Field Encapsulation Library

The FEL 2.2 User Guide

Patrick J. Moran Chris Henze
David Ellsworth

MRJ Technology Solutions
NASA Ames Research Center, M/S T27A-2
Moffett Field, CA, 94035, USA
{pmoran,chenze,ellswort}@nas.nasa.gov

NAS Technical Report NAS-00-002

January 3, 2000

Preface

This document describes version 2.2 of the Field Encapsulation Library (FEL), a library of mesh and field classes. FEL is a library for programmers — it is a "building block" enabling the rapid development of applications by a user. Since FEL is a library intended for code development, it is essential that enough technical detail be provided so that one can make full use of the code. Providing such detail requires some assumptions with respect to the reader's familiarity with the library implementation language, C++, particularly C++ with templates. We have done our best to make the explanations accessible to those who may not be completely C++ literate. Nevertheless, familiarity with the language will certainly help one's understanding of how and why things work the way they do. One consolation is that the level of understanding essential for using the library is significantly less than the level that one should have in order to modify or extend the library.

One more remark on C++ templates: Templates have been a source of both joy and frustration for us. The frustration stems from the lack of mature or complete implementations that one has to work with. Template problems rear their ugly head particularly when porting. When porting C code, successfully compiling to a set of object files typically means that one is almost done. With templated C++ and the current state of the compilers and linkers, generating the object files is often only the beginning of the fun. On the other hand, templates are quite powerful. Used judiciously, templates enable more succinct designs and more efficient code. Templates also help with code maintenance. Designers can avoid creating objects that are the same in many respects, but not exactly the same. For example, FEL fields are templated by node type, thus the code for scalar fields and vector fields is shared. Furthermore, node type templating allows the library user to instantiate fields with data types not provided by the FEL authors. This type of flexibility would be difficult to offer without the support of the language.

For users who may be having template-related problems, we offer the consolation that support for C++ templates is destined to improve with time. Efforts such as the Standard Template Library (STL) will inevitably drive vendors to provide more thorough, optimized tools for template code development. Furthermore, the benefits will become harder to resist for those who currently subscribe to the least-common-denominator "code it all in C" strategy.

May FEL bring you both increased productivity and aesthetic satisfaction.

Contents

1 Introduction **11**
- 1.1 An FEL example . . . 12
- 1.2 A graphic example . . . 15
- 1.3 FEL1 and FEL2 . . . 17
- 1.4 Acknowledgements . . . 17

2 Templates and Typedefs **19**
- 2.1 Templates . . . 19
- 2.2 Typedefs . . . 20

3 Vector and Matrix Classes **21**
- 3.1 The vector classes . . . 21
  - 3.1.1 FEL vectors as arguments to other libraries . . . 22
- 3.2 The matrix classes . . . 22

4 The FEL_time Class **25**
- 4.1 Multiple time representations: A caveat . . . 26

5 The FEL_cell Class **27**
- 5.1 The cell data members . . . 28
- 5.2 The FEL_vertex_cell class . . . 28

6 Positional Classes **29**
- 6.1 FEL_vertex_cell . . . 29
- 6.2 FEL_phys_pos . . . 29
- 6.3 FEL_structured_pos . . . 30
- 6.4 FEL_vector3f_and_int . . . 30

7 Memory Management **31**
- 7.1 Reference counted objects and pointers . . . 31
- 7.2 Reference counting and mutual exclusion . . . 33

8 Dynamic Casting **35**

9 Interpolation **37**
9.1 Setting interpolation modes . . . 37
9.2 Nearest neighbor interpolation . . . 38
9.3 Isoparametric interpolation . . . 38
9.4 Physical space interpolation . . . 39

10 Meshes and Fields **41**
10.1 Member function style . . . 41
10.2 Return values . . . 42

11 Meshes **43**
11.1 The mesh class hierarchy . . . 44
11.2 Setting and getting mesh properties . . . 44
11.3 Simplicial decomposition . . . 44
11.4 Point location and interpolation . . . 46
11.5 Coordinates . . . 47
11.6 Cell geometric properties . . . 48
11.7 Cell incidence relationships . . . 48
11.8 Adjacent cells . . . 49
11.9 Cardinality . . . 50
11.10 Cells and canonical enumeration . . . 50
11.11 PLOT3D IBLANK . . . 51

12 Structured Meshes **53**
12.1 Simplicial decomposition . . . 53
12.2 Cell incidence relationships . . . 53
12.3 Canonical cell enumeration . . . 54
12.4 Computational space support . . . 54
12.5 Structured mesh dimensions . . . 55
12.6 Axis-aligned structured meshes . . . 55
12.7 Curvilinear meshes . . . 56
12.8 Curvilinear mesh point location . . . 58
12.9 Curvilinear surface meshes . . . 59

13 Unstructured Meshes **61**
13.1 Constructing a tetrahedral mesh . . . 61
13.2 Cell incidence relationships . . . 62
13.3 Canonical cell enumeration . . . 62
13.4 Surfaces . . . 62
13.5 Point location . . . 63
13.6 Constructing a scattered vertex mesh . . . 63

14 Transformed Meshes **65**
14.1 How transformed meshes work . . . 67
14.2 The transformed mesh subclasses . . . 67

15 Multi-Zone Meshes **69**
15.1 Point location, IBLANK, and PLOT3D 69
15.2 Constructing a multi-zone mesh 70

16 Iterators **73**
16.1 Basic iterator usage 73
16.2 Iterators and ordering 75
16.3 Iterating over mesh subsets 75
16.4 Iterating over surfaces 77
16.5 Iterators and time 77

17 Fields **79**
17.1 Fields in general 79
17.2 Fields in context 79
17.2.1 Typeless fields 80
17.2.2 Typed fields 80
17.3 Fields in detail 83
17.3.1 Every field has a mesh 83
17.3.2 The `at_*()` calls 84
17.3.3 Iterating over fields 87
17.3.4 Eager fields 88
17.3.5 Field type “informant” functions 89
17.3.6 Odds and ends 90

18 Core Fields **93**
18.1 What are core fields? 93
18.2 The node buffer 93
18.3 Constructors and suppressed deallocation 94
18.4 An example 95
18.5 get_eager_field() 97

19 Derived Fields **99**
19.1 What is a derived field? 99
19.2 Lazy vs. eager evaluation 100
19.3 Mapping and interpolating 100
19.4 Built-in derived fields 101
19.4.1 Customizing the built-in derived fields 104
19.5 PLOT3D derived fields 105
19.6 Constructing a custom derived field 105
19.6.1 Writing a mapping function 106
19.6.2 Derived field declarations and constructors 108
19.6.3 Derived field checklist 109
19.6.4 A more or less complete derived field example 110

20 Differential Operator Fields **113**
20.1 Gradient, divergence, and curl 113
20.1.1 Grad 113
20.1.2 Div 113
20.1.3 Curl 114
20.2 First-order and second-order accuracy 114
20.3 Creating differential operator fields 115
20.4 “Chaining” differential operator fields 117

21 Instantiating Fields **119**
21.1 Basic type requirements 119
21.2 Differential operator field requirements 121

22 File I/O **123**
22.1 PLOT3D file reader functions 123
22.1.1 The PLOT3D flags 124
22.1.2 Automatic mesh type deduction 124
22.1.3 Reading mesh files 125
22.1.4 Getting information about structured mesh files 125
22.1.5 Reading solution files 126
22.1.6 Reading function files 127
22.1.7 Reading individual zones from multi-zone files 128
22.1.8 Making the PLOT3D readers more verbose 128
22.2 PLOT3D and paged file readers 128
22.3 The FITS file reader 130
22.4 The Vis5D file reader 130

23 Paged Meshes and Fields **131**
23.1 Introduction 131
23.2 How paged files work 132
23.3 Converting PLOT3D files to paged files 133
23.4 Using paged meshes and fields 133
23.5 Controlling memory usage 134
23.5.1 Pool size 134
23.5.2 Page priority hints 135

24 The PLOT3D Field Manager **137**
24.1 Constructing an `FEL_plot3d_field` 137
24.1.1 Constructing an `FEL_steady_plot3d_field` 137
24.1.2 Constructing a time-varying field manager 138
24.1.3 Constructing an `FEL_plot3d_q_field` 139
24.2 Creating primitive and derived PLOT3D fields 139
24.3 How the field manager works 141
24.4 Miscellaneous `FEL_plot3d_field` methods 142
24.5 An example 142
24.6 PLOT3D derived field “convenience functions” 143

24.7 PLOT3D derived field inventory arrays 143

25 Time-Varying Fields **149**
25.1 Working sets and callbacks 150
25.2 Time representations and conversions 152
25.3 Temporal interpolation 152
25.4 A time-varying field example 152

26 Time-Varying Meshes **155**
26.1 Single-zone time-varying meshes 156
26.2 Multi-zone time-varying meshes 157

A Glossary **159**

Chapter 1

Introduction

The Field Encapsulation Library (FEL) is a library for representing fields and meshes. The fields may represent the results from simulations or experimental observations. The main goals of FEL are to provide:

- a horizontal product supporting the quick development of new applications
- an extensible framework supporting a variety of mesh and field types
- a high-performance mesh-independent interface
- a library supporting work with large data sets

"Horizontal products" are reusable libraries that make the development of "vertical products" — applications — easier. FEL is a set of C++ classes designed to support not just the field and mesh types that are currently implemented, but also new mesh and field types in the future. A key part of the design is the development of a mesh-independent interface, i.e., an interface that allows users to write a single algorithm that will work with a variety of mesh and field types. The goal is to be able to introduce new types in the future and reuse algorithms written in terms of the FEL interface without modification. Finally, there are a number of features in FEL that are designed to make working with large data sets feasible for more users.

To develop an application using FEL, you should not find it necessary to read this entire document. The FEL release includes a primer which provides an overview of many features of FEL along with sample programs. The primer should appeal to those who prefer to "dive in" and learn by writing actual applications. On the other hand, the primer, in the name of brevity, focuses on library essentials. To get a more complete view of the features and design philosophy of FEL, you are encouraged to read on.

To convey the flavor of programming with FEL, we will now walk through a small example.

1.1 An FEL example

The following program reads in a mesh and field, then finds the cell with the greatest pressure gradient magnitude at its centroid:

```
#include "FEL.h"
int main(int argc, char* argv[])
{
  unsigned flags = FEL_deduce_mesh_type(argv[1]);
  FEL_mesh_ptr  mesh = FEL_read_mesh(argv[1], flags);
  FEL_plot3d_q_field_ptr q_field =
    FEL_read_q(mesh, argv[2], flags);

  FEL_float_field_ptr p_field =
    FEL_plot3d_make_pressure_field(q_field);

  FEL_vector3f_field_ptr grad_p_field =
    new FEL_gradient_of_float_field1(p_field);

  FEL_float_field_ptr mag_grad_p_field =
    new FEL_magnitude_of_vector3f_field(grad_p_field);

  mag_grad_p_field->set(FEL_SIMPLICIAL_DECOMPOSITION, 0);
  mag_grad_p_field->set(FEL_INTERPOLATION,
                        FEL_ISOPARAMETRIC_INTERPOLATION);
  int res;
  float field_value, max_value = 0.0;
  FEL_vector3f coords[8];
  FEL_phys_pos centroid;
  FEL_cell max_cell;
  FEL_cell_iter iter;
  for (mag_grad_p_field->begin(&iter); !iter.done(); ++iter) {
    mesh->coordinates_at_cell(*iter, coords);
    centroid.set(0.0, 0.0, 0.0);
    for (int i = 0; i < (*iter).get_n_nodes(); i++)
      centroid += coords[i];
    centroid /= (*iter).get_n_nodes();
    res = mag_grad_p_field->at_phys_pos(centroid, &field_value);
    if (res != FEL_OK) continue;
    cout << "Pressure gradient magnitude at " << *iter
         << " is " << field_value << endl;
    if (field_value > max_value) {
      max_value = field_value;
      max_cell = *iter;
    }
  }
  cout << "Maximum of " << max_value << " in " << max_cell << endl;
  return 0;
}
```

The example demonstrates several key features of FEL. The first is mesh independence — the same program works for fields based on a structured, unstructured, or multi-zone mesh. The mesh independence is achieved via FEL iterators and file reading functions that construct meshes and fields of the appropriate type. The particular iterator used in the example, a cell iterator, loops over the (hexahedral or tetrahedral) cells in a mesh. Using each cell, one can easily access mesh and field values. The example also illustrates how FEL makes it easy for the user to compose new fields in terms of existing ones. The field data in the example is known "solution data," in other words, data produced by a flow solver. Starting with a solution data field (`q_field`), one can construct a pressure field, a gradient field and a gradient magnitude field, each in one statement. The program also shows how one can access field values at arbitrary positions using just one function call (`at_phys_pos`).

There are many other capabilities beyond those shown in the example, such as support for time-varying fields and transformed fields. Rather than inventory all the capabilities of FEL here, the remainder of this section walks through the first example and mentions the chapters in this document in which to look for more information.

```
unsigned flags = FEL_deduce_mesh_type(argv[1]);
FEL_mesh_ptr mesh = FEL_read_mesh(argv[1], flags);
FEL_plot3d_q_field_ptr q_field =
  FEL_read_q(mesh, argv[2], flags);
```

The first four lines read in mesh and field data from the files named by `argv[1]` and `argv[2]`. The program first uses the mesh type deducer function to deduce `flags` indicating the particular file type (Chapter 22). FEL currently supports reading PLOT3D files and "Enterprise" paged files. Mesh and field objects based on paged files do not load the whole data set into memory, but instead work in a demand-driven style where only subsets of the data are loaded as needed (Chapter 23). The user can also read in data from a file in some other format and then construct meshes or fields directly. (Chapters 12, 18).

Fields and meshes in FEL are reference counted. The FEL type names with the `_ptr` suffix are all "smart pointers" that refer to reference counted objects (Chapter 7). For the most part, the pointers behave just like C-style pointers. FEL also uses C++ templates, which support parameterized types. For instance, fields in FEL are parameterized by their node type (Chapter 17). FEL uses typedefs to hide many of the common template instantiations, so that most FEL programs can be written without ever using template syntax directly (Chapter 2).

```
 FEL_float_field_ptr p_field =
   FEL_plot3d_make_pressure_field(q_field);

FEL_vector3f_field_ptr grad_p_field =
  new FEL_gradient_of_float_field1(p_field);

FEL_float_field_ptr mag_grad_p_field =
  new FEL_magnitude_of_vector3f_field(grad_p_field);
```

Here we create three *derived* fields from the fundamental solution data. The first field uses a built-in function to convert PLOT3D solution vectors to pressure values (Chapter 24). The second field applies a differential operator ("grad") to the pressure field, producing a vector field from a scalar field (Chapter 20). The third field takes the norm of the gradient field, yielding a scalar field (Chapter 19). This progression illustrates how FEL allows new fields to be defined in terms of preexisting ones.

The derived and differential operator fields require almost no storage, since all values are computed on demand. This demand-driven or "lazy" evaluation approach provides a significant advantage when working with large data sets, since the memory and computational requirements of precomputing a derived value over a whole field can be prohibitive.

```
mag_grad_p_field->set(FEL_SIMPLICIAL_DECOMPOSITION,0);
mag_grad_p_field->set(FEL_INTERPOLATION,
                      FEL_ISOPARAMETRIC_INTERPOLATION);
```

These directives tell FEL how to conduct some of its numerical business (Chapters 9, 11). FEL supports several spatial interpolation modes (Chapter 9). The simplicial decomposition flag controls whether FEL decomposes each mesh cell into simplices (e.g., each hexahedron into tetrahedra) for cell-based operations such as interpolation (Chapter 11).

```
int res;
float field_value, max_value = 0.0;
FEL_vector3f coords[8];
FEL_cell max_cell;
```

In addition to meshes and fields, FEL provides a number of fundamental data types, such as vectors (Chapter 3), positional classes (Chapter 6), and cells (Chapter 5). One can also declare arrays these types. For instance, the `coords` array above can be used to store the coordinates for each vertex of a cell.

```
FEL_cell_iter iter;
for (mag_grad_p_field->begin(&iter); !iter.done(); ++iter) {
```

FEL supports a mesh-independent way to loop over the cells of a mesh: *iterators*. Using an iterator, one can write algorithms that work with fields based on a variety of mesh types, without having to provide conditional statements that are type-dependent (Chapter 16).

```
  mesh->coordinates_at_cell(*iter, coords);
  centroid.set(0.0, 0.0, 0.0);
  for (int i = 0; i < (*iter).get_n_nodes(); i++)
    centroid += coords[i];
  centroid /= (*iter).get_n_nodes();
```

A fundamental operation in FEL is querying a mesh for geometric information, in this case the coordinates of the cell currently represented by the iterator. The `coordinates_at_cell` call returns coordinates for each vertex of the cell (Chapter 11). The cell, in turn, is queried for its number of nodes in order to calculate the centroid.

```
    res = mag_grad_p_field->at_phys_pos(centroid, &field_value);
    if (res != FEL_OK) continue;
    cout << "Pressure gradient magnitude at " << *iter
         << " is " << field_value << endl;
    if (field_value > max_value) {
      max_value = field_value;
      max_cell = *iter;
    }
  }
  cout << "Maximum of " << max_value << " in " << max_cell << endl;
}
```

One key group of member functions on FEL fields are the "`at`" calls, which support requests for field values. Some types of `at` calls merely retrieve field values at nodes, while others, such as `at_phys_pos` support field value queries at arbitrary physical positions. To comply with the `at_phys_pos` call in the example, FEL must find the mesh cell containing (`centroid`) and then use the geometry of the cell and the field values at its vertices to interpolate and get the `field_value` at `centroid`. FEL provides alternate versions of `at_phys_pos` (via function overloading) where the user can provide extra arguments intended to accelerate the point location (Chapter 11) or interpolation (Chapter 17) steps.

The `at` call returns 1 to signify success. It is possible that the `at` call in the example could fail in some cases. For instance, if the cell containing `centroid` were at the boundary of the mesh and had nonplanar faces, then the point location code might conclude that the given point is outside the mesh. In general, it is important that the user check the return value of the `at` calls in order to be assured that the field values produced by the call are valid.

This example is representative of the sorts of things one can do with FEL. The remainder of this document provides further details, options, examples, and possibilities. The accompanying FEL Reference Manual provides a list of the FEL classes and a summary of the public member functions for each class.

1.2 A graphic example

FEL *per se* does not do graphics, but some of the first applications written with FEL do. Graphics are fun, so we include one figure illustrating the output of a visualization tool called `gel`, which was written with FEL. Figure 1.1 is just a small taste of what one can do with FEL; there will be more pictures to follow.

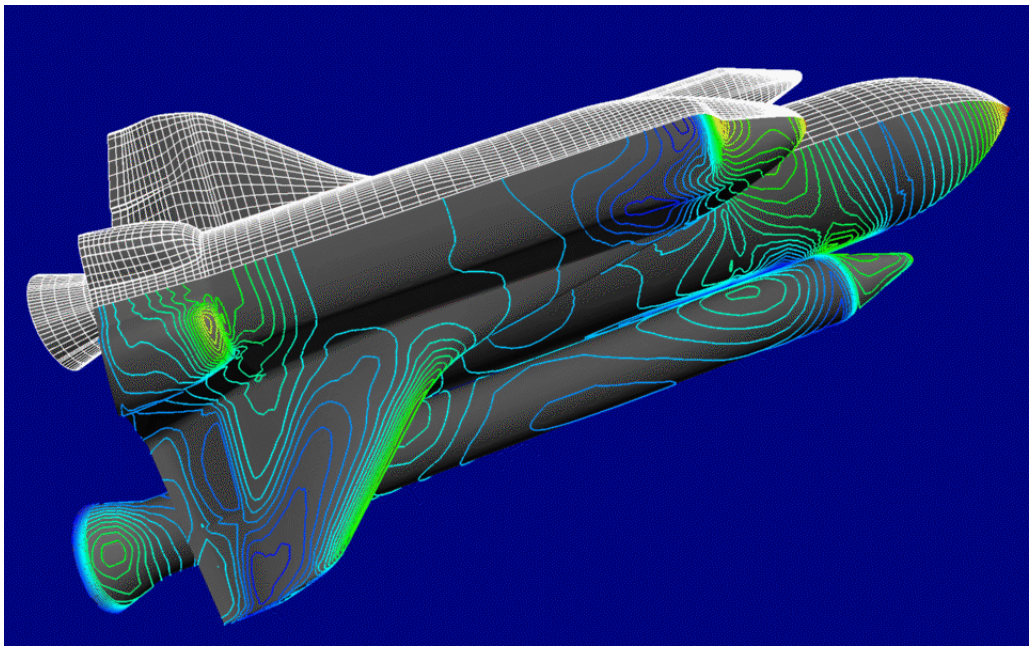


Figure 1.1: A space shuttle launch vehicle visualization: the left side of the model displays edges from the mesh, the right side shows contour lines for a pressure derived field. (Data by Pieter Buning, visualization courtesy of Tim Sandstrom.)

1.3 FEL1 and FEL2

An initial version of the Field Encapsulation Library was first presented at Visualization '96 [BKGY96]. The library described in this document represents a fundamental redesign and complete rewrite of "FEL1". Some of the features that are new to the 2.0 release of FEL include:

- derived fields, with demand-driven evaluation
- built-in support for the over 50 standard derived fields defined by PLOT3D [WBPE92]
- differential operators, employing either first- or second-order techniques, also with demand-driven evaluation
- transformed meshes and fields
- iterators
- support for arbitrary field node type, based on C++ templates
- an interface supporting the easy composition of fields
- much better opportunities for code reuse in the development of new classes
- support for a variety of cell types
- multiple, user-selectable spatial interpolation modes
- multiple, user-selectable simplicial decomposition modes
- working set management of time-series data
- temporal interpolation
- significantly improved memory management, including reference counting for meshes, fields, and interpolants
- support for more mesh types, including regular and unstructured
- support for surface meshes
- additional file reading capabilities, including support for reading PLOT3D "FORTRAN unformatted" files on either a workstation or Cray
- support for the reuse of the interpolation data associated with a cell
- integrated support for paged meshes and fields

1.4 Acknowledgements

We would like to thank the members of the NAS Data Analysis Group for their constructive feedback, patience, and encouragement during the FEL2 development process.

Chapter 2

Templates and Typedefs

FEL makes significant use of two C++ features–templates and typedefs–in order to increase the flexibility and power of the library without decreasing ease of use. Typedefs are available in C, thus they are likely to be familiar to many users. Templates, on the other hand, are new with C++ and are probably less familiar to most. In FEL the two features are used together in order to make it easy for the user to reap the benefits of templates without having to learn template syntax.

2.1 Templates

Templates are a feature of C++ allowing one to implement algorithms and classes where the specification of particular types is deferred. For instance, one could have a linked list of objects where the object type is specified by a token "`T`". The user can then declare that he or she wants a list of objects of a particular type, for instance `float`, by using the template syntax: `list<float>`. Informally, one can imagine the compiler and linker working together to generate the code required by a particular instantiation, such as a list of `float`s, by outputing customized source with the global substitution `float` for `T`, and then compiling the dynamically generated source. How and when template code is actually instantiated varies from system to system. The user experiences these differences as variations in how programs are compiled and linked, but the variations do not change the style in which one writes applications using FEL.

The most prominent use of templates in FEL is in the node type `T` of `FEL_typed_field<T>`. Templates allow the library to use one common set of classes for representing fields, regardless of field node type. There are two major benefits of the templated node approach. The first benefit has implications primarily for the user. The use of templated nodes means that one can create fields with node types which are not built into the library, without modifying the library. For instance, a computational scientist could create a scalar field where each scalar is represented by a double by writing `FEL_core_field<double>`. (The class `FEL_core_field<T>` is described in Chapter 18). The compiler and linker automatically handle generating the appropriate code. One could also instantiate fields where the node type is a structure

containing an aggregation of values, such as the solution vector used by a particular solver.

A second benefit of the template approach impacts the developers of FEL more directly than the users. The templated field node type allows the consolidation of classes, such as those for scalar fields and vector fields, which were formerly separate. This consolidation has many positive benefits: templated fields mean that there is less library code to maintain, and bug fixes in the field classes do not have to be propagated to fields of each node type variation.

2.2 Typedefs

Typedefs (provided by both C and C++) allow one to define convenient names for what are typically more complicated types. In FEL, typedefs are used primarily to hide template syntax from the users. For example, the statement:

```
typedef FEL_typed_field<float> FEL_float_field;
```

makes it possible to write `FEL_float_field` where one would have to write `FEL_typed_field<float>` without the typedef. Further typedefs are provided for working with `FEL_typed_field<float>` instances. For example, the statement:

```
typedef FEL_pointer<FEL_typed_field<float> >
  FEL_float_field_ptr;
```

makes it easy to declare a pointer to a float field (`FEL_float_field_ptr`) without using the more verbose template syntax. FEL provides typedef names for the most commonly used template instantiations; in particular, the types related to float fields and vector fields (where the vectors are composed of 3 floats) are included in the library. For many applications, one can use FEL without ever using template syntax. For more advanced users, the option of going to template notation is always available. For example, to instantiate a field with a new node type would require using at least some template notation.

The typedef names built into FEL are described in the following chapters, introduced with the templated classes for which they are written.

Chapter 3

Vector and Matrix Classes

FEL provides basic vector and matrix support for the user. The support is primarily for small vectors (2, 3, or 4 components) and small, square matrices. The vector and matrix classes are written using C++ templates, and typedefs are provided for the most commonly used instantiations.

3.1 The vector classes

The vector classes are as follows and are listed in Table 3.1. FEL provides specific classes for vectors of length 2, 3, or 4, rather than using the arbitrary length vector class `FEL_vector<N,T>`, for efficiency reasons.

The vector class typedefs built-in to FEL are listed in Table 3.2. The convention for typedef names is to append the original templated class name by the vector length, if not already provided, and a letter specifying the component type. The suffixes used in FEL are `i`, `f`, and `d`, corresponding to the C types `int`, `float`, and `double`, respectively.

FEL provides most of the basic math operators for the vector classes, so for example, one can write statements such as:

```
float f;
FEL_vector3f a, b;
FEL_vector3f c(1.0, 2.0, 3.0), d(13.0, 17.3, 21.1);
```

Class	Description
`FEL_vector2<T>`	vector of 2 type T components
`FEL_vector3<T>`	vector of 3 type T components
`FEL_vector4<T>`	vector of 4 type T components
`FEL_vector<N,T>`	vector of N type T components

Table 3.1: The basic vector class templates.

Typedef	Template Class	Component Type
FEL_vector2i	FEL_vector2<T>	int
FEL_vector2f	FEL_vector2<T>	float
FEL_vector2d	FEL_vector2<T>	double
FEL_vector3i	FEL_vector3<T>	int
FEL_vector3f	FEL_vector3<T>	float
FEL_vector3d	FEL_vector3<T>	double
FEL_vector4i	FEL_vector4<T>	int
FEL_vector4f	FEL_vector4<T>	float
FEL_vector4d	FEL_vector4<T>	double

Table 3.2: The vector typedefs.

```
a = 3 * c + d;
b = a + c - d;
f = a[0];
b[0] = 10.0;
b.set(10.0, 20.0 30.0);

cout << "a = " << a << endl;
```

The final statement shows the use of the C++ style output stream operator which can be handy for writing out the contents of vectors, for example for debugging. See the file `FEL_vector.h` for a complete listing of the operations supported for vector objects.

3.1.1 FEL vectors as arguments to other libraries

Occasionally, one may wish to use FEL vector values as arguments to routines provided by other libraries. For example, one may want to make calls to OpenGL graphics library routines taking vector arguments, without having to repackage the data into a different structure. The FEL vector method `v()` returns a pointer which can be used with routines requiring a C-style pointer to the beginning of an array. For instance:

```
FEL_vector3f v3f(1.0, 2.0, 3.0);
FEL_vector3d v3d(4.0, 5.0, 6.0);
glVertex3fv(v3f.v());
glVertex3dv(v3d.v());
```

The `v()` method is defined in `FEL_vector.h` and should be easily inlined by most compilers.

3.2 The matrix classes

FEL provides typedefs for the instantiations for float and double versions of the $N \times N$ matrices, where N is 2, 3, 4, 5, 6, or 8. The suffix convention follows along the same

Class	Description
FEL_matrix22<T>	2 by 2 matrix of type T components
FEL_matrix33<T>	3 by 3 matrix of type T components
FEL_matrix44<T>	4 by 4 matrix of type T components
FEL_matrix55<T>	5 by 5 matrix of type T components
FEL_matrix66<T>	6 by 6 matrix of type T components
FEL_matrix88<T>	8 by 8 matrix of type T components

Table 3.3: The matrix templates.

lines as that used for vectors. For instance, FEL_matrix33f signifies a 3 by 3 matrix with float components. Internally matrices are stored in row-major order, i.e., "C" style rather than FORTRAN style. FEL provides the basic math operators for matrices and for matrices with vectors, such as multiplication. The library also provides the function FEL_invert for inverting matrices. The inversion code is written using an analytical technique for FEL_matrix22<T> and FEL_matrix33<T>. For larger matrices the library uses Gauss-Jordan elimination with partial pivoting. See the file FEL_matrix.h for a complete listing of the operations supported for matrix objects.

Chapter 4

The FEL_time Class

FEL represents time using the class `FEL_time`. A class is used to represent time, rather than simply a C float or int, because there is more than one time representation in which a suer may want to work. `FEL_time` supports 3 representations:

- physical
- computational
- integer computational

Physical time corresponds to the non-dimensional time used by the flow solver in an unsteady flow simulation. Computational time corresponds to a temporal discretization, where a time-varying data set is represented as a sequence of time steps. The first time step would be at computational time 0, the second at 1, and so on. The user can request data at a computational time intermediate to the time steps by providing a value with a fractional part; e.g., computational time 0.5 to get data temporally interpolated half way between the first and second time steps. Integer computational time, also known as "time step", specifies a specific step in a time-series data set, without temporal interpolation.

`FEL_time` contains a tag indicating the representation currently in use and a union of an int and a float where the time itself is stored. By default the time representation is `FEL_TIME_REPRESENTATION_UNDEFINED`. The user can set time using calls such as `set_physical`. See `FEL_time.h` for the list of `set_*` and `get_*` calls. In most applications, one will typically work with one representation for time. Physical time provides a representation that is independent of how a flow solver chose to save snapshots through time, e.g., whether the data were written out at a fixed physical time interval or at adaptive intervals. Working in terms of time steps makes it easy for the user to access the data in a time series without regard to how the data are positioned in physical time, e.g., to compute statistics over all the time steps. Floating-point computational time supports the added flexibility of expressing times requiring temporal interpolation.

4.1 Multiple time representations: A caveat

In applications where one works with both physical and computational time simultaneously, there is a caveat. The class `FEL_time` defines the operator `==`, i.e., an equality test. When objects containing time, such as physical positions and vertex positions (described in the following chapters), are compared for equality, they compare their respective time values. Given a physical time value and a computational time value, an `FEL_time` object would have to convert one representation to the other in order to compare like representations. Unfortunately, `FEL_time` objects do not contain the information necessary to map between physical and computational time. In general the choice of which mapping to use is ambiguous, since there could be many time-varying fields, and they may not all use the same mapping. `FEL_time` operator `==` returns `false` if it cannot compare the times.

Mixing time representations in the arguments to FEL calls should not cause problems internal to FEL, to the best of our knowledge. Mixing time represention when doing `at_phys_pos` calls may cause some minor inefficiencies, though it should not cause the library to return incorrect answers. This case is revisited later when we describe the use of `at_phys_pos` (Chapter 17). The time mixing caveat should only be of concern to the user if he or she uses the `==` operator with the positional classes described in the next chapter: `FEL_vertex_cell`, `FEL_phys_pos` and `FEL_structured_pos`.

There is no easy solution to the "got one representation, need another, don't know how to convert" problem. The library could choose one time format to use throughout the application programmer's interface, but no matter what the choice, there will be cases where the selected format is awkward for the user. Physical time is a higher-level representation, but it is not convenient when one wants to loop over time steps. On the other hand, some applications would like to treat an unsteady data set as continuous in time and are not interested in how the samples were chosen, thus physical time is a better choice than computational (or time steps). The FEL design opts for flexibility and ease-of-use, at the risk of unexpected behavior in some obscure cases.

Chapter 5

The FEL_cell Class

The `FEL_cell` class defines a general purpose object for representing cells, the basic discretization building block for computational domains. FEL uses a general cell definition, where cell types include vertices, edges, triangles, quadrilaterals, tetrahedra, and hexahedra. Cells are used, for example, for point location: given a physical point and a hexahedral mesh, the library can return a cell containing the point. The cell types which can be represented by `FEL_cell` are listed in Table 5.1.

Cells may be grouped by their dimension and referred to as k-cells or k-dimensional cells. Every cell type has an associated dimension, the **Dimension** column of Table 5.1 lists the dimension for each cell type supported by FEL. Writing in terms of k-cells is more convenient in some cases, for instance, one can speak of the primary cells of a surface being 2-cells, without having to distinguish between triangles, quadrilaterals, or some other type of polygon.

An important concept related to cells is faces. A cell c in a mesh $\mathcal{M}$ is the *face* of another cell d in $\mathcal{M}$ if c is defined by a subset of the vertices defining d. The most common use of the term face is with hexahedral and tetrahedral cells: a hexahedron has quadrilateral faces, and a tetrahedron has triangle faces. But the term is more general: hexahedra and tetrahedra also have edge and vertex faces, even an edge has vertex faces. The topic of cells, faces, and incidence relationships will be revisited in the mesh chapter (Chapter 11).

Cell Type	Letter	Dimension	Vertices
FEL_CELL_VERTEX	V	0	1
FEL_CELL_EDGE	E	1	2
FEL_CELL_TRIANGLE	F	2	3
FEL_CELL_QUADRILATERAL	Q	2	4
FEL_CELL_TETRAHEDRON	T	3	4
FEL_CELL_HEXAHEDRON	H	3	8

Table 5.1: The FEL cell types.

Name	Type	Default
type	FEL_cell_type_enum	FEL_CELL_UNDEFINED
subid	short	-1
ijk	FEL_vector3i	(none)
zone	int	FEL_ZONE_UNDEFINED
time	FEL_time	FEL_time()

Table 5.2: The FEL_cell data members.

5.1 The cell data members

The 5 data members of an FEL_cell object are listed in Table 5.2. The type specifies one of the types listed in Table 5.1, or FEL_CELL_UNDEFINED. The zone would specify a zone in a multi-zone mesh. Time is used when working with time-varying data. The ijk and subid data members are used to specify a particular cell of a given type. The particular usage of ijk and subid for each type of cell depends on whether the mesh is structured or unstructured. See Chapters 12 and 13.

5.2 The FEL_vertex_cell class

An FEL_vertex_cell object represents a vertex in a mesh. The class is derived from FEL_cell and can be used as an argument to any routine expecting an FEL_cell argument. The FEL_vertex_cell class exists in order to make the development of algorithms written in terms of mesh vertices a bit easier. The class provides the opportunity to get some compile-time checking of routines which work only with vertices; and FEL provides routines for vertex cells which have some slight optimizations over the general cell routines.

Chapter 6

Positional Classes

The position of an individual point in space can be represented by an `FEL_vertex_cell`, `FEL_phys_pos`,or `FEL_structured_pos` object, see Table 6.1. All three classes also contain a representation for time, so the objects in general can represent a position in space and time.

