Contents

1 Introduction

This is the VisAD Java Component Library Developers Guide, describing the design and use of the VisAD Java component library for interactive analysis and visualization of numerical data. It also describes the design rationale, based on lessons learned from early mainframe visualization [5], interactive visualization [6], interactive computational steering [8], high-speed networks [7, 10], virtual reality [10], and supporting a broad user community [1, 9]. Key design decisions include:

- The use of pure Java for platform independence and to support data sharing and real-time collaboration among geographically distributed users. Support for distributed computing is integrated at the lowest levels of the system.
- A general mathematical data model that can be adapted to virtually any numerical data, that supports data sharing among different users, different data sources and different scientific disciplines, and that provides transparent access to data independent of storage format and location (i.e., memory, disk or remote).
- A general display model that supports interactive 3-D, data fusion, multiple data views, direct manipulation, collaboration, and virtual reality.
- Data analysis and computation integrated with visualization to support computational steering and other complex interaction modes.
- Support for two distinct communities: developers who create domain- specific systems based on VisAD, and users of those domain-specific systems. VisAD is designed to support a wide variety of user interfaces, ranging from simple data browser applets to complex applications that allow groups of scientists to collaboratively develop data analysis algorithms.
- Developer extensibility in as many ways as possible.

1.1 System Availability

The VisAD Java class library, including complete source code and installation instructions, is freely available from:
http://www.ssec.wisc.edu/ billh/visad.html

VisAD requires Java 1.2. VisAD displays are generated using either Java2D (included in Java 1.2) or Java3D. More information about these is available at:
http://java.sun.com/

1.2 Package Structure

The VisAD system consists of the following packages:

visad - the core VisAD package

visad.cluster - large data distributed on clusters

visad.collab - collaborative displays

visad.java3d - Java3D displays for VisAD

visad.java2d - Java2D displays for VisAD

visad.python - Python support for VisAD

visad.browser - JDK 1.1 browser interface to VisAD

visad.ss - the VisAD Spread Sheet

visad.formula - formula parser

visad.matrix - matrix operations via JAMA

visad.math - FFT and histogram operations

visad.util - VisAD user interface utilities

visad.data - VisAD data format adapters

visad.data.in - support for read-only VisAD adapters

visad.data.units - units database and parsing

visad.data.dods - VisAD - DODS server adapter

visad.data.fits - VisAD - FITS file adapter

visad.data.netcdf - VisAD - netCDF file adapter

visad.data.netcdf.units - units parser for netCDF adapter

visad.data.netcdf.in - data input for netCDF adapter

visad.data.netcdf.out - data output for netCDF adapter

visad.data.hdfeos - VisAD - HDF-EOS file adapter

visad.data.hdfeos.hdfeosc - native interface to HDF-EOS

visad.data.gif - VisAD - GIF / JPEG file adapter

visad.data.ij - VisAD adapter for image files via ImageJ

visad.data.jai - VisAD adapter for image files via JAI

visad.data.qt - VisAD - QuickTime file adapter

visad.data.tiff - VisAD - TIFF file adapter

visad.data.text - VisAD - ASCII file adapter

visad.data.vis5d - VisAD - Vis5D file adapter

visad.data.mcidas - VisAD - McIDAS file adapter

visad.data.biorad - VisAD - Biorad file adapter

visad.data.amanda - VisAD - F2000 file adapter & viewer

visad.data.hdf5 - VisAD – HDF-5 file adapter

visad.data.hdf5.hdf5objects - helper for HDF-5 adapter

visad.data.visad - VisAD (serial object) file adapter

visad.data.visad.object - VisAD (serial object) file adapter

The following packages are distributed with VisAD:
HTTPClient - complete http client library

nom.tam.fits - Java FITS file binding

nom.tam.util - Java FITS file binding

nom.tam.test - Java FITS file binding

ucar.netcdf - Java netCDF file binding

ucar.multiarray - Java netCDF file binding

ucar.util- logging functions for servers

ucar.tests - test Java netCDF file binding

dods.dap - DODS server core classes

dods.dap.parser - JavaCC generated DODS paersers

dods.dap.server - DODS servers

dods.util - utility classes for DODS

gnu.regexp - GNU regular expressions

edu.wisc.ssec.mcidas - Java McIDAS file binding

edu.wisc.ssec.mcidas.adde - Java McIDAS file binding

ij - ImageJ system package

ij.gui - ImageJ system package

ij.io- ImageJ system package

ij.measure - ImageJ system package

ij.plugin- ImageJ system package

ij.plugin.filter - ImageJ system package

ij.plugin.frame - ImageJ system package

ij.process - ImageJ system package

ij.text - ImageJ system package

ij.util - ImageJ system package

ncsa.hdf.hdf5lib - Java HDF-5 file binding

ncsa.hdf.hdf5lib.exceptions – Java HDF-5 file binding

visad.paoloa - GOES satellite analysis

visad.paoloa.spline - spline fitting

visad.aune - shallow fluid model

visad.benjamin - Milky Way galaxy model

visad.rabin - rainfall estimation spread sheet

visad.jmet - JMET – Java meteorology

visad.meteorology - classes useful for meteorology

visad.bom - classes for ABOM

visad.aeri - classes for AERI data

visad.georef - classes for georeferencing

visad.install - cluster installer for VisAD-in-a-box

The VisAD source distribution also includes a directory visad/examples that contains classes with the default package (i.e., no package statement).

VisAD is constantly being updated to fix bugs and add features and we don't even try to track all of these changes with VisAD version numbers. Rather, a file named 'DATE' is included in distribution jar files that gives the date and time the distribution file was created. We will change VisAD version numbers as new features accumulate.

1.3 Authorship, Copyright, History and Support

VisAD was written by programmers at the University of Wisconsin Space Science and Engineering Center (SSEC), at the Unidata Program Office and at the National Center for Supercomputer Applications (NCSA). They are:

Bill Hibbard - SSEC (contact author: hibbard@facstaff.wisc.edu)

Steve Emmerson - Unidata

Curtis Rueden - SSEC

Tom Rink - SSEC

Dave Glowacki - SSEC

Tom Whittaker - SSEC

Tommy Jasmin - SSEC

Don Murray - Unidata

Jeff McWhirter - Unidata

Nick Rasmussen - SSEC

Peter Cao - NCSA

James Kelly - ABOM

Andrew Donaldson - ABOM

Doug Lindholm - NCAR

Sylvain Letourneau - Canadian NRC

The following people made substantial intellectual contributions to the design:

John Anderson - SSEC Dave Fulker - Unidata Russ Rew - Unidata Glen Davis - Unidata

VisAD is freely available including source code. It is protected by copyright statements embedded in the source code and in the NOTICE, LICENSE and COPYING files distributed with the source code.

The VisAD Java class library is actually VisAD version 2.0. VisAD versions 1.0 and 1.1 were written in C by Bill Hibbard, Brian Paul (of SSEC) and Andre Battaiola (while visiting SSEC from INPE/CPTEC in Brazil) [8, 9], with substantial intellectual contributions from Charles Dyer of the UW Computer Sciences Dept.

VisAD has adopted the UD Units library developed by Steve Emmerson of Unidata. [http://www.unidata.ucar.edu/packages/udunits/index.html].

VisAD borrows design ideas and code from the Vis5D system for interactive visualization of numerical simulations of weather and other environmental phenomena [6, 9, 10]. Vis5D was written in C by Bill Hibbard, Johan Kellum (of SSEC), Brian Paul, Andre Battaiola, Dave Santek (of SSEC) and Marie-Francoise Voidrot-Martinez (while visiting SSEC from METEO France).

Vis5D grew out of the 4-D McIDAS system [5, 6], which was part of Verner Suomi's McIDAS system for visualizing data from his weather satellites. The 4-D McIDAS was the 3-D (plus animation) analog of Tom Whittaker's 2-D graphics subsystem of McIDAS, which was the first interactive weather graphics system.

The development of this software has been supported by NASA, EPA, NSF (via Unidata and NCSA), NOAA, ARPA and DOE. We especially want to thank Joe Bredekamp of NASA, Cliff Jacobs of NSF and Larry Smarr of NCSA for their support of the Java VisAD. We are also grateful to the Charles and Mamie van Doren Foundation for their support.

2 Overview

This is an overview of how applications are constructed using VisAD. Throughout this guide, we will capitalize the proper names of VisAD classes such as Data and Display, in accordance with Java custom. A VisAD application is a network of:

Data objects these may be simple real number values, text strings, vectors of real numbers, arrays such as images or grids, or complex hierarchies of data. They may include metadata for units, coordinate systems, complex sampling topologies, missing data indicators and error estimates, or they be simple values with minimal metadata. Data objects are described more thoroughly in Section 3. Section 3.12 explains the relation between data structures in VisAD and the C programming language.

Display objects these generate interactive 3-D depictions of Data objects on a workstation screen or in immersive virtual reality (such as a CAVE, ImmersaDesk, or helmet). Display objects are linked to Data objects, so that Data depictions are updated whenever Data values change. Some Displays implement direct manipulation, which enables users to change Data values by re-drawing Data depictions. Displays on different machines may be linked to the same Data objects, in which case geographically distributed users may collaboratively visualize and manipulate the same Data. Displays are described more thoroughly in Section 4.

Cell objects these are computations that are invoked whenever their input Data objects change value. They take their name from the cells of spread sheets. Like displays, Cells are linked to Data objects through DataReference objects (in fact, Displays and Cells both extend Action, the general class for objects whose actions are triggered by changing Data values). Cell objects are described more thoroughly in Section 5.

User interface (UI) objects these are generally part of a UI component package such as AWT or JFC, although there are a few specialized utility UI components in the VisAD class library (described in Section 8). UI objects may also link to Data objects. Data values may be changed by UI events (for example, sliders may change the values of real number data objects), or UI components may link to Actions so that they update whenever Data object values change.

DataReference objects these are pointers to Data objects. For example, in the statement "x = 3", x plays the role of a DataReference object and 3 plays the role of a Data object. The value of 3 cannot change just as many VisAD Data classes have values that cannot change (these are called immutable classes). So DataReference objects are necessary to represent variable data, just as the variable "x" is necessary in programming languages. Display, Cell and UI objects are linked to Data objects through DataReference objects. And DataReference objects would be used as symbol table entries in VisAD applications that implement programming language interpreters. DataReference objects are described more thoroughly in Section 3.11.

VisAD exploits Java Remote Method Invocation (RMI) so that Data, DataReference, Display, Cell and user interface objects may be linked together independent of their location on the network. Thus users at geographically remote workstations may collaborate by constructing Displays and linking them to the same Data object. Applications can be developed with VisAD that enable users to locate Data objects via web browsers and drag-and-drop them into Displays, link them into data analysis algorithms, and share visualizations of the results with colleagues at other locations. VisAD's use of RMI is described more thoroughly in Section 6.

The World Wide Web has created a shared network of generally passive text and image information. Distributed objects enabled by Java RMI will make this shared network much more active; that is, a network that includes execution threads. The VisAD system's general data model and thorough use of Java RMI provide a way to build a shared, active network of scientific data, displays and computations. This network could:

1. Change dynamically.
2. Have many simultaneous users with their own sets of display and user interface objects.
3. Have an indefinite life span, with users connecting and disconnecting but the basic network remaining.
4. Support numerous interacting execution threads.
5. Provide entrance points via web pages.

2.1 A Very Simple Application Example

We start with an application that reads a time sequence of images from a netCDF file and displays it with animation. There are only four executable lines of code in the application that have anything to do with VisAD in **code listing 2.1**:

1. creating the netCDF file reader,
2. reading the file,
3. creating a display of the file, and
4. linking the display into a JFrame.

