertex(NDIndex<Dim>&);
// Field of vertex-vertex, grid spacings of all cells:
Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Cell>&
getDeltaVertexField (Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Cell>&);

// Cell-cell grid spacing of indexed vertex:
Vektor<MFLOAT, Dim> getDeltaCell(NDIndex<Dim>&);

// Field of cell-cell grid spacings of all vertices:
Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Vert>&
getDeltaCellField (Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Vert>&);

// Array of surface normals to cells adjoining indexed cell:
Vektor<MFLOAT, Dim>* getSurfaceNormals(NDIndex<Dim>&);

// Array of (pointers to) Fields of surface normals to all cells:
void getSurfaceNormalFields (Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Cell>**);

// Similar functions, but specify the surface normal to a single face, using
// the following numbering convention: 0 means low face of 1st dim, 1 means
// high face of 1st dim, 2 means low face of 2nd dim, 3 means high face of 2nd dim, and so on:
Vektor<MFLOAT, Dim> getSurfaceNormal(NDIndex<Dim>&, unsigned);
Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Cell>& getSurfaceNormalField(
Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim, MFLOAT>, Cell>&, unsigned);

E.2.2 UniformCartesian Constructors

Aside from the default constructor, which you should not be used if you, don’t
know what it’s there for, there are three categories of UniformCartesian
constructors. All require an argument or set of Dim arguments specifying the
number of mesh nodes along each dimension. The one-argument form takes
an NDIndex<Dim>& to specify this: the length() value of each Index
and the NDIndex represents the number of mesh nodes (vertices); the multi-
argument forms take Dim Index&’s (these are implemented only up to Dim = 3).
Warning: be sure to use zero-based, unit-stride NDIndex Index objects;
UniformCartesian should eventually work for other cases, but for now the
implementation doesn’t account for non-zero base or non-unit stride for the index
space spanning the mesh nodes/cells. The first and simplest category of con-
structors has only the size arguments; here are examples of the one-argument and
multi-argument case:

Index I {5}, J {5}, K {5};
NDIndex<3> ndi;
UniformCartesian<3> umesh1(ndi);
UniformCartesian<3> umesh2(I, J, K);

Both of these UniformCartesian objects will have default mesh spacings
of 1.0 in all directions and an origin (location of first vertex) at (0.0, 0.0, 0.0). The
second category adds specification of the mesh spacings. You create an array of
type MFLOAT and pass it to the constructor:

double spacings[3] = {1.0, 1.0, 2.0}; //double because default of MFLOAT
UniformCartesian<3> umesh3{ndi, spacings};

The umesh3 object has mesh spacings $\Delta_x = 1.0$, $\Delta_y = 1.0$, and $\Delta_z = 2.0$. We used type double for the array of spacings because we used the default MFLOAT template parameter value of double when instantiating umesh3. If we had specified something else, UniformCartesian<3, float>, we would have had to specify float as the type of the spacings array. The umesh3 object has a default origin (location of first vertex) at (0,0,0).

The third category adds specification of the origin (location of the first vertex). You create a Vektor<MFLOAT, Dim> and pass it as the last constructor argument:

// Defining spacings and origin, use double because it's default of MFLOAT:
double spacings[3] = {1.0, 1.0, 2.0};
Vektor<double, 3> origin;
origin(0) = 5.0; origin(1) = 6.0; origin(2) = 7.0;
UniformCartesian<3> umesh4(ndi, spacings, origin);

The umesh4 object has mesh spacings $\Delta_x = 1.0$, $\Delta_y = 1.0$, and $\Delta_z = 2.0$ and origin at (5.0,6.0,7.0).

### E.2.3 UniformCartesian Member Functions and Member Data

UniformCartesian Member Data for Sizes and Spacings. There are several public data members representing mesh size and spacing information:

```c
unsigned gridSizes[Dim] ;
```

An array containing the mesh sizes-numbers of vertices along each dimension.

```c
typedef Cell DefaultCentering ;
```

Used by Field and other classes which are parameterized on Mesh and centering classes and require consistent defaults for both. The general user probably need never use this member.

```c
Vektor<MFLOAT, Dim> Dvc[1<<Dim] ;
```

Constants for derivatives in global differential operator functions such as Div(). Again, the general user probably never need know that this is publicly visible.

```c
bool hasSpacingFields ;
```