6.1 FEL_vertex_cell

The `FEL_vertex_cell` class is used for specifying an individual vertex in a mesh. `FEL_vertex_cell` is derived from the class `FEL_cell` and inherits most of its functionality from the cell class. `FEL_cell` and `FEL_vertex_cell` are described in the previous chapter.

6.2 FEL_phys_pos

An `FEL_phys_pos` object represents a position in physical space and time. One can request field values at a physical position using statements such as:

```
int res;
float f;
FEL_phys_pos p(1.2f, 3.3f, 0.0f);
res = float_field->at_phys_pos(p, &f);
if (res == 1)
  cout << "the field at " << p << " is " << f << endl;
```

Class	Derived From
`FEL_vertex_cell`	`FEL_cell`
`FEL_phys_pos`	`FEL_vector3f`
`FEL_structured_pos`	`FEL_vector3f`

Table 6.1: The FEL positional classes and their parent classes.

Since FEL_phys_pos is derived from FEL_vector3f, FEL_phys_pos can be used as with routines expecting an FEL_vector3f argument; the time component of the FEL_phys_pos is ignored. For example, to get the physical coordinates at a vertex, one could write:

```
mesh->coordinates_at_vertex_cell(vertex_cell, &phys_pos);
```

One can also use FEL_phys_pos objects with the mathematical operators += and -=, where the right-hand-side argument is of type FEL_vector3f. For instance, one could add to the spatial coordinates of a physical position by writing:

```
FEL_phys_pos p(10.0f, 11.0f, 12.0f);
cout << "before: " << p << endl;
p += FEL_vector3f(1.1f, 2.2f, 3.3f);
cout << "after:  " << p << endl;
```

The statements with cout illustrate the use of the C++ ostream operator defined for FEL_phys_pos. The ostream operator displays the physical coordinates; the time is also displayed if it is defined. The output from the statements above would look like:

```
before: (10, 11, 12)
after:  (11.1, 13.2, 15.3)
```

6.3 FEL_structured_pos

The class FEL_structured_pos represents positions in structured meshes described by floating-point i, j, and k values. FEL_structured_pos values are sometimes known as "computational coordinates", though they are only applicable to structured meshes (or a structured zone in a multi-zone mesh). FEL_structured_pos objects also contain an integer zone value and time. The zone comes into play when working with FEL_multi_mesh objects, and time is relevant to time-varying objects.

6.4 FEL_vector3f_and_int

The FEL_vector3f_and_int class, as its name suggests, represents objects consisting of a vector of 3 floats and an integer. The FEL_vector3f_and_int class is used primarily for working with meshes that include what is known in PLOT3D data sets as IBLANK [WBPE92]. IBLANK values are integers, one per vertex. Users see the FEL_vector3f_and_int type in the mesh calls coordinates_and_iblank_at_vertex_cell() and coordinates_and_iblank_at_cell(). See Chapter 11 for more on these routines.

Chapter 7

Memory Management

FEL allocates and deallocates memory using the standard C++ `new` and `delete` operators. In general, FEL objects that allocate memory are also responsible for deallocating the same memory. Likewise, memory allocated by the user is the user's responsibility for deallocation. There is one key exception to this convention with FEL: memory allocated for buffers passed to constructors of mesh and field subclasses becomes the responsibility of FEL to deallocate. For instance, a buffer with float values passed to the constructor of an `FEL_core_float_field` would become the responsibility of the core field to deallocate when the core field is destructed. Users who work solely with meshes and fields constructed via the file reader functions (see Chapter 22) do not have to be concerned with this issue, since the the convenience functions allocate and pass the buffers appropriately. If the user constructs a mesh or field explicitly, providing data buffer arguments, then it necessary that the buffers be allocated using the C++ `new []` operator, i.e. the C++ operator for allocating arrays of objects. This requirement is necessary because ultimately FEL will deallocate the buffer using the C++ `delete []` operator, and in general it is hazardous to allocate memory in one manner (e.g. `malloc`) and deallocate in another (e.g. `delete []`). With `FEL_core_field` instances, the user also has the option of providing a flag that will suppress the field's attempt to deallocate the solution data buffer. In this case, how the buffer was allocated is irrelevant to FEL, and the ultimate responsibility for deallocating the buffer would remain with the user.

7.1 Reference counted objects and pointers

Reference counting is a technique for tracking the number of references to each object in a set of objects. Reference counting supports sharing objects and the detection of a safe time to deallocate an object (i.e., when the number of references to an object goes to 0). In FEL, the most prominent use of reference counting is for mesh and field instances. Reference counting allows the user to create meshes and fields and then build additional fields, such as derived fields and differential operator fields, in terms of the original fields. The reference count book-keeping is done transparently. The

reference counting frees the user from some relatively tedious and error prone work and in general is unobtrusive.

FEL uses two classes that work together to implement reference counting: `FEL_reference_counted_object` and `FEL_pointer<T>`. The `FEL_mesh` and `FEL_field` classes inherit from `FEL_reference_counted_object`. The `FEL_pointer<T>` class is used to represent references to an object derived from `FEL_reference_counted_object`. The template type `T` stands for the particular class being pointed to, e.g., `FEL_mesh`. FEL provides typedef names for most pointers to reference counted objects, using the convention that pointer names take the class name of the object pointed to and append "`_ptr`". For example, `FEL_mesh_ptr` and `FEL_float_field_ptr` represent pointers to meshes and float fields. The pointer class provides definitions for the operations that can potentially change the reference count of an object. For instance, `FEL_pointer<T>` defines an assignment operator, so that when one pointer is assigned to another, one reference is potentially incremented and one is potentially decremented. We say "potentially" since pointers can have the value `NULL`, and no counts need to be changed when a pointer has a NULL value.

The `FEL_pointer<T>` class is designed so that one can use pointer instances in a manner similar to raw C-style pointers, e.g., one can write statements using "`->`" syntax. There are a few cases where FEL pointer instances differ in usage from raw pointers:

- default initialization
- casting
- comparisons with `NULL`
- deleting

FEL pointers, unlike raw C-style pointers, are guaranteed to always be initialized to NULL by default. This difference is not necessarily noticeable to the user, since one can still manually initialize pointers as one would do with raw pointers.

A second case where `FEL_pointer<T>` instances differ from raw pointers is in downcasting. For example, one may have a pointer to an `FEL_mesh`, but what one may need is a pointer to an `FEL_structured_mesh`, in order to access a method that is specific to structured meshes. With raw pointers one could write:

```
FEL_mesh* mesh;
FEL_structured_mesh* structured_mesh;
...
structured_mesh = (FEL_structured_mesh*) mesh;
```

Unfortunately, if `FEL_mesh*` and `FEL_structured_mesh*` were replaced by `FEL_mesh_ptr` and `FEL_structured_mesh_ptr`, the previous excerpt would not work. The cast fails because `FEL_mesh_ptr` (i.e., `FEL_pointer<FEL_mesh>`) technically is not a C pointer; there is no direct conversion from `FEL_mesh_ptr` to `FEL_structured_mesh_ptr` (i.e., `FEL_pointer<FEL_structured_mesh>`). The cast can be accomplished, but the required syntax is awkward. For the previous example:

```
FEL_mesh_ptr mesh;
FEL_structured_mesh_ptr structured_mesh;
...
structured_mesh = (FEL_structured_mesh*) (FEL_mesh*) mesh;
```

The less-than-obvious syntax is one motivation for providing downcasting macros for the user. See the following chapter for a list of the casting macros provided by FEL and some additional motivation for using the macros.

A third case where `FEL_pointer<T>` instances differ from raw pointers occurs when testing whether or not a pointer has the value `NULL`. `FEL_pointer<T>` class has the member function `null()` that can be used for this purpose, for example:

```
FEL_mesh_ptr mesh;  // default initialization
assert(mesh.null());

... assign non-NULL value to mesh
assert(!mesh.null());
```

Two `FEL_pointer<T>` instances (both instantiated with the same type for `T`) can also be tested for equality or lack there of using the `==` or `!=` operators.

FEL pointer objects differ from raw pointers in a fourth respect: deletion. Whereas in general one can call `delete` directly on a heap allocated object, the equivalent statement will not work with an FEL pointer instance. In a sense, this should not be surprising, reference counting takes responsibility for deallocating an object when there are no more references; the user should not have to take the same responsibility. Nevertheless, there are a few occasions where the user may want to cause the deallocation to happen sooner. For example, one may want to free memory used by a large mesh or field that one knows will not be used again. One can do this indirectly by assigning `NULL` to FEL pointers. Assuming that there are no other references to the mesh or field, assigning `NULL` to the remaining reference will decrement the count to 0 and induce the reference counting mechanism to do the deallocation.

Finally, some readers might observe in the previous discussion that it is possible to obtain the raw pointers to reference counted objects. One may be tempted to do this, since users are more familiar with C-style pointers and casting than the reference counting support classes of FEL. One should, however, resist this temptation. Reference counting requires little overhead, so performance is typically not an issue. Furthermore, the reference counting mechanism provided by FEL is robust, but all bets are off if one breaks through the abstraction and starts handling raw pointers directly. The potential consequences for defeating the reference counting are memory mismanagement bugs which can be very difficult to track down.

7.2 Reference counting and mutual exclusion

The FEL reference counting mechanism is also designed to support the use of reference counted objects in a multi-threaded environment. In a multi-threaded scenario, it is possible that several threads can take actions that change the reference count of a shared

object, such as mesh or field. Such actions include assigning one `FEL_pointer<T>` instance to another or passing a pointer object to a subroutine. Since such actions can take place simultaneously, it is important that the changes to the reference counts be made atomically. In other words, only one thread should be able to change the count at one time. FEL provides reference counting with critical section protection for count changes via the class `FEL_mutex_reference_counted_object` which is derived from `FEL_reference_counted_object`. The "`mutex`" version of the class uses a locking primitive provided by the task synchronization library (`TSL_mutex`) to ensure mutual exclusion. Meshes, fields, and interpolants in FEL are all derived from `FEL_mutex_reference_counted_object`, thus the reference counting for those objects should still work reliably in multi-threaded applications.

Chapter 8

Dynamic Casting

There are occasions when one has a pointer to a base class, but what one needs is a pointer to a class derived from the base class. For example, one may have a pointer to a mesh: `FEL_mesh_ptr`, but need a pointer to a structured mesh (a subclass of mesh): `FEL_structured_mesh_ptr`. The typical case where a pointer to a subclass is necessary arises when one needs to call a method that is available only from the subclass, for instance a method specific to structured meshes. Casting from one type to another type that is lower in the same class hierarchy is known as *downcasting*. FEL is designed to try to minimize the amount of downcasting required, nevertheless there are still cases where it is necessary. One hazard of downcasting is that it is possible to do it incorrectly, e.g., if the mesh pointer were referring to an unstructured mesh, a structured mesh cast would lead to dire consequences. Unfortunately, incorrect casts in general cannot be detected at compile-time; it is not until run-time that they become apparent.

C++ provides a standard way to downcast safely: dynamic casts. Using a dynamic cast, one can downcast a pointer to a particular subclass. If the cast is legal, then the result is a pointer to the subclass; if the cast is not legal, then the result is NULL. Thus one can downcast and then check if the cast completed successfully before continuing. Dynamic casting is a relatively new C++ feature, and it is not supported by some older compilers. FEL provides macros that behave like dynamic casts for the most common casts of FEL objects. The naming convention for the casts is `FEL_`*f*`_TO_`*t*`_CAST`, where an FEL pointer to a type *f* cast to an FEL pointer to a type *t*. For example `FEL_MESH_TO_STRUCTURED_MESH_CAST()` takes an `FEL_mesh_ptr` argument and returns an `FEL_structured_mesh_ptr`, or NULL if the cast is not legal. One difference between the FEL macros and C++ dynamic casting comes in the case where the argument is a NULL pointer. The C++ standard dictates that the dynamic cast of a NULL pointer should cause an exception, the FEL macro simply returns NULL. Table 8.1 lists the casting macros provided by FEL. Though there are many macros defined by the library, the need for them in user code is relatively infrequent. As the design of the library evolves, we are working towards further reducing the need for downcasting calls.

Cast
FEL_OBJECT_TO_MESH_CAST()
FEL_OBJECT_TO_FIELD_CAST()
FEL_MESH_TO_STRUCTURED_MESH_CAST()
FEL_MESH_TO_UNSTRUCTURED_MESH_CAST()
FEL_FIELD_TO_FLOAT_FIELD_CAST()
FEL_FIELD_TO_VECTOR3F_FIELD_CAST()
FEL_FIELD_TO_PLOT3D_Q_FIELD_CAST()
FEL_FIELD_TO_CORE_FLOAT_FIELD_CAST()
FEL_FIELD_TO_CORE_VECTOR3F_FIELD_CAST()
FEL_FIELD_TO_TIME_SERIES_FLOAT_FIELD_CAST()
FEL_FIELD_TO_TIME_SERIES_VECTOR3F_FIELD_CAST()
FEL_FIELD_TO_TIME_SERIES_PLOT3D_Q_FIELD_CAST()
FEL_INTERPOLANT_TO_HEXAHEDRAL_ISOPARAMETRIC_CAST()

Table 8.1: The FEL dynamic casting macros.

User code with an FEL dynamic cast macro would typically look like:

```
FEL_mesh_ptr mesh;
FEL_structured_mesh_ptr structured_mesh;
...
structured_mesh = FEL_MESH_TO_STRUCTURED_MESH_CAST(mesh);
if (!structured_mesh.null()) {
  ...
}
else {
  error ...
}
```

There is a second reason for the use of macros with downcasting. Pointers to FEL meshes and fields are not raw C-style pointers; rather the types such as `FEL_mesh_ptr` or `FEL_float_field_ptr` are typedef names for `FEL_pointer` objects. The classes `FEL_pointer` and `FEL_reference_counted_object` are used together to provide reference counting support in FEL (as described in the previous chapter). Unfortunately, straight-forward C-style downcasting does not work with FEL pointers, though it is still possible to downcast using a more arcane syntax. The macros are intended to hide that syntax.

Chapter 9

Interpolation

One of the most useful features of FEL is that it allows gridded, spatially discrete data to be treated as a continuous domain. In fact, this capability supports the key abstraction of a *field* and is one of the primary aims of the library. The central mechanism underlying the field abstraction is *spatial interpolation*. FEL supports temporal interpolation, too; this is described separately in Chapter 25.

9.1 Setting interpolation modes

In FEL, queries for field values at arbitrary physical space locations (i.e., `at_phys_pos()`, see Chapter 17) invoke a *point location* algorithm which finds the mesh cell containing the query point. FEL then uses the geometry of this cell, and the field values at its vertices, to determine a field value at an interior point. The last step can be done in one of three ways, and you tell FEL which method you want with a `set` command.

```
FEL_mesh_ptr mesh; // defined elsewhere...

mesh->set(FEL_INTERPOLATION, FEL_NEAREST_NEIGHBOR_INTERPOLATION);
mesh->set(FEL_INTERPOLATION, FEL_ISOPARAMETRIC_INTERPOLATION);
mesh->set(FEL_INTERPOLATION, FEL_PHYSICAL_SPACE_INTERPOLATION);
```

These `set` calls are really methods on `FEL_mesh`, but you can also call them on any field, and it will simply forward the call to its mesh. Note that however it is set, the interpolation mode on a mesh affects *all fields based on that mesh*. This rather unfortunate state of affairs will be amended in a future release of FEL so that fields will be less dependent entities. In the meantime, if you want different interpolation modes on fields sharing a common mesh, set the interpolation mode just prior to querying the field — but be aware that this may not be a robust strategy in a multithreaded application.

The interpolation modes work in concert with the simplicial decomposition mode, since the cell type which the interpolation routines receive may be altered by simplicial

decomposition. (Details on simplicial decomposition may be found in Chapter 12.) Thus the three interpolation modes listed above, combined with the three choices for simplicial decomposition —

```
// setting decomposition off, even and odd:
mesh->set(FEL_SIMPLICIAL_DECOMPOSITION, 0);
mesh->set(FEL_SIMPLICIAL_DECOMPOSITION, 1);
mesh->set(FEL_SIMPLICIAL_DECOMPOSITION, 2);
```

— form a 3×3 matrix of possibilities. If you don't explicitly set them, the interpolation mode defaults to `FEL_ISOPARAMETRIC_INTERPOLATION`, and "even" simplicial decomposition is turned on (`FEL_SIMPLICIAL_DECOMPOSITION, 1`). The different combinations of interpolation and simplicial decomposition may produce quite different numerical results on the same dataset, and they vary widely in the amount of numerical work involved. A few general considerations about choosing these modes are presented below (and see [KL95] for a comparison of isoparametric and physical space interpolation methods on tetrahedra), but for the most part, the choice is context dependent.

The combination of interpolation and decomposition modes affects the differential operator fields, since they obtain their required spatial derivatives by analytically differentiating the interpolating polynomials. This is the case even when the differential operator fields are queried at a vertex. For more information on the differential operator fields, see Chapter 20.

9.2 Nearest neighbor interpolation

`FEL_NEAREST_NEIGHBOR_INTERPOLATION` really doesn't do interpolation, rather it assigns to an interior point the field value of the nearest vertex of the enclosing cell. This "interpolation" method is the fastest of the bunch, but obviously not very smooth.

9.3 Isoparametric interpolation

`FEL_ISOPARAMETRIC_INTERPOLATION` transforms the enclosing cell and the query point into computational space and then interpolates a field value at the query location using linear basis functions. If the cell is a tetrahedron, the physical to computational space transformation of the query location is done analytically. For hexahedra, however, the physical to computational space transformation of the query location must be done numerically, using Newton's method, and this requires evaluating and inverting the Jacobian matrix for each iteration.

In computational space, the coordinates of the query point are ξ, η, ζ, and the field value $f = f(\xi, \eta, \zeta)$. Field values at cell vertices are referred to as f_n, where n ranges from 0 to 1 less than the number of nodes in the cell.

For tetrahedra, we use the linear basis function:

$$f(\xi, \eta, \zeta) = (1.0 - \xi - \eta - \zeta) f_0 + \xi f_1 + \eta f_2 + \zeta f_3$$

and for hexahedra:

$$\begin{aligned} f(\xi,\eta,\zeta) &= (1-\xi)(1-\eta)(1-\zeta)f_0 \\ &+ \xi(1-\eta)(1-\zeta)f_1 \\ &+ (1-\xi)\eta(1-\zeta)f_2 \\ &+ \xi\eta(1-\zeta)f_3 \\ &+ (1-\xi)(1-\eta)\zeta f_4 \\ &+ \xi(1-\eta)\zeta f_5 \\ &+ (1-\xi)\eta\zeta f_6 \\ &+ \xi\eta\zeta f_7 \end{aligned}$$

9.4 Physical space interpolation

`FEL_PHYSICAL_SPACE_INTERPOLATION` solves for the coefficients of an interpolating polynomial in physical space, using the locations and field values of the enclosing cell vertices as constraints. The calculation does not involve any physical to computational space transformations, but it does require inversion of a 4×4 (tetrahedron) or 8×8 (hexahedron) matrix.

In physical space, the coordinates of the query point are x, y, z, and the field value $f = f(x, y, z)$. Coordinates of cell vertices are referred to as x_n, y_n, z_n, and field values at the corresponding vertices are referred to as f_n, where n ranges from 0 to 1 less than the number of nodes in the cell. The coefficients of the interpolating polynomial are referred to as c_n.
For tetrahedra, we use the interpolating polynomial:

$$f(x,y,z) = c_0 + c_1 x + c_2 y + c_3 z$$

and determine the coefficients by:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & x_0 & y_0 & z_0 \\ 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \end{pmatrix}^{-1} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

For hexahedra, we use the interpolating polynomial:

$$f(x,y,z) = c_0 + c_1 x + c_2 y + c_3 z + c_4 xy + c_5 xz + c_6 yz + c_7 xyz$$

and determine the coefficients by:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} = \begin{pmatrix} 1 & x_0 & y_0 & z_0 & x_0y_0 & x_0z_0 & y_0z_0 & x_0y_0z_0 \\ 1 & x_1 & y_1 & z_1 & x_1y_1 & x_1z_1 & y_1z_1 & x_1y_1z_1 \\ 1 & x_2 & y_2 & z_2 & x_2y_2 & x_2z_2 & y_2z_2 & x_2y_2z_2 \\ 1 & x_3 & y_3 & z_3 & x_3y_3 & x_3z_3 & y_3z_3 & x_3y_3z_3 \\ 1 & x_4 & y_4 & z_4 & x_4y_4 & x_4z_4 & y_4z_4 & x_4y_4z_4 \\ 1 & x_5 & y_5 & z_5 & x_5y_5 & x_5z_5 & y_5z_5 & x_5y_5z_5 \\ 1 & x_6 & y_6 & z_6 & x_6y_6 & x_6z_6 & y_6z_6 & x_6y_6z_6 \\ 1 & x_7 & y_7 & z_7 & x_7y_7 & x_7z_7 & y_7z_7 & x_7y_7z_7 \end{pmatrix}^{-1} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{pmatrix}$$

The 4×4 and 8×8 matrices are inverted numerically, using Gauss-Jordan elimination with partial pivoting. We then check the "standard" matrix condition number using the infinity norm ($\|A\|_\infty \|A^{-1}\|_\infty$), and if this exceeds $1e^6$ we perform a singular value decomposition, zero the singular values $< 1e^{-3}$ and generate a pseudoinverse. We found the SVD to be necessary, especially in the 8×8 case, as many common data sets yield ill-conditioned matrices surprisingly often. The SVD routines will be made more flexible in a future release of FEL, with user-settable options replacing the hardcoded invocation and parameters. See, for example, [PTVF92] to learn more about SVD and related numerical issues.

Although the matrix inversions involved in the physical space interpolation routines are expensive, especially if the SVD routines are invoked, the inverted matrices are determined purely by the geometry of the cell. They can be reused for successive local interpolations on the same or a different field if the successive queries all fall in the same cell. The inverted matrix is cached in the `FEL_cell_interpolant` which can be resubmitted as part of an `at_phys_pos()` query. See Chapter 17 for details on the several variants of `at_phys_pos()`.

Chapter 10

Meshes and Fields

The heart of FEL is composed of mesh and field objects; the majority of the following chapters of this document will be on topics related to these two key types. Most of the common interface for meshes is defined by the class `FEL_mesh` (Chapter 11). Most of the interface for fields is defined by the classes `FEL_field` and `FEL_typed_field` (Chapter 17).

One goal in the design of FEL is that applications written in terms of the standard interfaces should work with a variety of mesh and field types. For example, a visualization technique written for scalar fields should work just as easily with a field where values are computed on demand, such as a derived pressure field (Chapter 19), as with a field where the data are precomputed and stored in memory (Chapter 18). The same code should also work with many other types of fields, such as those where data are paged in from disk on demand (Chapter 23) or fields that vary with time (Chapter 25).

While it is not hard to see the virtues of code reuse, it is not as easy in practice to design interfaces that make such reuse straightforward for the user. Even with the best of interfaces, it still may take some effort on the user's part to think in more general terms. Not every mesh is structured, not every field is steady. In general, the development of truly mesh and field type-independent algorithms requires effort on the part of both the FEL design team and the user. Making mesh and field independent interfaces and algorithms a reality continues to be a learning process for us all.

10.1 Member function style

The member functions of the mesh and field classes follow a general style where input arguments come first, followed by arguments pointing to the location where results should be written. The input arguments typically are C++ `const` references, so that their intended use should be clearer to the user, and so the compiler may have more opportunities to optimize. Most member functions return an integer indicating whether the call was successful. We look at the return values in a bit more detail next.

10.2 Return values

Most mesh and field member functions return an integer status value. The value can indicate success (`FEL_OK`) or it can indicate one of a variety of errors. The complete list of return values can be found in `FEL_returns.h`. One should use the names defined in `FEL_returns.h` rather than integer values explicitly, so that in the unlikely case that return codes get renumbered, one's code will still work. One return value where even the best of FEL programmers have lapsed into using the integer value directly is `FEL_OK`. `FEL_OK` is equal to 1. The interchangeability of 1 and `FEL_OK` is so ingrained in the FEL programming style that it is safe to assume that the number and symbol will be bound together for all time.

We conclude this chapter by strongly encouraging the users of FEL to check return values. There are two main reasons why one should get into this habit. First, if one is serious about developing algorithms that are mesh and field type-independent, then it is hazardous to assume when using some member function that "this call cannot fail". Even though there are already a large number of mesh and field types, it is easy to fall into the trap of thinking in terms of just one particular type. Even if one does consider all the ways that a call can fail, based on the types available in FEL today, one is still not completely safe. In the future there will surely be more mesh and field types introduced into the library. Algorithms written with careful error checking now should at least be able to gracefully indicate that they cannot work with a new type in the future.

A second reason to be conscientious about return value checking is that not doing the checking can lead to potentially insidious bugs. Since most member functions work by writing their results into a location passed into the call, one always has something in result location, whether or not the call succeeded. In some cases it may be obvious that the result location contains junk, but at other times the contents may seem plausible. For example, the result may contain a value from a previous, successful call. Checking return values is the only reliable way to guard against this type of problem.

Chapter 11

Meshes

Meshes are one of the two key types of objects in FEL, the other being fields. Meshes represent discretization of a domain into a set of cells. The cell types are drawn from the types represented by the class `FEL_cell`: vertices, edges, triangles, quadrilaterals, tetrahedra, and hexahedra. In an FEL mesh, it is assumed that a mesh containing a cell c also includes all the faces of c; so, for example, a mesh with a hexahedron c would also have all the quadrilateral, edge, and vertex faces of c. Meshes contain both geometric and topological information. Geometric information includes the coordinates of vertices and the volume of cells. Topological information includes neighbor relationships among the cells and other data about how the cells are organized. For example, a mesh containing a tetrahedron c can return the triangle faces or vertices of c.

In FEL, meshes are essential to the construction of fields, since every field has a mesh. For fields, meshes specify the location of nodes. *Nodes* are the points in the domain where solution values are generated by the solver or acquired by experiment. FEL supports vertex-centered fields, in other words fields where a node is associated with each vertex. There are other organizations for nodes; for instance, a hexahedral mesh may be "cell-centered", i.e., a node is associated with the interior of each hexahedron, but such configurations are currently unsupported by FEL.

One key responsibility of meshes is point location. Given a point p and a mesh containing some type of 3-cells, such as tetrahedra or hexahedra, the result of point location will be an integer return code and, if the location effort is successful, a cell containing p. The concept of point location in FEL has been generalized to meshes which do not contain 3-cells. For instance, FEL can represent surfaces in $\mathbf{R}^3$ consisting of triangles or quadrilaterals (2-cells). Point location with a surface mesh returns a 2-cell from the mesh. See Chapter 12 for details of how point location is defined for surfaces. Efficient point location is one of the keys in FEL to providing good performance overall.

A second key responsibility of meshes is to assist with interpolation. Given the cell c resulting from point location, a mesh can construct an "interpolant" for use later in interpolation. The interpolant contains information based on the geometry of c. The specific information contained in an interpolant depends upon the type of interpolation that the user has selected. For example, if c is a tetrahedron, and the prevailing inter-

polation mode is isoparametric, then the interpolant would contain the basis functions required to linearly interpolate within c. See Chapter 9 for more detail on interpolants.

11.1 The mesh class hierarchy

Figure 11.1 illustrates the class hierarchy for FEL meshes. `FEL_mesh` inherits from the class `FEL_mutex_reference_counted_object`, therefore meshes are reference counted in FEL, and there is critical section protection for the reference counting so that they can be used in multi-threaded applications. `FEL_mesh` specifies the interface that all mesh classes inherit. The mesh class also provides implementation of routines that are used by many of the mesh subclasses. Key subclasses of `FEL_mesh` are described in the following chapters.

11.2 Setting and getting mesh properties

FEL meshes support a general "`set`" interface for setting mesh properties:

```
void set(const FEL_set_keyword_enum k, int v);
```

The `set` call takes a keyword `k` and a value `v`; the complete list of keywords can be found in the file `FEL_set_keywords.h`. There is also a general "`get`" member function:

```
int get(const FEL_get_keyword_enum k, int* nv, int v[],
        int zone = FEL_ZONE_UNDEFINED) const;
```

The `get` call takes a keyword `k` and fills in `*nv` integer values into the array `v`. Note that some properties, such as the dimensions of a structured mesh, require more than one integer to describe. In most cases the user will know how many integers are needed to describe a particular property, i.e., how many values will be written into `v`. In those cases the user can pass the value `NULL` for the `nv` argument. The `get` function takes an optional final argument specifying a zone. Using the `zone` argument with `get` allows one to make queries of a particular zone in a multi-zone mesh. The `get` call returns 1 for success or an error value otherwise.

11.3 Simplicial decomposition

Some algorithms work in terms of the cells of a mesh, but require that the cells be simplices, i.e., that the cells be vertices, edges, triangles, or tetrahedra. FEL supports emulating the decomposition of a mesh into simplices through what is known as "simplicial decomposition". We say "emulating" because internally FEL does not change the representation of the mesh when simplicial decomposition is requested; it only changes the type of cells returned by methods such as point location. Currently only structured meshes contain non-simplicial cells (quadrilaterals and hexahedra), thus the following discussion only applies to structured mesh types. In the future other mesh types may

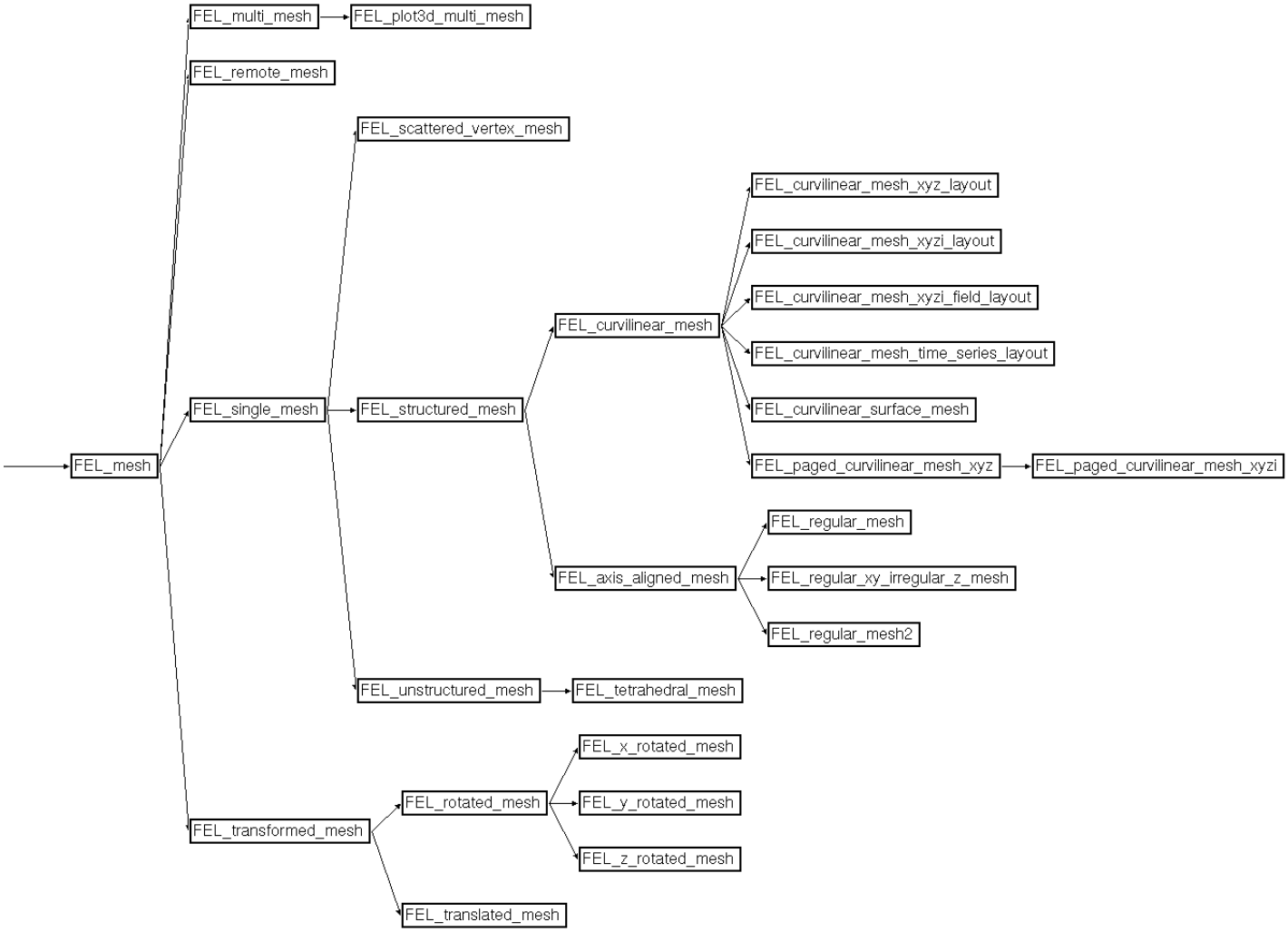


Figure 11.1: The FEL mesh class hierarchy.

also contain non-simplicial cells, but the interface for controlling the decomposition will be the same.

In general there is more than one way to subdivide a non-simplicial cell into simplices, even if one is not allowed to introduce new vertices as part of the subdivision. A quadrilateral can be broken into triangles in one of 2 ways, depending upon how the diagonal is chosen. There are 2 ways to decompose a hexahedron into 5 tetrahedra and many more decompositions consisting of 6 tetrahedra. A typical requirement is that the cells be decomposed consistently; in other words, if there is a non-simplicial face shared by two cells, then the decomposition chosen for each cell must result in the same decomposition for the shared face. The issue of consistent decompositions in FEL arises with structured hexahedral meshes, since adjacent hexahedra share quadrilateral faces. FEL supports 2 consistent simplicial decompositions for structured meshes, to be described in the next chapter. FEL also provides the call [1]:

```
int decomposition_cells(const FEL_cell c&,
                        int* nsc, FEL_cell sc[]) const;
```

which takes a non-simplicial cell `c` as an argument and returns `nsc` simplicial cells `sc` that `c` would be subdivided into, using the prevailing simplicial decomposition setting. (The user can query about the prevailing decomposition mode using the `get` member function described above.) The result of `decomposition_cells` is undefined if simplicial decomposition is not on when the call is made.

To set a particular value for simplicial decomposition, one can use the call `set(FEL_SIMPLICIAL_DECOMPOSITION, i)` on a mesh, where `i` would have the value 0, 1, or 2. The value 0 signifies that simplicial decomposition should be turned off. Values 1 and 2 each set one of 2 alternate decompositions for structured meshes.

11.4 Point location and interpolation

FEL meshes all inherit the following interface for point location:

```
int locate_close_vertex_cell(const FEL_phys_pos& p,
                             FEL_vertex_cell* v) const;
int locate(const FEL_phys_pos& p, FEL_cell* c) const;
int locate(const FEL_phys_pos& p, FEL_cell& start_cell,
           FEL_cell* c) const;
```

The `locate_close_vertex_cell` returns a vertex `v` that is close to `p`. The vertex is not guaranteed to be the closest, but sometimes close is good enough. The `locate` member functions are the workhorse routines for point location. Given a point `p`, `locate` produces a cell `c`. For most meshes, `c` is a hexahedron or tetrahedron containing

[1] For those less familiar with C++ notation, the `const` keyword may be new. When the keyword is part of an argument declaration, then `const` indicates that the function will not modify the argument. When `const` follows the closing parentheses of a class member function declaration, then `const` specifies that calling the function will not change the state of the object.

p. For other types of meshes, such as surface meshes, c may be a quadrilateral cell or triangle cell close to p.