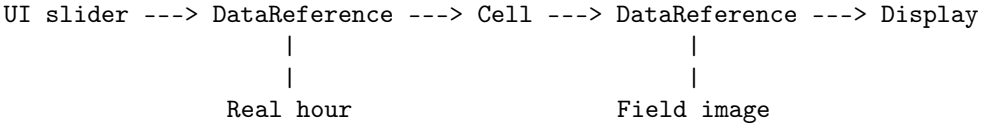
Listing 2.1: A very simple example of how easy a visualization program can get.

```
// import needed classes
import visad.*;
import visad.util.DataUtility;
import visad.java3d.DisplayImplJ3D;
import visad.data.netcdf.Plain;
import java.rmi.RemoteException;
import java.io.IOException;
import java.awt.*;
import javax.swing.*;

public class VerySimple {

  // type 'java VerySimple' to run this application
  public static void main(String args[])
         throws VisADException, RemoteException, IOException {

    // create a netCDF reader
    Plain plain = new Plain();

    // read an image sequence from a netCDF into a data object
    DataImpl image_sequence = plain.open("images.nc");

    // create a display for the image sequence
    DisplayImpl display = DataUtility.makeSimpleDisplay(image_sequence);

    // create JFrame (i.e., a window) for the display
    JFrame frame = new JFrame("VerySimple VisAD Application");

    // link the display to the JFrame
    frame.getContentPane().add(display.getComponent());

    // set the size of the JFrame and make it visible
    frame.setSize(400, 400);
    frame.setVisible(true);
  }
}
```

The VerySimple.java program is included in the visad/examples directory of the VisAD source distribution. To run it you also need to download and uncompress the images.nc file from ftp://ftp.ssec.wisc.edu/pub/visad-2.0/images.nc.Z into your visad/examples directory.

2.2 A Simple Application Example

The VerySimple application is so simple that it hides the network of VisAD objects it creates. Thus we present the Simple application which reads and displays the same image sequence, but provides some user interaction and makes the network of objects explicit. The diagram below shows the network of objects created by the Simple application. Its user controls a real number Data object (an hour value) via a UI slider, which in turn triggers a Cell to re-compute the value of a more complex Field Data object (for example, this may be an image array selected from an image sequence), whose depiction is updated in a Display.



This diagram corresponds to the simple application **code listing 2.2**.

Listing 2.2: A simple example of how easy a visualization program can get.

```
// import needed classes
import visad.*;
import visad.java3d.DisplayImplJ3D;
import visad.util.VisADSlider;
import visad.data.netcdf.Plain;
import java.rmi.RemoteException;
import java.io.IOException;
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;

public class Simple {

  // type 'java Simple' to run this application
  public static void main(String args[])
         throws VisADException, RemoteException, IOException {

    // create a DataReference for an image
    final DataReference image_ref = new DataReferenceImpl("image");

    // create a netCDF reader
    Plain plain = new Plain();

    // open a netCDF file containing an image sequence and adapt
    // it to a Field Data object
    final Field image_sequence = (Field) plain.open("images.nc");

    // create a Display using Java3D
    DisplayImpl display = new DisplayImplJ3D("image display");

    // extract the type of image and use
    // it to determine how images are displayed
```

```
    FunctionType image_sequence_type =
      (FunctionType) image_sequence.getType();
    FunctionType image_type =
      (FunctionType) image_sequence_type.getRange();
    RealTupleType domain_type = image_type.getDomain();
    // map image coordinates to display coordinates
    display.addMap(new ScalarMap((RealType) domain_type.getComponent(0),
                                 Display.XAxis));
    display.addMap(new ScalarMap((RealType) domain_type.getComponent(1),
                                 Display.YAxis));
    // map image brightness values to RGB (default is grey scale)
    display.addMap(new ScalarMap((RealType) image_type.getRange(),
                                 Display.RGB));

    // link the Display to image_ref
    // display will update whenever image changes
    display.addReference(image_ref);

    // create a DataReference and RealType for an 'hour' value
    final DataReference hour_ref = new DataReferenceImpl("hour");
    RealType hour_type =
      (RealType) image_sequence_type.getDomain().getComponent(0);
    // and link it to a slider
    VisADSlider slider = new VisADSlider("hour", 0, 3, 0, 1.0,
                                         hour_ref, hour_type);

    // create a Cell to extract an image at 'hour'
    // (this is an anonymous inner class extending CellImpl)
    Cell cell = new CellImpl() {
      public void doAction() throws VisADException, RemoteException {
        // extract image from sequence by evaluating image_sequence
        // Field at 'hour' value
        image_ref.setData(image_sequence.evaluate(
                                       (Real) hour_ref.getData()));
      }
    };
    // link cell to hour_ref to trigger doAction whenever
    // 'hour' value changes
    cell.addReference(hour_ref);

    // create JFrame (i.e., a window) for display and slider
    JFrame frame = new JFrame("Simple VisAD Application");
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
    });

    // create JPanel in JFrame
    JPanel panel = new JPanel();
    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
    panel.setAlignmentY(JPanel.TOP_ALIGNMENT);
    panel.setAlignmentX(JPanel.LEFT_ALIGNMENT);
    frame.getContentPane().add(panel);

    // add slider and display to JPanel
    panel.add(slider);
    panel.add(display.getComponent());

    // set size of JFrame and make it visible
    frame.setSize(500, 600);
    frame.setVisible(true);
  }
```

```
}
```

Creating the DataReferences for 'hour' and 'image' and linking them to the VisAD-Slider and Cell is simple. Creating the Display and linking it to the 'image' DataReference is also simple. Setting up the JFrame and JPanel are not too difficult and really independent of VisAD. The only complex part of this application is extracting the image's type information for use in setting up the Display. Every VisAD Data object has a MathType that describes its basic structure. Every real number value occurring in a complex Data object has a RealType, a subclass of MathType, that includes a name like "latitude", "time" or "temperature". The code in our simple application extracts the RealTypes from the MathType of the image so that it can define different display roles for the real number values occurring in the image. The image Data object is interpreted as a function that maps pixel locations into pixel brightnesses, and its MathType, denoted image_type, is a FunctionType that includes MathTypes for the function's domain and range. The image_type can be diagrammed as:

```
              FunctionType (image_type)
                  /               \
        function domain      function range
        RealTupleType        RealType (brightness)
        /           \                      |
RealType (line)     RealType (element)     |
       |                      |            |
       |                      |            |
       v                      v            v
     XAxis                  YAxis         RGB
```

Note that the bottom of the diagram includes the scalar mappings of image_type's RealType components to DisplayRealTypes: XAxis, YAxis and RGB (RGB indicates a pseudo color lookup table that maps brightness values to red, green and blue values).

The image_sequence Data object is treated as a function from time (hours) to images, so its MathType, denoted image_sequence_type, is also a FunctionType that can be diagrammed as:

```
    FunctionType (image_sequence_type)
        /                       \
function domain           function range
RealType (hour)         FunctionType (image_type)
                           /               \
                  function domain      function range
                  RealTupleType        RealType (brightness)
```

```
                /           \
RealType (line)     RealType (element)
```

Note that the image_type diagram is replicated in the range of this image_sequence_type diagram.

The call to the getType method of image_sequence returns its MathType, and then the calls to the getRange, getDomain and getComponent methods are used to parse the tree structure of the MathType to extract the RealTypes at the leaves of the tree. These RealTypes are then mapped to display coordinates such as Display.XAxis and Display.YAxis, and to display colors such as Display.RGB, using the ScalarMap constructors that are attached to the Display via its addMap method.

Note that image_sequence is treated as a function from a set of hour values to a set of images, and the evaluate method of image_sequence evaluates this function at an hour value and returns an image. Thus the doAction method of our computational Cell applies the evaluate method of image_sequence to an hour value to extract an image. Note also that image_sequence is declared as a Field, which is the VisAD class for functions represented by finite samplings.

In order to run the Simple application you need to download and uncompress the netCDF file "images.nc" from ftp://ftp.ssec.wisc.edu/pub/visad-2.0/images.nc.Z.

Response may be sluggish due to a problem with threads in early versions of Java3D. We should point out that the logic of this simple application, interactively selecting and displaying an image from an image sequence, can be implemented more simply and with faster response in a VisAD Display by mapping the "hour" RealType to Display.SelectValue. However, the Simple application is a nice illustration of how Data, DataReference, UI, Display and Cell objects can be linked together.

Section 12.3 describes a more complex application that creates a network of linked Data, DataReference, Display, Cell and UI objects distributed around the network to support collaboration among users at geographically remote locations. This application also includes direct manipulation Displays, where users change Data values by re-drawing their depictions. Appendix B is a complete source code listing of this application.

While the application described in Section 12.3 is more complex than the one presented here, it is still specific to a particular scientific problem. VisAD can be used to build much more flexible and generic applications. It would not be difficult to construct a generic spread sheet consisting of an array of Displays with one Data object per Display. UI components could let users add new Displays as needed and define the source of Data as: 1) a file, 2) direct manipulation in the Display, or 3) a mathematical expression involving Data objects in other Displays. VisAD could also be used as the basis for implementing a data flow system, or an interpreted numerical programming language.

2.3 Flexible Design by Reduction to Elements

The VisAD system offers a reductionist approach to design, as illustrated in the simple example of Section 2.2. Its image and image_sequence Data objects were defined as hierarchies of simple real values, and the Display for the image Data object was defined by mappings of its real values. This reductionist approach is very flexible in dealing with novel applications. The VisAD data model, described in Section 3, enables developers to define many different numerical data structures in terms of hierarchies built up from simple real numbers and text strings, and enables developers to attach various types of metadata to values at different levels in the hierarchy.

The integration of metadata could allow a developer to define a sophisticated type for 2-D image data as finite samplings of continuous functions from 2-D pixel locations, such as (line, element) or (latitude, longitude), to one or more pixel radiances. Image metadata may include units for location and radiance values (e.g., radians or degrees for latitude and longitude locations), sampling topologies and geometries for pixel locations (most images have rectangular topologies, rectangular geometries in (line, element) locations but curvilinear geometries in (latitude, longitude) locations), coordinate systems for pixel locations (images with (line, element) locations may specify mathematical transformations to (latitude, longitude) locations), missing radiance indicators, and error estimates for pixel radiances and locations. Developers also have the option to ignore most of these types of metadata, and implement images as simple arrays without units, coordinate transformations, missing data or error estimates, and sampled on rectangular integer lattices (i.e., pixels are addressed by integer line and element indices, much as they are in Fortran or C arrays).

The VisAD display model offers a similar reductionist approach. Developers define displays for complex numerical data objects in terms of mappings (the ScalarMap class) from their primitive real number elements (the RealType class) to the conceptual elements of displays (the DisplayRealType class). Developers can also attach various types of display metadata and interactive controls to these mappings. Developers may even define new kinds of display elements by defining new DisplayRealTypes. This is described in detail in Section 4.

Designing VisAD data types and displays is similar to designing database schemas and views. In fact, most of the differences between VisAD data types and database schemas can be traced to the fact that databases model discrete entities while numerical data are discrete approximations to continuous entities.

VisAD's reduction to elements is very powerful for adapting to new applications, but, like database schema design, can also be a challenge. The power comes from providing a context in which developers can answer questions like "What is the nature of an image?" However, an end user who merely wants to display an image should not have to first answer such questions. Thus VisAD user interfaces should present choices to end users in higher-level terms such as images, grids and tables. Of course, it is

possible to build user interfaces for VisAD that do defer such questions to end users, in order to give them the full power of the data model.

We also anticipate the development of intermediate class libraries between the core VisAD system and end user interfaces, which define higher-level application-specific data classes such as images, grids and tables. The methods of these higher-level data classes can encapsulate metadata manipulation in terms of higher-level data operations, including display methods that encapsulate manipulation of ScalarMaps from RealTypes to DisplayRealTypes. Such intermediate class libraries may simplify the task for those developing user interfaces for end users.