Flags whether the user has requested that the mesh object internally allocate and compute mesh-spacing BareField’s pointed to by VertSpacings and CellSpacings.
If you invoke the `storeSpacingFields()` function, UniformMesh will allocate two `BareField`'s and fill them with vertex-vertex and cell-cell mesh spacing values (stored as vectors whose components are the spacings along each dimension of the cell or shifted cell). These pointers provide public access to them. The general user can always use the functions like `getDeltaVertexField()` to put spacing values into his own `Field` objects, and this may be the best way to do this in general. Certain predefined IP2L global functions, such as the `Div()` differential operators, might rely on having these `BareField`'s internally available from the mesh object associated with the `Field`'s on which they operate. Note: These internal `BareField`'s are redundant for `UniformCartesian` in a couple of ways. First, the mesh spacing information is the same everywhere in the `Mesh`, and the vertex-vertex spacing is the same as the cell-cell spacing. The operators like `Div()` don’t need `BareField`'s of spacings for the uniform Cartesian case; and they don’t, in fact, use them or check if they exist. Both of these redundancies are removed in the nonuniform Cartesian case, however. In this case, storing this information in the internal `BareField`’s in the Cartesian objects is essential for functions like `Div()` - they return errors if `hasSpacingFields` is false.

### E.2.4 UniformCartesian Set/Accessor Functions for Member Data

```cpp
void storeSpacingFields ( );
void storeSpacingFields(e_dim_tag p1, int vnodes=-1);
void storeSpacingFields(e_dim_tag p1, e_dim_tag p2, int vnodes=-1);
void storeSpacingFields(e_dim_tag p1, e_dim_tag p2, e_dim_tag p3, int vnodes=-1);
void storeSpacingFields(e_dim_tag *p, int vnodes=-1);
```

The `UniformCartesian` class will optionally create internal `BareField`’s of appropriate sizes and fill them with vertex-vertex and cell-cell spacing values. You access these `BareField`’s via the `VertSpacings` and `CellSpacings` pointers described above. The `storeSpacingFields()` functions make this happen. Because they are constructing `BareField`’s, they provide prototypes based on those for `FieldLayout` so you can control the serial/parallel layout of the `BareField`’s, you specify which dimensions are serial or parallel using lists or an array of `e_dim_tag` values (SERIAL or PARALLEL). If you use the first prototype, with no arguments, you will get the default `FieldLayout` for the internal `BareField`’s: parallel for all dimensions.

```cpp
Vektor<MFLOAT,Dim> get_origin();
```
Returns the value of the origin of the mesh (position in space of the lowest mesh vertex).

MFLOAT get_meshSpacing(int d);

Returns the mesh spacing value for the specified direction. There is only one value for uniform spacing; thus the return type MFLOAT.

MFLOAT get_volume();

Returns the volume of a cell, which is the same everywhere in a uniform Cartesian mesh.

### E.2.5 Other UniformCartesian Methods

Most of the public member functions in UniformCartesian are designed to return some typical kinds of geometrical information about the Mesh that a user (programmer or class) might want. Where the information or input parameters are more complex than single MFLOAT values, the function return values or arguments are typically IP^2L classes such as Field. The set of functions evolved from general discussions among application programmers of, what is expected of a mesh, and we will continue to evolve its design iteratively as new applications demand new Mesh information. Many of these functions use NDIndex to index mesh vertices and cells. NDIndex has no intrinsic awareness of centering, but can clearly represent index values specifying mesh node or cell positions. The user must keep in mind that cell 0 along a dimension is between node 0 and node 1, and that there are one fewer cells than vertices.

MFLOAT getCellVolume(NDIndex<Dim>&);

Volume of single cell indexed by input NDIndex. The argument must describe a single element. That is, the range of every Index in the NDIndex must be 1. The getCellVolume() function returns an error otherwise.

Field<MFLOAT,Dim,UniformCartesian<Dim,MFLOAT>, Cell>& getCellVolumeField (Field<MFLOAT, Dim, UniformCartesian<Dim,MFLOAT>,Cell>&);

Field of volumes of all cells. This function basically assigns every element of the cell-centered Field to the return value of getCellVolume ( ). For a UniformCartesian mesh, this is obviously redundant, but the function is here for interface compatibility with Cartesian (which represents a nonuniform Cartesian mesh, for which this is not redundant). If you are only using UniformCartesian mesh objects, you should just use the single cell-volume value returned by getCellVolume; this will combine with other Field’s of values in your code the same way any scalar value will do.

MFLOAT getVertRangeVolume(NDIndex<Dim>&);
Volume of range of cells bounded by vertices specified by input NDIndex, which in this context will generally have a range greater than one in at least one dimension. The vertices represented by the lowest and highest index value set contained in the NDIndex mark the corners of a rectangular solid region; this function returns the volume of that region.

\[
\text{MFLOAT getCellRangeVolume(NDIndex<Dim>&);} 
\]

Volume of range of cells spanned by input NDIndex (index of cells). This is like getVertRangeVolume( ), except that the corners of the rectangular solid are cell-center positions rather than vertex positions.

\[
\text{NDIndex<Dim>& getNearestvertex(Vektbr<MFLOAT,Dim>&);} 
\]

Nearest vertex index to a point in space (x,y,z).