The second locate routine takes an extra start_cell argument. For several key mesh types, such as FEL_curvilinear_mesh and FEL_unstructured_mesh, a point is located by "walking" from cell to cell, until a result cell is found. If a start_cell argument is provided, then it is used to initialize the walking point location routine. Providing a start cell can significantly improve the point location performance if the cell is close to the given point.

An FEL_interpolant contains information based on the geometry of a cell used for interpolation. An interpolant is specific to a particular cell, and an FEL_cell_interpolant pairs a cell together with its interpolant. Since meshes contain cell geometric data, meshes are responsible for initializing interpolants. Initialization is done through the member functions:

```
int set_interpolant(FEL_cell_interpolant*) const;
int set_interpolant(const FEL_cell_interpolant& pci,
                    FEL_cell_interpolant* ci) const;
int locate_and_set_interpolant(const FEL_phys_pos&,
                               FEL_cell_interpolant*) const;
int locate_and_set_interpolant(const FEL_phys_pos&,
                               FEL_cell_interpolant&,
                               FEL_cell_interpolant*) const;
```

The second set_interpolant call provides the opportunity to reuse the interpolant loaded by a previous set_interpolant call, if the cell in ci is the same as in pci, and the interpolation mode (e.g., isoparametric) is the same. The latter two calls combine point location and setting an interpolant into one call.

11.5 Coordinates

The user can access the coordinates v of the vertices of a cell c via the calls:

```
int coordinates_at_cell(const FEL_cell& c,
                        FEL_vector3f v[]) const;
int coordinates_at_vertex_cell(const FEL_vertex_cell& c,
                               FEL_vector3f* v) const;
```

One can also convert between a structured mesh position s and physical coordinates via the call:

```
int coordinates_at_structured_pos(const FEL_structured_pos& s,
                                  FEL_vector3f* v) const;
```

Since the structured position includes a zone number, the "at structured pos" call can also be made on a multi-zone mesh, as long as the specified zone is structured. If the call is made on a mesh that does not have structured behavior, then the return value of the call will not be equal to 1.

11.6 Cell geometric properties

The member functions which answer queries regarding the geometric properties of cells are relatively self-explanatory:

```
bool cell_has_collapsed_edge(const FEL_cell&) const;
int volume_of_cell(const FEL_cell&, double*) const;
int centroid_of_cell(const FEL_cell&,
                     FEL_vector3f*) const;
int longest_edge_length_of_cell(const FEL_cell&,
                                float*) const;
int closest_vertex_of_cell(const FEL_phys_pos&,
                           const FEL_cell&,
                           FEL_vertex_cell*) const;
```

The `cell_has_collapsed_edge` call tests whether two vertices on the same edge have coordinates that are exactly equal; in other words there is no epsilon used in the floating-point coordinate equality test to allow for nearly-equal values.

11.7 Cell incidence relationships

Given a cell c, an application may require cells incident to c. Two distinct cells c and d are *incident* if c is the face of d or vice versa. For example, for a given triangle c in a mesh, one may need the vertices of c, or perhaps the tetrahedra for which c is a face. The incidence relationships among cells can be visualized with a graph. Figure 11.2 illustrates the incidence relationships for a small mesh in $\mathbf{R}^2$. The graph to the right contains a node for each cell in the mesh to the left. The nodes are organized into rows, each row containing cells of a particular dimension. The rows are ordered by ascending dimensionality: higher rows signify higher-dimension cells. A mesh containing 3-cells would have one extra row at the top. The example queries from above can be seen as starting at a particular node and following paths downward or upward. For example, to get the vertices of a triangle c, one could start at the node representing c and follow all the paths downwards. Likewise, in a mesh containing tetrahedra, one could start at a node representing a triangle c and follow the 0, 1, or 2 paths upward, depending on the number of tetrahedra that have c as a face. FEL meshes support queries based on cell incidence relationships via the calls `up_cells` and `down_cells`:

```
int up_cells(const FEL_cell& c, int d, int max,
             int* n, FEL_cell rc[], int = -1) const;
int down_cells(const FEL_cell& c, int d, int* n,
               FEL_cell rc[]) const;
```

Both calls take a cell `c` as the first argument, and the dimension `d` of the cells one wants in return as the second argument. Both calls write cells in the array `rc`. The number of cells produced is written into `n`. The `up_cells` call also takes an argument `max`, which specifies the maximum number of cells that the user wants back. The terms "`up`" and "`down`" can be thought of either as going up or down in dimension,

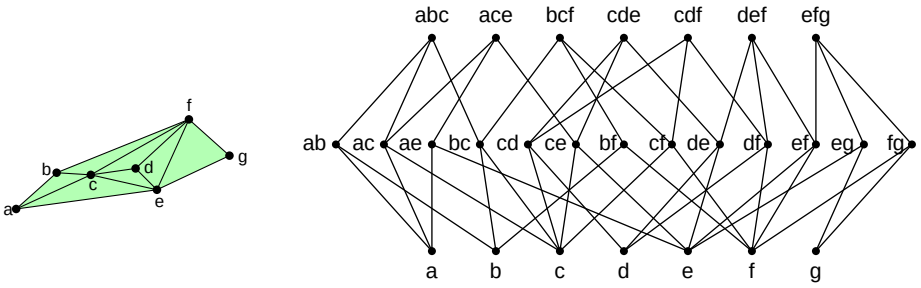


Figure 11.2: A small example mesh and its incidence graph.

or as going upward or downward in the incidence graph. In topology, the up and down operations are known as *star* and *closure*, respectively. Note that the concepts of incidence relationships and up and down calls are not specific to a particular type of mesh; thus algorithms written in terms of `up_cells` and `down_cells` have the potential of working with many types of meshes.

11.8 Adjacent cells

A concept related to the incidence relationships between cells is adjacency, also known as a neighbor relationship. The mesh member function `adjacent_cells` supports queries requesting the cells neighboring to a given cell. The function signature for `adjacent_cells` is:

```
int adjacent_cells(const FEL_cell&, int*, FEL_cell []);
```

where `c` is the cell for which to produce adjacent cells for, and `nac` and `ac` get the number of adjacent cells and the cells themselves, respectively. The most frequent usage of the `adjacent_cells` call is with a 3-cell argument. For example, given a hexahedron c from a hexahedral mesh, `adjacent_cells` would return the hexahedra which share a quadrilateral face with c. Likewise, given a tetrahedron c from a mesh (either a tetrahedral mesh or a hexahedral mesh with simplicial decomposition turned on), `adjacent_cells` will return the tetrahedra which share a triangle face with c. The `adjacent_cells` call is handy for algorithms that work breadth-first, starting from a seed cell. For example, one could construct an isosurface incrementally, processing cells outward from an initial 3-cell.

The concept of adjacency has a more formal definition. First, returning to the graph in Figure 11.2, imagine that each vertex is the face of a special (-1)-cell, i.e. that there is an extra row beneath the vertex (0-cell) row with one node, and arcs from each vertex to the (-1)-cell node. Furthermore, if the cells in the top row are k-cells, then imagine an extra row above the k-cells with a single $(k+1)$-cell that every k-cell is the face of. Given this augmented incidence relationship graph, one can define the

adjacent cells of a cell c via the up and down operations described above. Let S_{ud} be the set of cells produced by going up one dimension and then down one dimension, starting with c. Let S_{du} be the set of cells produced by going down one dimension and then up one dimension, starting with c. The cells *adjacent* to a cell c are the cells in $(S_{ud} \cap S_{du}) - c$. (There are more efficient ways to implement `adjacent_cells`, the preceeding definition is useful for its generality).

11.9 Cardinality

Since meshes are finite collections of cells, one basic query that the user might make is a count of the cells represented by a mesh. FEL provides this functionality via the method `card`:

```
int card(int k) const;
```

Given an integer argument `k`, `card` returns a count of the k-cells in a mesh. For instance, `card(0)` returns the number of vertices in a mesh. The value returned by `card` depends on whether simplicial decomposition is turned on or off. For example, if `card(2)` returns n when called on a structured hexahedral mesh with decomposition off, then `card(2)` will return $2n$ when decomposition is turned on.

11.10 Cells and canonical enumeration

For the k-cells in a mesh, one can imagine assigning a numbering so that each k-cell has a unique integer identifier. Such an enumeration could be handy, for example, for representing sets of k-cells, since each integer identity number would consume less memory than an `FEL_cell` object. FEL supports a canonical enumeration of k-cells, but not for every value of k and for every mesh type. The following chapters on structured and unstructured meshes list which enumerations are currently supported. Every enumeration follows the convention of going from 0 to `card(k)` - 1 for k-cells. To convert between the integer representation and the `FEL_cell` representation, and vice versa, FEL provides the methods:

```
int int_to_cell(int i, int k, FEL_cell* c,
                int s = -1) const;
int cell_to_int(const FEL_cell& c, int* i) const;
```

The conversion from integer to cell is influenced by the simplicial decomposition mode currently set for the mesh. One can override the prevailing decomposition mode for the duration of the `int_to_cell` call by providing an optional final argument specifying a decomposition mode. For `cell_to_int`, the decomposition mode is inferred from the incoming cell type.

11.11 PLOT3D IBLANK

The classes in the FEL mesh hierarchy, and the interface specified at the top hierarchy, are intended, as much as possible, to be independent of any particular mesh or file format standard. One case where FEL favors a particular standard is in its support of IBLANK. IBLANK is a standard defined by PLOT3D [WBPE92] where an integer "IBLANK" value is associated with each vertex in a mesh. Since meshes in FEL are currently all vertex-centered, having an integer at each vertex is equivalent to having an integer associated with each node in a field. The IBLANK value can serve one of three purposes. The first is as a flag indicating that the node associated with a vertex is invalid (indicated by an IBLANK of 0). The second IBLANK use is as a hint about overlapping zones in a multi-zone mesh. To indicate that a vertex may be within an overlapping zone z, PLOT3D specifies that the IBLANK value should be $-z$. (PLOT3D follows the FORTRAN style of numbering where the zones go from 1 to n rather than 0 to $n-1$). The third usage of IBLANK values are to flag vertices that are on an impenetrable surface, signified by the value 2. An IBLANK of 1 is the default, signifying that the node is OK and that there is no overlapping zone information. FEL meshes provide access to IBLANK values via the calls:

```
int iblank_at_cell(const FEL_cell& c, int i[]) const;
int iblank_at_vertex_cell(const FEL_vertex_cell& c,
                          int* i) const;
int combined_iblank_at_cell(const FEL_cell& c, int* ci) const;
int coordinates_and_iblank_at_cell(const FEL_cell& c,
                    FEL_vector3f_and_int[] ci) const;
int coordinates_and_iblank_at_vertex_cell(const FEL_vertex_cell& c,
                                    FEL_vector3f_and_int* ci) const;
```

For a cell `c`, `iblank_at_cell` and `iblank_at_vertex_cell` produce IBLANK values, one for each vertex in `c`. The call `combined_iblank_at_cell` produces a single integer `ci` that is a bitwise combination of the following 4 flags:

- FEL_PLOT3D_HAS_IBLANK_1
- FEL_PLOT3D_HAS_IBLANK_2
- FEL_PLOT3D_HAS_IBLANK_0
- FEL_PLOT3D_HAS_IBLANK_LT_0

The combined IBLANK call is handy for quickly determining whether a more thorough analysis of the IBLANK values for a cell is necessary. The flags are designed so that a bitwise combination of them will result in an integer value between 1 and 15. FEL meshes also provide the "coordinates and iblank" calls, where one can request both types of data simultaneously.

One can configure a mesh `m` so that a combined IBLANK value is returned by point location routines, using the call:

```
m->set(FEL_RETURN_IBLANK, 1);
```

If `locate` finds a cell containing the given point, then the combined IBLANK for the cell is returned. Note that the combined IBLANK is an integer between 1 and 15. The `locate` function can still return other values, for example to indicate that the given point is outside the mesh. See `FEL_returns.h` for a complete list of the return values.

Chapter 12

Structured Meshes

In FEL, structured meshes consist of hexahedral cells (and all their faces), with a regular organization such that the vertices of the mesh can be indexed by 3 indices, just as one would index a 3-dimensional array. The indices are usually written i, j, and k. Topological information, such as incidence relationships among cells, is represented implicitly. Geometric information, such as the coordinates of vertices, can be represented implicitly or explicitly, depending on the particular subclass of structured mesh. Subclasses of `FEL_structured_mesh`, in particular the curvilinear mesh subclasses, are some of the most heavily used classes in FEL.

12.1 Simplicial decomposition

When working with structured meshes, the user has the choice of 3 simplicial decomposition modes. Mode 0 corresponds to no decomposition. Modes 1 and 2 specify decompositions where each hexahedron is broken into 5 tetrahedra. There are two 5-tetrahedra decompositions possible for a hexahedron. In order for the decompositions to be consistent between each pair of adjacent hexahedra, the decomposition for each hexahedron must be the opposite of its adjacent neighbors. Thus, given the decomposition choice for one hexahedron in a structured mesh, the choices for all the remaining hexahedra are forced. FEL organizes the 2 decompositions in terms of "odd" and "even" vertices, where the odd and even designations come from the vertex i, j, and k indices. A vertex is even if the sum $(i + j + k)$ is even, otherwise it is odd. In decomposition mode 1, the diagonals added to decompose the quadrilaterals go between even vertices. The decomposition choices for the 6 quadrilateral faces of a hexahedron leave only one possible tetrahedral decomposition. In decomposition mode 2, the diagonals go between odd vertices, and the hexahedral decomposition follows suit.

12.2 Cell incidence relationships

`FEL_structured_mesh` implements the structured mesh support for the `up_cells` and `down_cells` member functions. Table 12.1 summarizes the combinations of

From \ To	0	1	2	3
0	•			u
1	d	•		
2	d	d	•	u
3	d	d	d	•

Table 12.1: The supported combinations for `up_cells` and `down_cells`. Each box in the grid represents a combination of From cell dimension and To cell dimension. Combinations marked with d are supported by `down_cells()`, combinations marked with u are supported by `up_cells()`. The boxes marked • are trivially supported by either `down_cells()` or `up_cells()`. Empty boxes indicate unsupported combinations.

"from" and "to" cell dimensions supported. From the table, for example, one can see that since there is a d in row 3, column 0, one can use the method `down_cells` to get from a hexahedron (3-cell) to its vertices (0-cells). The same support is available when simplicial decomposition is turned on; thus, given a subtetrahedron, one can get its vertices via `down_cells`. In general, the same "from" and "to" pairs are supported, regardless of the simplicial decomposition mode.

The FEL structured mesh class also implements the adjacent cells method for structured meshes. Currently `adjacent_cells` is implemented for hexahedra and subtetrahedra only.

12.3 Canonical cell enumeration

`FEL_structured_mesh` currently supports a canonical enumeration of vertices, hexahedra, and subtetrahedra. The enumeration for each type of cell corresponds to the ordering in which an iterator would produce the cells if iterating over the whole mesh. In terms of the cell i, j, and k indices, the i index would vary the fastest, k the slowest. If simplicial decomposition is turned on, then all the tetrahedra resulting from decomposing a particular hexahedron are numbered consecutively.

12.4 Computational space support

```
int coordinates_at_structured_pos(const FEL_structured_pos& s,
                                  FEL_vector3f* c) const;
int jacobian_at_vertex_cell(const FEL_vertex_cell& v,
                            FEL_matrix33f* m) const;
int contravariant_phys_to_comp_vector(const FEL_vertex_cell& vc,
                                      const FEL_vector3f& pv,
                                      FEL_vector3f* cv) const;
int contravariant_phys_to_comp_vector(const FEL_structured_pos& s,
                                      const FEL_vector3f& pv,
```

```
                                        FEL_vector3f* cv) const;
```

12.5 Structured mesh dimensions

FEL provides access to the dimensions of a structured mesh via the member function `get_structured_dimensions()`:

```
int get_structured_dimensions(int d[]) const;
```

The function is defined as part of the interface of `FEL_mesh`, thus one does not have to cast an `FEL_mesh_ptr` down to an `FEL_structure_mesh_ptr` in order to make the call. The function returns 1 on success or 0 if the call is inappropriate, e.g., if the call is made on an unstructured mesh.

The `get_structured_dimensions()` call can also be useful for distinguishing objects that have structured mesh behavior from those that do not. We say "has structured mesh behavior" rather than "is a structured mesh" because not all meshes with structured mesh behavior are subclasses of `FEL_structured_mesh`. In particular, a transformed mesh (Chapter 14) built in terms of a structured mesh has structured behavior, e.g., one can request the structured dimensions, yet it is not a structured mesh. Thus the call `get_structured_dimensions()` is preferable to `is_structured_mesh()`, since the fact a mesh is transformed should be transparent to a routine that requires an object with structured mesh behavior.

12.6 Axis-aligned structured meshes

Axis-aligned structured meshes are meshes where the cells are aligned with the coordinate axes in physical space. Axis-aligned meshes in FEL may have either regular or irregular axes. A *regular axis* is an axis where the spacing between neighboring vertices is constant. An *irregular axis* is an axis where the spacing is not obliged to be the same throughout. An axis-aligned structured mesh where all the axes are regular is also known as a *regular mesh*. Axis-aligned meshes can be thought of as meshes defined by the Cartesian product of regular or irregular axes aligned with the physical space axes.

Axis-aligned meshes have the advantage of requiring far less memory than curvilinear meshes (described later in this chapter), since coordinates can be represented implicitly. (PLOT3D IBLANK values are assumed to be 1 for every vertex.) Axis-aligned meshes also have the advantage of more efficient point location, since the regularity of the geometry admits significant optimizations over the more general curvilinear mesh case.

FEL provides classes representing axis-aligned meshes with all regular axes, meshes with regular x and y axes but an irregular z axis, and meshes where the z axis has dimension 1. The class constructors look like:

```
FEL_regular_mesh(int d0, int d1, int d2,
                 char* nm = "regular_mesh");
```

```
FEL_regular_mesh(int d0, int d1, int d2,
                 float s0, float s1, float s2,
                 char* nm = "regular_mesh");
FEL_regular_mesh(int d0, int d1, int d2,
                 float s0, float s1, float s2,
                 float o0, float o1, float o2,
                 char* nm = "regular_mesh");

FEL_regular_xy_irregular_z_mesh(int d0, int d1, int d2,
                                float o0, float s0,
                                float o1, float s1,
                                float* coordinates2,
                 char* nm = "regular_xy_irregular_z_mesh");

FEL_regular_mesh2(int d0, int d1,
                  float s0, float s1,
                  char* = "regular_mesh2");
```

The parameters `d0`, `d1` and `d2` specify the dimensions of the mesh in I, J, and K. The `nm` parameter gives the user the option of providing a character string name for the mesh. The second constructor for `FEL_regular_mesh` gives the user the opportunity to specify the spacing between adjacent vertices using the parameters `s0`, `s1`, and `s2`. The origin for regular axes can also be specified using the arguments `o0`, `o1`, and `o2` in the third constructor.

The class `FEL_regular_mesh2` is used for representing regularly gridded rectangles. The arguments follow the same pattern as for the hexahedral meshes. An `FEL_regular_mesh2` instance lies in the $z = 0$ plane. The third component of `FEL_phys_pos` and `FEL_structured_pos` arguments is ignored by `FEL_regular_mesh2`. Thus, the `locate` member function, given a point $p \in \mathbf{R}^3$, essentially projects p down to the $z = 0$ plane and then locates the quadrilateral to which p projects.

The default interpolation mode with `FEL_regular_mesh2` is of type nearest neighbor. Simplicial decomposition is not currently supported for this class.

12.7 Curvilinear meshes

Curvilinear meshes are the most general type of structured mesh in FEL. The class `FEL_curvilinear_mesh` is actually an abstract class. The classes derived from curvilinear mesh that the user can instantiate are:

- `FEL_curvilinear_mesh_xyz_layout`
- `FEL_curvilinear_mesh_xyzi_layout`
- `FEL_curvilinear_mesh_xyzi_field_layout`

There are also paged curvilinear mesh classes, described in Chapter 23. The "`layout`" suffixes distinguish how the data describing the coordinates and in some cases IBLANK are provided. The `xyz_layout` mesh works with an array of type `FEL_vector3f` that contains the coordinates of the vertices. The x, y, and z components of each vertex are contiguous in memory, as they are in each `FEL_vector3f`. In terms of the structured mesh I, J, and K coordinates, the vertices are ordered so that I varies the fastest and K the slowest. IBLANK values are assumed to be 1 everywhere. The `xyzi_layout` mesh works with an array of type `FEL_vector3f_and_int`. The layout is the same as for the `xyz` mesh, except that each vertex has an IBLANK value interleaved with the coordinates. The `xyzi_field_layout` is constructed with a field where the node type is `FEL_vector3f_and_int`. When a `FEL_curvilinear_mesh_xyzi_field_layout` is queried for coordinates or IBLANK data, it in turn queries the field it was constructed with. The `xyzi_field_layout` curvilinear mesh is typically used to represent unsteady meshes; see Chapter 26.

The constructors corresponding to the classes above are:

```
FEL_curvilinear_mesh_xyz_layout(
  int d0, int d1, int d2,
  FEL_vector3f* xyz,
  const char* nm = "curvilinear_mesh_xyz_layout");
FEL_curvilinear_mesh_xyzi_layout(
  int d0, int d1, int d2,
  FEL_vector3f_and_int* xyzi,
  const char* nm = "curvilinear_mesh_xyzi_layout");
FEL_curvilinear_mesh_xyzi_field_layout(
  int d0, int d1, int d2,
  FEL_vector3f_and_int_field_ptr xyzi_field,
  const char* nm = "curvilinear_mesh_xyzi_field_layout");
```

Each constructor takes the I, J, and K dimensions of the mesh as the first 3 arguments. The next argument is specific to the particular data layout: the argument is either a pointer to a buffer or a field pointer. Thus if the user has a buffer in an appropriate layout, then he or she can construct a curvilinear mesh directly.

In the future there may be other memory layouts supported by FEL via more subclasses of `FEL_curvilinear_mesh`. One layout that may be of particular interest is the case where the X, Y and Z coordinate values are in separate arrays, i.e., not interleaved together. Such a layout common in FORTRAN applications and in some file formats.

The convention in FEL for mesh and field constructors is that any buffer provided as a constructor argument becomes the reponsibility of the FEL object to deallocate. Since FEL will use the destructor `delete []` to do the deallocation, it is important that the user allocate the memory using the C++ allocation style for arrays, i.e., `new []`. For instance, if the buffer is an array of `FEL_vector3f`, then the allocation should look something like:

```
FEL_vector3f* xyz = new FEL_vector3f[n_vertices];
```

If the user has a buffer allocated in some other manner, or in general if the user does not want FEL doing the buffer deallocation, then for a mesh `m` one can use the statement:

```
m->set(FEL_SUPPRESS_DEALLOCATION, 1);
```

If the user specifies deallocation suppression, then he or she remains responsible for the management of the data buffers. See Chapter 7.

12.8 Curvilinear mesh point location

Curvilinear meshes inherit the interface of the two overloaded versions of `locate` declared in `FEL_mesh`. Both versions take an `FEL_phys_pos` as a first argument and a pointer to a cell where the result will be written as the last argument. The second version of `locate` takes an extra argument that is used as a start cell for walking point location. Given a start cell, `locate` for a curvilinear mesh will walk from cell to cell until a cell containing the given point is found. If `locate` with a start cell is unsuccessful, or if no start cell was provided, then curvilinear meshes uses four techniques to determine whether the mesh contains the given point. The techniques to locate a point p are:

(1) Test if p is in the mesh bounding box.

(2) Starting at the (computational coordinates) center of each of the 6 mesh boundary sides, use an adaptive vertex walk to get close to p. Starting from the walk destination closest to p, do a tetrahedral walk.

(3) Starting from any of the destinations in (2) that have not been tried, do a tetrahedral walk.

(4) For each 2-cell c on the boundary of the mesh, compute the centroid of c and the normal of c facing into the mesh. Choose the cell c from which p is visible and whose centroid is closest to p. Do a tetrahedral walk from there.

The techniques are ordered by computational cost; the fourth technique in particular is relatively expensive. The user can control how much effort a curvilinear mesh `m` puts into point location using the call:

```
m->set(FEL_LOCATE_EFFORT, level);
```

where the mesh may use the level i technique if $i \leq$ `level`. By default the level is 4, for multi-zone meshes the level for each zone is turned down to 3.

The curvilinear mesh locate routines return 1 to signify success. If the mesh finds a cell containing the given point, it automatically checks the IBLANK values of the vertices of the containing cell. If any of the IBLANK values are 0, then the return value for the locate call is 0.

12.9 Curvilinear surface meshes

Another subclass of `FEL_structured_mesh` is `FEL_curvilinear_surface_mesh`. The class represents surfaces in $\mathbf{R}^3$ composed of quadrilaterals. The class has several constructors:

```
FEL_curvilinear_surface_mesh(int d0, int d1,
                             FEL_vector3f* xyz,
                             const char* = name_default);
FEL_curvilinear_surface_mesh(int d0, int d1, int d2,
                             FEL_vector3f* xyz,
                             const char* = name_default);
FEL_curvilinear_surface_mesh(int d0, int d1,
                             FEL_vector3f_and_int* xyzi,
                             const char* = name_default);
FEL_curvilinear_surface_mesh(int d0, int d1, int d2,
                             FEL_vector3f_and_int* xyzi,
                             const char* = name_default);
```

The `d0`, `d1`, and `d2` arguments specify the structured mesh dimensions. If three "`d`" arguments are provided, then exactly one of the three must be equal to 1, otherwise the `d2` dimension is assumed to be 1. Following the "`d`" arguments is a pointer to a buffer with the coordinate data alone (`FEL_vector3f*`), or the coordinate data with IBLANK values (`FEL_vector3f_and_int*`). The user can query the mesh for coordinates and IBLANK values using the same "`at`" calls as for other types of meshes. The arguments should contain 0 in the "flat" dimension, i.e., the dimension given the value 1 in the constructor.

Point location for surface meshes is defined to mean locating the 2-cell (i.e., quadrilateral or triangle) whose centroid is closest to a given point location target p. If the distance from p to the closest cell centroid is shorter than the longest edge length of the cell, then the point location is considered successful. Currently it is not possible for the user to change the threshold for deciding whether a cell centroid is close enough.

FEL also supports iterating over a surface. The one difference when working with a cell iterator and a surface is that the highest-dimension cells in the mesh are now quadrilaterals (or triangles if simplicial decomposition is turned on), rather than hexahedra. Otherwise, the iterators are initialized and used just as with other structured meshes.

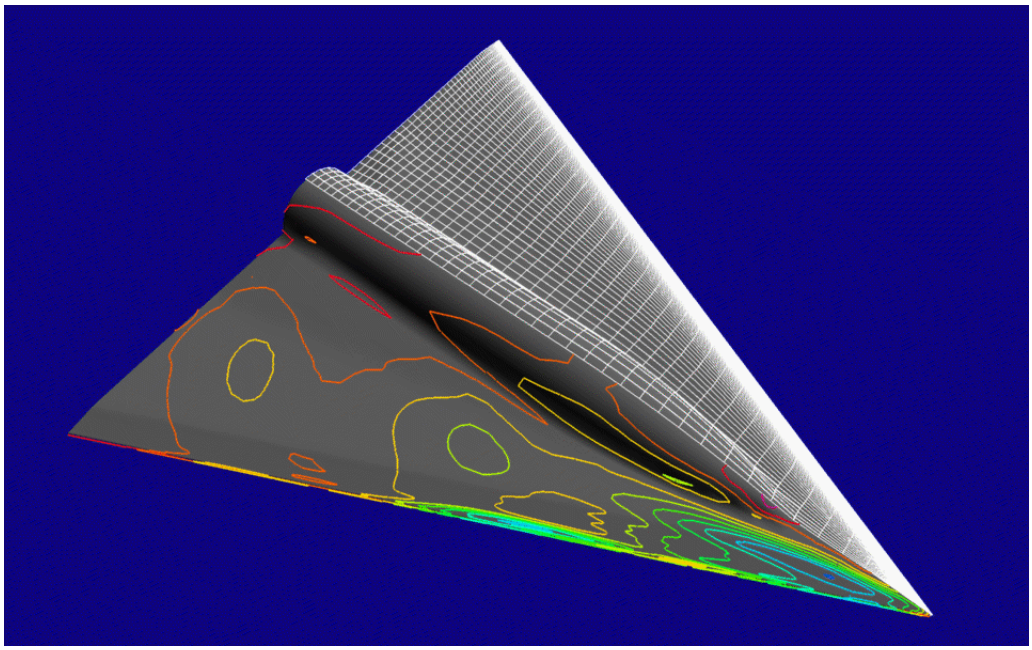


Figure 12.1: A CFD visualization of a delta wing in a single-zone, structured mesh. The left side of the wing displays edges from the mesh, the right side shows contour lines for a pressure derived field. See also Figures 1.1 and 13.1. (Data courtesy of Neal Chaderjian, visualization courtesy of Tim Sandstrom.)

Chapter 13

Unstructured Meshes

An unstructured mesh can be thought of as a collection of cells, without the regular organization of a structured mesh. Currently FEL supports two types of unstructured meshes. The first, `FEL_tetrahedral_mesh`, contains tetrahedra, triangles, edges, and vertices. As part of the tetrahedral mesh construction, the user can provide sets of triangles defining particular surfaces. Constructing a "tetrahedral" mesh with sets of triangles but no tetrahedra is also allowed. The second type of unstructured mesh in FEL is an `FEL_scattered_vertex_mesh`. Scattered vertex meshes consist solely of vertex cells.

13.1 Constructing a tetrahedral mesh

The constructor for an FEL tetrahedral mesh looks like:

```
FEL_tetrahedral_mesh(int n_vertices,
                     FEL_vector3f* coordinates,
                     int n_special_triangles,
                     FEL_vector3i* triangles,
                     int* triangle_ids,
                     int n_tetrahedra,
                     FEL_vector4i* tetrahedra,
                     const char* = "tetrahedral_mesh");
```

Here `n_vertices` specifies the number of vertices in the mesh, `coordinates` is an array containing the physical space coordinates of each vertex, `n_special_triangles` specifies the number of items in the `triangles` array and in the `triangle_ids` array. Each `FEL_vector3i` in the `triangles` array specifies the three vertex indices of a triangle. The arguments `n_tetrahedra` and `tetrahedra` specify the number of tetrahedra, and the vertices of each tetrahedron, respectively. The vertex numbering in the `triangles` and `tetrahedra` arrays is expected to be FORTRAN style, i.e., the numbers should refer to vertices as if they were numbered 1 to `n_vertices` rather than 0 to `n_vertices - 1`. The buffers

From \ To	0	1	2	3
0	•			u
1		•		
2	d		•	u
3	d			•

Table 13.1: The combinations supported by `FEL_tetrahedral_mesh` for `up_cells` and `down_cells` calls. Each box in the grid represents a combination of From cell dimension and To cell dimension. Combinations marked with `d` are supported by `down_cells()`, combinations marked with `u` are supported by `up_cells()`. The boxes marked • are trivially supported by either `down_cells()` or `up_cells()`. Empty boxes indicate unsupported combinations.

`coordinates`, `triangles`, `triangle_ids`, and `tetrahedra` passed in to the constructor become the responsibility of the tetrahedral mesh to deallocate when they are no longer needed.

13.2 Cell incidence relationships

Table 13.1 summarizes the currently implemented support for incidence relationship queries with tetrahedral meshes. From the table one can see, for example, that FEL tetrahedral meshes support queries requesting the vertices of a triangle (From 2 To 0), or the tetrahedra that a triangle is the face of (From 2 To 3). Queries for From/To combinations which are not currently supported return 0.

The `adjacent_cells` call is supported for tetrahedral arguments only.

13.3 Canonical cell enumeration

`FEL_tetrahedral_mesh` currently supports a canonical enumeration of vertices, triangles, and tetrahedra. No enumeration is currently supported for edges. The enumeration for each type of cell corresponds to the ordering in which an iterator would produce the cells when iterating over the whole mesh.

13.4 Surfaces

In structured hexahedral meshes, one can easily define a surface by holding one of the i, j, or k indices constant. Such surfaces are often used to represent entities such as the fuselage of an aircraft. Specifying a surface in a tetrahedral mesh is not as easy. To compensate for this, sets of triangles can be designated as representing a surface by the mesh generator, and a listing of the special triangles can then be included as part of the data set. For instance, the unstructured mesh file format defined by PLOT3D [WBPE92] supports the specification of triangle sets, each set with an integer

identity number. Note that the FEL_tetrahedral_mesh constructor starts with an array of triangles and an array of identity numbers, one for each triangle. As part of the construction, FEL_tetrahedral_mesh must determine the unique set of identity numbers and then rearrange the triangles into their corresponding sets. The user can query about the number of predefined triangle sets and get their identity numbers via the tetrahedral mesh calls:

```
int get_n_triangle_sets() const;
void get_triangle_set_ids(int ids[]) const;
```

Using one of the predefined set identity numbers, one can initialize an iterator and loop over the triangles or vertices of a given surface. See the chapter on iterators (Chapter 16) for more details.

13.5 Point location

As with structured meshes, point location for a point p in tetrahedral mesh works by "walking" from 3-cell to 3-cell, until a cell containing p is found. The tetrahedral mesh locate is overloaded, just as in the structured mesh case, so that the user can provide a start cell for the walking search. If the walking search with a given start cell is unsuccessful, or if no start cell is given, then a global search over the whole mesh is done.

13.6 Constructing a scattered vertex mesh

The constructor for a scattered vertex mesh looks like:

```
FEL_scattered_vertex_mesh(int n_vertices,
                          FEL_vector3f* coordinates,
                          const char* =
                            "scattered_vertex_mesh")
```

The array coordinates contains n_vertices of coordinates, i.e., coordinates for each vertex. The optional final argument gives the user the opportunity to give a specific name to the mesh.

As with any mesh subclass, the scattered vertex mesh supports the standard member functions inherited from FEL_mesh. Cell incidence relationships are trivial, since there are only 0-cells. The canonical enumeration of the vertices is the same ordering as provided to the mesh constructor. Point location is defined as locating the vertex closest to the given search point. Interpolation is simply nearest neighbor. The FEL_cell_iter and FEL_vertex_cell_iter both have the same behavior, i.e., both iterate over vertex cells.

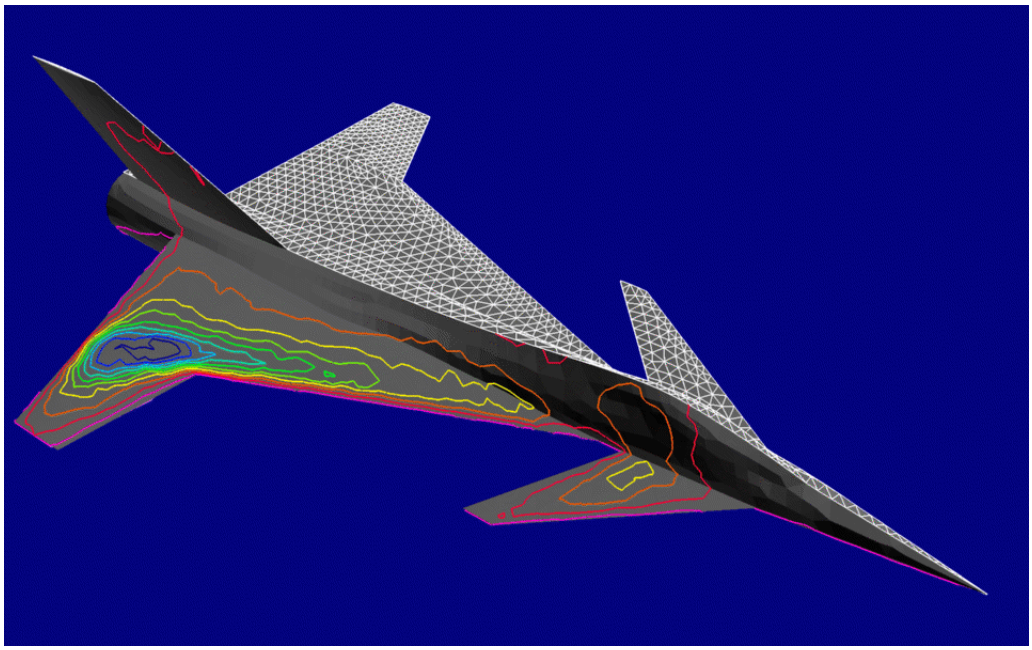


Figure 13.1: A CFD visualization of a fighter using an unstructured, tetrahedral mesh. The left side of the aircraft displays edges from the mesh, the right side shows contour lines for a pressure derived field. See also Figures 1.1 and 12.1. (Data courtesy of NASA Langley Research Center, visualization courtesy of Tim Sandstrom.)