2.4 The Value of Integrated Metadata

The goal of integrating metadata is actually to create systems that enable end users to ignore metadata (but also to manipulate metadata if they wish to). For example, a user might read weather model output grids from several different models and several different file formats, each sampled at different map projections, at different vertical coordinate systems and at different time steps. The file format adapters will read each file into a VisAD Data object that includes the grid data and metadata objects containing the grid's spatial and temporal sampling information. Display objects will use these metadata objects to display the grid data co-located in space and time. Furthermore, arithmetical operations will also co-locate the data. For example, if temperatures from one model are subtracted from temperatures from another model, the temperatures from the second will be resampled to the spatial and temporal locations of the first before they are subtracted. If the two models use different temperature units, these will be converted before values are subtracted, and before they are displayed together.

Section 3.12 uses code examples to illustrate how VisAD can be used for simple array operations like those used in the C programming language, but can also be used for high-level operations on arrays of data that integrate metadata.

Users who want to control all aspects of their computations may do so by explicitly manipulating and extending the VisAD metadata classes. Note in particular Section 3.3 on Units, 3.4 on CoordinateSystems, and Section 3.5.1 on Defining Interpolation Algorithms by Extending the Set Class.

As the Internet enables greater data sharing among scientists, it increases the problems associated with metadata and file format differences among scientists. Metadata integration in a common data model is an important tool for addressing these problems, both for those users who want to ignore metadata and those who want to control metadata.

2.5 Toolkit for Designing Interaction Techniques

Interactivity is the key to understanding numerical data and computations. This has been the driving principal behind the development of Vis5D and VisAD. The most basic interaction mode is rotating 3-D scenes, which resolves the inherent ambiguity problem of 3-D graphics. That is, while 3-D graphics are more dramatic than 2-D graphics, they suffer from the problem that every point on a 2-D display screen or on the viewer's 2-D retinas corresponds to many points in the 3-D scene. Rotating the scene, whether in response to mouse movements for workstation displays or in response to head motion in immersive virtual reality displays, is the most effective way to resolve this ambiguity.

Once the necessary graphics speed is attained for interactive 3-D rotation, it can be exploited for all sorts of other interaction modes, such as dragging plane slices and other specialized graphics through data volumes, selecting various combinations of fields to visually compare, animating time dynamics, editing color maps, etc. VisAD supports all of these 'ordinary' graphical interaction modes when used with sufficiently fast graphics hardware.

When computations can also be done with fast response times, they may be coupled with interactive graphics to create an interaction mode known as 'computational steering'. By allowing Data, computational Cells, Displays and user interface components to be connected flexibly, VisAD supports computational steering interactions.

Beyond ordinary graphical interactions and computational steering, VisAD is designed to support a number of more sophisticated graphical interaction modes. These include:

1. Exploring visualization designs: experimenting with different ways to display the same data. VisAD allows users to determine how Data are depicted by defining a set of ScalarMaps from data primitives (i.e., RealTypes) to display primitives (i.e., DisplayRealTypes). Graphical user interfaces can be developed for defining ScalarMaps, enabling users to interactively experiment with display designs. For example, users might define ScalarMaps by dragging graphical icons representing RealTypes onto graphical icons representing DisplayRealTypes.

2. Direct manipulation: user interaction directly with data depictions. In particular VisAD allows users to modify Data values by re-drawing their depictions. While many ordinary graphical interactions have direct manipulation interfaces, they are usually not user-definable and have simple parameterizations in terms of one or a few real numbers. VisAD allows changes to larger Data objects to be connected through computational Cells and back to graphical Displays for more complex and user-defined graphical interactions.

3. Event driven computations and displays: re-computation and re-displays are

triggered by data changes resulting from user interactions or running simulations. This extends the business spread sheet from simple numbers to complex numerical Data objects and their interactive 3-D visualizations. VisAD's Data, Display and Cell classes provide the tools for building numerical spread sheets.

4. Remote collaboration: geographically remote users share visualizations and user interfaces as if sitting in front of the same workstation. VisAD allows multiple remote Displays to share connections to a common set of Data objects and computational Cells.

Given this variety of basic interaction modes, VisAD can be viewed as a toolkit for building interaction techniques, in the same way that it and other systems are toolkits for building visualizations. The building blocks for interaction techniques are events, Display controls, direct manipulation, computational Cells, and shared access to Data across the network. Sections 4.4.2 and 6.3 present interesting small examples of building interaction techniques.

3 Data Model

The VisAD data model was designed to support virtually any numerical data. Rather than providing a variety of specific data structures like images, grids and tables, the VisAD data model defines a set of classes that can be used to build any hierarchical numerical data structures.

Data objects all have a class in the class hierarchy under Data, and all define a hierarchical composition of complex Data objects from primitive Data objects. The primitive (scalar) Data classes are Real and Text. A Real object contains a real number value (i.e., a member of R, the set of all real numbers) represented by a Java double. A Text object contains a text string. Complex hierarchical Data objects are built from these primitives using the Tuple, Set and Function classes. A Tuple object contains a set of components whose number, sequence and type are fixed by the MathType of the Tuple. A Set object represents a set of points in an n-dimensional real vector space (denoted by $\mathbb{R}^n$). There are a great variety of ways of representing such Sets, as described in Section 3.5. Note that a Tuple with n Real components is a RealTuple and represents a single point in $\mathbb{R}^n$. A Function object represents a function from $\mathbb{R}^n$ to values of some specific type. Field is the subclass of Function for functions represented by finite sets of samples of function values (for example, a satellite image samples a continuous radiance function at a finite set of pixel locations). The Data classes implement methods for various binary and unary mathematical operations (e.g., add, multiply, sqrt), as well as specialized operations such as Function evaluation and Tuple component access. The Data class hierarchy is described in more detail in Section 3.2.

Data objects include metadata defined by the classes: MathType, Unit, CoordinateSystem, Set (function domain sampling), ErrorEstimate and AuditTrail, as well as missing data indicators. The details of these different forms of metadata are described in Sections 3.1 and 3.3 - 3.8. Metadata are integrated into mathematical and visualization operations. For example Unit conversions and CoordinateSystem transforms are done implicitly as needed in Data operations.

3.1 MathTypes

Numerical data objects are finite approximations to idealized mathematical objects such as real numbers, vectors, sets and functions. Thus every Data object has a MathType, which indicates the type of mathematical object that it approximates.

The MathType class hierarchy is:

```
MathType
  ScalarType
    RealType
    TextType
  TupleType
    RealTupleType
  SetType
  FunctionType
```

The starting point for any new application of VisAD is defining a set of MathTypes for the Data objects involved. This set of MathTypes provides a context for defining metadata, data displays, and data analysis operations. This is similar to the way that database schemas provide a context for defining database applications. Developers using the VisAD class library can think about MathTypes using the following shorthand syntax:

```
MathType       := ScalarType | TupleType | SetType | FunctionType
ScalarType     := RealType | TextType
RealType       := name
TextType       := name
TupleType      := ( MathType , MathType , ..., MathType )
TupleType      := RealTupleType
RealTupleType  := ( RealType , RealType , ..., RealType )
SetType        := set ( RealTupleType )
FunctionType   := ( RealTupleType -> MathType )
FunctionType   := ( RealType -> MathType )
```

where TupleType and RealTupleType each have at least one component. For example, a satellite image of Earth may be a finite sampling of a continuous function with MathType:

```
( (latitude, longitude) ->
  (radiance_channel_1, ..., radiance_channel_N) )
```

The output of a weather model may be described using the MathType:

```
( time -> ( (latitude, longitude, altitude) ->
  (temperature, pressure, dewpoint, wind_u, wind_v, wind_w) ) )
```

And a set of map boundaries may be described using the MathType:

```
set ( (latitude, longitude) )
```

Note that the prettyString method of MathType returns a String with this shorthand notation for any VisAD MathType. The static stringToType method of MathType takes a String argument, which is assumed to be in this shorthand notation, and returns the corresponding MathType (of course, MathTypes returned by stringToType do not include any non-null default Units, CoordinateSystems or Sets).

MathTypes are a form of metadata that describe data organization. For example, weather model output are often stored in files as independent 2-D grids, where any higher-level organization must be deduced by comparing the metadata associated with each grid. MathTypes provide a way to explicitly document such higher-level data organizations.

Every scalar (i.e., primitive) value occurring in a Data object is associated with a named ScalarType occurring in the Data object's MathType. These names are used to control how the Data object is displayed, as described in Section 4.1.

Some MathTypes include default values for various kinds of metadata, including Units (see Section 3.3), CoordinateSystems (see Section 3.4), and samplings (see Section 3.5). Although these defaults may be over-ridden for Data values, the defaults define equivalence classes of convertible Units and CoordinateSystems among Data values with the same MathTypes, with convertibility enforced by the system. Note that application developers may opt out of Units, CoordinateSystems and any other form of metadata by setting that form of metadata to null in MathType and Data object constructors (however, developers may not opt out of MathTypes and Field samplings, which are mandatory).

MathType is abstract and serializable. A MathType object can only be local (see Section 6 for more information). Its subclasses are all immutable.

3.1.1 RealType Constructors

RealType includes the following constructors:

Listing 3.1: RealType Constructors

```
/** name of type (two RealTypes are equal if their names are equal);
default Unit for values of this type and may be null; default Set
used when this type is a FunctionType domain and may be null */
public RealType(String name, Unit default_unit, Set default_set)
       throws VisADException;
/** name of type (two RealTypes are equal if their names are equal);
default Unit and Set are null */
public RealType(String name) throws VisADException;
```

3.1.2 TextType Constructor

TextType includes the following constructor:

Listing 3.2: TextType Constructors

```
/** name of type (two TextTypes are equal if their names are equal) */
public TextType(String name) throws VisADException;
```

3.1.3 TupleType Constructor

TupleType includes the following constructor:

Listing 3.3: TupleType Constructors

```
/** array of component types */
public TupleType(MathType[] types) throws VisADException;
```

3.1.4 RealTupleType Constructors

RealTupleType includes the following constructors:

Listing 3.4: RealTupleType Constructors

```
/** array of component types;
default CoordinateSystem for values of this type (including
Function domains) and may be null; default Set used when this
type is a FunctionType domain and may be null */
public RealTupleType(RealType[] types,
  CoordinateSystem default_coordinate_system, Set default_set)
  throws VisADException;

/** construct a RealTupleType with one component */
public RealTupleType(RealType a,
  CoordinateSystem default_coordinate_system, Set default_set)
  throws VisADException;

/** construct a RealTupleType with two components */
public RealTupleType(RealType a, RealType b,
  CoordinateSystem default_coordinate_system, Set default_set)
  throws VisADException;

/** construct a RealTupleType with three components */
public RealTupleType(RealType a, RealType b, RealType c,
  CoordinateSystem default_coordinate_system, Set default_set)
  throws VisADException;

/** construct a RealTupleType with four components */
```

```
public RealTupleType(RealType a, RealType b, RealType c, RealType d,
  CoordinateSystem default_coordinate_system, Set default_set)
  throws VisADException;

/** array of component types;
default CoordinateSystem and Set are null */
public RealTupleType(RealType[] types) throws VisADException;

/** construct a RealTupleType with one component */
public RealTupleType(RealType a) throws VisADException;

/** construct a RealTupleType with two components */
public RealTupleType(RealType a, RealType b) throws VisADException;

/** construct a RealTupleType with three components */
public RealTupleType(RealType a, RealType b, RealType c)
  throws VisADException;

/** construct a RealTupleType with four components */
public RealTupleType(RealType a, RealType b, RealType c, RealType d)
  throws VisADException;
```

3.1.5 FunctionType Constructor

FunctionType includes the following constructor:

Listing 3.5: FunctionType Constructors

```
/** domain must be a RealType or a RealTupleType;
range may be any MathType */
public FunctionType(MathType domain, MathType range)
  throws VisADException;
```

3.1.6 SetType Constructor

SetType includes the following constructor:

Listing 3.6: SetType Constructors

```
/** domain must be a RealType or a RealTupleType */
public SetType(MathType domain) throws VisADException;
```

3.1.7 MathType Methods

Generally useful MathType methods include:

Listing 3.7: MathType Constructors

```
/** returns a missing Data object for any MathType */
public Data missingData() throws VisADException;

/** return a String that indents complex MathTypes
for human readability */
public String prettyString();

/** return an array of ScalarMaps that is an "intuitive"
guess at a good way to visualize this MathType;
returns null if it can't make a good guess */
public ScalarMap[] guessMaps(boolean threeD);

/** ScalarTypes are equal if they have the same name;
TupleTypes are equal if their components are equal;
FunctionTypes are equal if their domains and ranges
are equal */
public boolean equals(Object obj);

/** this is useful for determining compatibility of
Data objects for binary mathematical operations;
any RealTypes are equal; any TextTypes are equal;
TupleTypes are equal if their components are equal;
FunctionTypes are equal if their domains and ranges
are equal */
public boolean equalsExceptName(MathType type);

/** create a MathType from its string represnetation;
essentially the inverse of the prettyString method */
public static MathType stringToType(String s) throws VisADException;
```

3.1.8 ScalarType Methods

Generally useful ScalarType methods include:

Listing 3.8: ScalarType Methods

```
public String getName();
```

3.1.9 RealType Methods

Generally useful RealType methods include:

Listing 3.9: RealType Methods

```
/** return any RealType constructed in this JVM with name,
or null */
public static RealType getRealTypeByName(String name);
```

```
/** get default Unit */
public Unit getDefaultUnit();

/** get default Set*/
public Set getDefaultSet();

/** this is a violation of MathType immutability to allow a
a RealType to be an argument (directly or through a
SetType) to the constructor of its default Set;
this method throws an Exception if getDefaultSet has
previously been invoked */
public void setDefaultSet(Set set) throws VisADException;
```

3.1.10 TupleType Methods

Generally useful TupleType methods include:

Listing 3.10: TupleType Methods

```
/** return number of components */
public int getDimension();

/** return component for index between 0 and getDimension() - 1 */
public MathType getComponent(int index) throws VisADException;

/** return index of first component with type;
if no such component, return -1 */
public RealType getIndex(MathType) throws VisADException;

/** return index of first RealType component with name;
if no such component, return -1 */
public RealType getIndex(String name) throws VisADException;
```

3.1.11 RealTupleType Methods

Generally useful RealTupleType methods include:

Listing 3.11: RealTupleType Methods

```
/** get default Units of RealType components */
public Unit[] getDefaultUnits();

/** get default CoordinateSystem */
public CoordinateSystem getCoordinateSystem()

/** get default Set*/
public Set getDefaultSet();

/** this is an unavoidable violation of MathType immutability -
a RealTupleType must be an argument (directly or through a
```

```
SetType) to the constructor of its default Set;
this method throws an Exception if getDefaultSet has
previously been invoked */
public void setDefaultSet(Set set) throws VisADException;
```

3.1.12 FunctionType Methods

Generally useful FunctionType methods include:

Listing 3.12: FunctionType Methods

```
/** if the domain passed to constructor was a RealType,
getDomain returns a RealTupleType with that RealType
as its single component */
public RealTupleType getDomain();

public MathType getRange();
```

3.1.13 SetType Methods

Generally useful SetType methods include:

Listing 3.13: SetType Methods

```
/** if the domain passed to constructor was a RealType,
getDomain returns a RealTupleType with that RealType
as its single component */
public RealTupleType getDomain();
```

3.1.14 Application Example: Synthesizing MathTypes

Applications that construct Data objects from numerical values they compute generally need to synthesize MathTypes from their RealType components. Here's a sample of code for synthesizing a MathType appropriate for a Vis5D data set (this is roughly the inverse of the code in Section 3.1.15):

Listing 3.14: Application Example: Synthesizing MathTypes

```
// construct RealType components for grid coordinates
RealType row = new RealType("row", null, null);
RealType column = new RealType("column", null, null);
RealType level = new RealType("level", null, null);
```

```
// construct RealTupleType for grid coordinates
RealType[] types3d = {row, column, level};
RealTupleType domain = new RealTupleType(types3d);

// construct RealType components for grid fields
RealType temperature = new RealType("temperature", null, null);
RealType pressure = new RealType("pressure", null, null);
RealType water_vapor = new RealType("water_vapor", null, null);

// construct RealTupleType for grid fields
RealType[] field3d = {temperature, pressure, water_vapor};
RealTupleType range = new RealTupleType(field3d);

// construct FunctionType for grid
FunctionType grid_type = new FunctionType(domain, range);

// construct RealType and RealTupleType for time domain
RealType time = new RealType("time", null, null);
RealTupleType time_type = new RealTupleType(time);

// construct FunctionType for time sequence of grids
FunctionType vis5d_type = new FunctionType(time_type, grid_type);
```

3.1.15 Application Example: Analyzing MathTypes

Applications that get Data objects from file format adapters (described in Section 7) generally need to analyze MathTypes to extract their RealType components. The Vis5DForm class adapts Data objects from Vis5D files, whose MathTypes have the general form:

```
(time -> ((row, column, level) -> (field1, field2, ..., fieldN)))
```

That is, they are time sequences of multivariate 3-D grids. Here's a sample of MathType analysis code (this is roughly the inverse of the code in Section 3.1.14):

Listing 3.15: Application Example: Analyzing MathTypes

```
// get the MathType of a Data object named 'vis5d'
FunctionType vis5d_type = (FunctionType) vis5d.getType();

// extract time, the domain of the FunctionType
RealType time = (RealType) vis5d_type.getDomain().getComponent(0);

// get grid_type, itself a FunctionType and the range of the
// vis5d_type FunctionType
FunctionType grid_type = (FunctionType) vis5d_type.getRange();

// get the grid domain RealTupleType
RealTupleType domain = grid_type.getDomain();

// get the grid domain component RealType - they are grid coordinates
RealType row = (RealType) domain.getComponent(0);
```

```
RealType column = (RealType) domain.getComponent(1);
RealType level = (RealType) domain.getComponent(2);

// get the grid range - it is a RealTupleType of fields
RealTupleType range = (RealTupleType) grid_type.getRange();

// get the number of grid range components
int dim = range.getDimension();

// construct an array to hold the grid range RealTypes
RealType[] range_types = new RealType[dim];

// get the grid range RealTypes
for (int i=0; i<dim; i++) {
range_types[i] = (RealType) range.getComponent(i);
}
```

3.2 Data Class Hierarchy

The Data hierarchy is:

```
Data
  Scalar
    Real
    Text
  Tuple
    RealTuple
  Set
    (there is a large hierarchy under Set as described in Section 3.5)
  Function
    Field
      FlatField
```

To some extent the Data hierarchy mirrors the MathType hierarchy. However, it is important to note that MathType is not a synonym for Data class, since Data classes may be elaborated into different forms of finite representation of the corresponding MathTypes. For example, Set is elaborated into a large number of different ways of representing subsets of $\mathbb{R}^n$. Similarly, Function is elaborated into Field, for functions represented by finite samplings, and FlatField, for Fields with simple range values that can be represented by small numbers of Java's primitive data types rather than by objects. Developers may extend the Data classes to define new forms of representation. For example, a developer could extend Real to define a representation by ratios of infinite- precision integers rather than the Java primitive double used by Real (doubles are used by Real because experience has shown that using floats as the default can

cause round-off problems that are extremely difficult for application developers to detect and diagnose).

The Data hierarchy is also elaborated for various data storage locations and formats. Section 6 describes how the hierarchy for Data and other VisAD classes is structured for local and remote objects, and Section 7 describes how the Data class hierarchy is adapted to import data from various file formats. The Data hierarchy is being adapted to netCDF, HDF and FITS files, and developers may extend this to other file formats. Thus data are accessible via the VisAD Data API (Application Programming Interface) independent of storage location, file format and approximating representation.

The metadata classes described in Sections 3.1 and 3.3 - 3.8 define how Data objects approximate mathematical objects and how they model the world.

Data is an interface that may apply to both local and remote Data objects. DataImpl is an abstract class that only applies to local Data objects, and RemoteData is an interface that only applies to remote Data objects (see Section 6 for more information). DataImpl is cloneable and serializable. All of its subclasses except FieldImpl and FlatField are immutable. API documentation for the Set class hierarchy is described in Section 3.5 and for FlatFields is described in Section 3.9, rather than here.

3.2.1 Real Constructors

Real includes the following constructors:

Listing 3.16: The Real Constructors

```
/** unit and error may be null */
public Real(RealType type, double value, Unit unit, ErrorEstimate error)
  throws VisADException;

/** use RealType.Generic */
public Real(double value)
```

3.2.2 Text Constructor

Text includes the following constructor:

Listing 3.17: The Text Constructors

```
public Text(TextType type, String value) throws VisADException;

/** use TextType.Generic */
public Text(String value)
```

3.2.3 Tuple Constructors

Tuple includes the following constructors:

Listing 3.18: The Tuple Constructors

```
/** this constructs its MathType from the MathTypes of the
data array; components are copies of data */
public Tuple(Data[] data) throws VisADException, RemoteException;

/** only copy data if copy == true */
public Tuple(Data[] data, boolean copy)
throws VisADException, RemoteException;
```

3.2.4 RealTuple Constructors

RealTuple includes the following constructors:

Listing 3.19: The RealTuple Constructors

```
/** coordinate_system may be null; otherwise
coordinate_system.getReference() must equal
type.getCoordinateSystem.getReference() */
public RealTuple(RealTupleType type, Real[] reals, CoordinateSystem ↩
    coordinate_system)
  throws VisADException, RemoteException;

public RealTuple(Real[] reals) throws VisADException, RemoteException;
```

3.2.5 Field Constructors

Field is an interface implemented by FieldImpl for local Fields and RemoteFieldImpl for remote Fields. See Section 6 for more information about distributed computing. These classes have the following constructors:

Listing 3.20: The Field Constructors

```
/** FieldImpl is the most general sampled function;
domain_set defines the domain sampling;
if it is null, use the default Set of type.getDomain();
domain_set defines the Units and CoordinateSystem
of the Field domain */
public FieldImpl(FunctionType type, Set domain_set)
  throws VisADException;

/** use the default Set of type.getDomain() */
public FieldImpl(FunctionType type) throws VisADException;
```

```
/** construct a RemoteFieldImpl object to provide remote
access to field */
public RemoteFieldImpl(FieldImpl field)
  throws VisADException, RemoteException;
```

3.2.6 Data Methods

A Data object may be either local or remote, a DataImpl object may only be local and a RemoteData object may only be remote (see Section 6 for more information). The methods in this section define the universal operations applicable to all Data objects: getType returns a Data object's MathType, isMissing indicates whether the Data object has missing value (but note that even is a Data object is not missing, it may still have sub-objects with missing values), and local replaces a RemoteData object with a local DataImpl copy.

The binary and unary methods define basic mathematical operations on Data that are the building blocks for data analysis using VisAD. The binary and unary methods have wrapper methods for specific operations like add and sin. These operations are defined point-by-point for Tuple and Function Data objects, so that for example, the sin of a Function is a Function whose values are the sines of the original Function's values.

When add (or any other binary operation) is applied to two Fields the result is a Field whose values are the sums (or other operation) of the values of the two Functions, but only if the MathTypes of the two Fields match. MathType matching is defined recursively on TupleTypes and FunctionTypes in terms of their components, any RealType matches any RealType, and any TextType matches any TextType (thus matching Functions must have domains with the same dimension).