\[
\text{NDIndex<Dim>& getVertexBelow(Vektor<MFLOAT,Dim>&);} 
\]

Nearest vertex index with all vertex coordinates below a point in space (x,y,z).

\[
\text{NDIndex for the mesh cell containing the point (x,y,z). Use this, for example, to index a corresponding element in a cell-centered Field.} 
\]

\[
\text{Vektor<MFLOAT,Dim> getVertexposition(NDIndex<Dim>&);} 
\]

(x,y,z) coordinates of the Mesh vertex indexed by the NDIndex, which just have a range of one in all dimensions (that is, it must index a single point in index space).

\[
\text{Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim,MFLOAT>,Vert>& getVertexPositionField(Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim,MFLOAT>,Vert>&);} 
\]

Fills a vertex-centered Field with the (x,y,z) coordinates of all mesh vertices.

\[
\text{Vektor<MFLOAT, Dim> getDeltaVertex (NDIndex<Dim>&);} 
\]

Vertex-vertex grid spacing (\(\Delta x, \Delta y, \Delta z\)) of the cell indexed by the NDIndex.

\[
\text{Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim,MFLOAT>,Cell>& getDeltaVertexField(Field<Vektor<MFLOAT, Dim>, Dim, UniformCartesian<Dim,MFLOAT>,Cell>&);} 
\]

Fills a cell-centered Field with the vertex-vertex grid spacings (\(\Delta x, \Delta y, \Delta z\)) of all cells.

\[
\text{Vektor<MFLOAT,Dim> getDeltaCell(NDIndex<Dim>&);} 
\]

Cell-cell grid spacing (\(\Delta x, \Delta y, \Delta z\)) of indexed cell vertex. That is, this returns the distance between the cell centers on either side of the vertex position indexed by the NDIndex for each dimension.

\[
\text{Field<Vektor<MFLOAT, Dim>, Dim,uniformCartesian<Dim, MFLOAT>, Vert>& getDeltaCellField(Field<Vektor<MFLOAT,Dim>,Dim, UniformCartesian<Dim,MFLOAT>,Vert>&);} 
\]
Fills a vertex-centered Field with the cell-cell grid spacings \((\Delta_x, \Delta_y, \Delta_z)\) around all vertices.

\[
\text{Vektor}<\text{MFLOAT}, \text{Dim}> \star \text{getSurfaceNormals}(\text{NDIndex}<\text{Dim}>i);
\]

Array of surface normals to cells adjoining indexed cell. This is trivial for a Cartesian mesh, and is the same for every cell even in the nonuniform spacing cases represented by Cartesian. Future implementations in non-cartesian mesh classes would be more complicated.

\[
\text{void getSurfaceNormalFields(\text{Field}<\text{Vektor}<\text{MFLOAT}, \text{Dim}>, \text{Dim}, \text{UniformCartesian}<\text{Dim}, \text{MFLOAT}>, \text{Cell}>>)};
\]

Fills the Field’s pointed to by the array with of surface normals to all cells. Again, this is trivial for a cartesian mesh, and the values are the same everywhere; but future implementations in non-cartesian meshes would be more complicated.

\[
\text{Vektor}<\text{FLOAT}, \text{Dim}> \star \text{getSurfaceNormal}(\text{NDIndex}<\text{Dim}>i, \text{unsigned});
\]

\[
\text{Field}<\text{Vektor}<\text{FLOAT}, \text{Dim}>, \text{Dim}, \text{UniformCartesian}<\text{Dim}, \text{MFLOAT}>, \text{Cell}>> \star \text{getSurfaceNormalField(\text{Field}<\text{Vektor}<\text{FLOAT}, \text{Dim}>, \text{Dim}, \text{UniformCartesian}<\text{Dim}, \text{MFLOAT}>, \text{Cell}>>, \text{unsigned})};
\]

Similar functions to getSurfaceNormals() and getSurfaceNormalFields(), but specify the surface normal to a single face, using the following numbering convention: 0 means low face of 1st dimension, 1 means high face of 1st dimension, 2 means low face of 2nd dimension, 3 means high face of 2nd dimension; and so on.

### E.3 Cartesian Class

The Cartesian class represents the abstraction of a nonuniform-spacing Cartesian mesh discretizing a rectilinear region of space. The mesh spacings vary cell-by-cell along each dimension. As much as possible, the Cartesian interface is identical to the UniformCartesian interface described in Section xxx. It has the same mechanisms for returning various kinds of geometrical information from the Mesh: cell-cell and vertex-vertex spacings, nearest mesh vertex positions to a given point in space, and others. Like UniformCartesian, Cartesian is parameterized on dimensionality Dim, and another parameter MFLOAT which specifies the elemental type to use in storing and returning mesh geometrical information such as spacings and position coordinates. Generally, this should be a floating-point type, and it defaults to double. Vector values are represented using Vektor<MYSQL,Dim>. This chapter only discusses the places where the Cartesian interface differs from the UniformCartesian interface. Refer to Section xxx for all other information about Cartesian; substitute "Cartesian" for "UniformCartesian" in places such as the Mesh parameter for Field arguments to the member functions. To help with this, we show
the entire Cartesian public definition in the next section; in the subsequent sections discussing the member functions and data, we discuss only the cases that differ from UniformCartesian.