Chapter 14

Transformed Meshes

For domains where there is some type of symmetry, computational scientists often take advantage of the symmetry to model a smaller, fundamental domain. The results from modeling such a domain can then be replicated and transformed to fill the original domain. For example, in some turbomachinery studies, only one sector of a radially periodic domain may be simulated. When visualizing the results from such simulations, the scientist may only need to visualize the results within the fundamental domain, or the scientist may wish to see the results replicated to look like the actual turbomachine. See, for example, Figure 14.1. In some cases, it may be sufficient to generate the graphics primitives for the fundamental domain and then draw the primitives repeatedly, applying a different transformation each time. In other cases, replicating graphics primitives may not be enough. For instance, when using particle tracing visualization techniques, a scientist may be interested in seeing the traces continue beyond the fundamental domain. In general, there are occasions when one would like to treat the simulation results as if they filled the whole original domain, without regard to any symmetry optimizations that may have been employed.

FEL supports the representation of meshes with periodic symmetries through the subclasses of `FEL_transformed_mesh`. Transformed meshes are constructed with an original mesh and the data describing a particular transformation. Transformed meshes do not replicate the mesh data; thus one still enjoys most of the memory savings due to not constructing a mesh for the whole original domain. At the same time, transformed meshes have the same interface as ordinary meshes and can be used just as non-transformed meshes, with no special treatment. The transformed mesh classes emulate rigid body transformations: translation and rotation. A transformed mesh can be constructed given any FEL mesh instance, including single-zone meshes, multi-zone meshes, and even other transformed meshes. See the mesh class hierarchy figure (Figure 11.1) for a refresher on the FEL mesh family.

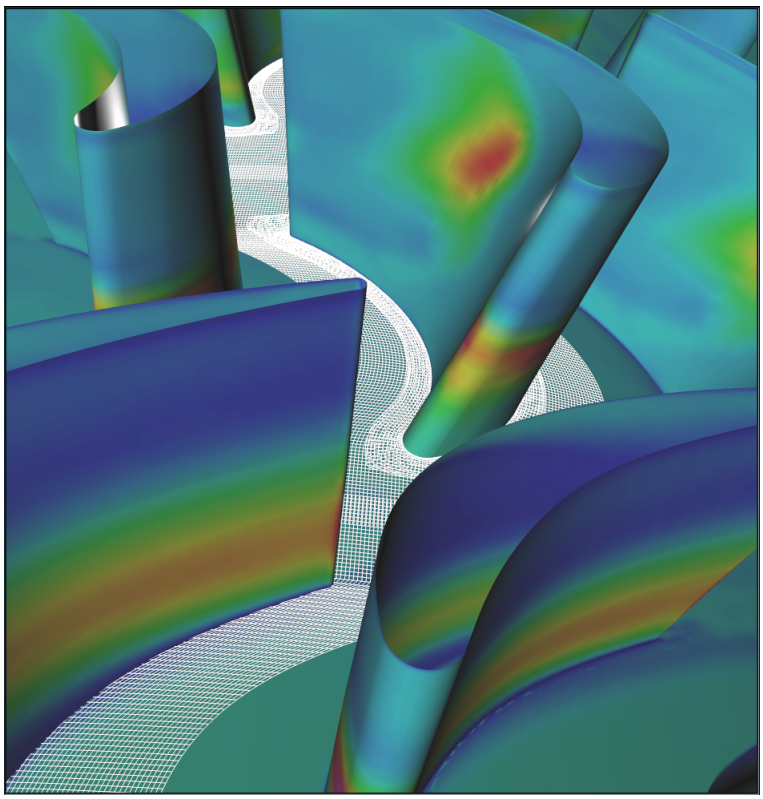


Figure 14.1: A close-up of a periodic domain modeling a turbine [GBD96]. The fundamental domain used for the simulation would extend upward from the white regions at the base of the blades. Data courtesy of Karen Gundy-Burlet, visualization by Tim Sandstrom.

14.1 How transformed meshes work

A subclass of `FEL_transformed_mesh` is constructed with the mesh to be transformed and the data required for a particular transformation T. For mesh member functions that do not depend on the mesh geometry, such as `card` and `up_cells`, the transformed mesh simply delegates the call to the original mesh. For member functions involving the mesh geometry, such as calls producing coordinates or the bounding box, a transformed mesh makes the corresponding call on the original mesh and then applies T to the result. For point location, a mesh representing a transformation T applies the inverse transformation T^{-1} to the given point and then calls the locate routine on the original mesh, using the inverse transformed point as an argument.

14.2 The transformed mesh subclasses

The transformed mesh subclasses are illustrated under `FEL_transformed_mesh` in the mesh hierarchy (Figure 11.1). The constructors for transformed mesh classes are:

```
FEL_translated_mesh(FEL_mesh_ptr m, const FEL_vector3f& t,
                    char* nm = "translated_mesh");
FEL_rotated_mesh(FEL_mesh_ptr, const FEL_matrix33f& r,
                 char* nm = "rotated_mesh");
FEL_x_rotated_mesh(FEL_mesh_ptr m, float a,
                   char* nm = "x_rotated_mesh");
FEL_y_rotated_mesh(FEL_mesh_ptr m, float a,
                   char* nm = "y_rotated_mesh");
FEL_z_rotated_mesh(FEL_mesh_ptr m, float a,
                   char* nm = "z_rotated_mesh");
```

Each constructor takes a mesh `m`, and an optional name `nm`. `FEL_translated_mesh` emulates a translation by a vector `t`. The `FEL_rotated_mesh` classes emulate rotation transformations. The rotations are represented by a 3×3 matrix `r` such that given an original point p and a matrix R, the transformed point $p' = R * p$. FEL also provides the classes `FEL_x_rotated_mesh`, `FEL_y_rotated_mesh`, and `FEL_z_rotated_mesh` for representing rotations about the x, y, and z axes, respectively. The angle `a` should be in degrees. These classes make it easier to construct a mesh rotated about a particular axis, since one does not have to remember the matrix terms. The rotation subclasses also make it possible to do some optimizations, since the transformations can be computed in fewer floating-point operations than required for the general rotation case.

Chapter 15

Multi-Zone Meshes

Multi-zone meshes are represented in FEL by the class `FEL_multi_mesh`. A multi-zone mesh consists of 1 or more zones, where each zone is a subclass of `FEL_single_mesh` or perhaps a transformed version of some single mesh object (see Chapter 14). Note that the zones do not have to be all of the same type, for example, one could construct a multi-zone mesh containing both structured and unstructured zones. A multi-zone mesh with a zone that itself is a multi-zone mesh is not allowed, because objects such as cells that need to specify a zone contain a single integer for that purpose, and the integer is used to index into a single level of hierarchy.

For most mesh member functions, a multi-zone mesh simply delegates the function call to a particular zone. In particular, member functions that take an incoming argument containing a zone number can be immediately delegated. The FEL classes `FEL_cell` and `FEL_structured_pos` both contain zone integers, but `FEL_phys_pos` does not. By default, the zone in an `FEL_cell` or an `FEL_structured_pos` is set to `FEL_ZONE_UNDEFINED`. It is an error to call a multi-zone mesh member function with an incoming argument containing an undefined zone.

15.1 Point location, IBLANK, and PLOT3D

Point location is the most difficult task that a multi-zone mesh must support. Given a point p to locate, a multi-zone mesh must find a zone and a cell containing p. In general, the task is complicated by the fact that zones can overlap; thus p may be located in not just a single zone, but possibly multiple zones. When there is more than one zone containing p, then a "best" zone must be chosen. Thus, to be completely thorough, a multi-zone mesh would attempt to locate p in every zone and then choose the best result. The test-every-zone approach is effective, but too expensive for most applications. FEL uses two strategies to improve upon the typical multi-zone point location performance, so that in most cases the test-every-zone approach is not necessary.

The first strategy for accelerating point location is the same as that used with other FEL mesh types: provide a start cell for the walking point location. Since FEL cells

contain a zone number, an initial cell argument effectively provides both a zone z to search first and a cell in z to start from. If the start cell is close to the location of the new point p, then it is possible that a cell containing p can be found quickly. Unfortunately, with a multi-zone mesh there is still the problem that the point p could be in more than one zone; thus even if the given start cell leads quickly to a cell c containing p, there is no assurance that c is the only cell containing p, or that c is the best cell containing p.

The second strategy for accelerating point location addresses the overlapping zones issue, and in general the issue of which zone to try next if point location in a particular zone fails. The strategy relies on PLOT3D [WBPE92] IBLANK values. Recall that in PLOT3D, an integer IBLANK of $-z$ can be used to suggest another zone z to search. If point location in a paricular zone fails, then a negative IBLANK still makes it possible to avoid resorting to the test-every-zone strategy. Furthermore, if point location in a particular zone is successful, then the lack of negative IBLANK values can be taken to mean that there are no other zones overlapping a given point, thus no more searching is needed.

IBLANK is also used by PLOT3D to flag nodes where the field data values are not valid (IBLANK of 0). As with curvilinear meshes, FEL returns an unsuccessful point location return code rather than a cell with a 0 IBLANK. In cases where there is more than one 0-IBLANK-free cell containing a given point, FEL must choose the best one. FEL chooses the cell with the smallest volume, since smaller cells typically come from higher-resolution meshes.

FEL is designed to be explicit about features that are specific to a particular file format or CFD standard, as much so as is practical. For multi-zone mesh point location, FEL depends upon IBLANKs, thus the point location code is PLOT3D specific. The PLOT3D dependency is made explicit in the FEL mesh class hierarchy: `FEL_multi_mesh` inherits from `FEL_mesh`, and `FEL_plot3d_multi_mesh` inherits from `FEL_multi_mesh` (see Figure 11.1). `FEL_multi_mesh` contains the vast majority of the interface and implementation for multi-zone meshes, the `plot3d` class implements point location. In the future, other multi-zone mesh classes can be derived from `FEL_multi_mesh`, each with its own system for representing information analogous to IBLANK.

15.2 Constructing a multi-zone mesh

The class `FEL_multi_mesh` is an abstract class; to make a concrete multi-zone mesh object, one must currently use the PLOT3D class. The constructor looks like:

```
FEL_plot3d_multi_mesh(int n_meshes, FEL_mesh_ptr* meshes,
                      char* nm = "FEL_plot3d_multi_mesh");
```

The parameter `n_meshes` specifies the number of meshes, `meshes` is an array of pointers to meshes. It is the responsibility of the multi-mesh class to deallocate the mesh pointer array when the multi-mesh is destructed.

All FEL meshes return IBLANK values of 1 by default if no IBLANK data are provided. Thus one can construct an `FEL_plot3d_multi_mesh` instance even if the individual zones do not contain explicit IBLANK information. Point location efforts

will tend to be slower, since there will be no zone jumping hints, and situations where a point is contained in more than one zone will not be detected.

Chapter 16

Iterators

Iterators provide an interface for looping over the cells in a mesh. The iterator interface is independent of the particular mesh type, for example whether the mesh is structured or unstructured. FEL iterators work with the generalized cell object `FEL_cell`; thus iterators can represent not just hexahedra or tetrahedra, but also lower-dimensional cells such as vertices, triangles, and quadrilaterals. The class `FEL_cell_iter` provides the general cell iterator functionality. FEL also provides the class `FEL_vertex_cell_iter`, which is used in a manner very similar to `FEL_cell_iter`, except that the iterator only represents vertices ("vertex cells").

16.1 Basic iterator usage

The basic use of iterators involves four operations: initialization, a done test, advancing to the next element, and dereferencing. The following code fragment illustrates all four operations:

```
FEL_mesh_ptr mesh;
FEL_vertex_cell_iter iter;
FEL_vector3f c;
int res;
...
for (mesh->begin(&iter); !iter.done(); ++iter) {
  res = mesh->coordinates_at_vertex_cell(*iter, &c);
  cout << *iter << " coordinates: " << c << endl;
}
```

The initialization is handled by the `begin` method supported by all FEL mesh classes. The test whether the iterator is done is accomplished via the `done()` method of iterators. Advancing the iterator is accomplished via the `++iter` call, and the iterator is dereferenced using the `*iter` syntax. Note how the dereferenced iterator can be used just as one would use an FEL_vertex_cell argument, e.g., as an argument to `coordinates_at_vertex_cell()`. One can also call methods on `*iter` that belong

to the FEL_cell class, but one must be a bit careful about syntax. In C++ (and C), the "." operator has higher precedence than "*". Thus, for example, the expression "*iter.get_ijk()" would parse the same as "*(iter.get_ijk())", which would not compile since the class FEL_cell_iter has no method get_ijk(). To access the methods belonging to the cell that an iterator represents, such as get_ijk(), one should write "(*iter).get_ijk()". (The operator required to support -> syntax is not currently defined in the library; therefore one cannot write "iter->get_ijk()".)

In a second example, we highlight two variations in the style of iterator usage:

```
FEL_field_ptr field;
FEL_vertex_cell_iter iter, end;
field->begin(&iter);
field->end(&end);
for ( ; iter != end; ++iter) {
  ...
}
```

The first variation shows how one tests whether an iterator is done: rather than calling the done() method, one can create a separate iterator object, initialized by the end() method of meshes, and then compare the original iterator with the end object using the != operator. The prefered FEL style is the former, i.e., use the done() member function rather than making a separate end object. The latter style is provided for use in the future with the Standard Template Library (STL) [MS96]. The example also highlights the fact that the iterator initialization routines (begin() and end()) are accessible via the field interface, so one does not have to get the mesh associated with a field in order to initialize iterators.

The usage of FEL_cell_iter iterators is nearly identical to the vertex cell iterator above. Where FEL_vertex_cell_iter appears above, one would instead write FEL_cell_iter. Calls taking FEL_vertex_cell arguments would have to be replaced with the appropriate calls taking an FEL_cell argument. FEL_vertex_cell_iter objects always iterate over vertices. FEL_cell_iter iterators, on the other hand, by default, loop over the highest-dimension cell type in the mesh. The cell type produced by FEL_cell_iter iterators is also a function of the simplicial decomposition mode for the mesh. So, for example, given a structured mesh containing hexahedra, an FEL_cell_iter produces hexahedra, or tetrahedra if simplicial decomposition is turned on. For a structured surface mesh (FEL_curvilinear_surface_mesh), FEL_cell_iter produces quadrilaterals, or triangles if simplicial decomposition is on. Iterators over FEL_tetrahedral_mesh instances produce tetrahedra, regardless of the simplicial decomposition state. Finally, cell iterators over FEL_multi_mesh instances produce the highest-dimension cell for each zone in the mesh.

16.2 Iterators and ordering

FEL iterators allow the user to specify subsets of cells over which to iterate, but the order in which the user sees the cells is fixed. For structured meshes, iterators advance the I index the fastest, followed by J, then K. See Chapter 12 for a description of how the data members of an `FEL_cell` are set in order to specify a particular structured mesh cell. With unstructured meshes, only the I index is used. For the `FEL_vertex_cell` iterator on an unstructured_mesh `m`, the I index goes from 0 to `m->card(0) - 1`. For the `FEL_cell` iterator on an unstructured_mesh `m`, the I index goes from 0 to `m->card(3) - 1`.

In the case of multi-zone meshes, FEL iterators process the zones in ascending order, using the default order of the mesh for each zone to control how the cells are produced.

16.3 Iterating over mesh subsets

In some cases, one may desire to iterate over a subset of the cells of a mesh rather than every one. In order to describe a subset of a mesh, one typically must know about the mesh type, e.g., whether the mesh is structured or unstructured. Unfortunately, this implies that one must give up some of the the mesh type independence afforded by the default behavior of FEL iterators. Nevertheless, mesh subset iteration is important in certain cases. FEL supports mesh subset iteration via optional pairs of keyword-value arguments provided to the `begin()` initialization statement.

Table 16.1 lists the keywords supported in iterator initialization. Also included in each of the S, U, and MM categories would be subclasses of the corresponding mesh class, for instance, `FEL_curvilinear_mesh` and `FEL_regular_mesh` would be included under the S category. The `FEL_I_MAX` parameter defaults to `dim[0] - 1` for a vertex iterator with a structured mesh. For unstructured meshes, `FEL_I_MAX` defaults to `card(0)` - 1. `FEL_J_MAX` and `FEL_K_MAX` are both set to 0.

The following excerpt illustrates an example where the iterator produces the vertices on the K = 0 surface of a structured mesh, with a stride of 2 in the I and J dimensions:

```
FEL_field_ptr field;
FEL_vert_pos_iter iter;
int res;
res = field->begin(&iter, FEL_K, 0,
                   FEL_I_STRIDE, 2, FEL_J_STRIDE, 2, 0);
if (res != 1) ...

for ( ; !iter.done(); ++iter) {
  ...
}
```

Note that the `begin()` statement above now has a return value, so that the library has the opportunity to indicate that one has provided initialization arguments that are

Keyword	Default	Mesh Types
FEL_I_MIN	0	S, U, MM*
FEL_I_MAX	dim[0] - 1	S, U, MM*
FEL_I_STRIDE	1	S, U, MM
FEL_J_MIN	0	S, MM*
FEL_J_MAX	dim[1] - 1	S, MM*
FEL_J_STRIDE	1	S, MM
FEL_K_MIN	0	S, MM*
FEL_K_MAX	dim[2] - 1	S, MM*
FEL_K_STRIDE	1	S, MM
FEL_I	(none)	S, MM*
FEL_J	(none)	S, MM*
FEL_K	(none)	S, MM*
FEL_UNSTRUCTURED_SURFACE	(none)	U, MM*
FEL_ZONE	0	MM

Table 16.1: The FEL iterator initialization keywords. The characters S, U, and MM in the column **Mesh Types** stand for FEL_structured_mesh, FEL_unstructured_mesh, and FEL_multi_mesh classes, respectively. The asterisk following MM indicates that the corresponding keyword is legal only if the initialization is for a specific multi-mesh zone.

not valid. For instance, the structured mesh initialization routine detects when the value for a parameter such as FEL_J_MAX is out of range, and returns a value not equal to 1 to indicate this error. Note also that the final argument to the begin() statement must always be 0 in order to indicate that are no more keyword/value pairs to follow.

In the case where one is initializing an iterator for a multi-zone mesh, some initialization keywords are allowed only if one also specifies that the iteration should be over a particular zone, using the FEL_ZONE keyword. In Table 16.1, the rows where MM is followed by an asterisk designate keywords which are allowed only in conjunction with multi-zone meshes if one is selecting a specific zone. This restriction is due to the fact that most parameters, such as FEL_I_MAX or FEL_UNSTRUCTURED_SURFACE, typically only make sense when applied to a particular zone.

The initialization of FEL_cell_iter instances is done using the same set of keywords as for FEL_vertex_cell iterators. The default values for the parameters are the same as for vertex iterators, except that the FEL_*_MAX values for structured meshes are initialized to dim[0] - 2, dim[1] - 2, and dim[2] - 2 for the I, J, and K indices, respectively.

For unstructured meshes, the FEL_I_MIN and FEL_I_MAX parameters allow the user to control the first and last cell in the sequence of cells produced. One can also specify a stride via FEL_I_STRIDE. Parameters controling J and K are ignored by unstructured meshes.

16.4 Iterating over surfaces

FEL iterators can also return vertices or cells from a surface. One can have a surface either because the underlying mesh is a surface mesh (e.g., `FEL_curvilinear_surface_mesh`) or because the iterator initialization keywords imply a surface. For structured meshes, one can specify a surface by holding either the I, J, or K index at a constant value. The iterator initialization keywords `FEL_I`, `FEL_J`, and `FEL_K` are used for this purpose. For example, in the code fragment above, the iterator initialization specifies the constant K = 0 surface. With structured meshes, iterating over a surface will produce quadrilaterals, or triangles if a simplicial decomposition mode is on. When iterating over a structured surface within a hexahedral mesh, the decomposition of the quadrilaterals into triangles matches the tetrahedral decomposition of the hexahedra in the original mesh.

With unstructured meshes, sometimes the input file specifies sets of triangles which are taken to be part of a surface. Each set of surface triangles has an associated indentification number. The user can access these triangle sets via the `FEL_UNSTRUCTURED_SURFACE` keyword, followed by a triangle set ID number.

16.5 Iterators and time

When working with time-varying data, one typically needs to initialize the time component of the cell represented by an `FEL_cell_iter` or an `FEL_vertex_cell_iter`. Both types of iterators support the methods `set_time()`, `set_physical_time()`, `set_computational_time()`, and `set_time_step()`. Using the set methods, one can initialize time for an iterator just as one would for an FEL cell. By default, the time associated with a cell is undefined. Note that FEL iterators do not currently iterate over time, i.e., advancing an iterator (using ++) does not change the time set by `set` calls described immediately above. On the other hand, the user is free to use the `set` calls within the iteration to manually change the time value.

Chapter 17

Fields

17.1 Fields in general

The field class hierarchy is the centerpiece of FEL. Conceptually, a field is a continuous region of space over which some physical quantity is continuously defined. At any given spatial location in the field, the associated physical quantity may vary with time, or it may be time invariant. In finite-difference computational simulations, field values are calculated only at discrete points, but the spatial and temporal sampling are chosen to yield a reasonable approximation to a continuous physical system. The FEL field class hierarchy attempts to provide a set of objects and methods (data types and functions) which allow the user to treat finite-difference simulation data abstractly as continuous fields, with minimal regard for the actual underlying discrete spatial and temporal representation. Moreover, the FEL field class hierarchy provides mechanisms for combining different fields, and for converting one field type to another, according to various mathematical or physical identities, so that as field values are retrieved they are operated on by various functions, and can thus be returned directly in a form suitable for a particular task.

17.2 Fields in context

The FEL field class hierarchy is rooted in `FEL_reference_counted_object` via `FEL_mutex_reference_counted_object`, so that fields can be managed using reference counting, and in a thread-safe manner. Reference counting can reduce unnecessary memory usage, a particularly desirable feature in the case of fields, which can be quite large. `FEL_reference_counted_object` also harbors a character string, which can be used to name any of its descendants, including fields.

Because fields are reference counted objects, they should be created only on the heap (via the `new` operator), using FEL's smart pointers as handles. More on instantiating fields later.

17.2.1 Typeless fields

Derived directly from the reference counting classes is `FEL_field`, the top of the field lineage proper. Unlike all of the other field classes, which inherit from it, `FEL_field` is not templated, and it contains not just a common interface, but common code shared by all fields, regardless of the type of field data. This node-type-independent code mostly provides an interface to the field's mesh, and to two types of iterators which traverse the mesh. There are also some functions for translating between physical and computational time, and for retrieving various global physical quantities associated with the field. Factoring out such code common to all fields, from the more parochial code of particular field types, not only embodies a clean conceptual separation, but it also may reduce redundant code generation by compilers that take an "all or nothing" approach to template instantiation.

From a client's point of view, `FEL_field`s are important because they are the common ancestor of all fields — hence a pointer to `FEL_field` (typedef'd as `FEL_field_ptr`) can refer to a field of *any* type. The `FEL_field` pointer thus provides the means by which one can store handles to potentially diverse field types in a single homogeneous container (such as an array), or construct functions to operate on arbitrary fields. This feature is invaluable in writing general purpose code. Of course, `FEL_field`s have such a generic interface that not much useful can be done with them without knowing some basic facts about their actual instantiations: methods to obtain these facts are provided by the `FEL_field is_*_field()` interface.

17.2.2 Typed fields

Derived from the generic `FEL_field` is the templatized `FEL_typed_field<T>`. `FEL_typed_field<T>` is parameterized by `<T>`, which is a placeholder for the type of data provided at each node. This parameterization allows a single body of code to support instantiation of fields of any type, so long as the type supports a few basic arithmetic operations (see Chapter 21). `FEL_typed_field<T>` specifies the type-dependent interface common to all fields – primarily the `at_*()` calls, which provide lazy evaluation of field values at given locations in space and time. For many applications, the `at_*()` calls are the heart of the FEL user interface. In addition, `FEL_typed_field` provides a convenience function for producing an "eagerly evaluated" field: this function precalculates field values at each node, and stores them into memory. By eliminating potentially redundant calculations, eager evaluation may be a logical choice if one plans on making heavy use of a highly derived field.

There are at present eleven immediate descendants of `FEL_typed_field` (see Figure 17.1). The most heavily used types will be described in more detail in following chapters, but for now here is a brief overview.

`FEL_core_field<T>`

A field whose node values reside in memory. Core fields are typically produced by reading a PLOT3D solution file from disk, by executing `get_eager_field()`, or



Figure 17.1: A portion of the FEL typed field hierarchy. The subclasses of `FEL_derived_field<T>` and `FEL_differential_operator_field<TO, FROM>` do not appear in this diagram. See Figure 20.1 for the differential operator subclasses.

by explicitly associating a mesh with a node buffer in a core field constructor. See Chapter 18.

`FEL_paged_field<T>`

This is a field whose node values are paged in from disk on demand — useful or even imperative for extremely large data sets. See Chapter 23.

`FEL_derived_field<TO>`

Under the lazy evaluation paradigm, derived fields are essentially filters which transform field data as part of their retrieval. Derived fields are always built on top of other fields, and while the derivation chains can be arbitrarily long, there must always be at least one field at the bottom which can retrieve or manufacture a field value "autonomously" to get the whole thing started. Derived fields are templatized by the type of data they produce. See Chapter 19.

`FEL_differential_operator_field1<TO,FROM>`
`FEL_differential_operator_field2<TO,FROM>`

These are specialized derived fields which apply the nabla operator to scalar or vector fields, producing other scalar or vector fields, using either first- or second-order approx-

imations for the required derivatives. The differential operator fields are templatized by their input and output types. See Chapter 20.

`FEL_constant_field<T>`

Constant fields return the same (constant) value from all locations. This can be useful for certain applications which want to combine fields algebraically — for example, one might shift the frame of reference of a velocity field by adding a constant velocity vector at all points. Note that the same effect can be achieved with derived field mapping functions.

`FEL_mesh_as_field<T>`

This field type in effect creates a vector field whose entry at each node is just the position vector of the node, as given by the mesh. Thus this field type constitutes an *adaptor*, allowing one to access a mesh using the field interface. Using this strategy, for instance, one can implement physical-space cutting surfaces on a mesh by extracting isosurfaces from the associated `FEL_mesh_as_field`. The mesh-as-field also allows one to convert computational coordinates to physical coordinates simply by using `FEL_mesh_as_field::at_structured_pos()`, but FEL provides a specialized and more efficient way of doing this: `coordinates_at_structured_pos()`, callable on any mesh.

`FEL_time_varying_field<T>`

This field type and its subclasses provide the additional interface necessary to support time-varying data. See Chapter 25.

`FEL_multi_field<T>`

This field type is basically a container class capable of managing multiple fields. It will be used primarily for supporting *transformed* fields.

`FEL_touch_counted_field<T>`

This field type simply sits on top of another field and records usage statistics from that field. Such statistics can be very informative during application development or tuning, as they may guide decisions about deployment of lazy evaluation, eager evaluation, or paged fields.

`FEL_cached_field<T>`

This field is built on top of another field, and caches node query results from that field. Field queries on revisited nodes can be fetched from the cache, potentially saving time by eliminating redundant calculations on a highly derived field.

`FEL_hash_cached_field<T>`

Similar to `FEL_cached_field`, but the cached data is stored in a dynamically created hash table, instead of in preallocated memory as in `FEL_cached_field`. The hashing scheme may save a lot of unnecessary storage space in sparsely accessed fields, at the cost of slightly higher cache retrieval times compared to `FEL_cached_field` (but still potentially faster than pure lazy evaluation, on a highly derived field).

17.3 Fields in detail

17.3.1 Every field has a mesh

A key data member of every field is its mesh, which is represented in the field by a `FEL_mesh_ptr`. Every instantiated field must contain precisely one valid mesh pointer. It is not possible to create a field without supplying a mesh of the same cardinality as the data buffer, since each node value must have an associated spatial location before it is possible to carry out such basic operations as point location or interpolation.

In contrast, a given mesh can be included by any number of fields, including none at all. A mesh by itself can support many purely geometric operations, whether or not any further data are attached to its vertices; and a single mesh can provide the spatial organization for many types of data, whether or not those data reside in memory or are constructed on the fly. A core field and any derived fields it supports will always share the same mesh.

Thus in FEL there is a considerable asymmetry between meshes and (non-positional) node data. The FEL paradigm depends on a *one-to-many* but nevertheless *tight binding* between a mesh and its node data. The main business of creating a core field is establishing this binding; any derived field merely inherits its input fields' mesh.

The `FEL_mesh_ptr` inside a field is protected, which means you can't access it directly. The access function `get_mesh()`, which you can call on any field, returns a pointer to the field's mesh. You can use this pointer to construct other fields, or to call any of the publicly available mesh functions. For example:

```
void foo(FEL_field_ptr field)
{
   FEL_mesh_ptr mesh = field->get_mesh();
   FEL_vector3f lo, hi;
   mesh->get_bounding_box(&lo,&hi);
   FEL_vector3f_field_ptr coord_field =
     new FEL_mesh_as_vector3f_field(mesh);
   FEL_cell_iter ci;
   FEL_vector3f pvec[FEL_CELL_MAX_NODES];
   for (mesh->begin(&ci); !ci.done(); ++ci)
     {
       coord_field->at_cell(*ci,pvec);
       isosurface(pvec, ...);
     }
```

```
}
```

Several of the commonly called methods on FEL_mesh have also been put in the FEL_field interface, so you can call them directly on a field without first retrieving the field's mesh pointer. In these cases, the field merely forwards your function call to its mesh. Functions in this category include:

```
int card(int);
int get_n_zones();
void set(FEL_set_keyword_enum, int) ;
int coordinates_at_cell(const FEL_cell&, FEL_vector3f []);
int coordinates_at_vertex_cell(const FEL_vertex_cell&,
                               FEL_vector3f*);
int convert_time(const FEL_time&,
                 FEL_time_representation_enum,
                 FEL_time*) const;
```

and the iterator functions:

```
void begin(FEL_vertex_cell_iter*);
int begin(FEL_vertex_cell_iter*, int, ...);
void end(FEL_vertex_cell_iter*);
void begin(FEL_cell_iter*);
int begin(FEL_cell_iter*, int, ...);
void end(FEL_cell_iter*);
```

See Chapter 11 for details of these functions.

17.3.2 The at_*() calls

The at_*() calls allow one to retrieve field values at arbitrary locations and times within the computational domain. Locations and times can be specified in either computational or physical coordinates, with the previously noted exception that fields based on unstructured meshes don't (can't!) support computational spatial dimensions. Since the at_*() calls are declared to return field values of a specific type, they are introduced to the field interface in the templatized FEL_typed_field<T>, the common ancestor of all instantiable fields.

In addition to being parameterized by the type of field value being retrieved, as just mentioned, the at_*() calls are named according to the type of the field location being queried. This scheme was adopted, rather than overloading on location type, for reasons involving C++ function hiding. FEL's design dictates that various specialized field types redefine certain functions that are initially defined as high as possible in the class hierarchy. The at_*() calls are virtual, so that this redefinition gives rise to polymorphism. However, in C++ it is impossible to selectively override only a subset of a group of overloaded functions: if even a single member of the group is overridden, the rest are effectively hidden. This function hiding can only be overcome by redefining the *entire* group of overloaded functions, and for those that are unchanged, providing

explicit redirection up the class hierarchy. This solution is inefficient — it may take multiple (redirected) virtual function calls to reach the actual method — and it also results in needlessly cluttered class declarations and definitions.

FEL largely sidesteps these issues by renaming each of the `at_*()` calls according to the type of location being queried. This permits the different varieties of `at_*()` to be independently overridden, at the cost of a slightly more cumbersome function name. Renaming functions according to their argument types is known as "name mangling", and is the strategy employed by C++ compilers to distinguish between overloaded functions. FEL is "partially mangling" the `at_*()` interface, but hasn't abandoned overloading altogether: the `at_phys_pos()` calls are overloaded with respect to other argument types.

The partial mangling and overloading in the `at_*()` interface results in 8 distinct calls:

```
int at_vertex_cell(const FEL_vertex_cell&, T*);
int at_cell(const FEL_cell&, T[]);
int at_cell_interpolant(const FEL_cell_interpolant&, T[]);
int at_structured_pos(const FEL_structured_pos&, T*);
int at_phys_pos(const FEL_phys_pos&, T*);
int at_phys_pos(const FEL_phys_pos&, FEL_cell_interpolant*, T*);
int at_phys_pos(const FEL_phys_pos&, FEL_cell_interpolant&,
                FEL_cell_interpolant*, T*);
int at_phys_pos(const FEL_phys_pos&,
                const FEL_cell_interpolant&, T*);
```

The first three of these calls retrieve field values directly from nodes (or sets of nodes), and do not involve any spatial interpolation. `at_vertex_cell()` returns field values from an individual vertex in the mesh, and `at_cell()` returns an array of field values from the group of mesh vertices making up a cell. The cell can be any type supported by FEL (see Table 5.1), and since one cell type is FEL_CELL_VERTEX, `at_vertex_cell()` is really just a specialized version of the more general `at_cell()`. Because no spatial interpolation is necessary to fetch field values at cell vertices, `at_cell_interpolant()` simply disregards the interpolant part of the cell_interpolant, and in all other respects is essentially the same as `at_cell()`. It is provided as a convenience to the user.

The calls `at_structured_pos()` and `at_phys_pos()` can retrieve field values at arbitrary locations, as specified in computational (structured meshes only!) or physical coordinates, respectively. The requested location may not coincide with a mesh vertex, so some type of spatial interpolation is necessary. In `at_structured_pos()` the interpolation is always performed in computational space, using the computational coordinates supplied with the query. This is tantamount to isoparametric interpolation, discussed in Chapter 9. With `at_phys_pos()`, the spatial interpolation is done in either computational space or physical space, depending on the current interpolation mode (see Chapter 11 for details about setting the interpolation mode). In either case, if repeated and relatively localized `at_phys_pos()` queries will be made, much of the "set-up" work associated with the interpolation can

be profitably saved and potentially reused. Previous point location and interpolation information can be cached in an `FEL_cell_interpolant` object, and the overloaded versions of `at_phys_pos()` support the reuse of these hard-won data. For this reason, there is a hierarchy of `at_phys_pos()` calls, and the most appropriate call for a given field query depends on how much prior knowledge is on hand:

- `at_phys_pos(const FEL_phys_pos&, T*)` does global point location to find the mesh cell enclosing the queried physical position, builds an interpolant based on that cell's geometry, uses the interpolant to produce an interpolated field value which it stores into T*, then deletes the interpolant. This is the most expensive and most extravagant `at_phys_pos()` call, and is generally used only when an isolated field value is required.