Most important, binary and unary operations on Data objects involve their metadata. When two Fields are added, the domain samples of one are resampled to the domain samples of the other, including any necessary Unit conversions of Real components of the domains and any necessary CoordinateSystem transformations between RealTuple domains. The range values of one Field are estimated at the domain sample locations of the other Field using either nearest neighbor or weighted average algorithms, as specified in the optional resampling_mode argument to binary methods. Unit conversions and CoordinateSystem transformations are also applied as needed to range values of Fields before they are added. Furthermore, ErrorEstimates attached to Field range values are modified to reflect error effects of binary and unary operations. ErrorEstimate propagation may assume either that operand errors are independently or dependently distributed, or ErrorEstimate propagation may be disabled, using the error_mode argument to binary and unary methods.

In some cases Data objects may be combined in binary operations even if their

MathTypes do not match. For example, a Real object may be combined with any other Data object, and a Functions may be combined with Data objects that match the MathType of the Function's range.

Listing 3.21: The MathType Constructors

```
public MathType getType()
throws VisADException, RemoteException;

/** flag indicating whether Data object has missing value */
public boolean isMissing()
throws VisADException, RemoteException;

/** if remote, return a local copy;
if local, return this */
public DataImpl local()
throws VisADException, RemoteException;

/** general binary operation between this and data; operation may
be Data.ADD, Data.SUBTRACT, etc; these include all binary
operations defined for Java primitive data types; new_type
is the MathType of the result; sampling_mode may be
Data.NEAREST_NEIGHBOR or Data.WEIGHTED_AVERAGE; error_mode
may be Data.INDEPENDENT, Data.DEPENDENT or Data.NO_ERRORS */
public Data binary(Data data, int operation, MathType new_type,
int sampling_mode, int error_mode)
throws VisADException, RemoteException;

/** like previous signature of binary, except the result takes
the MathType of this unless the default Units of that MathType
conflict with Units of the result, in which case a generic
MathType with appropriate Units is constructed */
public Data binary(Data data, int operation, int sampling_mode,
int error_mode)
throws VisADException, RemoteException;

public Data add(Data data, int sampling_mode, int error_mode)
  throws VisADException, RemoteException;

/** use Data.NEAREST_NEIGHBOR and Data.NO_ERRORS */
public Data add(Data data) throws VisADException, RemoteException;

public Data subtract(Data data, int sampling_mode, int error_mode)
throws VisADException, RemoteException;

/** use Data.NEAREST_NEIGHBOR and Data.NO_ERRORS */
public Data subtract(Data data) throws VisADException, RemoteException;

/** similar methods are defined for the following binary operators:
multiply, divide, pow, max, min, atan2, atan2Degrees and
remainder */

/** general unary operation; operation may be Data.ABS, Data.ACOS, etc;
these include all unary operations defined for Java primitive data
types; new_type is the MathType of the result; sampling_mode may be
Data.NEAREST_NEIGHBOR or Data.WEIGHTED_AVERAGE; error_mode may be
Data.INDEPENDENT, Data.DEPENDENT or Data.NO_ERRORS */
public Data unary(int operation, MathType new_type, int sampling_mode,
```

```
int error_mode)
throws VisADException, RemoteException;

/** like previous signature of unary, except the result takes
the MathType of this unless the default Units of that MathType
conflict with Units of the result, in which case a generic
MathType with appropriate Units is constructed */
public Data unary(int operation, int sampling_mode, int error_mode)
throws VisADException, RemoteException;

/** clone this Data object except give it new_type */
public Data changeMathType(MathType new_type)
throws VisADException, RemoteException;

public Data abs(int sampling_mode, int error_mode)
throws VisADException, RemoteException;

/** use Data.NEAREST_NEIGHBOR and Data.NO_ERRORS */
public Data abs() throws VisADException, RemoteException;

public Data acos(int sampling_mode, int error_mode)
throws VisADException, RemoteException;

/** use Data.NEAREST_NEIGHBOR and Data.NO_ERRORS */
public Data acos() throws VisADException, RemoteException;

/** similar methods are defined for the following unary operators:
acosDegrees, asin, asinDegrees, atan, atanDegrees, ceil, cos,
cosDegrees, exp, floor, log, rint, round, sin, sinDegrees,
sqrt, tan, tanDegrees, negate */
```

3.2.7 Real Methods

A Real object may only be local. Binary operations may be performed between a Real and any other Data object that does not contain Text components; such operations are applied independently with each Real component. Generally useful Real methods include:

Listing 3.22: The Real Methods

```
public final double getValue();

/** get double value converted to unit */
public final double getValue(Unit unit) throws VisADException;

public Unit getUnit();

public ErrorEstimate getError();
```

3.2.8 Text Methods

Text may only be local. The only binary operation that works for Text is `Data.ADD`, which is interpreted as string concatenation. No unary operations work for Text. Generally useful Text methods include:

Listing 3.23: The text methods

```
public String getValue();
```

3.2.9 Tuple Methods

A Tuple object may only be local. Generally useful Tuple methods include:

Listing 3.24: The tuple methods

```
/** return number of components */
public int getDimension();

/** return component for index between 0 and getDimension() - 1 */
public MathType getComponent(int index) throws VisADException;

/** construct Tuple; used for constructing Tuples in Spreadsheet;
e.g., link(visad.Tuple.makeTuple(A2, B1, B2)) */
public static Tuple makeTuple(Data[] datums)
throws VisADException, RemoteException
```

3.2.10 RealTuple Methods

A RealTuple object may only be local. Generally useful RealTuple methods include:

Listing 3.25: The RealTuple methods

```
/** get Units of Real components */
public Unit[] getTupleUnits();

/** get ErrorEstimates of Real components */
public ErrorEstimate[] getErrors() throws VisADException;

/** get CoordinateSystem */
public CoordinateSystem getCoordinateSystem();
```

3.2.11 Function Methods

A Function object may be either local or remote, a FunctionImpl object may only be local and a RemoteFunction object may only be remote (see Section 6 for more information). Generally useful Function methods are listed below. Note in particular the resample method which is invoked implicitly for many visualization and mathematical operations on Functions and can be invoked by applications for image remapping and a variety of similar Function operations.

Listing 3.26: The Function methods

```
/** get dimension of Function domain */
public int getDomainDimension()
throws VisADException, RemoteException;

/** get Units of domain Real components */
public Unit[] getDomainUnits()
throws VisADException, RemoteException;

/** get domain CoordinateSystem */
public CoordinateSystem getDomainCoordinateSystem()
throws VisADException, RemoteException;

/** evaluate Function at domain_value, for 1-D domains */
public Data evaluate(Real domain_value, int sampling_mode,
int error_mode)
throws VisADException, RemoteException;

/** evaluate Function at domain_value, for 1-D domains,
using Data.NEAREST_NEIGHBOR and Data.NO_ERRORS */
public Data evaluate(Real domain_value)
throws VisADException, RemoteException;

/** evaluate Function at domain_value */
public Data evaluate(RealTuple domain_value, int sampling_mode,
int error_mode)
throws VisADException, RemoteException;

/** evaluate Function at domain_value using
Data.NEAREST_NEIGHBOR and Data.NO_ERRORS */
public Data evaluate(RealTuple domain_value)
throws VisADException, RemoteException;

/** return a Field of Function values at samples in set;
this combines unit conversions, coordinate transforms,
resampling and interpolation */
public Field resample(Set set, int sampling_mode, int error_mode)
throws VisADException, RemoteException;

/** return the derivative of this Function with respect to d_partial;
d_partial may occur in this Function's domain RealTupleType, or,
if the domain has a CoordinateSystem, in its Reference
RealTupleType; propogate errors according to error_mode */
public abstract Function derivative(RealType d_partial,
int error_mode) throws VisADException, RemoteException;
```

```
/** return the derivative of this Function with respect to d_partial;
set result MathType to derivType; d_partial may occur in this
Function's domain RealTupleType, or, if the domain has a
CoordinateSystem, in its Reference RealTupleType;
propogate errors according to error_mode */
public abstract Function derivative(RealType d_partial,
MathType derivType, int error_mode)
throws VisADException, RemoteException;

/** return the tuple of derivatives of this Function with respect to
all RealType components of its domain RealTupleType;
propogate errors according to error_mode */
public abstract Data derivative(int error_mode)
throws VisADException, RemoteException;

/** return the tuple of derivatives of this Function with respect
to all RealType components of its domain RealTupleType;
set result MathTypes of tuple components to derivType_s;
propogate errors according to error_mode */
public abstract Data derivative(MathType[] derivType_s,
int error_mode) throws VisADException, RemoteException;

/** return the tuple of derivatives of this Function with respect
to the RealTypes in d_partial_s; the RealTypes in d_partial_s
may occur in this Function's domain RealTupleType, or, if the
domain has a CoordinateSystem, in its Reference RealTupleType;
set result MathTypes of tuple components to derivType_s;
propogate errors according to error_mode */
public abstract Data derivative(RealTuple location,
RealType[] d_partial_s, MathType[] derivType_s, int error_mode)
throws VisADException, RemoteException;
```

3.2.12 Field Methods

A Field object may be either local or remote, a FieldImpl object may only be local and a RemoteField object may only be remote (see Section 6 for more information). Generally useful Field methods include:

Listing 3.27: The Field methods

```
/** set the values of the Field (at the domain Set samples)
using the values in range (the length of range must
equal the length of the domain Set);
make copies of range values if copy is true */
public void setSamples(Data[] range, boolean copy)
throws VisADException, RemoteException;

/** get the domain Set */
public Set getDomainSet()
throws VisADException, RemoteException;

/** get the Units of the Real components of the domain Set */
public Unit[] getDomainUnits()
throws VisADException, RemoteException;
```

```
/** get the CoordinateSystem of the domain Set */
public CoordinateSystem getDomainCoordinateSystem()
throws VisADException, RemoteException;

/** get the Field value at the index-th sample in the
domain Set */
public Data getSample(int index)
throws VisADException, RemoteException;

/** get the 'Flat' components of this Field's range values
in their default range Units (as defined by the range of
the Field's FunctionType); if the range type is a RealType
it is a 'Flat' component, if the range type is a TupleType
its RealType components and RealType components of its
RealTupleType components are all 'Flat' components; the
return array is dimensioned:
double[number_of_flat_components][number_of_range_samples] */
public double[][] getValues()
throws VisADException, RemoteException;

/** set Field value at the index-th sample in the
domain Set, to range */
public void setSample(int index, Data range)
throws VisADException, RemoteException;

/** set Field value at the sample in the domain Set nearest
domain, to range */
public void setSample(RealTuple domain, Data range)
throws VisADException, RemoteException;

/** return an Enumeration of RealTuple values in domain Set */
public Enumeration domainEnumeration()
throws VisADException, RemoteException;

/** return true is this is a FlatField */
public boolean isFlatField();

/** assumes the range type of this is a Tuple and returns
a Field with the same domain as this, but whose range
samples consist of the specified Tuple component of the
range samples of this; in shorthand, this[].component */
public Field extract(int component)
throws VisADException, RemoteException;

/** combine domains of two outpost nested Fields into a single
domain and Field; for examples transform the MathType
(a -> ((b, c) -> d)) into ((a, b, c) -> d) */
public Field domainMultiply()
throws VisADException, RemoteException;

/** factor Field domain into domains of two nested Fields (with
factor as outer domain); for examples transform the MathType
((a, b, c) -> d) into (a -> ((b, c) -> d)) (where factor = a) */
public Field domainFactor(RealType factor)
throws VisADException, RemoteException;
\end{environment-name}

\subsection{FieldImpl Method}
This describes a single static method of FieldImpl:
```

```
\begin{lstlisting}[
caption={[FieldImpl methods]The FieldImpl methods},
label=code:fieldImplMethods,
]
/** resample all elements of the fields array to the domain
set of fields[0], then return a Field whose range samples
are Tuples merging the corresponding range samples from
each element of fields; if the range of fields[i] is a
Tuple without a RangeCoordinateSystem, then each Tuple
component of a range sample of fields[i] becomes a
Tuple component of a range sample of the result -
otherwise a range sample of fields[i] becomes a Tuple
component of a range sample of the result; this assumes
all elements of the fields array have the same domain
dimension */
public static Field combine(Field[] fields)
throws VisADException, RemoteException;
```