E.3.1 Cartesian Definition (Public Interface)

Enumeration used for specifying mesh boundary conditions. Mesh BC are used for things like figuring out how to return the mesh spacing for a cell beyond the edge of the physical mesh, as might arise in stencil operations on Field's on the mesh.

```cpp
enum MeshBC_E { Reflective, Periodic, None };


template<unsigned Dim, class MFOAT=double>
class Cartesian : public Mesh<Dim>
{
    public:
        // Public member data:
        typedef Cell<Dim> DefaultCentering;  // Default centering (used by Field, etc.)
        Vektor<MFOAT, Dim> Dvc[1<<Dim];  // Constants for derivatives.

        bool hasSpacingFields;  // Flags allocation of the following:
        BareField<Vektor<MFOAT, Dim>, Dim>* VertSpacing;
        BareField<Vektor<MFOAT, Dim>, Dim>* CellSpacing;

        // Public member functions:
        // Constructors
        Cartesian();  // Default constructor

        // Non-default constructors
        Cartesian(NDIndex<Dim>& ndi);
        Cartesian(Index& i);
        Cartesian(Index& i, Index& j);
        Cartesian(Index& i, Index& j, Index& k);

        // These also take a MFOAT** specifying the mesh spacings:
        Cartesian(NDIndex<Dim>& ndi, MFOAT** delX);
        Cartesian(Index& i, MFOAT** delX);
        Cartesian(Index& i, Index& j, MFOAT** delX);
        Cartesian(Index& i, Index& j, Index& k, MFOAT** delX);

        // These further take a Vektor<MFOAT, Dim>** specifying the origin:
        Cartesian(NDIndex<Dim>& ndi, MFOAT** delX, Vektor<MFOAT, Dim>** orig);
        Cartesian(Index& i, MFOAT** delX, Vektor<MFOAT, Dim>** orig);
        Cartesian(Index& i, Index& j, MFOAT** delX, Vektor<MFOAT, Dim>** orig);
        Cartesian(Index& i, Index& j, Index& k, MFOAT** delX, Vektor<MFOAT, Dim>** orig);

        // These further take a MeshBC_E array specifying mesh boundary conditions,
        Cartesian(NDIndex<Dim>& ndi, MFOAT** delX, Vektor<MFOAT, Dim>** orig, MeshBC_E* mbc);
        Cartesian(Index& i, MFOAT** delX, Vektor<MFOAT, Dim>** orig, MeshBC_E* mbc);
        Cartesian(Index& i, Index& j, MFOAT** delX, Vektor<MFOAT, Dim>** orig, MeshBC_E* mbc);
        Cartesian(Index& i, Index& j, Index& k, MFOAT** delX, Vektor<MFOAT, Dim>** orig, MeshBC_E* mbc);
        ~Cartesian();

};
```
Set functions for member data: create BareField’s of vertex and cell spacings; allow for specifying layouts via the FieldLayout e_dim_tag and vnodes parameters (these get passed by constructing the FieldLayout and used to, construct the BareField’s).

```cpp
void storeSpacingFields(); // Default will have default layout

// Special cases for 1-3 dimensions, a la FieldLayout ctars:
void storeSpacingFields(e_dim_tag p1; int vnodes=-1);
void storeSpacingFields (e_dim_tag p1, e_dim_tag p2, int vnodes=-1);
void storeSpacingFields(e_dim_tag p1, e_dim_tag p2, e_dim_tag p3, int vnodes=-1);

// Next we have one far arbitrary dimension, a la FieldLayout ctor:
// All the others call this one internally:
void storeSpacingFields(e_dim_tag *p, int vnodes=-1);

// Accessor functions far member data:
// Get the origin of mesh vertex positions:
Vektor<MFLOAT, Dim> get_origin();

// Get the spacings of mesh vertex positions along specified direction:
MFLOAT* get_meshSpacing(int d);

// Get mesh boundary conditions:
MeshBC_E get_MeshBC(unsigned face); // One face at a time
MeshBC_E* get_MeshBC(); // All faces at once

// Formatted output of Cartesian object:
void print(ostream&);

// Stream formatted output of Cartesian object:
friend ostream& operator<<(ostream&, const Cartesian<Dim, MFLOAT>&);

// Other Cartesian methods
// Volume of a single cell indexed by input NDIndex
MFLOAT getCellVolume(NDIndex<Dim>&);

// Field of Volumes of all cells
Field<MFLOAT, Dim, Cartesian<Dim, MFLOAT>, Cell>& getCellVolumeField( Field<_FLOAT, Dim, Cartesian<Dim, MFLOAT>, Cell>&);

// Volume of range of cells bounded by vertices specified by input NDIndex:
FLOAT getVertRangeVolume (NDIndex<Dim>&);

// Volume of range of cells spanned by input NDIndex (index of cells)
FLOAT getCellRangeVolume(NDIndex<Dim>&);

// Nearest vertex index to, (x, y, z)
NDIndex<Dim>& getNearestVertex (Vektor<MFLOAT, Dim>&);

// Nearest vertex index with all vertex coordinates below (x, y, z)
NDIndex<Dim>& getVertexBelow (Vektor<MFLOAT, Dim>&);

// NDIndex for cell in cell-ctrd Field containing the point (x, y, z)
NDIndex<Dim>& getCellContaining (Vektor<MYSQL, Dim>&);

// (x, y, z) coordinates of indexed vertex
Vektor<MYSQL, Dim> getVertexPosition (NDIndex<Dim>&);
```
// Field of (x,y,z) coordinates of all vertices:
Field<Vektor<MFLOAT,Dim>,Dim, Cartesian<Dim,MFLOAT>,Vert>& getVertexPositionField(
Field<Vektor<MFLOAT,Dim>, Dim, Cartesian<Dim,MFLOAT>,Vert>&);