- `at_phys_pos(const FEL_phys_pos&, FEL_cell_interpolant*, T*)` does global point location to find the mesh cell enclosing the queried physical position, builds an interpolant based on that cell's geometry, uses the interpolant to produce an interpolated field value which it stores into T*, then stores the interpolant into `FEL_cell_interpolant*`, whence it is returned to the client. This is just as expensive as the previous call, but at least now we have the *possibility* of reusing the information stored in the cell interpolant!

- `at_phys_pos(const FEL_phys_pos&, FEL_cell_interpolant&, FEL_cell_interpolant*, T*)` does "local" point location, i.e. starting in the cell of the supplied `FEL_cell_interpolant&`. This will converge much more quickly than a global search, if the queried physical point is nearby; and there is the added bonus that the interpolant part of the supplied `FEL_cell_interpolant` can be reused, if the queried physical point falls in the supplied cell. In any case, the current cell and interpolant are returned via `FEL_cell_interpolant` for the next go around. This is a safe and efficient method if successive `at_phys_pos()` calls are highly spatially correlated, as when integrating a stream line, for instance. Be warned, however, that the local search can be much *slower* than a global search, if the desired physical point is far from the supplied cell.

- `at_phys_pos(const FEL_phys_pos&, const FEL_cell_interpolant&, T*)` doesn't do point location at all, but simply assumes the provided `FEL_cell_interpolant` is appropriate and goes ahead with the interpolation. This is certainly the fastest `at_phys_pos()` call, but risky if you're not sure the `FEL_cell_interpolant` matches the `FEL_phys_pos`. If it doesn't, the `at` call may fail, but even worse, it may well succeed, and silently produce garbage that may or may not be easily recognized as such. Be sure you know what you're doing if you use this streamlined `at_phys_pos` variant.

All the `at_*()` calls apply to time-varying fields, in which case temporal interpolation may be necessary (*may* because temporal interpolation is bypassed if the requested time falls squarely on a timestep). The positional classes in the various `at_*()` calls

all have an entry for time, which is simply ignored in time-invariant fields. See Chapter 25 for a more detailed discussion of time-varying fields.

The `at_*()` calls store the field values they retrieve into locations specified by a user-supplied pointer (`T*`). The return value of the function itself signifies the outcome of the retrieval: FEL_OK for success, and other values resulting from various mishaps. If the `at()` call fails, the returned field value slot (`T*`) will probably be unchanged, and the returned `FEL_cell_interpolant` will probably be bogus. Since the `T*` is frequently reused, it may still point to a valid looking value, and be unwittingly used. Reusing a bogus `FEL_cell_interpolant` may result in gross inefficiencies, further failed `at` calls, or even a core dump (typical scenario: failure to create an interpolant results in it being set to NULL, and then one blindly uses it in an `at_phys_pos(const FEL_phys_pos&, const FEL_cell_interpolant&, T*)` call ... SIGSEGV lies this way). For these reasons, good coding style, and general peace of mind, one should always check the return values of the `at_*()` calls.

17.3.3 Iterating over fields

As a convenience, `FEL_field` supports the same iterator interface as `FEL_mesh`. In fact, iterator methods invoked on a field are merely forwarded to the field's mesh. This shorthand somewhat compromises the abstraction of a field as a continuous domain, but allows a more relaxed coding style in which one simply makes all function calls off a field, without having to remember which ones are more mesh-related. For instance:

```
int foo(FEL_float_field_ptr field)
{
   float f[FEL_CELL_MAX_NODES];
   field->set(FEL_SIMPLICIAL_DECOMPOSITION,0); // forwarded
   FEL_cell_iter ci;
   for (field->begin(&ci); !ci.done; ++ci)     // forwarded
     {
       field->at_cell(*ci,f);                  // field method
       if (any(f)<0.0)
         field->coordinates_at_cell(*ci,c);    // forwarded
       field->at_phys_pos(average(c),&f)       // field method
       cout << (*ci)
            << "has negative vertex: "
            << "scalar at centroid = "
            << f
            << endl;
     }
}
```

See Chapter 16 for a more complete discussion on the capabilities and uses of iterators.

17.3.4 Eager fields

FEL defaults to "lazy evaluation" of field values. This means that all field values FEL returns via the `at_*()` calls, and any intermediate values required for their evaluation, are generated only when needed to satisy a given `at_*()` call. The sole exceptions to this rule are the `FEL_core_field` nodal values, which reside in a buffer in memory, or perhaps on disk in the case of a `FEL_paged_field`. The creation of a derived field merely sets up an appropriate filtering mechanism, which is not pressed into action until derived values are requested. Particularly in the case of highly derived fields, lazy evaluation saves a lot of storage space and setup time, at the expense of slower turnaround when any values are actually needed, and potential redundant calculations if the same locations are queried over and over again.

Sometimes it might make better sense to evaluate derived fields at all their mesh vertices and store these results in memory. Then when node values are needed, FEL merely has to fetch them, for immediate return, or for derivative or interpolation purposes. This "eager evaluation" strategy essentially converts a derived field into a core field. It requires more storage space, but particularly in the case of highly derived fields, can provide faster immediate access, and can also provide longer term savings by eliminating redundant calculations if the same locations are repeatedly queried. An eager field is a field level cache.

One creates an eager field by invoking `get_eager_field()`, which is a method on `FEL_typed_field`, so it can be called on any typed field. `get_eager_field()` returns a `FEL_pointer` to the same type of field that calls it. Thus, assuming `lazy_vector_field` is already defined:

```
FEL_vector3f_field_ptr eager_vector_field =
  lazy_vector_field->get_eager_field();
```

In the case of time-varying fields, `get_eager_field()` requires an argument specifying the computational time of the newly produced field; see Chapter 25 for details. If `get_eager_field()` fails — most likely because it can't allocate enough memory or, in the time-varying case, if the time is invalid — it returns NULL. Be sure to check the return value of `get_eager_field()` before you do something embarassing like trying to dereference a NULL pointer.

Eagerly evaluated fields are probably a logical choice if you are faced with some combination of the following:

1. you need the quickest possible access to a field
2. you are repeatedly accessing lots of locations in the field
3. you have a very highly and expensively derived field
4. you have enough memory

An application supporting interactive sweeping of a gridplane through a velocity $\times$ vorticity magnitude field is a good example meeting the first three criteria above.

17.3.5 Field type "informant" functions

The FEL field class hierarchy strives for polymorphic behavior, but sometimes you just can't pretend anymore and you simply *must* know what specific type of field is really being represented by some generalized FEL field pointer. For such desperate times, FEL provides type-specific "informant" functions which return true or false depending on whether the invoking field meets the criterion encoded in the function name. There are six such functions:

```
bool is_core_field();
bool is_time_series_field();
bool is_float_field();
bool is_vector3f_field();
bool is_plot3d_q_field();

bool varies_with_time();
```

These functions are methods on `FEL_field`, so they can be called on any type of field whatsoever. Note that a given field can return true for more than one of the queries; only the return values from `is_float_field()`, `is_vector3f_field()`, and `is_plot3d_q_field()` are mutually exclusive.

An affirmative response to an `is_*_field()` call indicates that the queried field really *is* an object of the queried type, so that a valid downcast to the queried type is possible. In fact, type confirmation before downcasting is the primary intended use of the `is_*_field()` calls.

On the other hand, `varies_with_time()` can return true for any of the field types which can be built on top of a time-varying field. The `varies_with_time()` query is generally used as part of a switch or optimization in a general application, since the methods and resources for managing steady and unsteady data are so different.

Note that on a given field, `varies_with_time()` can return true and `is_time_series_field()` may return false. This would happen, for example, if the field in question were a derived field built on top (directly or indirectly) of a time series field. Keep in mind that an `is_*_field()` call tells you where a field is declared in the FEL class hierarchy, whereas `varies_with_time()` tells you only that a field has a time-varying member somewhere in its individual lineage (the group of fields that are "chained" together by successive definitions). In biological terms, `is_*_field()` calls are queries about *phylogeny*, and `varies_with_time()` pertains to *ontogeny*.

Using the informant functions, one can be tempted to write code like this:

```
int foo(FEL_field_ptr field)
{
   if (field->is_float_field())
      do_scalar_stuff(FEL_FIELD_TO_FLOAT_FIELD_CAST(field));
   else if (field->is_vector3f_field())
      do_vector_stuff(FEL_FIELD_TO_VECTOR3F_FIELD_CAST(field));
   else
      return 0;
```

```
   return 1;
}
```

Suit yourself, but be aware that this explicit style may become hard to maintain if used with abandon. If you have lots of type-dependent code, it may be better to localize the branch points, say by pushing them all inside a "Factory" class, which manufactures polymorphic objects that can be passed to type-independent functions.

17.3.6 Odds and ends

Min/max values

Sometimes it is useful to know the minimum and maximum values of a scalar field, for example if one is fine-tuning a transfer function. To this end, FEL provides the method `get_min_max()`, which returns the minimum and maximum nodal values of a scalar field:

```
int res;
float min,max;
res = some_float_field->get_min_max(&min,&max);
if (res == FEL_OK) ...
```

For time-varying fields the method requires an additional argument specifying the computational time, and as usual, the function return value signals the final outcome of the call. Although the method is declared on `FEL_typed_field`, it produces meaningful results only on float fields, i.e. fields on which `is_float_field()` returns `true`. For all other field types, `get_min_max()` prints a message on standard error, leaves the pointer arguments untouched, and returns something other than FEL_OK.

In addition to taking care of the dirty work of iterating over the field in search of extrema, `get_min_max()` caches the minimum and maximum field values once it finds them. On any given field, the second and subsequent invocations of `get_min_max` merely conjure up the cached values. This is very fast, and makes it unnecessary for the client to store the minimum and maximum field values explicitly.

User data

All FEL fields have two slots for user-defined data:

```
int user_type;
void* client_data;
```

These are part of the top level `FEL_field` interface, so they are available in any field.

The `user_type` entry is a single user-managed integer that is meant to be used as a simple type-tag if, say, the user wants to categorize fields differently than FEL does. One scenario has the user assigning to the tag at field creation time, using a custom enumeration. Functions can then be written to branch in their treatment of the fields

after examining the tag (but see warnings above in Section 17.3.5. The tag could also be used to track individual *instances* of fields, rather than *types*.

The `client_data` is a general purpose pointer that can be used to associate arbitrary data with a field. One simple possibility for using `client_data` is just a generalization of the `user_type` scenarios. The user could attach a `struct` to a field, which contained type and instance records, and perhaps some other useful descriptive information. Another idea is to use the `client_data` hook to turn traditional data-driven frameworks inside-out: Instead of constructing a framework which *manages fields*, by explicitly associating them with particular meta-data, visualization techniques, and so on, one could stuff all the bookkeeping inside the field itself, together with appropriate behaviors for interacting with certain environments — so that, in the extreme, fields could largely *manage themselves*. Developers should keep in mind, however, that while the `client_data` mechanism allows fields to be arbitrarily enhanced and extended, such modifications are likely to be relatively domain specific, and should not be confused with enhancements and extensions to the Field Encapsulation Library itself.

Chapter 18

Core Fields

The previous chapter discussed the generic interface declared for all fields on the `FEL_field` and `FEL_typed_field` base classes. This chapter presents a more specific treatment of a particularly important subclass of typed field called a core field.

18.1 What are core fields?

Core fields (`FEL_core_field<T>`) are fields whose node values reside in memory. This is in contrast to the various sorts of derived fields, for which node values are generated only on demand. Core fields live at the roots of "derivation chains" (see Chapter 19) and provide raw values which derived fields modify.

Core fields are produced by reading a solution file from disk, by executing `FEL_typed_field::get_eager_field()`, or by explicitly associating a mesh with a node buffer in a core field constructor.

Reading solution files from disk is the subject of Chapter 22, and `get_eager_field()` is discussed in Chapter 17. Here we will describe how to construct a core field "manually" in memory. This operation is essential, for example, if one wants to load data directly from a field simulation into FEL, or if one wants to write a file reader for FEL.

18.2 The node buffer

As with any other type of field, construction of a core field requires a preexisting mesh. Meshes may be created by reading in a file from disk (see Chapter 22), or by declaring a `FEL_regular_mesh`. Details for creating a mesh "manually" in memory are given in Chapter 12 (for structured meshes) and Chapter 13 (for unstructured meshes).

Assuming a mesh is already on hand, the central task of creating a core field is allocating a buffer and filling it with the node values. In most cases, the buffer should be allocated on the free store using either the `new` operator or `operator new` (or `operator new[]`, if your compiler supports it). This is because in most cases, once

the core field is created, memory management of the node buffer is turned over to FEL, and when the reference count to a core field reaches zero, FEL will try to deallocate its node buffer using `delete[]`. Attempting to `delete[]` memory which hasn't been obtained via `new` is disastrous.

If you want to retain responsibility for memory management of core field node buffers, you can do so by requesting that the core field "suppress deallocation". If this option is chosen, the node buffer is left untouched when the core field is destructed. In this case, therefore, the memory need not be allocated with `new` — the memory may be obtained by some other dynamic allocator, or it may be static. In any event, the user is responsible for maintaining a handle to the node buffer, and for freeing the memory if so desired. Of course, the user should not deallocate the node buffer while the core field is still in use! Directions for choosing the "suppress deallocation" option are given below.

Core fields are templatized, and can be constructed with any node type `T` supporting a few basic arithmetic operations (see Chapter 21). Since the node buffer must hold a type `T` for every vertex of its associated mesh, the buffer must be at least of size (in bytes):

```
mesh->card(0) * sizeof(T)
```

The elements of the node buffer must be arranged in the same order as the corresponding vertices of the associated mesh. For structured meshes of dimension (`idim, jdim, kdim`), this means that the sequential memory layout has `idim` varying most rapidly, and `kdim` most slowly. C/C++ programmers should be wary here, as this column-major convention may seem out of place (but in general is more convenient for data coming from predominantly FORTRAN solvers). For unstructured meshes, the sequential layout is arbitrary and completely determined by the vertex ordering in the mesh. In both the structured and unstructured cases, FEL vertex cell iterators follow the linear arrangement of mesh vertices in memory, so if you are creating a node buffer whose values depend on vertex coordinates, an `FEL_vertex_cell_iter` can be employed to guarantee vertex-node correspondence.

For core fields with multi-element node types, based on either structured or unstructured meshes, the multiple elements associated with a given node should be layed out as a contiguous group in memory. Note that this natural layout differs from certain file formats, including the PLOT3D solution file format, in which vector *components* are grouped together.

18.3 Constructors and suppressed deallocation

There are three core field constructors. With a preexisting mesh and an allocated and filled node buffer, one creates a core field with one of the following two constructors:

```
FEL_core_field(FEL_mesh_ptr, T*, char* = "core_field");
FEL_core_field(FEL_mesh_ptr, T*, bool, char* = "core_field");
```

The `FEL_mesh_ptr` argument takes a pointer to the intended mesh of the core field. `T*` should point to the head of a node buffer of type `T` which you have allocated and filled with type `T`'s, according to the scheme outlined above. The `bool` argument in the second constructor is used to indicate whether or not you want to suppress deallocation of the node buffer when the core field's reference count reaches zero: `true` for suppressed deallocation (you are responsible for deallocation), `false` for automatic deallocation (FEL assumes management of the node buffer). If you use the first constructor, "suppressed deallocation" defaults to `false`. The last core field constructor argument is an optional name, which defaults to "core_field" if you don't supply a more imaginative name of your own.

The third core field constructor —

```
FEL_core_field(FEL_mesh_ptr, FEL_pointer< FEL_core_field<T> >,
               char* = "shared_core_field");
```

— takes a pointer to a preexisting core field rather than a pointer to a node buffer, and creates a new core field by binding the node buffer of the preexisting core field to the incoming mesh represented by `FEL_mesh_ptr`. This results in two core fields which share a single node buffer but associate the data with different meshes. For this to make sense, the different meshes must have the same number of vertices (`mesh1->card(0) == mesh2->card(0)`) — if this is not the case, bad things will happen. In a "shared core field" the node buffer is a shared resource, so the suppress deallocation option defaults to `true`.

As we have just seen, the suppress deallocation option on a core field is set at construction time, either implicitly or explicitly, but it can also be changed at any time by a `set` call, if you change your mind about node buffer memory management:

```
// FEL deletes buffers:
my_core_field->set(FEL_SUPPRESS_DEALLOCATION, 0);
// you're on your own:
my_core_field->set(FEL_SUPPRESS_DEALLOCATION, 1);
```

It's probably not a good idea to set suppress deallocation to false (0) on a shared core field.

18.4 An example

Here is a rather contrived example which constructs a core field "manually". See Chapter 12 for a description of `FEL_regular_mesh`, and Chapter 16 for details on `FEL_vertex_cell_iter`.

```
#include "FEL.h"

int main()
{
  int idim=3,jdim=4,kdim=5;

  FEL_mesh_ptr mesh =
    new FEL_regular_mesh(idim,jdim,kdim,1,1,1);

  FEL_vector3f* node_buffer =
    new FEL_vector3f[mesh->card(0)];

  assert (node_buffer);

  int n=0;
  FEL_vertex_cell_iter iter;

  // fill node buffer:
  for (mesh->begin(&iter);!iter.done();++iter)
    {
      int i = (*iter)[0];
      int j = (*iter)[1];
      int k = (*iter)[2];
      node_buffer[n++].set(atan2(j,i),
                           atan2(k,sqrt(i*i+j*j)),
                           sqrt(i*i+j*j+k*k));
    }

  // create core field
  // (suppress deallocation default is false)
  FEL_vector3f_field_ptr my_core_field =
    new FEL_core_field<FEL_vector3f>
    (mesh, node_buffer);

  FEL_vector3f v;

  for (mesh->begin(&iter);!iter.done();++iter)
    {
      my_core_field->at_vertex_cell(*iter,&v);
      cout << *iter << " : " << v << endl;
    }

  my_core_field = NULL;  // node_buffer delete[]'d

  return 0;
}
```

18.5 get_eager_field()

Core fields are also generated by the `get_eager_field()` call. This method, available on any field, evaluates the field at every vertex and writes the results into a `new`ly allocated node buffer, which is bound to the mesh of the calling field. This function is typically used to convert a lazily evaluated derived field into a core field, for quicker access time, but could conceivably be used to duplicate an existing core field, in the unlikely event this was useful. More verbiage on `get_eager_field` can be found in Chapter 17.

Chapter 19

Derived Fields

19.1 What is a derived field?

A "derived field" is a field whose values are computed (derived) from one or more other fields. The derivation referred to here involves some kind of mathematical mapping, and should not be confused with the C++ sense of derivation (although the FEL derived field classes *do* inherit from other classes).

A derived field is essentially a filter, built on top of some number of *component fields*. The derived field and all its component fields necessarily share the same mesh. When the derived field is queried for a value at some location, it forwards the request to its component fields, gathers the values returned by the component fields, and then transforms these field values according to some function before passing them back to the caller.

The function that the derived field uses to produce its values from its component field values is called a *mapping function*. As its name suggests, the mapping function maps component field values into derived field values. The mapping functions are either predefined by the library (more on these below) or user-defined. User-defined mapping functions allow arbitrary transformations of field data to be encapsulated in the retrieval process, so that field queries on derived fields can return data directly in a form suitable for a particular task.

The component fields of a derived field can be of any type, including other derived fields. This last possibility allows "derivation chains" of arbitrary depth, and at any level the three types of component fields can be freely mixed. Naturally, several different derived fields can share component fields. Thus an application can build an arbitarily complex set of derived fields, but their relationships can always be described by a directed acyclic graph of one or more components. Since queries on derived fields are always forwarded to component fields, any given derivation "lineage" must eventually terminate in a field type capable of producing a value autonomously. The field types that fit this bill are core fields (which pull a value from memory), paged fields (which pull a value from memory or disk), "mesh as fields" (which use mesh coordinates as field values), and constant fields (which simply return the same value for all

queries).

19.2 Lazy vs. eager evaluation

By default, evaluation of derived field quantities is completely demand driven. That is, no field values are generated when a derived field is created: derived field quantities are computed only in response to a client query. This strategy is variously described as *lazy evaluation*, or *deferred evaluation*, or a *pull model*.

Because of the lazy evaluation paradigm, creation of derived fields requires very little time and storage. Internal creation of a derived field consists mainly of registering the component fields and mapping function, which are represented by pointers in the derived field. On our workstations (SGI R10Ks), creation of a derived field requires about 30 microseconds, and about 200 bytes of memory. Thus it is quite reasonable for an application to present many derived fields (hundreds, or more!) to a user as selectable options, creating them either at startup time or on demand, even if most of the fields are never used.

The downside of the lazy evaluation scheme is lengthier derived field value retrieval times, compared to core fields, since the derived field must manufacture its values on demand. For simple derived fields (single core component field, mapping function requiring only local values) retrieval times are about 25% longer than core field retrieval. As the derivation chain comprises more component fields and more involved mapping functions, the derived field retrieval times increase accordingly.

If retrieval time is at a premium and ample storage space is available, it may be desirable to precalculate and store derived field values across the entire domain, or at least retain and reuse those derived field values which are generated to satisfy client requests. The first of these aims can be accomplished by way of `get_eager_field()`, a method on `FEL_typed_field` which iterates over all vertices of its invoking field and writes their values into newly allocated storage. This converts a derived field into a core field[1], and may be the logical choice if one wants to minimize retrieval time on a highly derived field. The second option above can be realized by creating a `FEL_cached_field` on top of a derived field, which retains derived field values in a cache, and attempts to satisfy client requests from the cache before deriving anew. Cached fields may be useful if a derived field is going to be accessed repeatedly. Examples of creating eager and cached fields are given below.

19.3 Mapping and interpolating

There are two distinct types of derived fields in FEL: `FEL_map_then_interpolate_derived_field*` and `FEL_interpolate_then_map_derived_field*`. As the names suggest, the difference between these two types depends on whether the mapping function is applied before or after any necessary interpolation. More specifically: To satisfy a general physical or computational space query (*i.e.*, `at_phys_pos()` or `at_structured_pos()`), a field must first

[1] ... or can also be used to generate a copy of a core field, in the unlikely event this was desired.

perform point location to determine which gridcell contains the query point. Then field values are obtained for each vertex of this enclosing cell and used to interpolate a field value at the enclosed point.

- In an `FEL_map_then_interpolate_derived_field*`, the derived field queries its component field(s) at each vertex of the enclosing cell, applies the mapping function at each vertex to convert component field quantities to derived field quantities, then interpolates the derived field quantity at the query location.

- In an `FEL_interpolate_then_map_derived_field*`, the derived field queries its component field(s) at each vertex of the enclosing cell, interpolates the component field quantities at the query location, then applies the mapping function to convert the component field quantities to the derived field quantity.

The relative order of mapping and interpolation can make a big difference in the final derived field value, particularly if the mapping function is nonlinear. Which ordering is most "appropriate" depends on the problem at hand. In addition to numerical implications, there are efficiency considerations: mapping before interpolation invokes the mapping function on every vertex (potentially expensive for a complex mapping function) but interpolates only a single variable; interpolating before mapping invokes interpolation on all component field values (potentially expensive if there are several component fields, with complex node types) but applies the mapping function only once. The built-in derived fields are all of type `FEL_map_then_interpolate_derived_field*`. This option produces the same results one gets with precalculated (eagerly evaluated) component fields. Note that for vertex-based queries (*i.e.*, `at_cell()`), which entail no interpolation, the distinction between `FEL_map_then_interpolate_derived_field*` and `FEL_interpolate_then_map_derived_field*` is immaterial.

19.4 Built-in derived fields

FEL provides several "prepackaged" derived fields. The differential operator fields are specialized derived fields which are discussed separately in Chapter 20. The rest of the FEL built-in derived fields are each briefly described in the following list. The sample declarations indicate the type of the derived field produced, and also the type(s) of the component fields which must be provided as arguments. The generic component fields `float_field` and `vector3f_field` represent fields already created in an application, from which you want to derive some new fields. The newly created derived fields are called `derived_float_field` and `derived_vector3f_field`. All derived fields are templatized, but for clarity the typedef'd names are used here. Therefore further variations on these built-in derived fields can be generated by using the templatized declarations directly.

- ```
  FEL_float_field_ptr derived_float_field =
  new FEL_magnitude_of_vector3f_field(vector3f_field);
  ```
  ```

derived_float_field now points at a newly created scalar field whose values are the lengths (Euclidean norms) of the vectors at the corresponding points in vector3f_field.

- FEL_float_field_ptr derived_float_field = new FEL_absolute_value_of_float_field(float_field);

  derived_float_field now points at a newly created scalar field whose values are the absolute values of the scalar values at the corresponding points in float_field.

- FEL_float_field_ptr derived_float_field = new FEL_negate_of_float_field(float_field);

  derived_float_field now points at a newly created scalar field whose values are the additive inverses of the scalar values at the corresponding points in float_field.

- FEL_vector3f_field_ptr derived_vector3f_field = new FEL_negate_of_vector3f_field(vector3f_field);

  derived_vector3f_field now points at a newly created vector field whose values are the additive inverses of the vector values at the corresponding points in vector_field – that is, the corresponding vectors in vector_field and derived_vector_field have equal magnitudes but opposite directions.

- FEL_float_field_ptr derived_float_field = new FEL_sum_of_float_field(float_field1, float_field2);

  derived_float_field now points at a newly created scalar field whose values equal the sum float_field1 and float_field2 at corresponding points – that is, derived_float_field = float_field1 + float_field2.

- FEL_vector3f_field_ptr derived_vector3f_field = new FEL_sum_of_vector3f_field(vector3f_field1, vector3f_field2);

  derived_vector3f_field now points at a newly created vector field whose values equal the vector sum of vector3f_field1 and vector3f_field2 at corresponding points – that is derived_vector_field = vector3f_field1 + vector3f_field2.

- FEL_float_field_ptr derived_float_field = new FEL_difference_of_float_field(float_field1, float_field2);

  derived_float_field now points at a newly created scalar field whose values equal the differences between float_field1 and float_field2 at corresponding points – that is, derived_float_field = float_field1 - float_field2.

- `FEL_vector3f_field_ptr derived_vector3f_field = new FEL_difference_of_vector3f_field(vector3f_field1, vector3f_field2);`

  `derived_vector3f_field` now points at a newly created vector field whose values equal the difference vectors between `vector3f_field1` and `vector3f_field2` at corresponding points – that is, `derived_vector_field = vector3f_field1 - vector3f_field2`.

- `FEL_float_field_ptr derived_float_field = new FEL_product_of_float_field(float_field1, float_field2);`

  `derived_float_field` now points at a newly created scalar field whose values are the products of `float_field1` and `float_field2` at corresponding points – that is, `derived_float_field = float_field1 * float_field2`.

- `FEL_float_field_ptr derived_float_field = new FEL_quotient_of_float_field(float_field1, float_field2);`

  `derived_float_field` now points at a newly created scalar field whose values are the quotients of `float_field1` and `float_field2` at corresponding points – that is, `derived_float_field = float_field1 / float_field2`.

- `FEL_float_field_ptr derived_float_field = new FEL_dot_of_vector3f_field(vector3f_field1, vector3f_field2);`

  `derived_float_field` now points at a newly created scalar field whose values are the scalar or dot products of `vector3f_field1` and `vector3f_field2`.

- `FEL_vector3f_field_ptr derived_vector_field = new FEL_cross_of_vector3f_field(vector3f_field1, vector3f_field2);`

  `derived_vector_field` now points at a newly created vector field whose values are the vector or cross products of `vector3f_field1` and `vector3f_field2`.

- `FEL_float_field_ptr derived_float_field = new FEL_component_of_vector3f_field(vector3f_field, i);`

  `derived_float_field` now points at a newly created scalar field whose values are the i^{th} component of the vectors at the corresponding points in `vector3f_field`. Component numbering starts at 0.

- `FEL_float_field_ptr derived_float_field = new FEL_component_of_plot3d_q_field(plot3d_q_field, i);`

`derived_float_field` now points at a newly created scalar field whose values are the i^{th} component of the PLOT3D solution vectors at the corresponding points in `plot3d_q_field`. Component numbering starts at 0.

19.4.1 Customizing the built-in derived fields

As mentioned above, derived fields are templatized, and the built-in fields are no exception: the examples just shown use typedef'd versions to hide the template syntax. You can achieve some level of customization of the built-in derived fields by using the template syntax directly.

The templated declarations of the built-in derived fields look like this:

```
template <class TO, class FROM>
class FEL_magnitude_field;

template <class TO, class FROM>
class FEL_absolute_value_field;

template <class TO, class FROM>
class FEL_negate_field ;

template <class TO, class FROM1, class FROM2>
class FEL_difference_field;

template <class TO, class FROM1, class FROM2>
class FEL_sum_field;

template <class TO, class FROM1, class FROM2>
class FEL_product_field;

template <class TO, class FROM1, class FROM2>
class FEL_quotient_field;

template <class TO, class FROM1, class FROM2>
class FEL_dot_field;

template <class TO, class FROM1, class FROM2>
class FEL_cross_field;

template <class TO, class FROM>
class FEL_component_field;
```

In these declarations, the `TO` and `FROM` classes are the template parameters, or "type placeholders". The `FROM` parameters represent the type(s) of the component

field(s), and the TO parameter represents the type that is derived from them. These parameters must be replaced with the actual input and output types of the derived field you wish to construct – using the appropriate syntax, of course.

For example, say you want to multiply a vector field by a scalar field, that is, you want a derived vector field which scales the vectors of one of its component fields by the scalar values of its other component field. Here's a declaration using FEL_product_field:

```
FEL_vector3f_field_ptr scaled_vector_field =
 new FEL_product_field<FEL_vector3f, FEL_vector3f, float>
       (unscaled_vector_field, float_field);
```

The arguments in the <>'s are the template parameters. The first such argument is the TO type: this derived field will produce vectors. The second and third template parameters are the FROM types – in other words, the types of the component fields from which the derived field will produce its values.

The arguments in the ()'s are fodder for the derived field's constructor. These arguments are pointers to fields of type FROM1 and FROM2 – in this case, an FEL_vector3f_field_ptr, and an FEL_float_field_ptr, respectively. These fields must both share the same mesh, and must be properly initialized at the time the derived field is constructed. All the constructors for the built-in derived fields require pointers to fields of the FROM types (and, optionally, a name for the field).

In this example derived field (scaled_vector_field), the vectors of unscaled_vector_field are scaled by the floating point values of float_field at corresponding locations. One could invoke a *single* (location invariant) scaling factor by making float_field an FEL_constant_field (see Chapter 17).

19.5 PLOT3D derived fields

In addition to the generic built-in derived fields described in the last section, FEL supplies over fifty specific derived fields defined by PLOT3D. These predefined derived fields can be created by calls to specialized convenience functions, or by enum'ed requests to a PLOT3D "field manager". For more information on both of these options, see Chapter 24.

19.6 Constructing a custom derived field

If the built-in derived fields don't offer the functionality you need, you can define your own from scratch. And even if you can achieve the same functionality by other means, encapsulating certain data transformations in a derived field can be an effective programming strategy. The most important and involved step in defining a derived field is creating the mapping function.

19.6.1 Writing a mapping function

The mapping function defines how derived field data are actually derived from the component field(s). The mapping function must be declared and defined by the user, and passed to a derived field constructor as a pointer. Each of the derived field constructors expects a pointer to a function with one of three prototypes, depending on whether the derived field operates on one, two, or three component fields:

```
int map_func(const FEL_solution_globals&,
             const FROM1*,
             void*,
             TO*)

int map_func(const FEL_solution_globals&,
             const FROM1*, const FROM2*,
             void*,
             TO*)

int map_func(const FEL_solution_globals&,
             const FROM1*, const FROM2*, const FROM3*,
             void*,
             TO*)
```

Of course, you can name your own mapping function whatever you'd like; "map_func" is used here solely as an illustration.

`TO`, `FROM1`, `FROM2`, and `FROM3` are template "type placeholders", which are replaced by actual types in actual mapping function declarations and definitions. The only difference among the three prototypes is the number of `FROM`s, which corresponds to the number of component fields in the derived field. In all cases, the `TO` parameter represents the node type of the derived field itself. Thus, the `FROM`s represent the input to the mapping function, the `TO` represents the output, and the guts of the mapping function just specify how the `FROM`s produce the `TO`.

Here is an example of a mapping function, which calculates the magnitude of the difference between two input vectors. This can be used to construct a derived scalar field, whose isosurfaces, for instance, show the spatial pattern of discrepancy between two vector fields.