3.2.13 Application Example: Synthesizing Fields

In this example we assume that:

```
grid_type = ((row, column, level) -> (temperature, pressure, water_vapor))
```

and:

```
vis5d_type = (time -> grid_type)
```

These are the types appropriate for Vis5D data sets synthesized by the example in Section 3.1.14. This example includes constructors for an Integer3DSet and an Integer1DSet, which are described in detail in Section 3.5.3.3, and a constructor for a FlatField, which is an efficient sub-class of FieldImpl described in Section 3.9. The Integer3DSet is an integer lattice of 50 by 50 by 20 points for a Vis5D grid, and the Integer1DSet is a sequence of hour values from 0 to 23. FlatField includes a version of the setSamples method that takes an array of floats, in addition to the version of setSamples inherited from FieldImpl that takes an array of Data objects. Here's a sample of code for synthesizing a FieldImpl appropriate for a Vis5D data set:

Listing 3.28: Synthesizing a FieldImpl appropriate for a Vis5D data set

```
// construct an integer 3-D grid
Set grid_set = new Integer3DSet(50, 50, 20);

// construct a sequence of 24 hours
Set time_set = new Integer1DSet(24);

// construct a FieldImpl for a time sequence of grids
FieldImpl vis5d = new FieldImpl(vis5d_type, time_set);
```

```
for (int i=0; i<24; i++) {
// conbstruct a FlatField for the i-th time step
FlatField grid = new FlatField(grid_type, grid_set);

// construct an array to hold the gridded field values;
// data[0] is an array of temperatures, data[1] an array
// of pressures, and data[2] an array of water_vapors
float[][] data = new float[3][50 * 50 * 20];

// ... code to set data values ...

// set the data values into the grid
grid.setSamples(data);

// set grid as the i-th time sample of vis5d
vis5d.setSample(i, grid);
}
```

3.3 Units

The Unit class defines units for Real values in terms of a user-extensible list of BaseUnits and associated physical quantities. The system-intrinsic list is:

```
ampere    electric current
candela   luminous intensity
kelvin    temperature
kilogram  mass
meter     length
second    time
mole      amount of substance
radian    angle
```

A Unit is defined by a set of BaseUnits with associated integer exponents, plus a real coefficient and offset. For example, yard = 0.9144 x meter, fahrenheit = (1 / 1.8) x kelvin + 459.67, and joule = kilogram x meter x second^{-2}. Two Units are convertible if they have the same set of BaseUnits and integer exponents, or if the exponents of one are negatives of the exponents of the other.

Units with non-zero offsets are dangerous. For example, the conversion of fahrenheit temperature differences to kelvin differences is not correct unless the offset is ignored. In order to avoid this problem, arithmetic operations implicitly convert all inputs to Units with zero offsets.

3.3.1 Unit Methods

Unit is abstract and serializable. A Unit object can only be local (see Section 6 for more information). Its subclasses are all immutable. Applications do not invoke Unit constructors explicitly. Rather they derive new Units be invoking methods of existing Units, or they create new BaseUnits by invoking a static factory method in BaseUnit. Generally useful Unit methods include:

Listing 3.29: The Unit methods

```
/** create a new Unit by raising this (which may not include
an offset) to power */
public Unit pow(int power) throws UnitException;

/** create a new Unit by multiplication by amount;
for example, Unit yard = meter.scale(0.9144); */
public Unit scale(double amount) throws UnitException;

/** create a new Unit by adding offset;
for example, Unit celsius = kelvin.shift(273.15); */
public Unit shift(double offset) throws UnitException;

/** create a new Unit by multiplying this (which may not
include an offset) by that */
public Unit multiply(Unit that) throws UnitException;

/** create a new Unit by dividing this (which may not
include an offset) by that */
public Unit divide(Unit that) throws UnitException;
```

3.3.2 SI Variables

The system intrinsic BaseUnits are defined in the SI class as follows:

```
BaseUnit SI.ampere;
BaseUnit SI.candela;
BaseUnit SI.kelvin;
BaseUnit SI.kilogram;
BaseUnit SI.meter;
BaseUnit SI.second;
BaseUnit SI.mole;
BaseUnit SI.radian;
```

3.3.3 BaseUnit Methods

Generally useful BaseUnit methods include:

Listing 3.30: The BaseUnit methods

```
/** create a new BaseUnit with the given quantityName and
unitName */
public static BaseUnit addBaseUnit(String quantityName,
String unitName) throws UnitException;

/** return any baseUnit created in this JVM with the given
unitName */
public static baseUnit unitNameToUnit(String unitName)

/** return any baseUnit created in this JVM with the given
quantityName */
public static baseUnit quantityNameToUnit(String quantityName)
```

3.3.4 CommonUnit Variables

The CommonUnit class defines commonly used Units, including:

Listing 3.31: The CommonUnit Variables

```
Unit CommonUnit.degree;
Unit CommonUnit.radian;
Unit CommonUnit.second;
/** all BaseUnits have exponent zero in dimensionless */
Unit CommonUnit.dimensionless;
/** promiscuous is compatible with any Unit; useful for constants;
not the same as null Unit, which is only compatible with
other null Units */
Unit CommonUnit.promiscuous;
```

3.4 CoordinateSystems

CoordinateSystem is an abstract class whose sub-classes define invertable transformations of the form $\mathbb{R}^n$ <—> $\mathbb{R}^n$ between values of various RealTupleTypes. A CoordinateSystem always refers to its reference RealTupleType. On the other hand, a RealTupleType might or might not refer to a default CoordinateSystem. Consequently, a RealTupleType can be one of three kinds with respect to CoordinateSystems:

1. Reference: the RealTupleType doesn't refer to a default CoordinateSystem but a CoordinateSystem refers to the RealTupleType.
2. Equivalent: the RealTupleType refers to a default CoordinateSystem and, thus, refers indirectly to a reference RealTupleType.
3. Uninvolved: the RealTupleType neither refers to a default CoordinateSystem nor is referred to by a CoordinateSystem.

Thus CoordinateSystems define equivalence classes of those RealTupleTypes with the same reference. For example, (polar_stereographic_row, polar_sterographic_column), (lambert_conformal_row, lambert_conformal_column) and other map projections could form an equivalence class relative to, and including, the Reference (latitude, longitude). Each of the map projections would include a default CoordinateSystem that defined its mathematical transformation between (row, column) and (latitude, longitude).

The default CoordinateSystem defined by a RealTupleType can be over-ridden for RealTuple values of that type, in order to support data-dependent CoordinateSystems. For example, meteorologists use (latitude, longitude, pressure) as a CoordinateSystem with Reference (latitude, longitude, altitude), where the mathematical transformation can vary depending on the vertical distribution of pressures. A default CoordinateSystem can only be over-ridden by a CoordinateSystem with the same Reference.

3.4.1 CoordinateSystem Constructors

CoordinateSystem is abstract and serializable. A CoordinateSystem object can only be local (see Section 6 for more information). Applications generally do not invoke CoordinateSystem methods, but they construct new CoordinateSystem objects and define new CoordinateSystem subclasses.

Note that care should be taken to make sure that:

1. The order of RealType components in a reference RealTupleType is consistent with the computations of the toReference and fromReference methods.
2. The Units of the RealType components in a reference RealTupleType are consistent with the values assumed by the toReference and fromReference methods.
3. The order of RealType components of a RealTupleType with a CoordinateSystem is consistent with the computations of the toReference and fromReference methods.

The constructor for the abstract CoordinateSystem class is:

Listing 3.32: The Abstract CoordinateSystem Constructor

```
/** user-defined subclasses must supply reference and units */
public CoordinateSystem(RealTupleType reference, Unit[] units)
  throws VisADException;
```

Constructors for specific CoordinateSystems included with VisAD include:

Listing 3.33: Some concrete CoordinateSystem Constructors

```
/** construct a CoordinateSystem for (latitude, longitude,
radius) relative to a 3-D Cartesian reference;
this constructor supplies units =
{CommonUnit.Degree, CommonUnit.Degree, null} to the super
constructor, in order to ensure Unit compatibility with its
use of trigonometric functions */
public SphericalCoordinateSystem(RealTupleType reference)
  throws VisADException;

/** construct a CoordinateSystem for (longitude, radius)
relative to a 2-D Cartesian reference;
this constructor supplies units = {CommonUnit.Degree, null}
to the super constructor, in order to ensure Unit
compatibility with its use of trigonometric functions */
public PolarCoordinateSystem(RealTupleType reference)
  throws VisADException;

/** construct a CoordinateSystem that whose transforms invert
the transforms of inverse (i.e., toReference and
fromReference are switched); for example, this could be
used to define Cartesian coordinates relative to a
refernce in spherical coordinates */
public InverseCoordinateSystem(RealTupleType reference, CoordinateSystem ↩
    inverse)
  throws VisADException;

/** construct a CoordinateSystem for grid coordinates (e.g.,
(row, column, level) in 3-D) relative to the value space
of set; for example, if satellite pixel locations are
defined by explicit latitudes and longitude, these could
be used to construct a Gridded2DSet which could then be
used to construct a GridCoordinateSystem for (ImageLine,
ImageElement) coordinates relative to reference coordinates
(Latitude, Longitude) */
public GridCoordinateSystem(GriddedSet set)
  throws VisADException;
```

3.4.2 CoordinateSystem Methods

Extensions of CoordinateSystem must implement the following methods:

Listing 3.34: The CoordinateSystem methods

```
/** convert RealTuple values to Reference coordinates;
for efficiency, input and output values are passed as
double[][][] arrays rather than RealTuple[] arrays; the
array indexes are:
double[tuple_dimension][number_of_tuples] */
public double[][][] toReference(double[][][] tuples)
  throws VisADException;

/** convert RealTuple values from Reference coordinates */
public double[][][] fromReference(double[][][] tuples)
  throws VisADException;
```

The following methods are implemented in CoordinateSystem in terms of the above methods, but for efficiency's sake extensions of CoordinateSystem may override those with direct implementations:

Listing 3.35: The methods, a concrete CoordinateSystem class may override

```
public float[][][] toReference(float[][][] tuples)
throws VisADException;

public float[][][] fromReference(float[][][] tuples)
throws VisADException;
```

3.5 Sets

A Field object approximates a function by interpolating its values at a finite subset of its domain [3]. A Field object includes a Set object that defines the finite sampling of the function's domain. This Set object also defines the CoordinateSystem of the Field's domain and the Units of the domain's RealType components. The Set class has many sub-classes for different ways of defining finite subsets of the Set's domain $\mathbb{R}^n$ (n is called the domain dimension of the Set). A partial Set class hierarchy is:

```
Set
  SimpleSet
    DoubleSet
    FloatSet
    SampledSet
      ProductSet
      UnionSet
      GriddedSet
        LinearNDSet
          IntegerNDSet
        Gridded1DSet
          Linear1DSet
            Integer1DSet
          Gridded1DDoubleSet
        Gridded2DSet
          Linear2DSet
            LinearLatLonSet
            Integer2DSet
          Gridded2DDoubleSet
        Gridded3DSet
```

```
      Linear3DSet
        Integer3DSet
      Gridded3DDoubleSet
  IrregularSet
    Irregular1DSet
    Irregular2DSet
    Irregular3DSet
```

A SimpleSet is embedded on a sub-manifold of dimension m in $\mathbb{R}^n$ (m is called the manifold dimension of the Set). A DoubleSet with domain dimension n is just the large but finite set of values in $\mathbb{R}^n$ representable by n IEEE double precision floating point values. Similarly for FloatSet and single precision. The SampledSet class implements some common methods for its subclasses. The samples of a GriddedSet are organized in an m-dimensional grid. For a LinearSet this grid is aligned to the axes of the domain $\mathbb{R}^n$ and for an IntegerSet the grid points form an integer lattice based at the origin. The samples of an IrregularSet are not organized. ProductSets and UnionSets allow Sets to be defined as products and unions of other Sets.