// Vertex-vertex grid spacing of indexed vertex
Vektor<MFLOAT,Dim> getDeltaVertex(NDIndex<Dim>&);

// Field of vertex-vertex grid spacings of all vertices
Field<Vektor<MFLOAT,Dim>, Dim, Cartesian<Dim,MFLOAT>,Vert>& getDeltaVertexField(
Field<Vektor<MFLOAT,Dim>, Dim, Cartesian<Dim,MFLOAT>,Vert>&);

// Cell-cell grid spacing of indexed cell:
Vektor<MFLOAT,Dim> getDeltaCell(NDIndex<Dim>&);

// Field of cell-cell grid spacings of all vertices:
Field<Vektor<MFLOAT,Dim>, Dim, Cartesian<Dim,MFLOAT>,Cell>& getDeltaCellField(
Field<Vektor<MFLOAT,Dim>, Dim, Cartesian<Dim,MFLOAT>,Cell>&);

// Array of surface normals to cells adjoining indexed cell:
Vektor<MFLOAT,Dim>* getSurfaceNormals(NDIndex<Dim>&);

// Array of (pointers to) Fields of surface normals to all cells:
void getSurfaceNormalFields(Field<Vektor<MFLOAT,Dim>,Dim, Cartesian<Dim,MFLOAT>,Cell>**);

Similar functions, but specify the surface normal to a single face, using the following numbering convention: 0 means low face of 1st dim, 1 means high face of 1st dim, 2 means low face of 2nd dim, 3 means high face of 2nd dim, and so on:

Vektor<MFLOAT,Dim> getSurfaceNormal(NDIndex<Dim>&, unsigned);
Field<Vektor<Parcelable<FLOAT,Dim>,Dim, Cartesian<Dim,MFLOAT>,Cell>& getSurfaceNormalField(
Field<Vektor<Parcelable<FLOAT,Dim>,Dim, Cartesian<Dim,MFLOAT>,Cell>&);

E.3.2 Cartesian Constructors

Aside from the default constructor, which you should not be using if you don’t know what it’s there for, there are four categories of Cartesian constructors. The first three are the same as for UniformCartesian (see Section xxx), except that specifying mesh spacings requires more than just a single FLOAT value for Cartesian( ). You must pass in an array of arrays of FLOAT values (FLOAT T * * ) one array for each dimension, having size given by the number of cells along that dimension. The new fourth constructor category adds specification of mesh-spacing boundary conditions. You create an array of type MeshBC_E, an enumeration having values {Reflective, Periodic, None}. This tells Cartesian how to provide geometry information such as mesh spacings beyond the physical edge of the mesh, as might arise in implementing differential operators using finite differencing of Field’s centered on the mesh. There are basically only two ways to do this: wrap around periodically to mesh spacing values inside the physical mesh, or reflect the mesh spacing values across the
boundary. The user should avoid specifying None for mesh boundary-condition
types. The following code example illustrates using this fourth constructor cate-
gory for Cartesian(), and how to set up mesh spacing and boundary condition
specifiers:

```cpp
const unsigned Dim = 3U;
unsigned nx=5, ny=5, nz=5;
Index I(nx), J(nz), K(nz);

NDIndex<Dim> ndi;

// Defining spacings and origin, use double because it’s default of MFLOAT:
double* spacings[Dim];
spacings[0] = new double [nx];
spacings[1] = new double [ny];
spacings[2] = new double [nz];

int vert;
for (vert=0; vert < nx; vert++) (delX[0]) [vert] = 1.0 + vert*1.0;
for (vert=0; vert < ny; vert++) (delX[1]) [vert] = 2.0 + vert*2.0;
for (vert=0; vert < nz; vert++) (delX[2]) [vert] = 3.0 + vert*3.0;

Vektor<double,Dim> origin;
origin(0) = 5.0;
origin(1) = 6.0;
origin(2) = 7.0;

MeshBC_E meshbc[2*Dim];
for(unsigned face=0; face < (2*Dim); face++)
  meshbc[face] = Reflective;

Cartesian<Dim> cmesh(ndi, spacings, origin, meshbc);
```