```
int vector_difference_mag(const FEL_solution_globals&,
                          const FEL_vector3f* v0,
                          const FEL_vector3f* v1,
                          void*,
                          float* mag)
{
  *mag = FEL_magnitude(*v0 - *v1);
  return 1;
}
```

This function just subtracts one input vector (`v1`) from the other (`v0`) and passes the vector result to `FEL_magnitude`, whose floating point return value is stored into the location pointed to by `mag`. The `FEL_solution_globals&` and `void*` are unnamed here to preclude compiler warnings about unused variables. The first three arguments must be declared `const` to inform the compiler that they are not modified by the mapping function. You must be sure to include these `const` declarations, and you must be sure to honor them (*i.e.*, don't try to modify these arguments in the mapping function), or your code will not compile. Also note that the first argument is a *reference* and must be so declared. Omitting either the `const` qualifier or the reference declarator ("&") will usually cause a compiler error in the derived field constructor taking this mapping function (with a message like "no instance of constructor [for whatever derived field you are trying to make] matches the argument list").

The transformation of `FROM`s to `TO` may involve any sort of mathematical gymnastics one wishes. Additional *data* which may (or may not) figure in the transformation are available from the `FEL_solution_globals` and (if you provide it) the `void*`.

Using the `FEL_solution_globals`

The `FEL_solution_globals` contain a collection of values associated with the derived field. Four of these values –

```
float free_stream_mach;
float alpha;              // angle of attack
float reynolds_number;
float time;               // usually not used
```

– are ultimately taken from the PLOT3D solution file header of the *first* component field (or one of its ancestors, if it is a derived field itself). Since the multiple component fields of a derived field must share a common mesh, they will typically also share the same header data; but be careful if you are somehow combining disparate data. There are a handful of other fields in the `FEL_solution_globals` structure which are used by various built-in mapping functions.

Here is another sample mapping function, which is set up to receive a velocity vector input and uses the the `free_stream_mach` and `alpha` values from the `FEL_solution_globals` to calculate the perturbation velocity:

```
int perturbation_velocity(const FEL_solution_globals& sg,
                          const FEL_vector3f* v,
                          void*,
                          FEL_vector3f* pv)
{
   FEL_vector3f vinf;
   vinf[0] = sg.free_stream_mach * cos((M_PI/180.) * sg.alpha);
   vinf[1] = sg.free_stream_mach * sin((M_PI/180.) * sg.alpha);
   vinf[2] = 0.;  // no sideslip angle beta ...
   *pv = *v - vinf;
   return 1;
}
```

Client data via `void*`

The `void*` can be used to provide arbitrary client data to the mapping function. At creation time, you can register a `void*` with the derived field, and this pointer will be passed to the mapping function every time it is invoked. This mechanism provides the mapping function with directions to find any additional data it may require – data which can be specifically updated for a given query on the derived field.

An example using client data passed to a mapping function will be given below.

Handling exceptional cases

Mapping functions may encounter problematic data. For example, a mapping function may receive a "0" input value which figures in the denominator of some expression. Singular cases like these can be indicated by the `int` return value of the mapping function. The derived field `at_call`s forward this return value to the client. As usual, the client should check the return value of the `at_call`, and be prepared to respond appropriately. Ignoring the return value is perilous: if the `at_call` fails, the field value "return slot" will not be updated, and therefore will contain stale and possibly misleading data.

19.6.2 Derived field declarations and constructors

There are two types of derived fields, which allow you to determine the relative ordering of mapping and interpolation (see Section 19.3). If you want mapping to precede interpolation, use `FEL_map_then_interpolate_derived_field*`. If you want interpolation to precede mapping, use `FEL_interpolate_then_map_derived_field*`.

The `*`'s in these class names are wildcards for the number of component fields of the derived field. The `*` should be replaced by "1", "2", or "3", as appropriate.

The non-built-in derived fields must be declared using template syntax. The template arguments are enclosed in angle brackets (`<>`s), immediately following the derived class type specifier. The template arguments specify the type of the derived field itself (the output type of the mapping function), and the type(s) of the component field(s) (the input type(s) of the mapping function). The derived field type (`TO`) is given first, followed by the component field type(s) (`FROM`).

A derived field which implements a "dot product" operation takes two vector fields (`FROM1` and `FROM2`) and uses them to calculate a floating point value (`TO`). The template part of the declaration would look like:

```
FEL_map_then_interpolate_derived_field2<float, FEL_vector3f,
                                                FEL_vector3f>
```

or

```
FEL_interpolate_then_map_derived_field2<float, FEL_vector3f,
                                                FEL_vector3f>
```

Following the angle-bracketed template arguments is a parenthesized list of arguments for the derived field's constructor. In order, these are a list of pointers to the component fields, a pointer to the mapping function, a pointer to client data, and an optional name. The component field pointers should be listed in the same order as they appear in the mapping function – this ordering establishes the correspondence between component field values sent to and received by the mapping function. In both `C` and `C++` using "`&`" to get the address of a function is optional, so the name of the mapping function will serve as a pointer; however, by all means use "`&`" if it makes you feel better.

Assuming that we already have on hand component vector fields `input_vector_field1` and `input_vector_field2`, and a mapping function `make_dot_product` that converts two vectors into a float, complete derived field declarations follow:

```
FEL_float_field_ptr dot_product_field =
  new FEL_map_then_interpolate_derived_field2
        <float, FEL_vector3f, FEL_vector3f>
        (input_vector_field1, input_vector_field2,
         make_dot_product, NULL);

FEL_float_field_ptr dot_product_field =
  new FEL_interpolate_then_map_derived_field2
        <float, FEL_vector3f, FEL_vector3f>
        (input_vector_field1, input_vector_field2,
         make_dot_product, NULL, "my_dot_product_field");
```

The ``NULL'' argument fills the slot for the (unused) `void*` client data. In the second example, we've given the derived field an (optional) name, which can be retrieved via `dot_product_field->get_name()`. It often improves readability to break up the template and constructor arguments over several lines, as we've done here. A healthy compiler won't mind.

19.6.3 Derived field checklist

Declaring and constructing a derived field requires some planning. Here is a little checklist:

- Map then interpolate, or interpolate then map? The relative ordering of these two operations is determined by the type of derived field you construct; *i.e.*, there are separate types of derived fields (`FEL_map_then_interpolate_derived_field*` and `FEL_interpolate_then_map_derived_field*` for each option. See Section 19.3 above.

- How many component fields are there? Component fields are the "input" to the derived field, from which its values are derived. FEL allows one, two, or three component fields. This number immediately follows "`field`" in the complete

derived field class name. It is unusual to need more than three component fields. If you have more than this number of component fields, you may be able to "factor" the derivation into parts, each involving three or fewer components, and construct the overall derivation incrementally.

- What are the component field types? These types (typically `float`, `FEL_vector3f`, or `FEL_plot3d_q`) are needed for the template arguments in the derived class declaration.

- What are the component field instances? The component fields (of the types just mentioned in the item above) on which the derived field is built need to exist when the derived field is constructed. FEL pointers to these fields are passed as arguments to the derived field constructor.

- What is the mapping function? The mapping function converts component field types to the type of the derived field itself. A pointer to this function must be provided to the derived field constructor. (Or at least one must provide a function pointer of the appropriate type, which should actually point to a function of the approriate type when the derived field is queried. By redirecting the function pointer registered with the derived field, one could dynamically change the derivation, based on some conditions, just prior to querying the field...)

- Are there any client data which need to be passed to the mapping function? A pointer can be registered with the derived field at construction time, which provides arbitrary data to the mapping function. The data can be changed at any time, for instance, just prior to querying the field.

19.6.4 A more or less complete derived field example

Here is an example which constructs a derived field representing the signed distance from a plane. Such a field could be used in conjunction with an iosurface routine, for instance, to implement a cutting plane routine.

```
typedef struct
{
  FEL_vector3f n;   // normal
  float p;          // point
} hessian;          // Hessian normal form of a plane

// the mapping function
int signed_distance(const FEL_solution_globals&,
                    const FEL_vector3f& test_point,
                    void* plane,  // client data
                    float* dist)
{
  hessian* hnf = (hessian*)plane;
  // calculate signed distance
```

```
    *dist = FEL_dot(hnf->n, test_point) + hnf->p;
    return 1;
  }


  int cutting_plane(FEL_plot3d_field_ptr plot3d_field,
                    hessian* plane,
                    FEL_float_field_enum scalar_field)
  {

     FEL_mesh_ptr mesh = plot3d_field->get_mesh();

     // "convert" mesh to position vector field
     FEL_vector3f_field_ptr pvec =
       new FEL_mesh_as_vector3f_field(mesh);

     FEL_float_field_ptr distance_field =
       new FEL_map_then_interpolate_derived_field1
       <float,FEL_vector3f>
       (pvec, signed_distance, plane);

     FEL_float_field_ptr color_by_field =
      plot3d_field->make_float_field(scalar_field);

    // colormapping via "color_by_field"
    isosurf(distance_field, color_by_field, ...);
  }
```

In this example, the typedef'd struct `hessian` contains a point and normal defining a plane. The mapping function `signed_distance()` uses the point and normal to calculate the signed distance from the plane (negative on one side, positive on the other) to an incoming `test_point`. The mapping function is used by the derived field `distance_field`, along with the plane equation, which is provided as client data (actual argument `plane`) to the derived field constructor. The derived field is built on top of a single component field — `pvec` — which is a field of the position vectors of the underlying mesh, courtesy of the adaptor class `FEL_mesh_as_vector3f_field` (see Chapter 17). When `distance_field` is queried for a value at a given location, it returns a signed scalar indicating the distance of the query location from the plane described in `plane`. Thus an `at_cell()` call on `distance_field` will return an array of values indicating the distance of each vertex of a cell from a plane. Interpolating the 0-valued surface on this field (via marching cubes, or the like) will yield the cutting plane specified by `plane`.

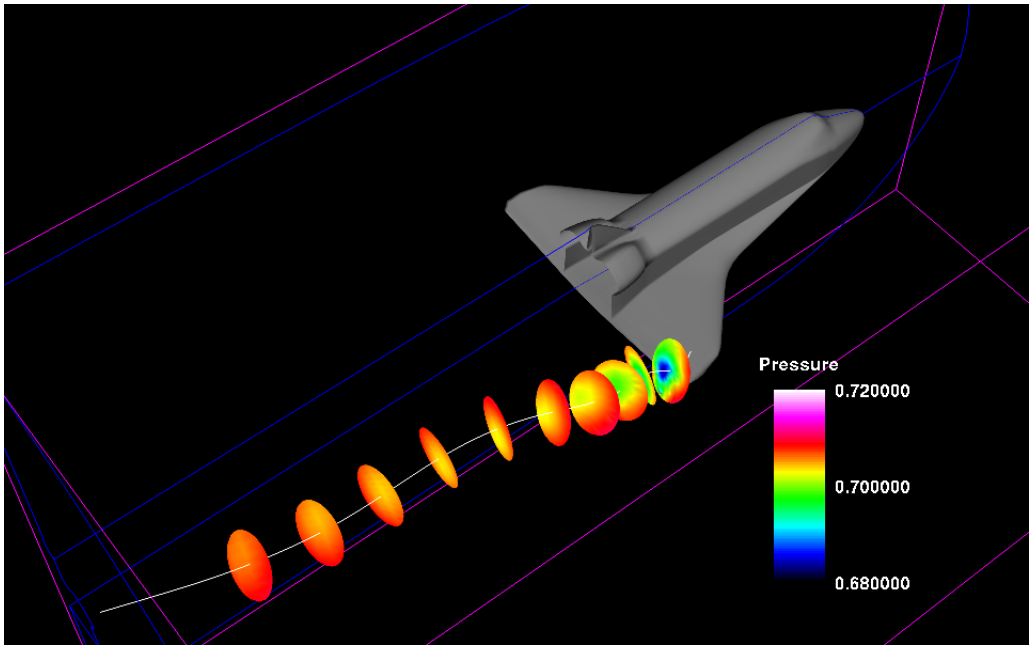


Figure 19.1: A visualization illustrating several uses of derived fields. The white line originating from the trailing edge of the shuttle wing is a streamline traced in the velocity derived field. Using the streamline as a local frame of reference, cutting planes were constructed as in the example in this chapter, and then each plane was trimmed to a disk. Textured on each disk is the pressure derived field. Visualization by Chris Henze.

Chapter 20

Differential Operator Fields

20.1 Gradient, divergence, and curl

The FEL differential operator fields are specialized derived fields which apply the differential operators grad, div, and curl, to a preexisting *base field*, in three-dimensional Euclidean space. The three operators are typically written in terms of a single operator, the nabla operator (∇).

20.1.1 Grad

The gradient operator acts on a scalar field and produces a vector field indicating the local direction and magnitude in which the scalar field is changing most rapidly. In Cartesian coordinates, where f is a scalar field, $f = f(x, y, z)$:

$$grad\ f = \nabla f = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j} + \frac{\partial f}{\partial z}\mathbf{k}$$

The gradient vectors are always perpendicular to the level surfaces (isosurfaces) of f. The spatial derivative of f in any direction is just the projection of the gradient vector on that direction.

20.1.2 Div

The divergence operator acts on a vector field and produces a scalar field indicating the local net outward flow per unit of time and volume. In Cartesian coordinates, where $\mathbf{F}$ is a vector field, $\mathbf{F} = \mathbf{F}(x, y, z)$, and F_x is the x-component of $\mathbf{F}$ (likewise for y and z):

$$div\ \mathbf{F} = \nabla \cdot \mathbf{F} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z}$$

Locations in a vector field $\mathbf{F}$ where $\nabla \cdot \mathbf{F} > 0$ are called *sources* (more outflow than inflow), locations where $\nabla \cdot \mathbf{F} < 0$ are called *sinks* (more inflow than outflow), and locations where $\nabla \cdot \mathbf{F} = 0$ are called *source-free*. If the entire field is source-free, $\mathbf{F}$ is called *solenoidal*.

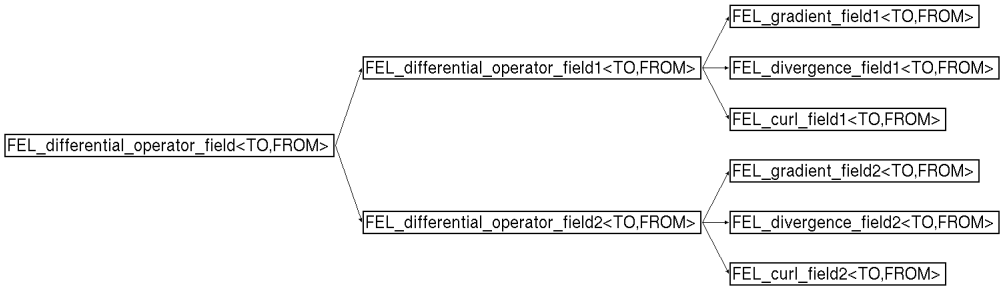


Figure 20.1: The FEL differential operator field class hierarchy.

20.1.3 Curl

The curl operator acts on a vector field and produces another vector field indicating the local direction and magnitude of the rotation of the original vector field. In Cartesian coordinates, where $\mathbf{F}$ is a vector field, $\mathbf{F} = \mathbf{F}(x, y, z)$, and F_x is the x-component of $\mathbf{F}$ (likewise for y and z):

$$curl\ \mathbf{F} = \nabla \times \mathbf{F} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}\right)\mathbf{i} + \left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x}\right)\mathbf{j} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y}\right)\mathbf{k}$$

At a given location in a swirling vector field, the curl is proportional to the local angular velocity. The curl is nonzero in "straight" vector fields with shear. Vector fields with $\nabla \times \mathbf{F} = 0$ everywhere are called *irrotational*.

20.2 First-order and second-order accuracy

As you can see from the descriptions in the previous section, all FEL differential operator fields require *spatial derivatives* of their scalar or vector base field values. In FEL, these spatial derivatives are estimated by either a first-order accurate scheme, or a second-order accurate scheme, depending on the type of differential operator field you construct. Figure 20.1 shows that the FEL differential operator field class hierarchy has two parallel lineages: the "1" or "2" in the class names determines whether a first-order or a second-order accurate scheme is used in estimating the derivatives used by a given differential operator.

In the first-order differential operator fields (descendants of `FEL_differential_operator_field1<TO,FROM>`), the same interpolating polynomial which is used to interpolate values on the base field is analytically differentiated to produce an expression yielding interpolated derivatives. If the differential operator field interpolation mode (see Chapter 9) is `FEL_ISOPARAMETRIC_INTERPOLATION`, the computational space shape functions on a given cell are differentiated with respect to the computational coordinates, the resulting partial derivatives are evaluated at the appropriate locations, and then transformed into physical space by way of the metrics. If the interpolation mode is `FEL_PHYSICAL_SPACE_INTERPOLATION`, the physical

space interpolating polynomial on a given cell is differentiated with respect to physical space coordinates, and the requisite partial derivatives can be evaluated directly where needed. In either case, the resulting physical space partial derivatives are used to generate the required differential operator quantities.

In the second-order differential operator fields (descendants of `FEL_differential_operator_field2<TO,FROM>`), partial derivatives are estimated at each vertex of a cell by central differencing in computational space, and these derivatives are transformed into physical space by the metrics. Then the differential operator quantities are generated at each cell vertex and, if necessary, interpolated at interior locations.

In general, the second-order differential operator fields will produce more accurate and reliable results than the first-order fields, but the second-order fields require more work, since the central difference scheme demands field values from a larger stencil.

At present, second-order differential operator fields are not supported on unstructured meshes. Techniques exist for an analogue of the central difference scheme on unstructured meshes [Bar91], and may be included in a future release. Note, however, that second-order differential operator fields *are* supported on the tetrahedral meshes derived from structured meshes by simplicial decomposition.

20.3 Creating differential operator fields

Differential operator fields are built on top of preexisting fields, which we refer to here as *base fields*. As long as the base field has the proper node type (scalar or vector) to which the differential operator applies, it may itself have been created in any number of ways, i.e., the base field may be a core field, a derived field, or perhaps another differential operator field.

The base field is provided to the differential operator field constructor as an FEL pointer. In fact, the only other argument to the differential operator field constructors is an optional name, in the form of a character string. You choose between first- and second-order numerical schemes by instantiating a differential operator field with a "1" (first-order) or "2" (second-order) as the last character in the class name.

The differential operator fields are all templated (parameterized) by two types: the type they return ("`TO`") and the type of the base field they operate on ("`FROM`"). As usual, there is a selection of predefined typedefs for the most common instantiations, which allow you to bypass the template syntax in many cases. See the FEL Reference Manual for a complete list of available typedefs.

Using a first-order gradient field as an example, the prototypical differential operator field declaration looks like this:

```
FEL_pointer< FEL_typed_field<TO> > grad_field =
 new FEL_gradient_field1<TO, FROM>
  (FEL_pointer< FEL_typed_field<FROM> >);
```

The differential operator field is declared to operate on base field type `FROM` and produce type `TO`. Therefore the constructor argument is a pointer to a field of type

FROM, and the differential operator field itself, returned by new, is represented by a pointer to a field of type TO.

Thus,

```
FEL_float_field_ptr temperature_field;
...

FEL_vector3f_field_ptr grad_of_temperature_field =
 new FEL_gradient_field1<FEL_vector3f, float>
  (temperature_field);
```

produces a vector field which at any location returns the gradient of temperature_field. Using a typedef, the same declaration could be accomplished as:

```
FEL_float_field_ptr temperature_field;
...

FEL_vector3f_field_ptr grad_of_temperature_field =
 new FEL_gradient_of_float_field(temperature_field);
```

Here are a few more annotated examples.

```
FEL_float_field_ptr temperature_field;
FEL_vector3f_field velocity_field;
...

// 1st order
FEL_float_field_ptr div_of_velocity_field =
 new FEL_divergence_of_vector3f_field1(velocity_field);

// 2nd order
FEL_float_field_ptr div_of_velocity_field =
 new FEL_divergence_of_vector3f_field2(velocity_field);

// 2nd order, double precision output
FEL_double_field_ptr div_of_velocity_field =
 new FEL_divergence_of_vector3f_field2<double,FEL_vector3f>
   (velocity_field);

// curl of velocity = vorticity
FEL_vector3f_field_ptr vorticity_field =
 new FEL_curl_field2<FEL_vector3f,FEL_vector3f>(velocity_field);

// typedef'd version
FEL_vector3f_field_ptr vorticity_field =
 new FEL_curl_of_vector3f_field2(velocity_field);
```

20.4 "Chaining" differential operator fields

As mentioned previously, the base field of a differential operator field can itself be a differential operator field. This allows operator "chaining" to create new operators. For example, the Laplacian operator (∇^2, sometimes written Δ), in Cartesian coordinates, operating on a scalar field $f = f(x, y, z)$, is defined to be:

$$\Delta f = div\ grad\ f = \nabla \cdot (\nabla f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2}$$

In light of this definition, one can declare a scalar Laplacian differential operator field as follows:

```
FEL_float_field_ptr pressure;  // a derived field: de-
fined elsewhere ...

FEL_vector3f_field_ptr grad_pressure =
 new FEL_gradient_of_vector3f_field2(pressure);

FEL_float_field_ptr laplacian_pressure =
 new FEL_divergence_of_vector3f_field2(grad_pressure);
```

Now `laplacian_pressure` is a scalar field which returns the second spatial derivative, or "curvature", of the pressure field. Note that `pressure` is itself a derived field, whose values are only calculated on demand.

Chaining the differential operator fields is a powerful technique, and FEL supports chains of arbitrary length; but one must be aware of the numerical limitations of gridded data. Numerical differentiation by way of finite difference schemes magnifies noise in the data, and is subject to truncation and roundoff error. These factors can quickly overwhelm any meaningful signature in a dataset, due to the repeated numerical differentiation entailed by a chain of differential operator fields.

The numerical problems associated with repeated differentiation are particularly severe on unstructured grids. Unstructured grids support only first-order differential operators, and these produce constant values across a given tetrahedron. Subsequent differentiation at the nodes of the constant-valued tetrahedra invariably yields zero derivatives. On structured grids, chained differential operator fields should be second-order, to avoid the derivatives from "bottoming out" prematurely.

In addition to these numerical issues, there are performance considerations associated with chained differential operator fields. In the second-order central difference scheme, evaluating the derivatives at a given vertex requires base field values from a neighborhood of adjacent vertices. If the base field itself requires central difference values, it will have to fetch values from yet a wider neighborhood, encompassing the first. This "expanding neighborhood" (or "stencil") around each vertex results in significant overhead for a field query. In addition, adjacent vertices have largely overlapping "expanded neighborhoods", so adjacent field queries result in a lot of duplicate work. For these reasons, if performance is an issue, it may make sense to eagerly evaluate (see Chapter 17) or at least cache (see Chapter 17) the results of a chained differential operator field, particularly if it involves highly derived fields.

Here is an example combining differential operator fields and derived fields. The Laplacian operator, in Cartesian coordinates, operating on a vector field $\mathbf{F} = \mathbf{F}(x, y, z)$, is defined to be:

$$\Delta\mathbf{F} = grad\ div\ \mathbf{F} - curl\ curl\ \mathbf{F} = \nabla(\nabla \cdot \mathbf{F}) - \nabla \times (\nabla \times \mathbf{F}) = \frac{\partial^2 \mathbf{F}}{\partial x^2} + \frac{\partial^2 \mathbf{F}}{\partial y^2} + \frac{\partial^2 \mathbf{F}}{\partial z^2}$$

We can define a vector Laplacian operator field as follows:

```
FEL_vector3f_field_ptr v_field; // some vector field, de-
fined elsewhere

FEL_float_field_ptr div_field =
 new FEL_divergence_of_vector3f_field2(v_field);

FEL_vector3f_field_ptr grad_div_field =
 new FEL_gradient_of_float_field2(div_field);

FEL_vector3f_field_ptr curl_field =
 new FEL_curl_of_vector3f_field2(v_field);

FEL_vector3f_field_ptr curl_curl_field =
 new FEL_curl_of_vector3f_field2(curl_field);

FEL_vector3f_field_ptr vector_laplacian_field =
 new FEL_difference_of_vector3f_field
  (grad_div_field, curl_curl_field);

// if desired, convert into core field
vector_laplacian_field =
 vector_laplacian_field->get_eager_field();
```

Chapter 21

Instantiating Fields

The most prominent use of templates in FEL is for the node type of fields. The templating makes it possible for the user to construct fields with a new node type with a minimal amount of code. FEL requires a few basic operations be supported for the node type, so that the library can interpolate if necessary when queried about field values. The number of operators required is intentionally kept small to make it easier to introduce custom types.

To demonstrate the minimal requirements of the node type, we present a few example types below. Keep in mind that most built-in numerical types support all the required operations, and then some.

21.1 Basic type requirements

The first example type is the "`foo`" type. A `foo` has basically the behavior of a spartan scalar.

```
class foo {
  float value;
public:
  foo() { }
  foo(float v) : value(v) { }
  friend foo operator*(double d, const foo& f) {
    return foo((float) (d * f.value));
  }
  friend foo operator+(const foo& lhs, const foo& rhs) {
    return foo(lhs.value + rhs.value);
  }
  // ostream operator not necessary, but handy
  friend ostream& operator<<(ostream& strm, const foo& f) {
    return strm << f.value;
  }
};
```

A "`foo`" must have a default constructor (so that one can allocate an array of them), and we provide a constructor with a float argument so the example is not too trivial. The `*` and `+` operators support multiplying a `foo` by a scalar and adding two `foo` objects together. In a program, the instantiation of a `foo` field would look like:

```
main() {
  // make a dummy mesh
  FEL_mesh_ptr mesh = new FEL_regular_mesh(5, 7, 11);

  // convenience typedefs for foo fields
  typedef FEL_field<foo> FEL_foo_field;
  typedef FEL_pointer<FEL_foo_field> FEL_foo_field_ptr;
  typedef FEL_core_field<foo> FEL_core_foo_field;

  // make a foo field and iterate over it
  foo* foo_data = new foo[mesh->card(0)];
  // should fill in the foo data buffer ...
  FEL_foo_field_ptr foo_field;
  foo_field = new FEL_core_foo_field(mesh, foo_data);
  FEL_vertex_cell_iter iter;
  int res;
  for (foo_field->begin(&iter); !iter.done(); ++iter) {
    foo f;
    res = foo_field->at_vertex_cell(*iter, &f);
    assert(res == FEL_OK);
    cout << "field value at " << *iter << " is " << f << endl;
  }
```

The `typedef` statements are not essential, but they come in handy further down the line, since the template syntax can get a bit tedious.

An example closer to what one might do in practice involves the construction of a field where each node has a vector of 10 doubles:

```
  typedef FEL_vector<10,double> FEL_vector10d;
  typedef FEL_field<FEL_vector10d> FEL_vector10d_field;
  typedef FEL_pointer<FEL_vector10d_field> FEL_vector10d_field_ptr;
  typedef FEL_core_field<FEL_vector10d> FEL_core_vector10d_field;
  FEL_vector10d* vector10d_data =
    new FEL_vector10d[mesh->card(0)];
  FEL_vector10d_field_ptr vector10d_field =
    new FEL_core_vector10d_field(mesh, vector10d_data);
  for (vector10d_field->begin(&iter); !iter.done(); ++iter) {
    FEL_vector10d v;
    res = vector10d_field->at_vertex_cell(*iter, &v);
    assert(res == FEL_OK);
    cout << "field value at " << *iter << " is " << v << endl;
  }
```

```
}
```

The vector template is defined by FEL (see Chapter 3); there is no need to provide the required math operators since they are already in the library.

21.2 Differential operator field requirements

If differential operator fields will be constructed with the new node type, then a few more operators must be defined:

```
class bar {
  float value;
public:
  bar() { }
  bar(float v) : value(v) { }
  friend bar operator*(double d, const bar& b) {
    return bar((float) (d * b.value));
  }
  friend bar operator+(const bar& lhs, const bar& rhs) {
    return bar(lhs.value + rhs.value);
  }
  friend bar operator-(const bar& lhs, const bar& rhs) {
    return bar(lhs.value - rhs.value);
  }

  // the following should not be necessary,
  // but are needed for some SGI compilers, which can
  // get confused at instantiation time
  friend bar operator*(const bar& b, double d) {
    return bar(b.value * d);
  }
  friend bar operator-(const bar& b) {
    return bar(-b.value);
  }
  friend bool operator==(const bar& lhs, const bar& rhs) {
    return lhs.value == rhs.value;
  }
};
```

Technically speaking, the only extra operator that should be required is `bar operator-(const bar&, const bar&)`. Unfortunately, as noted in the example, a few more operator definitions may be necessary if the compiler and linker that one is working with get confused. Fortunately, the error messages indicating the need for a particular operator are typically not too difficult to decipher, and the operators are fairly straight-forward to define.

Currently FEL contains `typedef` statements for fields with the following node types:

- `float`
- `double`
- `FEL_vector2f`
- `FEL_vector3f`
- `FEL_vector3d`
- `FEL_matrix33f`
- `FEL_plot3d_density_momentum`
- `FEL_plot3d_q`

The last two types are aggregates defined for the solution vector data standard to PLOT3D [WBPE92]. The "`q`" type contains the entire PLOT3D solution, the "`density_momentum`" field contains the subset of the variables necessary for velocity-related derived fields. The math operators for both types simply define the same operators in turn for each component in the object. So, for instance, operator `+` with two `q` objects returns a new object where the density, momentum, and energy components are each the sum of the corresponding components in the two arguments.

Chapter 22

File I/O

To construct instances of most mesh and field classes in FEL, one provides, as arguments, parameters and pointers to buffers containing data. Typical parameters include specification of structured mesh dimensions, and data buffers typically include vertex coordinates or solution data. For most scientists, mesh data and solution data are stored in files of various formats. FEL provides file reader functions that extract the data from files and, where appropriate, that load the data into main memory with a layout appropriate for the class to be constructed. Using the reader functions, one can simply specify the file name and optional flags describing the file format, and the appropriate objects will be constructed.

FEL has families of file reader functions for several major file formats. The most extensive support in FEL is for the PLOT3D [WBPE92] format. A second family of readers accepts paged PLOT3D files (see Chapter 23). Paged PLOT3D files contain the data of standard PLOT3D files, but the data are reorganized into page-sized chunks that can be read in on demand. FEL contains less extensive support for two more file formats: FITS [FIT] and Vis5D [Vis]. Finally, the library provides a generic set of reading functions that are independent of a particular file format. The idea is that an application written in terms of the generic routines can work with a variety of file formats. The generic routines currently support work with PLOT3D and paged PLOT3D files, with limited support for the Vis5D format.

File readers are built using standard FEL operations. Thus, it is possible to write a reader for a new file type without having to modify FEL.

22.1 PLOT3D file reader functions

The visualization application PLOT3D [WBPE92] defines a family of file formats for storing meshes and fields. The FEL file reader functions handle files designated as “3D (/WHOLE)” by PLOT3D. The PLOT3D “1D”, “2D”, and “3D (/PLANES)” formats are not supported. This section contains descriptions for the three PLOT3D-related file reader families. Most of this section describes the family of generic file readers, with notes describing how the other two families differ. A later subsection shows how the

functions correspond among the three families.

Most applications should use the generic functions so that they work with both types of files. While the generic functions have some amount of overhead over the specific functions, the overhead is quite minimal and is outweighed by allowing your program to work with multiple file formats. Future versions of FEL may enhance the generic file reader functions which would let your application read those files with little or no effort.

22.1.1 The PLOT3D flags

The FEL PLOT3D flags allow the user to specify the type of a file. They specify the type of the file (PLOT3D or paged file) as well as the particular variation of the PLOT3D file format. The generic file readers use the file type flags (either `FEL_PLOT3D_3D_WHOLE` or `FEL_PAGED_PLOT3D_3D_WHOLE`) to determine the file-type-specific routine that should be called to do the actual work. The variation flags tell the standard PLOT3D routines which PLOT3D variation the file contains. The paged PLOT3D routines ignore the variation flags because the paged files are self-identifying. If you call a standard-PLOT3D- or paged-PLOT3D-specific reader routine directly, the flags argument must correspond to the file type that the routine handles.

Several flags can be expressed simultaneously using the C "`|`" bitwise-or operator. For example, "`FEL_PLOT3D_3D_WHOLE | FEL_PLOT3D_MULTIZONE | FEL_PLOT3D_IBLANK`" specifies a PLOT3D multi-zone file which includes IBLANK information.

The PLOT3D readers recognize the following flags:

`FEL_PLOT3D_3D_WHOLE` for all PLOT3D (non-paged) files

`FEL_PAGED_PLOT3D_3D_WHOLE` for PLOT3D paged files

`FEL_PLOT3D_IBLANK` file contains IBLANK information

`FEL_PLOT3D_MULTIZONE` file has multiple zones

`FEL_PLOT3D_UNSTRUCTURED` file contains an unstructured (tetrahedral) mesh

`FEL_PLOT3D_FORTRAN_UNFORMATTED` binary with FORTRAN control words

`FEL_PLOT3D_FORTRAN_FORMATTED` PLOT3D ASCII format

`FEL_PLOT3D_FUNCTION` file is a function file

`FEL_PLOT3D_LITTLE_ENDIAN` bytes are in little endian order

22.1.2 Automatic mesh type deduction

FEL provides an automatic format deducing function, `FEL_deduce_mesh_type()`, which makes it easier for the user to read meshes without having to remember the particular variation of the file format at hand. The deducer will determine the type of the file (standard PLOT3D or paged), and the file's PLOT3D variation. The deducer

allows one to pretend that all PLOT3D files are self-describing. The deducer takes a character string file name and returns an unsigned integer representing the flags, or 0 if it could not successfully deduce the mesh type. The deducer essentially works by hypothesizing that the file is a particular format and examining the file to see whether the file is consistent with that format. With paged PLOT3D files, the deducer just examines the first few bytes of the file to see if the file has a valid paged PLOT3D file header.

Deducing a standard PLOT3D file takes more work. The PLOT3D deducer must hypothesize a PLOT3D file variation, interpret the first words in the file as if in the hypothetical variation, and then compute the size of the file in bytes implied by the header words. If the number of bytes implied matches the actual number of bytes in the file, then the deducer returns the specific variation. Otherwise, the deducer continues to the next variation hypothesis. The technique used by the deducer is not fool-proof. For example, if for some reason the file has extra bytes appended to it, then the implied count will not match the actual, and the deducer will fail.

22.1.3 Reading mesh files

The function `FEL_read_mesh()` is the function for reading meshes. The function takes one required argument, the character string mesh file name, and returns an FEL_mesh subclass. If the reader encounters an error, then it returns `NULL`. Immediately following the file name argument is an optional argument with the format flags for the given file. By default the reader calls the deducer function to compute the flags. The third argument allows reading a specific zone from a multi-zone data set. The default value of `-1` specifies that every zone should be read in; other values specify a particular zone to be read. The function's declaration is:

```
FEL_mesh_ptr FEL_read_mesh(char*, unsigned = 0,
                           int = -1);
```

22.1.4 Getting information about structured mesh files

Three functions read part of a structured mesh file and return information about it. These functions abort if they are called on an unstructured mesh file. All three functions return a status code of `FEL_OK` on success and `FEL_FAILED` on failure. The first argument for the functions specifies the name of the file. The second argument specifies the type of the file and must be non-zero (use the deducer if you don't know the type).

The same information can be retrieved from a mesh once you have read it, using `get_n_zones` and `get_structured_dimensions`. If you do not need the information before reading the mesh, it is more efficient to read the entire mesh and then retrieve the information from it.

The functions are:

```
int FEL_read_mesh_dimensions(char*, unsigned, int*,
  FEL_vector3i**);
int FEL_read_mesh_n_zones(char*, unsigned, int*);
```

```
int FEL_read_mesh_zone_dimensions(char*, unsigned, int,
  FEL_vector3i*);
```

The first function, `FEL_read_mesh_dimensions`, returns the number of zones and the dimensions of each zone. The dimensions are returned in a dynamically-allocated array. This array must be deallocated using `delete[]` after you have finished using it. The array is returned by setting the `FEL_vector3i*` that is pointed to by the third argument.

The second function, `FEL_read_mesh_n_zones`, returns the number of zones by setting the integer pointed to by the third argument. The last function, `FEL_read_mesh_zone_dimensions`, returns the dimensions of the zone specified by the third argument by modifying the `FEL_vector3i` pointed to by the last argument. These last two functions call `FEL_read_mesh_dimensions` and return part of what it returns, so it is more efficient to call that function if you need information about all the zones.

22.1.5 Reading solution files

FEL provides a set of solution reader functions, all with the prefix `FEL_read_`, e.g., `FEL_read_density`. Each solution reader function take, as a first argument, the mesh upon which the solution is based and a character string file name as the second argument. The third required argument is the set of flags specifying the specific file format. The flags are the same as those used for the mesh reading functions, though some flags (specifically `FEL_PLOT3D_IBLANK` and `FEL_PLOT3D_UNSTRUCTURED`) are irrelevant to solution files and are ignored. If the flags include `FEL_PAGED_PLOT3D_3D_WHOLE`, i.e., if one will be reading from a paged file, then the remaining flags are not necessary, as paged files are self describing. The data in the paged file overrides options spelled out by the flags. Immediately following the three required arguments to the solution reader routines there is an optional argument specifying a specific zone to read. By default the data for all the zones of a multi-zone file are read.

Unlike the mesh case, the library does not provide a format deducing function for solution files. Typically, one can use the flags returned by `FEL_deduce_mesh_type` as an argument to both the mesh and solution reading functions, since both are usually the same PLOT3D variation, e.g., “FORTRAN unformatted”. Thus the lack of a solution format deducer is typically not an inconvenience. In the cases where the mesh and solution are not in the same format, one can still use the mesh deducer result for reading the mesh, but the flags for the solution reader must be determined by the user and provided manually.

The typical usage of the deducer and reader functions is:

```
unsigned flags;
FEL_mesh_ptr mesh;
FEL_float_field_ptr density_field;

flags = FEL_deduce_mesh_type(mesh_file_name);
```

```
mesh = FEL_read_mesh(mesh_file_name, flags);
density_field = FEL_read_density(mesh, soln_file_name, flags);
```

Code written for actual use should check that the return values for each of the calls is not `0` or `NULL`. The following solution reader functions are provided:

```
FEL_float_field_ptr FEL_read_density(FEL_mesh_ptr,
  const char*, unsigned, int = -1);
FEL_float_field_ptr FEL_read_momentum_x(FEL_mesh_ptr,
  const char*, unsigned, int = -1);
FEL_float_field_ptr FEL_read_momentum_y(FEL_mesh_ptr,
  const char*, unsigned, int = -1);
FEL_float_field_ptr FEL_read_momentum_z(FEL_mesh_ptr,
  const char*, unsigned, int = -1);
FEL_vector3f_field_ptr FEL_read_momentum(FEL_mesh_ptr,
  const char*, unsigned, int = -1);
FEL_float_field_ptr FEL_read_energy(FEL_mesh_ptr,
  const char*, unsigned, int = -1);
FEL_plot3d_q_field_ptr FEL_read_q(FEL_mesh_ptr,
  const char*, unsigned, int = -1);
FEL_vector3f_field_ptr FEL_read_velocity(FEL_mesh_ptr,
  const char*, unsigned, int = -1);
FEL_plot3d_density_momentum_field_ptr
  FEL_read_density_momentum(FEL_mesh_ptr, const char*,
                            unsigned, int = -1);
```

22.1.6 Reading function files

FEL provides two routines for reading PLOT3D "function" files: `FEL_read_float_function` and `FEL_read_vector3f_function`. The first two arguments to the routines, the mesh and file name, are the same as for the solution readers. The last two arguments are also the same: the second to last argument specifies the file format flags; the final argument, which is optional, specifies a particular zone to read. The function file reader routines have an extra, third argument, specifying how many scalar variables to skip over in order to get to the desired values. For example, given a function file having a zone containing a vector (3 floats) followed by a scalar, one would use a value of 3 for the third argument in order to read the scalar variable. The paging code does not currently support function files, so the paged versions of these functions do not exist. The functions are:

```
FEL_float_field_ptr FEL_read_float_function(FEL_mesh_ptr,
  char*, int, unsigned, int = -1);
FEL_vector3f_field_ptr FEL_read_vector3f_function(FEL_mesh_ptr,
  char*, int, unsigned, int = -1);
```

22.1.7 Reading individual zones from multi-zone files

On some occasions one may desire to read the data for individual mesh or solution zones from multi-zone files. For example, when working with very large data sets, one may not be able to load the whole data set into memory at once, so processing zones one at a time may be the only option. All the PLOT3D mesh, solution, and function file readers support an optional final argument, allowing the user to specify an individual zone to read.

In order to make it easier to write code that works with both single and multi-zone meshes, the reading routines allow one to provide a zone argument, even when the mesh is single-zone. In the single-zone case, the argument value must be 0. This allowance makes it possible to write code such as:

```
unsigned flags = FEL_deduce_mesh_type(mesh_file);
int n_zones;
FEL_read_mesh_n_zones(mesh_file, flags, &n_zones);

for (int zone = 0; zone < n_zones; zone++) {
  FEL_mesh_ptr m = FEL_read_mesh(mesh_file, flags, zone);
  FEL_float_field_ptr f =
    FEL_read_energy(m, soln_file, flags, zone);

  // ... do processing with field f
}
```

The excerpt above will work with both single- and multi-zone files. Of course, code for actual use should check the return values from the reader functions to ensure success.

22.1.8 Making the PLOT3D readers more verbose

In general, the PLOT3D reader functions do not write any output to the terminal. In some cases it may be handy to request more output, for example, when trying to determine why the reader cannot successfully read a particular file. The PLOT3D reader routines all check a public global variable: `FEL_reader_verbose`, which can be set to `true` by the user. When this flag is true the readers provide more information about what they are doing.

22.2 PLOT3D and paged file readers

The readers described so far are generic file readers. FEL also provides reading functions that operate only on a single file type (standard or paged PLOT3D files). They are mainly provided for backward compatibility with older programs. Most programs should use the generic functions. Table 22.1 shows the correspondences between the three PLOT3D reader families.

Generic Function	PLOT3D Function	Paged PLOT3D Function
FEL_deduce_mesh_type	FEL_plot3d_deduce_mesh_type	FEL_plot3d_deduce_paged_mesh_type
FEL_read_mesh	FEL_plot3d_read_mesh	FEL_plot3d_read_paged_mesh
FEL_read_mesh_dimensions	FEL_plot3d_read_mesh_dimensions	FEL_plot3d_read_paged_mesh_dimensions
FEL_read_mesh_zone_dimensions	(use generic function)	(use generic function)
FEL_read_mesh_n_zones	(use generic function)	(use generic function)
FEL_read_density	FEL_plot3d_read_density	FEL_plot3d_read_paged_density
FEL_read_momentum_x	FEL_plot3d_read_momentum_x	FEL_plot3d_read_paged_momentum_x
FEL_read_momentum_y	FEL_plot3d_read_momentum_y	FEL_plot3d_read_paged_momentum_y
FEL_read_momentum_z	FEL_plot3d_read_momentum_z	FEL_plot3d_read_paged_momentum_z
FEL_read_momentum	FEL_plot3d_read_momentum	FEL_plot3d_read_paged_momentum
FEL_read_energy	FEL_plot3d_read_energy	FEL_plot3d_read_paged_energy
FEL_read_q	FEL_plot3d_read_q	FEL_plot3d_read_paged_q
FEL_read_velocity	FEL_plot3d_read_velocity	FEL_plot3d_read_paged_velocity
FEL_read_density_momentum	FEL_plot3d_read_density_momentum	FEL_plot3d_read_paged_density_momentum
FEL_read_float_function	FEL_plot3d_read_float_function	(not implemented)
FEL_read_vector3f_function	FEL_plot3d_read_vector3f_function	(not implemented)

Table 22.1: PLOT3D file reader correspondences.

22.3 The FITS file reader

FEL provides a simple reader for FITS files. The reader handles data for 2- or 3-dimensional regular meshes, with a single float value at each node. The reader takes as its first argument the character string specifying the file name. The next two arguments each are a pointer to an integer. FITS file headers consist of a sequence of tags, each with a corresponding value. Some users have extended their use of the FITS format by defining new, non-standard tags. In particular, the tags "XPOS" or "YPOS", each accompanied by an integer, are used by McDonnell-Douglas to signify the lower corner of a subimage (where images are represented by a 2-D mesh). The FEL FITS file reader function recognizes the "XPOS" and "YPOS" tags, returning the corresponding values in the integers pointed to by the second and third arguments of the reader call. By default "XPOS" and "YPOS" are set to 0.

In general, the FITS reader function does not write any output to the terminal. In some cases, it may be handy to request more output, for example, when trying to determine why the reader cannot successfully read a particular file. The FITS reader routine checks a public global variable, `FEL_fits_reader_verbose`, which can be set to `true` by the user. When this flag is true, the reader provides more information about what it is reading, including displaying the tags that it encounters in the file.

22.4 The Vis5D file reader

FEL provides a simple reader for files in the VIS5D format. There are two routines available to the user: `FEL_vis5d_get_scalar` and `FEL_vis5d_get_vector`. The scalar routine returns a float field, the vector routine returns a `FEL_vector3f` field. Both routines take a mesh as a first argument and a character string file name as a second argument. The scalar field reader takes a third argument: a character string providing the name of the variable to read. Finally, both routines take an integer final argument specifying the time step to retrieve when working with time series data.

Chapter 23

Paged Meshes and Fields

23.1 Introduction

Paged meshes and fields are similar to standard FEL meshes and core fields, but they use much less memory. Instead of keeping all of the data in memory, they bring in the portions that are used. Data are placed into a pool of memory; the size of the pool can be specified by the user. When the pool is full, some data that have not been recently used are replaced with the new data.

A paged mesh or field can be much faster than the corresponding in-memory version if your system does not have enough main memory to hold all of the data. While you can rely on the operating system's virtual memory to bring in the data when you are using in-core meshes and fields, the paged versions use memory more efficiently and may be able to keep what is currently needed in memory while the operating system cannot. A program will run much faster if its data can be kept in memory. Even if you have enough memory, a paged mesh or field can be faster if you only need to use a fraction of the data since only that fraction is read from disk. However, a paged mesh or field can be slower if you have sufficient memory and repeatedly use all or nearly all of the data. This happens because the data are read in smaller amounts compared to when the entire file is read at once, and because, for the same amount of data, reading data in smaller amounts takes longer than reading it all at once. Also, accessing the data is a bit slower with the paged routines because they do some additional checks and calculations. More information about the advantages and disadvantages of paged files and out-of-core visualization in general can be found in [CE97].

From a coding standpoint, the use of a paged mesh or field is very similar to using the corresponding in-core objects. The same operations are supported, except that the names of some setup calls are different.

As of this writing, only PLOT3D structured grid and solution files can be paged. Support for function files and unstructured grids may be available in a future release. Also, the current paging code is not thread-safe, so paged files should not be used in multithreaded programs. This should be corrected in the next release.

23.2 How paged files work

The technique that paged files use to bring data from disk as it is used is similar to how virtual memory is implemented in computer systems, but using software instead of hardware. Like virtual memory systems, the paging code reads fixed-sized blocks of data, called *pages* of data.

One reason that paged meshes and fields use memory more efficiently than the operating system's virtual memory system is that they can select a different page size. The paging code uses pages of 2 kilobytes, while current workstations use page sizes of 4, 8, or 16 kilobytes. Smaller pages use memory more efficiently than larger pages because there are smaller amounts of unreferenced data surrounding the referenced data that must still be read into memory.

The second reason for paged meshes' and fields' more efficient memory usage is that they reorganize the data in the file, which places different data on each page. For structured grids, an 8x8x8 cube of data from a single zone is placed in a page. The original PLOT3D files would place a plane of data on a page. Again, the reason that this file organization uses memory more efficiently is that placing cubes of data on each page has smaller amounts of unreferenced data surrounding the referenced data. For most 3D traversals of the data, placing 8x8x8 cubes around the referenced data covers a smaller portion of the overall data compared to placing (for example) 32x64 planes of data around the referenced data. In practice, this reorganization cuts the memory usage in half compared to PLOT3D when computing streamlines. If memory is tight, using half the memory can translate into larger factors of speed improvement. One drawback from reorganizing the data is that the files must be converted to a new format. The next section describes how to convert files.

When you start to use a file, it is opened, and data structures are set up that have an entry for each page in the file; each entry indicates that the page has not been loaded into memory. When you access some data (e.g., by using an `at_cell` call), the pages where the data are found are computed and checked to see if they are present. If they are not present (which would be true if the file was just opened), the pages are read. Then, the values are retrieved from the pages and returned.

Pages read from disk are placed in memory allocated from a pool of pages. There is a single memory pool, which means that it is shared among all of the open paged files. If all pages are in use when a new page is needed, a page that has not been recently used is reused. The size of the pool is user-configurable (see Section 23.5 below). Preliminary estimates indicate that you can visualize a dataset interactively when the pool will hold 20% of the dataset's grid files and 5% of the solution files. These numbers depend on how the data are used and thus will vary considerably.

There is some memory overhead associated with paged files. When you start using a file, the paging code creates data structures that use memory equal to about 0.3% of the file size. This can be significant: a 10 gigabyte dataset would use about 30 megabytes of memory. This memory is allocated separately from the memory pool used to hold file data.

23.3 Converting PLOT3D files to paged files

As mentioned above, you need to convert your PLOT3D files to the paged file format. The `p3dtopaged` program will convert one file to the new format. This program can be found in the FEL source directory.

For most files, the program can deduce the type of input file. In that case, the usage is:

```
p3dtopaged input-file output-file
```

The program cannot figure out some file types, especially FORTRAN unformatted files with more than 5 solution variables. With these files the type must be specified on the command line using flags. The flags are:

`-g` indicates that the input is a grid file with no IBLANKS

`-i` indicates that the input is a grid file with IBLANKS

`-s` indicates that the input is a solution file

`-1` (a one) indicates that the input file has one zone

`-m` indicates that the input file has multiple zones

`-b` indicates that the input is a binary file

`-f` indicates that the input is a FORTRAN unformatted file

If the program cannot deduce the type of the file, you will have to specify three of the above options: one from `g`, `i`, or `s`; `1` or `m`; and `b` or `f`. For example, `p3dtopaged -imb` would specify that the input file is a multi-zone, binary, PLOT3D grid file that includes IBLANKS.

Paged files can be converted to the PLOT3D format with the `pagedtop3d` program. It only outputs binary files; FORTRAN unformatted is not supported. If your original input file was a FORTRAN unformatted solution file with seven parameters per node, when you convert the paged file to a PLOT3D file only the first five parameters will be in the file. This happens because paged files contain only the first five parameters. If the original PLOT3D file was a binary file, converting the file to a paged file and then back to a PLOT3D file will give you an identical file.

23.4 Using paged meshes and fields

A program uses paged meshes and fields in nearly the same way as it uses in-core meshes and fields. The main difference is how these objects are created. In-core meshes and fields can be created by reading a file or by giving a block of data to a constructor. Paged meshes and fields should only be created by using functions similar to the ones used to read a file for an in-core mesh or field.

For example, the `FEL_plot3d_read_mesh` function will read a PLOT3D file into memory and return a `FEL_mesh_ptr`. The `FEL_plot3d_read_paged_mesh` function will open the paged-format file and initialize the paging data structures, and then return a `FEL_mesh_ptr`. Both functions take the same arguments. The only difference is that the `FEL_plot3d_read_paged_mesh` function ignores the flags argument. See Chapter 22 for more details about these functions.

Your program can also use file-reading functions that work for both PLOT3D files and paged files. Using these generic functions is encouraged since your program will then work with both in-core and paged meshes and fields. These functions will first look at the flags argument to determine the type of the file. If the flags argument specifies a file type (i.e., it is non-zero), that type is used. Otherwise, the functions look at the file itself to determine the file type. If the file is a PLOT3D file, these functions will read the file and return an in-core mesh or field; if it is a paged file, they will open the file and return a paged mesh or field. In general, if the in-core function is `FEL_plot3d_read_xxx`, the paged function will be named `FEL_plot3d_read_paged_xxx`, and the generic function will be named `FEL_read_xxx`. Table 22.1 shows the correspondences between the generic and paged-file functions.

Another minor difference between in-core and paged meshes and fields is how the `compute_bounding_box` and `compute_min_max` functions are implemented. In-core meshes and fields must compute the results at run time. The paged versions of these functions instead use values computed when the file was converted to a paged file and thus are much faster. Because of this, you should use the FEL functions to compute these values instead of writing your own.

23.5 Controlling memory usage

If you are concerned about the performance of your program, you will probably want to avoid having it use more memory than is installed on the system where it is running. You can control how the paging code uses memory by two methods. The first method lets you control the total amount of memory used. The second method allows you to specify which meshes or fields are more important and should be kept in memory in preference to other meshes or fields.

23.5.1 Pool size

You can control the amount of memory used by changing the size of the memory pool. The paging code uses this pool to hold pages from paged mesh and field files that are brought into memory. By default, the maximum pool size is 50% of the system's memory. On most Unix systems, the "system memory size" is the value returned from the `getrlimit` system call for the maximum resident set size. Many shells (sh, csh, etc.) allow this value to be changed with the `limit` command using the `memoryuse` option. For example, `limit memoryuse 100m` would set the overall maximum memory usage to 100 megabytes and the default paging pool size to 50 megabytes.

On Solaris systems, the "system memory size" is determined using a different call that returns the actual physical memory size.

The maximum size of the pool can also be changed with two procedure calls. The first, `FEL_set_paging_memory_megabytes`, sets the size of the pool directly. The second call, `FEL_set_paging_memory_fraction`, changes the fraction of the system's memory that is used; the fraction should range from 0 to 1. Both calls take a single `float` argument. The size of the pool can only be changed before opening the first paged file. Calls to these two functions after opening a file have no effect. The two calls are defined as follows:

```
extern void FEL_set_paging_memory_megabytes(float);
extern void FEL_set_paging_memory_fraction(float);
```

Memory will only be allocated to the pool as data are read from disk. This means that, if the maximum pool size is much larger than the files that are used, the pool will grow to at most the size of the files. (The pool can be somewhat larger than the total file sizes because partially-filled 8x8x8 cubes of data take up a full page when in memory but take only a partial page when on disk.)

One difference between the paged and standard file reader calls is that the same data are loaded in memory only once with the paged calls. For example, if you call `FEL_read_paged_plot3d_density` and `FEL_read_paged_plot3d_density_momentum` for the same file and use data from both, the density values will only be read once. But, if you use the non-paged calls, `FEL_read_plot3d_density` and `FEL_read_plot3d_density_momentum`, the density values will be read twice and stored twice.

23.5.2 Page priority hints

The second memory control method allows a program to control which file's pages are retained in the memory pool. When the memory pool is full and more data are needed from a file, a page that currently contains data is selected and reused. Each mesh or field has a priority that determines the length of time before its pages can be reused. Increasing or decreasing the priority gives the paging code a hint on which data are important or unimportant so that pages containing less important data can be reused first. These hints have not yet been used in an application, so they should be regarded as experimental.

For example, priority hints could be used with an unsteady dataset in an interactive application that has a concept of the current simulation time. In such an application, only the two timesteps that bracket the current simulation time are used. If all the timesteps but the current ones are set to low priority, then they will always be the ones reused.

The following priorities can be used:

- `FEL_LOW_PAGE_PRIORITY` allows the mesh's or field's pages to be deallocated and reused immediately.

- FEL_STANDARD_PAGE_PRIORITY allows the mesh's or field's pages to be deallocated after they have not been used for a while. This is the default priority.
- FEL_HIGH_PAGE_PRIORITY allows the mesh or field's pages to be deallocated after they have not been used for a longer time than standard-priority pages.

Priorities are set by using the `set` member function with one of the following keywords:

- FEL_FIELD_PAGE_PRIORITY changes the priority for both the field (solution) data and the mesh data when invoked on a field, and changes the priority of the mesh data when invoked on a mesh.
- FEL_MESH_PAGE_PRIORITY changes the priority of the mesh (if invoked on a field, changes priority of the field's mesh).
- FEL_SOLUTION_PAGE_PRIORITY changes the priority for the field data when invoked on a field, and has no effect when invoked on a mesh.

The current interface allows you to set the priority for an entire field, an entire mesh, or both, or for an individual zone in a mesh. You cannot set the priority for individual zones of the data for a multi-zone field. Some examples are below:

```
void priority_example()
{
  // code assumes "grid" and "solution" files are
  // multi-zoned
  unsigned int flags = FEL_deduce_mesh_type("mesh");
  FEL_mesh_ptr mesh = FEL_read_mesh("mesh", flags);
  FEL_float_field_ptr field = FEL_read_density(mesh,
    "solution", flags);

  // change priority of mesh for all zones
  mesh->set(FEL_MESH_PAGE_PRIORITY, FEL_LOW_PAGE_PRIORITY);
  // change priority of mesh for zone 1
  mesh->get_zone(1)->set(FEL_MESH_PAGE_PRIORITY,
                         FEL_LOW_PAGE_PRIORITY);

  // change priority of mesh for all zones
  field->set(FEL_MESH_PAGE_PRIORITY, FEL_LOW_PAGE_PRIORITY);
  // change priority of mesh for zone 1
  field->get_mesh()->get_zone(1)->set(FEL_MESH_PAGE_PRIORITY,
                                      FEL_LOW_PAGE_PRIORITY);
  // change priority of field data for all zones
  field->set(FEL_SOLUTION_PAGE_PRIORITY, FEL_LOW_PAGE_PRIORITY);
  // change priority of mesh and field data for all zones
  field->set(FEL_FIELD_PAGE_PRIORITY, FEL_LOW_PAGE_PRIORITY);
}
```

Chapter 24

The PLOT3D Field Manager

The PLOT3D field manager is a class hierarchy which can help create and manage fields based on PLOT3D data files. Once you create a field manager object, you can ask it to create any of over fifty predefined derived fields by name, and it will take care of any necessary file reading and derived field construction, returning a field ready for use. A function which receives a pointer to a field manager object essentially has been sent an entire collection of fields.

24.1 Constructing an `FEL_plot3d_field`

The base class of the field manager hierarchy is `FEL_plot3d_field`. This class is an abstract class. Three derived classes create instances of `FEL_plot3d_field` for three types of fields: steady fields, time-varying fields, and PLOT3D Q fields. Most applications should assign instances of the derived classes to a `FEL_plot3d_field_ptr` so that their code is independent of the actual field manager type.

24.1.1 Constructing an `FEL_steady_plot3d_field`

Two constructors are available for constructing an `FEL_steady_plot3d_field`:

```
FEL_steady_plot3d_field(char* mesh_file, char* soln_file,
                        unsigned flags = 0);
FEL_steady_plot3d_field(FEL_mesh_ptr mesh, char* soln_file,
                        unsigned flags = 0);
```

In the first constructor, you supply the names of your PLOT3D mesh file (`mesh_file`) and solution file (`soln_file`), and, optionally, flags describing your file formats. In the second constructor, you supply a pointer to a preexisting mesh (`mesh`) instead of the name of a mesh file.

24.1.2 Constructing a time-varying field manager

Time-varying field manager objects create time-varying fields in response to requests for derived fields. Since manager objects create time-varying fields, the arguments to the two constructors for a time-varying field manager are nearly the same as the constructors for time-varying fields. Chapter 25 describes time-varying fields and the common arguments.

Three of the field manager constructor's arguments are different from the time-varying field's constructor. The first argument can be a string specifying the mesh file instead of a mesh object. The third argument, the callback function, is different in that it does not return a field object. Instead, it fills in the name of the solution file and returns a value indicating success or failure. Finally, the field manager has a seventh argument giving the file flags for the solution files.

The definitions for the constructors are:

```
const int FEL_plot3d_filename_len = 1024;
typedef int (*FEL_plot3d_filename_callback)
                 (int, void*, char[FEL_plot3d_filename_len]);

FEL_fixed_interval_time_series_plot3d_field(
  char* mesh_file,
  int num_time_steps,
  FEL_plot3d_filename_callback cb,
  void* user_data,
  float physical_time0,
  float physical_time1,
  unsigned flags = 0);

FEL_fixed_interval_time_series_plot3d_field(
  FEL_mesh_ptr mesh,
  int num_time_steps,
  FEL_plot3d_filename_callback cb,
  void* user_data,
  float physical_time0,
  float physical_time1,
  unsigned flags = 0);
```

You can specify time-varying meshes using the second constructor and passing a time-varying mesh as the first argument (see Chapter 26).

An example of a time-varying manager constructor is shown below. The example code reads a steady mesh file called `mesh` and 20 solution files named `soln00`, `soln01`, ..., `soln19`. The physical time values for the timesteps start at 0 and increase by 1 for each timestep. The code also has an example of how to use the `user-data` argument: the first four letters of the filename are passed to the callback function using that argument.

```
int soln_callback(int timestep, void* userdata,
  char filename[FEL_fixed_interval_time_series_plot3d_field::
  FEL_plot3d_filename_len])
{
  sprintf(filename, "%s%02d", (char*) userdata,
          timestep);
  return 1;
}

FEL_plot3d_field_ptr make_time_varying_manager()
{
  unsigned flags = FEL_deduce_mesh_type("mesh");
  FEL_plot3d_field_ptr fp = new
    FEL_fixed_interval_time_series_plot3d_field("mesh", 20,
    soln_callback, "soln", 0, 1, flags);
  return fp;
}
```

24.1.3 Constructing an `FEL_plot3d_q_field`

The third type of field manager constructor allows using fields that are not handled by the first two cases, such as transformed fields or multizoned fields with mixed steady and unsteady zones. This constructor allows an application to create an arbitrary PLOT3D Q field and then to use the field manager to create derived fields from it. The constructor simply takes the Q field as its only argument:

```
FEL_q_plot3d_field(FEL_plot3d_q_field_ptr q_field);
```

This class name can unfortunately be easily confused with another type name, `FEL_plot3d_q_field`. The latter type is a synonym for `FEL_typed_field<FEL_plot3d_q>`.

24.2 Creating primitive and derived PLOT3D fields

Once you have a properly initialized `FEL_plot3d_field`, creating the PLOT3D primitive and derived fields is straightforward. Access to the primitive PLOT3D fields is provided by five methods:

```
FEL_float_field_ptr get_density_field();
FEL_vector3f_field_ptr get_momentum_field();
FEL_float_field_ptr get_energy_field();

FEL_plot3d_density_momentum_field_ptr
  get_density_momentum_field();
FEL_plot3d_q_field_ptr get_q_field();
```

The first three methods return fields representing the momentum variable and two thermodynamic state variables which make up the PLOT3D solution output. The latter two methods return "conglomerate" fields, the last one containing the entire PLOT3D solution vector.

PLOT3D derived fields are obtained via two functions:

```
FEL_float_field_ptr make_float_field(FEL_float_field_enum);
FEL_vector3f_field_ptr
  make_vector3f_field(FEL_vector3f_field_enum);
```

These functions take a single argument indicating the desired field and they return a pointer to a field of the appropriate type. The `enums` representing the supported fields are shown in Table 24.1. The identity of the fields produced by the various `enums` should be evident from their names; precise definitions can be found in [WBPE92] or in the FEL code itself (in the file `FEL_plot3d_map_functions.C`). The `enums` correspond exactly to the original PLOT3D "function numbers" ([WBPE92]), so you can use those directly if you prefer. If you need a derived field which isn't predefined, you can always create one using some combination of the primitive and predefined fields — see Chapter 19.

```
FEL_density = 100                                  FEL_normalized_density = 101
FEL_stagnation_density = 102                       FEL_normalized_stagnation_density = 103
FEL_pressure = 110                                 FEL_normalized_pressure = 111
FEL_stagnation_pressure = 112                      FEL_normalized_stagnation_pressure = 113
FEL_pressure_coefficient = 114                     FEL_stagnation_pressure_coefficient = 115
FEL_pitot_pressure = 116                           FEL_pitot_pressure_ratio = 117
FEL_dynamic_pressure = 118                         FEL_temperature = 120
FEL_normalized_temperature = 121                   FEL_stagnation_temperature = 122
FEL_normalized_stagnation_temperature = 123        FEL_enthalpy = 130
FEL_normalized_enthalpy = 131                      FEL_stagnation_enthalpy = 132
FEL_normalized_stagnation enthalpy = 133           FEL_internal_energy = 140
FEL_normalized_internal energy = 141               FEL_stagnation_energy = 142
FEL_normalized_stagnation energy = 143             FEL_kinetic_energy = 144
FEL_normalized_kinetic_energy = 145                FEL_u_velocity = 150
FEL_v_velocity = 151                               FEL_w_velocity = 152
FEL_velocity_magnitude = 153                       FEL_mach_number = 154
FEL_speed_of_sound = 155                           FEL_cross flow velocity = 156
FEL_divergence_of_velocity = 158                   FEL_x_momentum = 160
FEL_y_momentum = 161                               FEL_z_momentum = 162
FEL_energy = 163                                   FEL_entropy = 170
FEL_entropy_s1 = 171                               FEL_x_component_of_vorticity = 180
FEL_y_component_of_vorticity = 181                 FEL_z_component_of_vorticity = 182
FEL_vorticity_magnitude = 183                      FEL_swirl = 184
FEL_velocity_cross_vorticity_magnitude = 185       FEL_helicity = 186
FEL_pressure_gradient_magnitude = 192              FEL_density_gradient_magnitude = 193
FEL_shock = 400                                    FEL_filtered_shock = 401
```

```
FEL_velocity = 200                                 FEL_vorticity = 201
FEL_momentum = 202                                 FEL_perturbation_velocity = 203
FEL_velocity_cross_vorticity = 204                 FEL_pressure_gradient = 210
FEL_density_gradient = 211
```

Table 24.1: Scalar (above the line) and vector (below the line) derived fields predefined in PLOT3D [WBPE92] and supported by `FEL_plot3d_field`. The FEL `enums` are given, along with the PLOT3D "function number".

All the derived fields created by the `FEL_plot3d_field` are of type `FEL_map_then_interpolate_derived_field*()`. Any required differential operator fields are second-order if the `FEL_plot3d_field`'s mesh is structured, and first-order if the mesh is unstructured. Future versions of the `FEL_plot3d_field` manager may permit the client to specify the types of derived fields and differential operator fields to be used in the construction of an individual field.

24.3 How the field manager works

The PLOT3D field manager works by reading core or paged fields and returning derived fields from these underlying fields. When it creates a core or paged field, it saves a pointer to it and when possible reuses the field for later derived fields. For example, if you call `make_float_field(FEL_pressure)` and `make_float_field(FEL_temperature)`, they will share the underlying Q field.

This sharing of multiple derived fields only happens when the the same field manager object is used to create all of the derived fields. An application that creates a field manager object, creates a derived field, destroys the field manager, and then creates a new field manager and a second derived field will read the solution twice.

Memory usage with the field manager is not always optimal when the field manager is using PLOT3D solution files. When a field manager is given a PLOT3D solution file, in most cases it reads the entire file and creates a Q field when the first derived field is requested. This is not efficient if only part of the solution field is used, which would happen if `make_float_field(FEL_density)` is called. However, reading all of the data allows multiple derived fields to share the solution data when the derived fields need different but overlapping portions of the solution file.

The primitive PLOT3D field functions (the `get_*_field` functions) for steady and time-varying field managers operate differently. Each reads a portion of the solution the first time it is called. If all file primitive field functions are called, the density and energy solution data will be read twice, and the momentum data will be read three times.

If you first create a primitive PLOT3D field and then create a derived field, the field manager may use the primitive field instead of first creating a Q field. For example, if `get_density_momentum_field` and then `make_vector3f_field(FEL_velocity)` are called, the velocity field will be derived from the density-momentum field created in the first call. If the density-momentum field had not existed, the velocity field would be derived from a Q field. This behavior depends on which primitive PLOT3D fields are needed for each derived field. See the source file `FEL_plot3d_field.C` for details.

Memory usage with paged files is not an issue. The paging code insures that the data needed is read and stored only once, even if it is used for multiple primitive PLOT3D and derived fields. Memory usage is also not an issue when the field manager is given a PLOT3D Q field. In this case, the primitive and derived field functions return fields derived from the Q field handed to the field manager constructor.

24.4 Miscellaneous FEL_plot3d_field methods

FEL_plot3d_field provides a get_mesh() and a generic set() method as a convenience. This code:

```
FEL_plot3d_field_ptr plot3d_manager;
...
FEL_mesh_ptr mesh = plot3d_manager->get_mesh();
```

returns the mesh associated with plot3d_manager. This mesh is associated with any of the fields created by plot3d_manager.