Note that Set is a sub-class of Data, so Sets are full-fledged Data objects in addition to being a form of metadata for Fields. For example, a set of map boundaries would be a Set with domain dimension n = 2 and manifold dimension m = 1.

Attention 1 (Possible class name conflict) *Note also that there is a Set class in the java.util package as of JDK 1.2. Thus applications should avoid combining* `import visad.*;` *with* `import java.util.*;`.

3.5.1 Defining Interpolation Algorithms by Extending the Set Class

The resample method of the Field class is the workhorse of the system. It takes a Set as an argument and returns a new Field containing values of the original Field sampled at the Set locations. It also does any necessary Unit conversions and CoordinateSystem transformations. The resample method is invoked implicitly whenever needed for mathematical and visualization operations involving Fields. The resample method includes options to interpolate Field values by either nearest neighbor or weighted average. Any degree polynomial interpolation, single stage Barnes and Cressman analyses, and a wide variety of other interpolation schemes can be expressed as weighted averages. Fields get weights from the valueToInterp method of SimpleSet. Thus developers may implement new interpolation algorithms by extending the Set class.

Implementation of interpolation methods not consistent with weighted average would require extensions of Field and FlatField. Nearest neighbor resampling uses the valueToIndex method of Set.

The getWedge method of SimpleSet is important for the efficiency of Field resampling and interpolation. The samples of one Set are passed to the valueToInterp and valueToIndex of another set in an order defined the first Set's getWedge method. Sets use getWedge to define a spatially coherent order of their samples. It is important that developers who extend SimpleSet try to define spatially coherent orders in their implementations of getWedge.

Note that valueToInterp and valueToIndex generally throw an Exception for any Set whose manifold dimension is less than its domain dimension. Thus the resample method does not work for Fields whose domain Sets have manifold dimension less than their domain dimension. In order to resample a Field X over a domain of dimension N with manifold dimension M $<$ N, applications must explicitly copy values of X to another Field Y whose domain has dimension M and is a parameterization of the submanifold containing the samples of X. For example, if N = 3 and M = 2, then the samples of X lie on a 2-D surface embedded in a 3-D space, and the domain of Y should be a parameterization of this surface, with samples locations corresponding to X's sample locations on the surface.

3.5.2 The Delaunay Class for Irregular Sets

The topology of IrregularSets is recorded, and in some cases computed, in the Delaunay classes, which form the following hierarchy:

```
Delaunay
  DelaunayClarkson
  DelaunayWatson
  DelaunayFast
  DelaunayCustom
```

The DelaunayClarkson class computes Delaunay triangulations in any dimension between 2 and 8 using Ken Clarkson's algorithm. DelaunayCustom constructors accept sampling topologies from applications. The DelaunayWatson class computes Delaunay triangulations in 2 or 3 dimensions using David Watson's algorithm. The DelaunayFast class computes non-Delaunay triangulations quickly.

Note that any computation of Delaunay or approximate Delaunay topology is extremely slow and apt to exceed available memory for large Sets. Hence, where an irregular topology is known to the application, we strongly recommend that the topology be supplied by the application through the DelaunayCustom constructor.

3.5.3 Set Constructors

Set is a subclass of DataImpl. A Set object may only be local. The Set classes include the following constructors.

DoubleSet and FloatSet Constructors

These are the finite but very large sets of values representable with N IEEE floats or doubles. Because of their size, they may not be used as Field domains. They are primarily used (with N = 1) for FlatField range values, where they cause range values to be stored in IEEE floats or doubles.

Listing 3.36: The DoubleSet and FloatSet constructors

```
/** the set of values representable by N doubles;
type must be a RealType, a RealTupleType or a SetType;
coordinate_system and units must be compatible with defaults
for type, or may be null;
a DoubleSet may not be used as a Field domain */
public DoubleSet(MathType type, CoordinateSystem coordinate_system,
Unit[] units) throws VisADException;

/** the set of values representable by N floats;
type must be a RealType, a RealTupleType or a SetType;
coordinate_system and units must be compatible with defaults
for type, or may be null;
a FloatSet may not be used as a Field domain */
public FloatSet(MathType type, CoordinateSystem coordinate_system,
Unit[] units) throws VisADException;
```

LinearSet Constructors

LinearSet is an interface implemented by Linear1DSet, Linear2DSet, Linear3DSet and LinearNDSet. Linear1DSets are finite arithmetic progressions of values. Higher dimensional LinearSets are product sets of Linear1DSets. All LinearSets have manifold dimension equal to their domain dimension, although any of the component Linear1DSets may consist of a single sample (in this case, the valueToIndex and valueToInterp methods will throw an Exception).

Linear1DSet, Linear2DSet, Linear3DSet are redundant with LinearNDSet but have more efficient implementations.

The samples of a LinearSet are in raster order, with component values for the first dimension changing fastest and component values for the last dimension changing slowest (this is the same as the ordering of elements in a multi- dimensional Fortran array). For example, given a Linear2DSet with domain type (X, Y) that is a product of six X samples and five Y samples, the 2-D samples are ordered as:

```
          Y (second) component

     X     0   6  12  18  24
           1   7  13  19  25
 (first)   2   8  14  20  26
           3   9  15  21  27
component  4  10  16  22  28
           5  11  17  23  29
```

LinearSets extend GriddedSets, described in Section 3.5.3.3. GriddedSets have rectangular topology while LinearSets have rectangular topology and geometry.

Listing 3.37: The LinearSet Constructors

```
/** an arithmetic progression of length values between first and last;
coordinate_system and units must be compatible with defaults
for type, or may be null; errors may be null */
public Linear1DSet(MathType type,
double first, double last, int length,
CoordinateSystem coordinate_system, Unit[] units,
ErrorEstimate[] errors) throws VisADException;

/** a 1-D arithmetic progression with null errors and generic type */
public Linear1DSet(double first, double last, int length)
throws VisADException;

/** a 2-D cross product of arithmetic progressions;
coordinate_system and units must be compatible with defaults
for type, or may be null; errors may be null */
public Linear2DSet(MathType type,
double first1, double last1, int length1,
double first2, double last2, int length2,
CoordinateSystem coordinate_system, Unit[] units,
ErrorEstimate[] errors) throws VisADException;

/** a 2-D cross product of arithmetic progressions with
null errors and generic type */
public Linear2DSet(double first1, double last1, int length1,
double first2, double last2, int length2)
throws VisADException;

/** a 3-D cross product of arithmetic progressions;
coordinate_system and units must be compatible with defaults
for type, or may be null; errors may be null */
public Linear3DSet(MathType type,
double first1, double last1, int length1,
double first2, double last2, int length2,
double first3, double last3, int length3,
CoordinateSystem coordinate_system, Unit[] units,
ErrorEstimate[] errors) throws VisADException;

/** a 3-D cross product of arithmetic progressions with
null errors and generic type */
public Linear3DSet(double first1, double last1, int length1,
double first2, double last2, int length2,
```

```
    double first3, double last3, int length3)
    throws VisADException;

    /** a 2-D cross product of arithmetic progressions that whose east
    and west edges may be joined (for interpolation purposes);
    coordinate_system and units must be compatible with defaults
    for type, or may be null; errors may be null */
    public LinearLatLonSet(MathType type,
50  double first1, double last1, int length1,
    double first2, double last2, int length2,
    CoordinateSystem coordinate_system,
    Unit[] units, ErrorEstimate[] errors)
    throws VisADException;

    /** a 2-D cross product of arithmetic progressions that whose east
    and west edges may be joined (for interpolation purposes), with
    null errors, CoordinateSystem and Units are defaults from type */
    public LinearLatLonSet(MathType type,
60  double first1, double last1, int length1,
    double first2, double last2, int length2)
    throws VisADException;

    /** construct an N-dimensional set as the product of N Linear1DSets;
    coordinate_system and units must be compatible with defaults
    for type, or may be null; errors may be null */
    public LinearNDSet(MathType type, Linear1DSet[] sets,
    CoordinateSystem coordinate_system,
    Unit[] units, ErrorEstimate[] errors)
70  throws VisADException;

    /** construct an N-dimensional set as the product of N Linear1DSets,
    with null errors, CoordinateSystem and Units are defaults from
    type */
    public LinearNDSet(MathType type, Linear1DSet[] sets)
    throws VisADException;

    /** construct an N-dimensional set as the product of N arithmetic
    progressions; coordinate_system and units must be compatible
80  with defaults for type, or may be null; errors may be null */
    public LinearNDSet(MathType type, double[] firsts, double[] lasts,
    int[] lengths, CoordinateSystem coordinate_system,
    Unit[] units, ErrorEstimate[] errors)
    throws VisADException;

    /** construct an N-dimensional set as the product of N arithmetic
    progressions, with null errors, CoordinateSystem and Units are
    defaults from type */
    public LinearNDSet(MathType type, double[] firsts, double[] lasts,
90  int[] lengths) throws VisADException;
```

IntegerSet Constructors

IntegerSet is an interface implemented by Integer1DSet, Integer2DSet, Integer3DSet and IntegerNDSet. These classes are simple extensions of the corresponding LinearSet classes that constrain arithmetic progressions to sequences of consecutive integers based at zero. Integer1DSet, Integer2DSet, Integer3DSet are redundant with

IntegerNDSet but have more efficient implementations.

IntegerSets are useful as the domains of Fields that are really just simple 1-D, 2-D, 3-D or N-D arrays of values.

Listing 3.38: The IntegerSet constructors

```
/** construct a 1-dimensional set with values {0, 1, ..., lengthX-1};
coordinate_system and units must be compatible with defaults for
type, or may be null; errors may be null */
public Integer1DSet(MathType type, int lengthX,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors)
throws VisADException;

/** a 1-D set with null errors and generic type */
public Integer1DSet(int lengthX)
throws VisADException;

/** construct a 2-dimensional set with values
{0, 1, ..., lengthX-1} x {0, 1, ..., lengthY-1};
coordinate_system and units must be compatible with defaults for
type, or may be null; errors may be null */
public Integer2DSet(MathType type, int lengthX, lengthY,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors)
throws VisADException;

/** a 2-D set with null errors and generic type */
public Integer2DSet(int lengthX, lengthY)
throws VisADException;

/** construct a 3-dimensional set with values {0, 1, ..., lengthX-1}
x {0, 1, ..., lengthY-1} x {0, 1, ..., lengthZ-1};
coordinate_system and units must be compatible with defaults for
type, or may be null; errors may be null */
public Integer3DSet(MathType type, int lengthX, lengthY, lengthZ,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors)
throws VisADException;

/** a 3-D set with null errors and generic type */
public Integer3DSet(int lengthX, lengthY, lengthZ)
throws VisADException;

/** construct an N-dimensional set with values in the cross product
of {0, 1, ..., lengths[i]-1}
for i=0, ..., lengths[lengths.length-1];
coordinate_system and units must be compatible with defaults for
type, or may be null; errors may be null */
public IntegerNDSet(MathType type, int[] lengths,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors)
throws VisADException;

/** an N-D set with null errors and generic type */
public IntegerNDSet(int[] lengths)
throws VisADException;
```

GriddedSet Constructors

GriddedSets are N-dimensional sets with rectangular topologies but not necessarily rectangular geometries. GriddedSet implements the general N-dimensional case (although that implementation is not complete in the initial release) and is extended by Gridded1DSet, Gridded2DSet and Gridded3DSet, which are complete.