The cmesh object has mesh spacings $\Delta x = 1.0, \Delta y = 2.0, \Delta z = 3.0$ and reflective mesh boundary conditions
on all faces. The other three cases of Cartesian() are like this but with
fewer parameters. As in UniformCartesian, if the origin is unspecified it
defaults to (0.0, 0.0, 0.0), if the mesh spacings are unspecified they default to uniform
values of 1.0 for all cells along all axes. The mesh boundary conditions default to
Reflective on all faces.

E.3.3 Cartesian Member Functions and Member Data

Cartesian Member Data for Sizes and Spacings

The types are all the same as for UniformCartesian (See Appendix E.2 ).
The values of and interpretation of the Dvc data member are completely different;
functions such as Div() which use this information to implement differential
operators must account properly for the variation of mesh spacing values and the
difference between cell-cell and vertex-vertex spacings in the nonuniform case.
For UniformCartesian, the single Dvc array could hold all the necessary information; for Cartesian, a slightly different Dvc coordinates with VertSpacing and CellSpacings, which must exist before the user can invoke operators such as Div().

**Cartesian Set/Accessor Functions for Member Data**

Except for the return value of `get_meshSpacing()`, and the new functions `get_MeshBC()`, the interfaces and descriptions are all the same as for UniformCartesian. (See Appendix E.2).

```c
FLOAT* get_meshSpacing(int d);
```

Returns the array of mesh-spacing values along the specified direction. The number of elements in the arrays the number of cells in the mesh along that dimension.

```c
MeshBC_E* get_MeshBC(unsigned face);
```

Returns the value of the mesh boundary-condition specifier for the requested face of the mesh. The numbering convention for faces is: 0 means low face of 1st dimension, 1 means high face of 1st dimension, 2 means low face of 2nd dimension, 3 means high face of 2nd dimension, and so on.

```c
MeshBC_E* get_MeshBC();
```

Returns the array of values of mesh boundary-condition specifiers for all faces of the Mesh, following the numbering convention described in the description of the one-argument prototype of this function above. The number of elements in the array is two times the number of dimensions.

**Other Cartesian Methods**

As for UniformCartesian, most of the public member functions in Cartesian are designed to return some typical kinds of geometrical information about the mesh that a user (programmer or class) might want. So far, all of the member functions mirror UniformCartesian exactly, except for the occasional replacement of "Cartesian" for "UniformCartesian" in mesh template-parameters for Field arguments.
Appendix F

Centering

Classes in this chapter represent the abstraction of centering at particular positions on a Mesh. For any type of mesh, vertex (mesh-node) centering is well-defined. For Cartesian meshes, cell centering is well defined, because each rectangular mesh cell has a well-defined center, the midpoint along each dimension. Also, for Cartesian meshes, combinations of cell and vertex centering along the various dimensions clearly define common centerings such as face centering and edge centering. IP^2L predefines classes for cell and vertex centering, as well as classes to represent all possible combination centerings for Cartesian meshes. These classes are all static in the current implementation (you do not instantiate objects of the class types, you just refer to the centerings by class name). The centering classes typically serve as the centering template parameter for Field and other IP^2L classes parameterized on centering.

F.1 Cell Class

The static Cell class represents the abstraction of cell-centering of something on a Mesh. That ”something” is usually a Field. Cell can serve as a value for the Centering template parameter of the Field class. Every Field element, whatever its type, is centered at the position of the corresponding cell center in the Mesh using the same index space for mesh cell centers as (or the Field elements. If the Field elements (T parameter for Field class) is some kind of multi component type such as Vektor, the whole object of that type in every Field element is cell-centered (that is, all components have the same centering). The CartesianCentering class allows you to center components of Field elements independently. If you center a Field on a Mesh using Cell, you must make sure that you construct the Field to have the proper number of elements so that it has one-for every cell in the mesh. When you use Field constructors
taking a Mesh object argument (instance of the class specified as the Mesh template parameter for Field), the extents of the NDIndex object or index field constructor arguments match those used to instantiate that Mesh. When you instantiate a Field with Cell centering using constructors without a mesh object argument, the internally-constructed mesh object will automatically have the right number of cells. subsectionCell Definition (Public Interface)

class Cell {
public:
    static char* CenteringName ;
    static void print_Centerings(ostream&) ;
};

F.1.1 Cell Constructors

Cell is a static class you don’t instantiate objects, but rather only refer to it by its class name.