```
FEL_plot3d_field::
  set(const FEL_set_keyword_enum keyword, int value);
```

passes the requested options to the mesh and primitive fields of the FEL_plot3d_field object. Note that since all fields created by the FEL_plot3d_field share the same mesh and primitive fields, the set() call on a FEL_plot3d_field object affects all fields which *have been* created, and even fields which *will be* created, by the FEL_plot3d_field. Furthermore, set() calls on any field created by the FEL_plot3d_field will have the same wide-reaching effect. At present, this interdependence of a group of fields sharing a common mesh is a limitation of FEL which will be resolved in a future release. In the meantime, be aware of potentially unwanted side effects of the set() calls.

24.5 An example

```
#include "FEL.h"

int main(int argc, char** argv)
{
 // mesh file, solution file in argv[1] and argv[2]
 FEL_plot3d_field_ptr plot3d_manager =
  new FEL_steady_plot3d_field(argv[1], argv[2]);

 FEL_float_field_ptr helicity_field =
  plot3d_manager->make_float_field(FEL_helicity);

 FEL_vector3f_field_ptr grad_pressure_field =
  plot3d_manager->make_vector3f_field(FEL_pressure_gradient);

 // grad_pressure_field is ready for at_calls, or whatever

 return 0;
}
```

This example first creates a FEL_steady_plot3d_field using the mesh and solution files specified in the first two arguments. It then creates a float field containing helicity and a vector field containing the gradient of pressure.

24.6 PLOT3D derived field "convenience functions"

The FEL_plot3d_field takes care of file reading and caching of primitive fields by maintaining internal state, but in response to an incoming FEL_float_field_enum or FEL_vector3f_field_enum the manager creates the requested derived field by calling one of a number of external "convenience functions". You can call these functions directly, whether or not you've created an FEL_plot3d_field. All of the functions take a pointer to an FEL_plot3d_q_field as their only required argument and, as usual, an optional character string name. A few of the functions have overloaded versions which accept pointers to fields comprising a subset of the PLOT3D solution vector. The functions return a pointer to either an FEL_float_field or an FEL_vector3f_field. For example:

```
FEL_plot3d_q_field_ptr q_field;
...
FEL_float_field_ptr sp_field =
  FEL_plot3d_make_stagnation_pressure_field(q_field);
```

Although this looks very similar to a derived field constructor invocation, this wrapper function calls any required constructors on your behalf, so you don't want a new after the "=".

Tables (24.2 and 24.3) present the available PLOT3D derived field convenience functions. These functions correspond one-to-one with the FEL enums shown in Table 24.1, modulo an obvious naming convention. The first column in the table lists the function. The second column lists the arguments accepted by the functions. Many functions are overloaded so multiple arguments types are accepted. Table 24.4 lists the abbreviations for the argument types. See the header file FEL_plot3d_derived_field.h for more details.

The default name of the created field is the name of the function called with the FEL_ and make_ deleted. For example, FEL_plot3d_make_density_field creates a field with a default name of FEL_plot3d_density_field You can override the default name by specifying the name as the second argument (the third argument for the FEL_plot3d_make_velocity_field function that accepts separate density and momentum fields).

24.7 PLOT3D derived field inventory arrays

An application may want to present *all* PLOT3D derived fields to a user as a selectable list. Since the fields are predefined and lazily evaluated, they can all be created ahead of time with minimal time and storage overhead and, when selected, they can be easily conjured forth without any need for a complicated input routine and interpreter or the

Function Name	Arguments
FEL_plot3d_make_density_field	Q
FEL_plot3d_make_normalized_density_field	Q, D
FEL_plot3d_make_stagnation_density_field	Q
FEL_plot3d_make_normalized_stagnation_density_field	Q
FEL_plot3d_make_pressure_field	Q
FEL_plot3d_make_normalized_pressure_field	Q
FEL_plot3d_make_stagnation_pressure_field	Q
FEL_plot3d_make_normalized_stagnation_pressure_field	Q
FEL_plot3d_make_pressure_coefficient_field	Q
FEL_plot3d_make_stagnation_pressure_coefficient_field	Q
FEL_plot3d_make_pitot_pressure_field	Q
FEL_plot3d_make_pitot_pressure_ratio_field	Q
FEL_plot3d_make_dynamic_pressure_field	Q, DM
FEL_plot3d_make_temperature_field	Q
FEL_plot3d_make_normalized_temperature_field	Q
FEL_plot3d_make_stagnation_temperature_field	Q
FEL_plot3d_make_normalized_stagnation_temperature_field	Q
FEL_plot3d_make_enthalpy_field	Q
FEL_plot3d_make_normalized_enthalpy_field	Q
FEL_plot3d_make_stagnation_enthalpy_field	Q
FEL_plot3d_make_normalized_stagnation_enthalpy_field	Q
FEL_plot3d_make_internal_energy_field	Q
FEL_plot3d_make_normalized_internal_energy_field	Q
FEL_plot3d_make_stagnation_energy_field	Q
FEL_plot3d_make_normalized_stagnation_energy_field	Q
FEL_plot3d_make_kinetic_energy_field	Q, DM
FEL_plot3d_make_normalized_kinetic_energy_field	Q, DM
FEL_plot3d_make_u_velocity_field	Q, DM
FEL_plot3d_make_v_velocity_field	Q, DM
FEL_plot3d_make_w_velocity_field	Q, DM
FEL_plot3d_make_velocity_magnitude_field	Q, DM
FEL_plot3d_make_mach_number_field	Q
FEL_plot3d_make_speed_of_sound_field	Q
FEL_plot3d_make_cross_flow_velocity_field	Q, DM
FEL_plot3d_make_divergence_of_velocity_field	Q, DM
FEL_plot3d_make_x_momentum_field	Q, M
FEL_plot3d_make_y_momentum_field	Q, M
FEL_plot3d_make_z_momentum_field	Q, M
FEL_plot3d_make_energy_field	Q
FEL_plot3d_make_entropy_field	Q
FEL_plot3d_make_entropy_s1_field	Q
FEL_plot3d_make_x_component_of_vorticity_field	Q, DM
FEL_plot3d_make_y_component_of_vorticity_field	Q, DM
FEL_plot3d_make_z_component_of_vorticity_field	Q, DM
FEL_plot3d_make_vorticity_magnitude_field	Q, DM
FEL_plot3d_make_swirl_field	Q, DM
FEL_plot3d_make_velocity_cross_vorticity_magnitude_field	Q, DM
FEL_plot3d_make_helicity_field	Q, DM, V
FEL_plot3d_make_pressure_gradient_magnitude_field	Q
FEL_plot3d_make_density_gradient_magnitude_field	Q, D

Table 24.2: Derived field convenience functions that return float fields.

Function Name	Arguments
FEL_plot3d_make_velocity_field	Q, DM, D+M
FEL_plot3d_make_vorticity_field	Q, DM
FEL_plot3d_make_momentum_field	Q
FEL_plot3d_make_perturbation_velocity_field	Q, DM
FEL_plot3d_make_velocity_cross_vorticity_field	Q, DM
FEL_plot3d_make_pressure_gradient_field	Q
FEL_plot3d_make_density_gradient_field	Q, D

Table 24.3: Derived field convenience functions that return vector fields.

Abbreviation	Argument Description
D	Density field
DM	Density-momentum field
D+M	Two separate arguments: a density field and a momentum field
M	Momentum field
Q	Q field
V	Velocity field

Table 24.4: Abbreviation key for the derived field function tables.

like. As an aid for this kind of usage, FEL provides two arrays containing all the FEL PLOT3D `enum`s — one with all the float field `enums` and one with all the vector field `enum`s. A sentinel ("0") marks the end of each array.

The "inventory arrays" are found in `FEL_plot3d_field.h`:

```
FEL_float_field_enum FEL_plot3d_float_fields[];
FEL_vector3f_field_enum FEL_plot3d_vector3f_fields[];
```

By traversing these arrays, one can sequentially construct all PLOT3D derived fields and, by calling `get_name()` on each field, one can accumulate a list of canonical field names to be presented to the user. If you are listing the names, the field manager should be created using a `FEL_plot3d_q_field` instead of actual files. If you use actual files, the names may vary from the canonical `plot3d_`*xxx*`_field` form. Here is a short demonstration showing how to list all the derived scalar fields using `FEL_plot3d_float_fields[]`:

```
#include "FEL.h"

char** get_derived_float_names(int* nsfields)
{
  // make dummy field manager using dummy mesh and field
  FEL_mesh_ptr mesh = new FEL_regular_mesh(4, 4, 4);
  FEL_plot3d_q_field_ptr q_field =
    new FEL_constant_field<FEL_plot3d_q>(mesh, FEL_plot3d_q());
  FEL_solution_globals sg;
  sg.set_alpha(0); sg.set_free_stream_mach(0);
  sg.set_time(0); sg.set_reynolds_number(0);
```

```
  q_field->set_solution_globals(sg);
  FEL_plot3d_field_ptr plot3d_field =
    new FEL_q_plot3d_field(q_field);
  assert(plot3d_field!=NULL);

  *nsfields = 0;
  // no sizeof on unspecified dim array
  while (FEL_plot3d_float_fields[(*nsfields)++] != 0);
  --(*nsfields);
  FEL_float_field_ptr* scalar_fields =
    new FEL_float_field_ptr[*nsfields];
  char** scalar_names = new char*[*nsfields];
  for (int i=0; i<*nsfields; ++i) {
    FEL_float_field_ptr f =
      plot3d_field->make_float_field
        (FEL_plot3d_float_fields[i]);
    assert(f!=NULL);
    const char* n = f->get_name();
    scalar_names[i] = strcpy(new char[strlen(n)+1], n);
    f=NULL;
  }

  // make names without "plot3d", "field", and "_"
  char t[256];
  char* s;
  for (i=0;i<*nsfields;++i) {
    t[0]='\0';
    for (s=scalar_names[i];(s=strtok(s,"_"))!=NULL;s=NULL)
      if ( strcmp(s,"plot3d") && strcmp(s,"field") ) {
        strcat(t,s);
        strcat(t," ");
      }
    t[strlen(t)-1] = '\0';
    // t can't be longer than original
    strcpy(scalar_names[i],t);
  }
  return scalar_names;
}

int main()
{
  int nsfields;
  char** scalar_names =
    get_derived_float_names(&nsfields);
  cout << nsfields <<
    " scalar fields ready and waiting in scalar_fields[]"
```

```
    << endl;
  for (int i=0;i<nsfields;++i)
    cout << scalar_names[i] << endl;
  return 0;
}
```

Chapter 25

Time-Varying Fields

Fields that vary with time, also known as *unsteady* fields, are often represented by a series of time steps, where each time step represents the field at some instant in time. The time steps may or may not be at fixed intervals during a simulation, though in many cases a fixed interval is used. FEL represents unsteady fields via the classes `FEL_time_series_field` and `FEL_fixed_interval_time_series_field`. The interface for time-series fields inherits from the standard field interface defined in `FEL_field` and `FEL_typed_field`. Just as with steady fields, one can make "`at`" calls to request data using an `FEL_phys_pos`, `FEL_cell`, `FEL_vertex_cell` or `FEL_structured_pos` argument. All the argument types contain an `FEL_time` data member. For steady data, the time is ignored; for unsteady data, it is essential that time be specified. Given an argument containing a time value, time-series fields produce field values by accessing data from the appropriate time step. If necessary, time-series fields can also do temporal interpolation (described later in this chapter) to produce values at a time intermediate to the time steps.

In general, a time-varying field with node type `T` can be used anywhere a field of node type `T` can be used; for instance, one can build derived fields on top of an unsteady field. There are a few cases where, due to precomputation or caching, unsteady fields are not interchangeable with steady fields. Recall that the member function `get_eager_field` provides a way for the user to get a new field where every node is eagerly evaluated. With a time-varying field, `get_eager_field` essentially returns a snapshot in time, i.e., a steady field. Thus if one wants to call `get_eager_field`, then the final argument, specifying a time value, is no longer optional. It is a run-time error if one calls `get_eager_field` on an unsteady field without providing a time.

A second case where unsteady fields require special treatment occurs when working with the FEL caching fields. The fields `FEL_cached_field` and `FEL_hash_cached_field` provide an option for users who want the computational frugality of caching results, without having to pay the costs of eagerly precomputing values over a whole field. Unfortunately, the FEL caching fields are not designed to work with time-varying fields; caching fields ignore time. Thus, with caching fields as they are currently implemented, a cached time-varying field would result in better performance, but in general wrong answers. To prevent what could be an insidious

problem, the cached field constructors check the type of field they are caching (at run-time) and fail if the argument is not steady.

FEL currently supports time-series data where there is a fixed interval in time between each time step. The library is designed so that adding support for data that are not temporally spaced at fixed intervals should not be too difficult in the future. The library supports both time-varying fields and time-varying meshes. The time-varying mesh support is described in the next chapter. The topology of the mesh is currently required to be constant over time, in other words, meshes (and the fields based on those meshes) that adapt over time are not currently supported.

An individual time step in an unsteady data set may be very large, the set of all the time steps may be hundreds of times larger. For many data sets it is not feasible to load every time step into memory. FEL time-series fields support the automated management of a subset of the time steps in memory using a working set. We describe the `FEL_time_series_field` working set control interface next.

25.1 Working sets and callbacks

The `FEL_time_series_field` class manages a working set of field objects, each object corresponding to a time step. When the user requests data at some point within the field represented by the `FEL_time_series_field` instance, the time-series field must check to see if the necessary time steps are currently in the working set. If so, then the time-series field requests data from the time steps, does temporal interpolation if necessary, and produces the result. If a needed time step is not in the working set, and the working set is not full, then the time step is acquired using the user-provided callback function. The callback function is described below. If the working set is full, then the field replaces a time step using a least recently used policy to choose the field to be replaced.

For most users, the working set management is completely automatic, in other words there is no need for the user to manually load and unload data. For those who do want to directly control which time steps are loaded, the `FEL_time_series_field` class provides an interface where one can dictate the working set size and contents. The working set management member functions are:

```
void set_working_set_size(int n);
bool load(int time_step);
bool load_all();
void unload(int time_step);
void unload_all();
void unload_least_recently_used();
void set_verbose(bool);
void show_working_set(ostream& s);
```

The function `set_working_set_size` allows the user to control the size of the working set used by the field. The default working set size is 5. The function `load` allows the user to make the field load a particular time step. The function `load_all` resizes the working set to the total number of time steps and loads each one. The

`load_all` function is handy when an application can afford to load every time step. The `unload` functions are relatively self-explanatory. The `set_verbose` function can be used to make the field output updates when time steps are loaded and unloaded. Watching the output from verbose mode can be educational for those not familiar with the use of working sets.

Users working with applications that are multi-threaded need to be careful with time-series fields as they are currently implemented. The routines that manage the working set do not have critical section protection. This means that, if multiple threads make accesses that cause the working set to change, then there is a chance that the working set data structure will become inconsistent. To avoid this problem the user should ensure that the working set does not change while in multi-threaded code. The simplest way to ensure no change is to load all the time steps initially. The `load_all` call is handy for this purpose. If loading all the time steps is not an option, then the user must make sure that working set changes occur only in single-thread mode. Providing critical section protection for the working set is on the list of future enhancements to FEL.

`FEL_time_series_field` relies on a callback function provided by the user at field construction time to produce the field corresponding to a particular time step when needed. For example, a callback returning a density field might look like:

```
unsigned flags = FEL_PLOT3D_3D_WHOLE;
const char* file_names[] = {"file1", "file2", "file3", "file4"};

FEL_float_field_ptr my_callback(FEL_mesh_ptr m, int time_step,
                                void*)
{
  return FEL_read_density(m, file_names[time_step], flags);
}
```

The callback function takes three arguments: the mesh `m` that the return field should be based on, a time step, and a `void*` pointer to "client data", i.e., data provided by the user when the time-series field is constructed that gets handed back to the callback function. The client data, for instance, could be a pointer to a structure that contains parameters such as the file names and file reader flags, parameters that are globals in the example above. The callback should return the field for the given time step. The callback can return NULL to indicate some type of failure, if a file could not be read, for example.

Callbacks give the user a great deal of flexibility in complying with requests for time step data. For example, the callback could construct and return a derived field. Another possibility would be for the callback to construct a new `FEL_core_field` using a buffer already in memory, such as the buffer used by a simulation. Yet another possibility would be for the callback to construct some type of analytic field that could be used for testing.

25.2 Time representations and conversions

The objects in FEL, such as `FEL_phys_pos`, that contain a time value use the type `FEL_time`. `FEL_time`, as described in Chapter 4, can represent three types of time: physical, computational or integer computational (time step). Usually it is not necessary to convert from one representation to another, since the user can choose to work in any of the three representations. If the user does need to convert, then the library provides the function:

```
int convert_time(const FEL_time& from,
                 FEL_time_representation_enum to_representation,
                 FEL_time* to) const;
```

The routine converts from `from` to `to`, given the desired `to_representation`.

25.3 Temporal interpolation

Given a time value that does fall on a time step, time-series fields do temporal interpolation. FEL currently supports linear interpolation between the pair of time steps bracketing a particular time. No temporal interpolation will take place if the user is working in the integer time step representation. Temporal interpolation is also suppressed if working in physical or computational time, and a given time value implies no fractional part between time steps.

In the case where both temporal and spatial interpolation are necessary, for instance, to produce a field value at a physical position in an unsteady field, temporal interpolation occurs first. For instance, consider a hexahedral structured mesh with simplicial decomposition turned off. Given an `at_phys_pos` call with a point p, FEL will locate the hexahedron containing p, temporally interpolate to get the field values at the time specified in p and then spatially interpolate to get the final result.

25.4 A time-varying field example

To illustrate the construction and use of a time-varying field, we present a small example. The callback and globals used for this example are the same as used for the example above (`my_callback`). See also the example program included as part of the FEL primer: `primer_13a.C`.

```
#include "FEL.h"
int n_time_steps = 4;
float physical_time_0 = 12000.0;
float physical_fixed_interval = 1.0;

int main()
{
  int res;
  flags = FEL_deduce_mesh_type(argv[1]);
```

```
    FEL_mesh_ptr mesh = FEL_read_mesh(argv[1], flags);
    assert(mesh != NULL);
    FEL_float_field_ptr field =
      new FEL_fixed_interval_time_series_float_field(
        mesh,
        n_time_steps,
        my_callback,
        NULL,
        physical_time_0,
        physical_fixed_interval);

    // find an arbitrary physical point inside mesh
    FEL_cell cell;
    FEL_phys_pos phys_pos;
    res = mesh->int_to_cell(mesh->card(3) / 2,
                            3, &cell);
    assert(res == 1);
    res = mesh->centroid_of_cell(cell, &phys_pos);
    assert(res == 1);

    phys_pos.set_physical_time(physical_time_0);
    float f;
    int res = field->at_phys_pos(phys_pos, &f);
    assert(res == 1);
  }
```

The mapping between physical time and time steps is specified by the `physical_time_0` and `physical_fixed_interval` arguments to the field constructor. For a time step t, the corresponding physical time p would be:

$$p = \texttt{physical_time_0} + t * \texttt{physical_fixed_interval}.$$

When the user works in physical time, `FEL_time_series_field` solves for (floating-point) computational time t using the same equation. If t has no fractional part, then temporal interpolation can be avoided. If t does have a fractional part, then the bracketing time steps are equal to the floor and ceiling of t.

Chapter 26

Time-Varying Meshes

It is not difficult to imagine unsteady data sets where not only the field values change with time but also the mesh geometry. For instance, the simulation of a helicopter or windmill typically involves rotating blades, and often there are meshes that move with the blades. FEL currently supports time-varying meshes using the time-varying field mechanism described in the previous chapter. The time-varying support is currently limited to curvilinear meshes. Once the time-varying mesh is constructed, it can be used just as any other mesh. As with time-varying fields, the biggest difference when using an unsteady mesh is that the arguments to mesh calls (e.g., the `FEL_cell` passed to `coordinates_at_cell`) must have their time value set.

Unfortunately, to construct a time-varying mesh, one must follow a somewhat circuitous route. A curvilinear mesh can be made to be unsteady by constructing the mesh with a time-series field. The field has node type `FEL_vector3f_and_int`, where the vector part represents coordinates, and the int part represents an IBLANK value. A curvilinear mesh constructed with a `FEL_vector3f_and_int` field consults the field whenever coordinates or IBLANK data are needed. If the field provided at mesh construction time is a time-varying field, then presto, one has a time-varying mesh.

In the following sections we walk through the construction of an unsteady (single-zone) curvilinear mesh and discuss what would need to be done in the multi-zone case. Building time-varying meshes is currently one of the more challenging things that one can do in FEL, and the reader is warned that more than one pass over this chapter may be necessary in order for everything to make sense. In the future our plan is to factor out the time-series mechanism from the time-series field class, so that it can be used for both meshes and fields, without some of the gymnastics described below. In the mean time, we hope that the following example is illuminating.

In the example we omit the construction of the solution field. Typically a data set with an unsteady mesh also has unsteady solution data. See the previous chapter for the details on building time-varying fields. See also the program `primer_13b.C` from the FEL primer for an example where a time-varying mesh is constructed.

26.1 Single-zone time-varying meshes

To construct an unsteady curvilinear mesh, follow these steps:
1. Assume a few globals, e.g.:

```
const int n_time_steps = 5;
const char* mesh_names[n_time_steps] =
  {"m0", "m1", "m2", "m3", "m4"};
unsigned file_reader_flags;
const float physical_time_0 = 0.0;
const float physical_fixed_interval = 1.0;
```

2. Provide a callback to be used by the `FEL_vector3f_and_int` field, where we use an adapter to make a mesh behave like a field, e.g.:

```
FEL_vector3f_and_int_field_ptr
my_mesh_callback(FEL_mesh_ptr, int time_step, void*) {
  FEL_mesh_ptr mesh;
  mesh = FEL_read_mesh(mesh_names[time_step], file_reader_flags);
  if (mesh == NULL) return NULL;
  return new FEL_mesh_as_vector3f_and_int_field(mesh);
}
```

3. Determine the structured dimensions of the mesh, e.g.:

```
int res;
FEL_mesh_ptr mesh;
char* mesh_name = "my_mesh";
file_reader_flags = FEL_deduce_mesh_type(mesh_name);
assert(file_reader_flags != 0);
FEL_vector3i dim;
res = FEL_read_mesh_zone_dimensions(mesh_name,
                                    file_reader_flags, 0, &dim);
```

4. Build a structured mesh:

```
mesh = new FEL_structured_mesh(dim[0], dim[1], dim[2]);
```

5. Build a time-series field for the coordinates and IBLANK, e.g.:

```
FEL_vector3f_and_int_field_ptr xyzi_field =
  new FEL_fixed_interval_time_series_vector3f_and_int_field(
    mesh,
    n_time_steps,
    my_mesh_callback,
    NULL,
    physical_time_0,
    physical_fixed_interval);
```

6. Build the unsteady curvilinear mesh, e.g.:

```
mesh =
  new FEL_curvilinear_mesh_xyzi_field_layout(dim[0], dim[1],
                                             dim[2], xyzi_field);
```

26.2 Multi-zone time-varying meshes

To construct an `FEL_multi_mesh`, one should follow the same pattern as above, once for each zone. If the user callback reads data from a multi-zone mesh file, then typically the callback should just read data for a particular zone. If only some zones vary with time then a time-series mechanism need only be built for those zones. Thus, if an application has some information about which zones are steady and which are not, there is an opportunity for optimization. By not building time-varying objects for steady zones, an application should get better performance, since unnecessary temporal interpolation and time step loading will be avoided. There should also be memory savings, since the mesh for a particular zone will only occur once in memory. For meshes that contain PLOT3D IBLANK information, one should be sure that if a zone is assumed to be steady, then the IBLANK and coordinate data both should not vary with time.

Appendix A

Glossary

abstract class An *abstract class* defines interface and implementation which are inherited by derived classes. Abstract classes in C++ cannot be instantiated. See also concrete class.

abstract factory See factory.

adapter An *adapter* provides an alternate interface for a class. For example, in FEL the class `FEL_mesh_as_field<T>` serves as an adapter that provides a field interface for a mesh object.

adjacent cells Given n-complex C, by convention there exists one (null) $(n+1)$-cell of which every n-cell of C is a face; likewise there exists one (null) $(n-1)$-cell which is the face of every vertex. Distinct k-cells c and d (for $0 \leq k \leq n$) are then said to be *adjacent* if:

(i) there exists some $(k-1)$-cell of C that is a face of both c and d, and

(ii) there exists some $(k+1)$-cell of which each of c and d is a face.

For example, two hexahedra are adjacent if they share a quadrilateral face. Two vertices are adjacent if they are the endpoints of a common edge.

affine combination Let $P = \{p_1, ..., p_k\}$ be a finite set of points in $\mathbf{R}^d$. A point x is an *affine combination* of P if:

$$x = \sum_{i=1}^{k} \lambda_i p_i,$$

where $\sum_{i=1}^{k} \lambda_i = 1$. See also linear combination.

affinely independent A finite set P of k points is *affinely independent* if there is not a point $p_i \in P$ such that p_i is an affine combination of $P - \{p_i\}$.

block A *block* is an alternate name for zone.

Cartesian grid A *Cartesian grid* is a grid where the cells are aligned with the coordinate axes. The grid looks similar to the output from a quad-tree or oct-tree decomposition, though Cartesian grids are not necessarily oct-trees. The term "Cartesian grid" is also used by some as a synonym for uniform grid.

cell-centered field A *cell-centered field* is a field where there is a node associated with each cell in the mesh. Typically the cells are hexahedra or tetrahedra.

CFD Computational Fluid Dynamics.

cell a *k-cell* of a topological space T is a subspace of T whose interior is homeomorphic to $\mathbf{R}^k$ and whose boundary is non-null. Less formally, typically when one speaks of cells in CFD one means hexahedra or tetrahedra (i.e., 3-dimensional cells or 3-cells), though there are also 0-cells (vertices), 1-cells (edges) and 2-cells (polygons).

cell complex A *cell complex* of a topological space T is a finite collection C of cells of T such that:

(i) the relative interiors of cells of C are pairwise disjoint,

(ii) for each cell $c \in C$, the boundary of cell c is the union of elements of C,

(iii) if $c, d \in C$ and $c \cap d \neq \emptyset$, then $c \cap d$ is the union of elements of C.

CGNS CGNS (Complex Geometry Navier Stokes) is a file format currently under development for storing CFD data.

Chimera Chimera, a composite monster from Greek mythology, is used in CFD when speaking of multi-zone grids.

Chimera file A *Chimera file* is an auxiliary file to the main multi-zone grid description file. The Chimera file contains definitive information on how the solver should handle regions where zones intersect.

class A *class* is a basic concept in object-oriented programming. A class encapsulates both data and member functions for operating on the data.

closure The *closure* of a cell c in a cell complex C consists of all the faces of c.

computational space A *computational space* is a space defined in terms of a particular discretization used to decompose a physical space. One typical computational space corresponds to a hexahedral curvilinear (structured) mesh in physical space: in computational space each physical space hexahedron corresponds to a unit cube. Each vertex in such a computational space can be indexed as if in a 3-dimensional array, and the indices are typically labeled i, j, and k.

concrete class A *concrete class* is a class that can be instantiated. See also abstract class.

curvilinear mesh A *curvilinear mesh* is the most general form of structured mesh.

demand-driven evaluation *Demand-driven evaluation* is a technique where computations are done only when requested, rather than in advance.

derived field A *derived field* is a field defined in terms of one or more other fields. For example, a velocity derived field can be defined in terms of momentum and density fields.

dynamic binding The run-time association of a request to an object and one of its operations. In C++, virtual functions are dynamically bound.

dynamic casting *Dynamic casting* is a new feature in the C++ standard designed to support safe casts down or across an inheritance hierarchy. With dynamic casts, the success of the cast of a pointer can be verified at run-time by testing whether the result is non-NULL. For example, given a class `B` and a class `D` derived from `B`:

```
B* b;
D* d;
b = new D();
d = (D*) b;                    // C-style cast
d = dynamic_cast<D*> b;        // C++ dynamic cast
assert(d != NULL);
```

The C++ standard only requires that dynamic casting be supported for classes with virtual functions. Dynamic casting is not supported by some older C++ compilers. See also RTTI.

eager evaluation *Eager evaluation* is the opposite of lazy evaluation.

edge-centered field An *edge-centered field* is a field where there is a node associated with each edge of the mesh.

Enterprise "Enterprise" is the name of the file format used for working with paged meshes and fields. See Chapter 23.

exceptions *Exceptions* are a new feature of the C++ standard designed to support the handling of exceptional conditions. Exceptions are not supported by some older C++ compilers.

face A cell c in a collection C is the face of a cell $d \in C$ if c can be defined in terms of a subset of the vertices of d. For example, a tetrahedron has triangle, edge, and vertex faces. See also proper face.

facet A *facet* is a face. Typically facets refer to 2-cells, such as triangles and quadrilaterals.

face-centered field A *face-centered field* is a field where a node is associated with each face in the mesh. Typically the faces refer to 2-cells, such as quadrilaterals and triangles.

factory A *factory* encapsulates the procedures required to build various types of objects. For example, in FEL the class `FEL_plot3d_field` encapsulates the steps required to build each of the over 50 standard derived fields defined by PLOT3D.

FAST The *Flow Analysis Software Toolkit* is a CFD visualization post-processing application developed at NASA Ames Research Center.

FEL The *Field Encapsulation Library*.

field A *field* represents a function within a domain via a finite set of nodes. Every field has a mesh that contains the location and organization of the nodes.

friend class In C++, a class *A* can be declared to be a *friend* of another class *B*, enabling *A* to have the same access rights to the member functions and data of *B* instances as *B* itself.

general position The definition of *general position* is context dependent, i.e., dependent upon the particular application. Typically it is easier to define general position in terms of what it is not: a configuration is not in general position if an infinitesimal perturbation can change how the configuration is classified. For example, 4 points in $\mathbf{R}^3$ are not in general position if they are coplanar.

grid The term *grid* is used as a synonym for mesh by many. For some, grid implies structured. Thus mesh is preferred to grid if one wishes to refer to unstructured objects, or both structured and unstructured objects.

***has-a* relationship** A *has-a relationship* denotes containment. *Has-a* is interchangeable with *is-part-of* or *uses-for-implementation*. See also *is-a* relationship.

homeomorphic Two spaces are *homeomorphic* if they are topologically equivalent.

hybrid mesh A *hybrid mesh* is a mesh that is partially structured and partially unstructured. See Figure A.1.

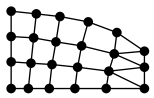


Figure A.1: An example of a hybrid mesh.

IBLANK IBLANK is a standard from PLOT3D [WBPE92] for associating extra information with each vertex in a mesh, using one integer per vertex. IBLANK values can be used to provide hints about overlapping zones in multi-zone meshes, or to indicate that the node value associated with a vertex is not valid.

incident cells Cells c and d are *incident* if c is a proper face of d, or vice versa. For example, there are vertices, edges, and quadrilaterals that are incident with a hexahedron in a mesh.

instantiation *Instantiation* is the creation of a new instance of a given class. With templated classes instantiation also involves the generation at compile-/link-time of the code implied by a particular set of template parameters.

irregular mesh *Irregular mesh* is an alternate name for rectilinear mesh.

***is-a* relationship** An *is-a relationship* expresses a refinement between types or classes. In C++ class hierarchies, the relationship between a derived class and a base class is usually *is-a*. For example, a curvilinear mesh *is-a* structured mesh, and a structured mesh *is-a* mesh. See also *has-a* relationship.

iterator *Iterators* are an abstraction for processing items one-by-on in a collection. Meshes can be thought of as collections of cells, and FEL provides iterators for the mesh classes.

Jacobian matrix A *Jacobian matrix* is used to do transformations from one coordinate space to another. For example, to do transformations between computational space and physical space one can use the Jacobian matrix $\mathcal{J}$:

$$\mathcal{J} = \frac{\partial \vec{x}}{\partial \vec{\xi}} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{pmatrix}$$

where x, y, and z refer to physical coordinates and ξ, η, and ζ (xi, eta and zeta) refer to computational coordinates.

One typical use of the Jacobian matrix is to convert partial derivatives computed with respect to computational space into physical space derivatives. For instance, to compute the physical space spatial gradient at a point in a scalar field, one can compute partial derivatives in computational space and then convert to physical space using the following equation, based on the chain rule for derivatives:

$$\frac{\partial f}{\partial \vec{x}} = \frac{\partial f}{\partial \vec{\xi}} \frac{\partial \vec{\xi}}{\partial \vec{x}} = \begin{pmatrix} \frac{\partial f}{\partial \xi} & \frac{\partial f}{\partial \eta} & \frac{\partial f}{\partial \zeta} \end{pmatrix} \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{pmatrix}^{-1}$$

The partial terms with respect to computational space are relatively easy to approximate using finite differencing techniques.

lazy evaluation Lazy evaluation is a synonym for demand-driven evaluation.

linear combination Let $P = \{p_1, ..., p_k\}$ be a finite set of points in $\mathbf{R}^d$. A point x is a *linear combination* of P if:

$$x = \sum_{i=1}^{k} \lambda_i p_i.$$

See also affine combination.

member function A *member function* is a function defined as part of a class.

mesh A *mesh* is a collection of vertices and information about the connectivity between the vertices. Based on the connectivity one can also define cells, for example, hexahedral cells in a structured mesh.

method *Method* is an alternate name for member function.

metrics The terms in the inverse of a Jacobian matrix are known as *metrics*.

node A *node* is a point in the domain where the solution is calculated.

overloading An operation is *overloaded* if the code that is executed is dependent upon the type of the arguments. For example, C supports a limited kind of overloading for operators such as "+", since the compiler automatically generates the appropriate machine instructions, such as for fixed-point or floating-point addition, based on the argument types. C++ supports the same built-in overloading, as well as a much more general system for user-defined overloading of operators and functions.

The choice of which version of an overloaded function to use in C++ is made at compile-time, in contrast to dynamically-bound functions, where the choice is made at run-time. See virtual function and polymorphism.

overset grids An overset grid is a grid where there are multiple, overlapping zones.

periodic mesh In a *periodic mesh* one takes advantage of the symmetry or periodicity of the computational domain in order to use a mesh that covers only part of the domain. For example, in a turbomachinery simulation where there is radial symmetry, one may use a mesh that covers only a pie-shaped wedge within the domain.

parameterized types A *parameterized type* is a type where some of the constituent types are specified with parameters. For example, a list data structure may be parameterized by the type of value contained by each element in the list. Parameterized types are supported in C++ via templates.

physical space *Physical space* is the space that corresponds to the "real world" domain, such as the region surrounding an aircraft, where one wishes to model phenomena such as flow. See also computational space.

placement new *Placement new* is a special version of the C++ *operator new* where the user explicitly specifies the memory to be used for an object.

polymorphism *Polymorphism* is the ability to substitute objects of matching interface for one another at run-time. In C++ a class is *polymorphic* if it declares or inherits a virtual function.

PLOT3D PLOT3D was originally a CFD visualization post processing tool developed by Pieter Buning of NASA Ames Research Center. PLOT3D lives on primarily in some of the standards it defined such as those for file formats and IBLANKs.

proper face A cell c is a *proper face* of a cell d if c is a face of d and $c \neq d$.

pure virtual function A *pure virtual function* of a class *A* is a member function that is required to be defined by subclasses of *A*. A class that has a pure virtual function is an abstract class.

rectilinear mesh A *rectilinear mesh* is a structured mesh where the cells are aligned with the coordinate axes, but the spacing between adjacent vertices can be irregular. Rectilinear meshes are also known as irregular meshes or non-uniform meshes. See Figure A.2.

reference counting *Reference counting* is a technique for tracking how many other objects are using (have a reference to) a particular object. Reference counting is typically part of a memory management strategy where objects are automatically deallocated (garbage collected) when their count goes to 0. Reference counting is also known as "use counting".

regular mesh A *regular mesh* is a structured mesh where the cells are aligned with the coordinate axes and the spacing between adjacent vertices is equal in each dimension. See Figure A.2. In some of the literature a regular mesh is considered to be a uniform mesh.

RTTI *Run-Time Type Identification* is a new capability defined in the C++ standard allowing the user to query an object about its type. RTTI is required to be supported only for classes which have virtual functions. RTTI is not supported by some older C++ compilers. Some new C++ features, such as dynamic casting, depend upon the availability of RTTI.

signature An operation's *signature* is defined by its name, parameters, and return value.

simplex A *k-simplex* is the convex hull of $k + 1$ affinely independent points. A 0-simplex is a vertex, a 1-simplex is an edge, a 2-simplex is a triangle and a 3-simplex is a tetrahedron.

simplicial decomposition *Simplicial decomposition* is the subdivision of mesh cells into simplices. For example, hexahedra would be subdivided into tetrahedra, and quadrilaterals would be subdivided into triangles.

specialization In C++ templated functions, a *specialization* is an implementation for a specific template type that overrides the generic implementation provided by the template.

star The *star* of a cell c in a cell complex C is the subset of C consisting of all the cells of which c is a face.

steady A *steady* mesh or field is one that does not change over time. See also unsteady.

STL The *Standard Template Library* is a library of C++ templated classes supporting container data structures, such as sets.

structured mesh In a structured mesh the vertices and cells are organized in a regular pattern. In one of the most typical types of structured meshes, the vertices can be addressed as if they were in a multi-dimensional array. In 3-d, all the 3-dimensional cells are hexahedral. Structured meshes can be further classified as uniform, regular, rectilinear and curvilinear. See Figure A.2.

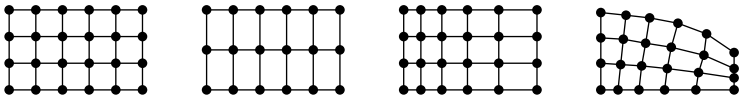


Figure A.2: FEL structured mesh types.

templates *Templates* are C++'s support for parameterized types.

uniform mesh A *uniform mesh* is a structured mesh where the cells are aligned with the coordinate axes and the spacing between adjacent vertices is uniform throughout. See Figure A.2.

unsteady An *unsteady* mesh or field is one that changes over time. See also steady.

unstructured mesh An *unstructured mesh* in 3-d consists of polyhedral cells, with no implicit connectivity. The cells are not necessarily all tetrahedra, though this is one of the most common unstructured types. See Figure A.3.

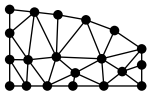


Figure A.3: An unstructured mesh.

use counting See reference counting.

vertex-centered field A *vertex-centered field* is a field where a node is associated with each vertex in the mesh. The standard file formats defined by PLOT3D are all for vertex-centered fields.

virtual function A virtual function is a member function where the particular implementation is selected at run-time, based on the type of the object for which the operation is called. C++ supports polymorphism via virtual functions.

working set A *working set* is a subset of a much larger set, where maintaining a subset improves performance in some respect. For instance, the caching done in the memory hierarchies used by most microprocessor-based systems can be thought of as a performance improvement strategy based on working sets.

zone A *zone* refers to a particular submesh in a multi-mesh.

Bibliography

[Ale61] P. Alexandroff. *Elementary Concepts of Topology*. Dover Publications, Inc., New York, 1961. Translated by Alan E. Farley.

[B+90] G. Bancroft et al. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Proceedings of Visualization '90*, pages 14–24. IEEE Computer Society Press, October 1990.

[Bar91] T. Barth. Numerical aspects of computing viscous high Reynolds number flows on unstructured meshes. In *29th Aerospace Sciences Meeting*, Reno, Nevada, January 1991.

[BKGY96] S. Bryson, D. Kenwright, and M. Gerald-Yamasaki. FEL: The field encapsulation library. In *Proceedings of Visualization '96*, pages 241–247. IEEE Computer Society Press, October 1996.

[CE97] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of Visualization '97*, pages 235–244. IEEE Computer Society Press, October 1997.

[FIT] FITS. `http://www.gsfc.nasa.gov/astro/fits/fits_home.html`.

[GBD96] K. Gundy-Burlet and D. Dorney. Three-dimensional simulations of hot streak clocking in a 1-1/2 stage turbine. In *32nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, Lake Buena Vista, FL, July 1996.

[KL95] D. Kenwright and D. Lane. Optimization of time-dependent particle tracing using tetrahedral decomposition. In *Proceedings of Visualization '95*. IEEE Computer Society Press, October 1995.

[MS96] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison Wesley Publishing Company, Menlo Park, California, 1996.

[PTVF92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, New York, second edition, 1992.

[Vis] Vis5D. `http://www.ssec.wisc.edu/~billh/vis5d.html`.

[WBPE92] P. Walatka, P. Buning, L. Pierce, and P. Elson. *PLOT3D User's Manual*. National Aeronautics and Space Administration, July 1992. NASA Technical Memorandum 101067.