F.1.2 Cell Member Functions and Member Data

static char* CenteringName;
Public data member, having static value "Cell"
static void print_Centerings(ostream&) ;
Invoked as Cell::printCenterings(ostream&) prints the value of the string Cell::CenteringName.

F.2 Vert Class

The static Vert class represents the abstraction of vertex-centering of something on a mesh. That ”something” is usually a Field. Vert can serve as a value for the Centering template parameter of the Field class. Every Field element, whatever its type, is centered at the position of the corresponding vertex center in the mesh-using the same index space for Mesh vertex centers as for the Field elements. If the Field elements (T parameter for Field class) is some kind of multicomponent type such as Vektor, the whole object of that type in every Field element is vertex-centered (that is, all components have the same centering). The CartesianCentering class allows you to center components of Field elements independently. If you center a Field on a Mesh using Vert, you must make sure that you construct the Field to have the proper number of elements so that it has one for every vertex in the mesh. When you use Field constructors taking a mesh object argument (instance of the class specified as the Mesh template parameter for Field), the extents of the NDIndex object or
Index Field constructor arguments match those used to instantiate that Mesh. When you instantiate a Field with Vert centering using constructors without a mesh object argument, the internally constructed mesh object will automatically have the right number of vertices.

subsection Vert Definition (Public Interface)

class Vert {
public:
    static char* CenteringName;
    static void print_Centerings(ostream&);
};

F.2.1 Vert Constructors

Vert is a static class you don’t instantiate objects, but rather only refer to it by its class name.

F.2.2 Vert Member Functions and Member Data

static char* CenteringName;

Public data member, having static value "Vert".
static void print_Centerings(ostream&);

Invoked as Vert::printCenterings(ostream&) prints the value of the string Vert::CenteringName.

F.3 CommonCartesianCenterings Class

The static CommonCartesianCenterings class is a wrapper class for commonly-used special cases of the CartesianCentering class, predefined by IP2L as a convenience for the user. Basically, it is a collection of typedef’s. CartesianCentering is a parameterized static class representing the abstraction of componentwise centering of a multicomponent object (typically a Field) on a cartesian mesh. Via the template parameters, the user specifies the centering of the various components along the various directions. See next section for more details. CommonCartesianCenterings provides a, shorthand definition for some of the common cases expressible by CartesianCentering. Also shown in the definition of CommonCartesianCenterings below are the CenteringEnum which it uses and the static wrapper class commonCartesianCenteringEnums. This last class contains the special-case arrays of CenteringEnum values (CELL or VERT) which represent the various centerings for various specializations of the template parameters. CommonCartesianCenterings parameterized on dimensionality Dim, which
has the same meaning as the dimensionality parameter in Field or UniformCartesian (for example).

The unsigned value NComponents and the unsigned value Direction. NComponents represents the number of components in a multicomponent object to be centered on the mesh, for example, if you are centering a Field<Vektor<double, 3U>,the number of components is three. Direction represents a specifying direction, and is only really used in some of the CommonCartesianCenterings members. For example, to specify face centering of a scalar field (or single component of a non-scalar field), you must specify which direction is perpendicular to the faces where you are centering. If you wanted to center a scalar field on the xy faces in 3D, you would use the value 2U for Direction; because the xy faces are perpendicular to the z direction (the directions are numbered sequentially, x is 0U, y is 1U, and z is 2U).

F.3.1 CommonCartesianCenterings Definition (Public Interface)

```cpp
enum CenteringEnum {CELL=0, VERTEX=1, VERT=1};

template<unsignedDim, unsigned NComponents=1U, unsigned Direction=0U>
class CommonCartesianCenteringEnums
{
    public:
        // CenteringEnum arrays Classes with simple, descriptive names

        // All components of Field cell-centered in all directions:
        static CenteringEnum allCell [NComponents*Dim] ;

        // All components of Field vertex-centered in all directions:
        static CenteringEnum allVertex[NComponents*Dim] ;

        // All components of Field face-centered in specified direction (meaning vertex centered in that direction, cell-centered in others):
        static CenteringEnum allFace[NComponents*Dim];

        // All components of Field edge-centered along specified direction
        // (cell centered in that direction, vertex-centered in others):
        static CenteringEnum allEdge[NComponents*Dim];

        // Each vector component of Field face-centered in the corresponding direction
        static CenteringEnum vectorFace[NComponents*Dim];
    }

template<unsigned Dim, unsigned NComponents=1U, unsigned Direction=0U>
class CommonCartesianCenterings
{
    public:
        typedef CartesianCentering<CommonCartesianCenteringEnums<Dim,NComponents, Direction>::allCell, Dim, NComponents> allCell;
        typedef CartesianCentering<CommonCartesianCenteringEnums<Dim,NComponents, Direction>::allVertex, Dim, NComponents> allVertex;
}```
```cpp
typedef CartesianCentering<CommonCartesianCenteringEnums<Dim, NComponents, Direction>::allFace, Dim, NComponents> allFace;
typedef CartesianCentering<CommonCartesianCenteringEnums<Dim, NComponents, Direction>::allEdge, Dim, NComponents> allEdge;
typedef CartesianCentering<CommonCartesianCenteringEnums<Dim,NComponents, Direction>::vectorFace, Dim, NComponents> vectorFace;
```

### F.3.2 CommonCartesianCenterings Constructors

CommonCartesianCenterings is a static class, you don’t instantiate objects, but rather only refer to it by its class name. Refer to its members his way also for example: CartesianCenterings<3U, 1U> for the member representing centering on a 3D mesh of a scalar (one-component) object.

### F.3.3 CommonCartesianCenterings Member Data

```cpp
typedef CartesianCentering<CommonCartesianCenteringEnums<Dim,NComponents, Direction>::allCell, Dim, NComponents> allCell;
```

Specifies CELL centering of all components along all dimensions. Functionally equivalent to the Cell centering class.

```cpp
typedef CartesianCentering<CommonCartesianCenteringEnums<Dim,NComponents, Direction>::allVertex, Dim, NComponents> allVertex;
```

Specifies VERTEX centering of all components along all dimensions. Functionally equivalent to the Vert centering class.

```cpp
typedef CartesianCentering<CommonCartesianCenteringEnums<Dim,NComponents, Direction>::allFace, Dim, NComponents> allFace;
```

Specifies centering of all components on the faces which are orthogonal to the specified direction (Direction=0 means x, 1 means y, 2 means z). That is, vertex-centering along direction Direction, and cell-centering along all other directions.

```cpp
typedef CartesianCentering<CommonCartesianCenteringEnums<Dim,NComponents, Direction>::allEdge, Dim, NComponents> allEdge;
```

Specifies centering of all components on the edges which are parallel to the specified direction (Direction=0 means x, 1 means y, 2 means z). That is, cell-centering along direction Direction, and vertex-centering along all other directions.

```cpp
typedef CartesianCentering<CommonCartesianCenteringEnums<Dim,NComponents, Direction>::vectorFace, Dim, NComponents> vectorFace;
```

Specifies componentwise face centering of the components of a Vector. Usually you use this for centering a \(1^2\) Field whose elements are Vector’s. For example,

```
Field<Vector<double, 3>, 3, UniformCartesian<3>, CommonCartesianCenterings<3, 3, 3>>
```
F.4 CartesianCentering Class

The static CartesianCentering class represents the abstraction of component-wise centering of a multicomponent object (typically a Field) on a cartesian mesh. Via the template parameters, the user specifies the centering of the various components along the various directions.

F.4.1 CartesianCenteringDefinition(Public Interface)

template<const CenteringEnum* CE, unsigned Dim, unsigned NComponents=1U>
class CartesianCentering
{
  public:
  static char* CenteringName;
  static void print_Centerings(ostream&);
};

F.4.2 CartesianCentering Constructors

CartesianCentering is a static class, you don’t instantiate objects, but rather only refer to it by its class name, with fully-specified template parameter values. If you have defined a static array of type CenteringEnum called myCEArray, the identifier CartesianCenterings<myCEArray,3U,1U> refers to the static class representing centering on a 3D mesh of a scalar (one-component) object according to the centering specifiers in myCEArray. In this case, myCEArray would have to have three elements (each having the value CELL or VERT). In general, the number of elements in the array of CenteringEnum (the CE template parameter value) must equal the value of the Dim parameter multiplied by the value of the NComponent parameter. The ordering of the elements is so that the component indices vary fastest, and the dimensions indices vary the slowest (like a 2D C array dimensioned as [Dim][NComponent]). You must declare the CenteringEnum array you use as the value of the CE template parameter as static, and at global scope.

F.4.3 CartesianCentering Member Functions and Member Data

All the member functions of CartesianCentering are of course static. Refer to them with the syntax classname::membername, where classname is a name identifying a particular CartesianCentering class specialization, as described in this Appendix.

static char* CenteringName;
A string containing a identifying name for a particular template instance of CartesianCentering. Currently, there is only a single name, so that the value of CartesianCentering<CE,Dim,NComponent> is the same for any values of the template parameters.

static void print_Centerings(ostream&);  

Prints a formatted version of the CartesianCentering class. Specifically, it prints the values of the elements of the CE template parameter (as an array with Dim*NComponents elements).