GriddedSets may have manifold dimension less than (or equal to) their domain dimension. A GriddedSet with domain dimension N and manifold dimension M defines an M-dimensional grid of samples embedded in an N-dimensional space. In the GriddedSet constructors, the arguments lengthX, lengthY and lengthZ define the numbers of samples along each dimension of the grid (so the number of length arguments defines the manifold dimension), and the samples array argument defines the locations of grid points in N-dimensional domain space. The samples array has type float[][] with dimensions float[N][number_of_samples]. Thus the i-th point in the grid is located at:

`(samples[0][i], samples[1][i], ..., samples[N-1][i]).`

The samples are in raster order, with the first grid dimension changing fastest and the last grid dimension changing slowest. That is, the first lengthX samples form the first 'column' of the grid, the first (lengthX * lengthY) samples for the first sub-plane of the grid, and so on.

If the manifold dimension is less than the domain dimension or any of the grid sizes (i.e., lengthX, lengthY or lengthZ) is 1, then the valueToIndex and valueToInterp methods will throw an Exception. If the manifold dimension equals the domain dimension and all of the grid sizes is greater than 1, then the GriddedSet constructor will perform numerical checks on the samples array to ensure that form a valid grid (e.g., to ensure that they are sorted in the 1-D case).

Listing 3.39: The GriddedSet constructors

```
/** a 1-D sorted sequence with no regular interval; samples array
is organized float[1][number_of_samples] where lengthX =
number_of_samples; samples must be sorted (either increasing
or decreasing); coordinate_system and units must be compatible
with defaults for type, or may be null; errors may be null */
public Gridded1DSet(MathType type, float[][] samples, int lengthX,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors)
throws VisADException;

/** a 1-D sequence with no regular interval with null errors,
CoordinateSystem and Units are defaults from type */
public Gridded1DSet(MathType type, float[][] samples, int lengthX)
throws VisADException;

/** a 1-D sorted sequence with no regular interval; samples array
```

```
    is organized double[1][number_of_samples] where lengthX =
    number_of_samples; samples must be sorted (either increasing
    or decreasing); coordinate_system and units must be compatible
    with defaults for type, or may be null; errors may be null */
    Gridded1DDoubleSet is useful for sequences of DataTime values
    represented as double precision seconds */
    public Gridded1DDoubleSet(MathType type, double[][] samples,
    int lengthX, CoordinateSystem coordinate_system,
    Unit[] units, ErrorEstimate[] errors)
    throws VisADException;

    /** a 1-D sequence with no regular interval with null errors,
    CoordinateSystem and Units are defaults from type;
    Gridded1DDoubleSet is useful for sequences of DataTime values
    represented as double precision seconds */
    public Gridded1DDoubleSet(MathType type, double[][] samples,
    int lengthX)
    throws VisADException;

    /** a 2-D set whose topology is a lengthX x lengthY grid;
    samples array is organized float[2][number_of_samples] where
    lengthX * lengthY = number_of_samples; samples must form a
    non-degenerate 2-D grid (no bow-tie-shaped grid boxes); the
    X component increases fastest in the second index of samples;
    coordinate_system and units must be compatible with defaults
    for type, or may be null; errors may be null */
    public Gridded2DSet(MathType type, float[][] samples, int lengthX,
    int lengthY, CoordinateSystem coordinate_system,
    Unit[] units, ErrorEstimate[] errors)
    throws VisADException;

    /** a 2-D set whose topology is a lengthX x lengthY grid, with
    null errors, CoordinateSystem and Units are defaults from type */
    public Gridded2DSet(MathType type, float[][] samples, int lengthX,
    int lengthY) throws VisADException;

    /** a 2-D set with manifold dimension = 1; samples array is
    organized float[2][number_of_samples] where lengthX =
    number_of_samples; no geometric constraint on samples;
    coordinate_system and units must be compatible with defaults
    for type, or may be null; errors may be null */
    public Gridded2DSet(MathType type, float[][] samples, int lengthX,
    CoordinateSystem coordinate_system,
    Unit[] units, ErrorEstimate[] errors)
    throws VisADException;

    /** a 2-D set with manifold dimension = 1, with null errors,
    CoordinateSystem and Units are defaults from type */
    public Gridded2DSet(MathType type, float[][] samples, int lengthX)
    throws VisADException;

    /** a 3-D set whose topology is a lengthX x lengthY x lengthZ
    grid; samples array is organized float[3][number_of_samples]
    where lengthX * lengthY * lengthZ = number_of_samples;
    samples must form a non-degenerate 3-D grid (no bow-tie-shaped
    grid cubes);  the X component increases fastest and the Z
    component slowest in the second index of samples;
    coordinate_system and units must be compatible with defaults
    for type, or may be null; errors may be null */
    public Gridded3DSet(MathType type, float[][] samples, int lengthX,
    int lengthY, int lengthZ,
```

```
    CoordinateSystem coordinate_system,
    Unit[] units, ErrorEstimate[] errors)
80  throws VisADException;

    /** a 3-D set whose topology is a lengthX x lengthY x lengthZ
    grid, with null errors, CoordinateSystem and Units are
    defaults from type */
    public Gridded3DSet(MathType type, float[][] samples, int lengthX,
    int lengthY, int lengthZ) throws VisADException;

    /** a 3-D set with manifold dimension = 2; samples array is
    organized float[3][number_of_samples] where lengthX * lengthY
90  = number_of_samples; no geometric constraint on samples; the
    X component increases fastest in the second index of samples;
    coordinate_system and units must be compatible with defaults
    for type, or may be null; errors may be null */
    public Gridded3DSet(MathType type, float[][] samples, int lengthX,
    int lengthY, CoordinateSystem coordinate_system,
    Unit[] units, ErrorEstimate[] errors)
    throws VisADException;

    /** a 3-D set with manifold dimension = 2, with null errors,
100 CoordinateSystem and Units are defaults from type */
    public Gridded3DSet(MathType type, float[][] samples, int lengthX,
    int lengthY) throws VisADException;

    /** a 3-D set with manifold dimension = 1; samples array is
    organized float[3][number_of_samples] where lengthX =
    number_of_samples; no geometric constraint on samples;
    coordinate_system and units must be compatible with defaults
    for type, or may be null; errors may be null */
    public Gridded3DSet(MathType type, float[][] samples, int lengthX,
110 CoordinateSystem coordinate_system, Unit[] units,
    ErrorEstimate[] errors)
    throws VisADException;

    /** a 3-D set with manifold dimension = 1, with null errors,
    CoordinateSystem and Units are defaults from type */
    public Gridded3DSet(MathType type, float[][] samples, int lengthX)
    throws VisADException;
```

IrregularSet Constructors

IrregularSets are N-dimensional sets with irregular topologies consisting of lists of (N+1)-gons (i.e., line segments in 1 dimension, triangles in 2 dimensions, tetrahedra in 3 dimensions, etc). IrregularSet implements the general N-dimensional case (although that implementation is not complete in the initial release) and is extended by Irregular1DSet, Irregular2DSet and Irregular3DSet, which are complete.

The samples array argument to the IrregularSet constructors defines the locations of sample points in N-dimensional domain space. The samples array has type float[][] with dimensions float[N][number_of_samples]. Thus the i-th sample point is located at:

`(samples[0][i], samples[1][i], ..., samples[N-1][i])`.

IrregularSets may have manifold dimension less than or equal to their domain dimension. If the manifold dimension is less than the domain dimension, then the valueToIndex and valueToInterp methods throw Exceptions.

In 1 dimension the topology is constructed merely by sorting the samples. In higher dimensions the topology may be constructed by a Delaunay triangulation or may be specified in the constructor (using the DelaunayCustom class). See Section 3.5.5 for more information about Delaunay classes.

Listing 3.40: The IrregularSet constructors

```
/** a 1-D irregular set; samples array is organized
float[1][number_of_samples]; samples need not be
sorted - the constructor sorts samples to define
a 1-D "triangulation";
coordinate_system and units must be compatible with
defaults for type, or may be null; errors may be null */
public Irregular1DSet(MathType type, float[][] samples,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors)
throws VisADException;

/** a 1-D irregular set with null errors, CoordinateSystem
and Units are defaults from type */
public Irregular1DSet(MathType type, float[][] samples)
throws VisADException;

/** a 2-D irregular set; samples array is organized
float[2][number_of_samples];  no geometric constraint on
samples; if delan is non-null it defines the topology of
samples (which must have manifold dimension 2), else the
constructor computes a topology with manifold dimension 2;
note that Gridded2DSet can be used for an irregular set
with domain dimension 2 and manifold dimension 1;
coordinate_system and units must be compatible with
defaults for type, or may be null; errors may be null */
public Irregular2DSet(MathType type, float[][] samples,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors,
Delaunay delan)
throws VisADException;

/** a 2-D irregular set with null errors, CoordinateSystem
and Units are defaults from type; topology is computed
by the constructor */
public Irregular2DSet(MathType type, float[][] samples)
throws VisADException;

/** a 3-D irregular set; samples array is organized
float[3][number_of_samples];  no geometric constraint on
samples; if delan is non-null it defines the topology of
samples (which may have manifold dimension 2 or 3), else
the constructor computes a topology with manifold dimension
3; note that Gridded3DSet can be used for an irregular set
with domain dimension 3 and manifold dimension 1;
```

```
coordinate_system and units must be compatible with
defaults for type, or may be null; errors may be null */
public Irregular3DSet(MathType type, float[][] samples,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors,
Delaunay delan)
throws VisADException;

/** a 3-D irregular set with null errors, CoordinateSystem
and Units are defaults from type; topology is computed
by the constructor */
public Irregular3DSet(MathType type, float[][] samples)
throws VisADException;
```

ProductSet and UnionSet Constructors

ProductSets are SampledSets that are defined as products of other SampledSets (called the ProductSet's factor sets). The domain dimension of a ProductSet is the sum of the domain dimensions of its factors and similarly its manifold dimension is the sum of the manifold dimensions of its factors. The order of samples in a ProductSet is the rasterization of the orders of samples of its factors. As the index of the ProductSet increases, the index of the first factor varies fastest and the index of the last factor varies slowest.

UnionSets are SampledSets that are defined as unions of other SampledSets. All the sets in the union must have the same domain dimension and they must all have the same manifold dimension. Note that the valueToInterp method is not implemented for UnionSets but the valueToIndex method is. Thus if a UnionSet is the domain set of a Field, arithmetic operations involving the Field must specify the Data.NEAREST_NEIGHBOR resampling mode rather than Data.WEIGHTED_AVERAGE. The order of samples in a UnionSet is the serialization of the orders of samples of its components. As the index of the UnionSet increases, the samples of the first component are enumerated first and the samples of the last component are enumerated last.

Listing 3.41: The ProductSet and UnionSet constructors

```
/** create the product of the sets array; coordinate_system
and units must be compatible with defaults for type,
or may be null; errors may be null */
public ProductSet(MathType type, SampledSet[] sets,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors)
throws VisADException;

/** create the product of the sets array, with null errors,
CoordinateSystem and Units are defaults from type */
public ProductSet(MathType type, SampledSet[] sets)
throws VisADException;
```

```
/** create the union of the sets array; coordinate_system
and units must be compatible with defaults for type,
or may be null; errors may be null */
public UnionSet(MathType type, SampledSet[] sets,
CoordinateSystem coordinate_system,
Unit[] units, ErrorEstimate[] errors)
throws VisADException;

/** create the union of the sets array, with null errors,
CoordinateSystem and Units are defaults from type */
public UnionSet(MathType type, SampledSet[] sets)
throws VisADException;
```

3.5.4 Set Methods

Applications generally do not invoke Set methods, but they construct new Set objects and may define new Set subclasses. New Set subclasses must either implement or inherit these methods:

Listing 3.42: The Set methods

```
/** return an enumeration of sample indices in a spatially
coherent order; this is useful for efficiency */
public int[] getWedge();

/** return an enumeration of sample values in index order
(i.e., not in getWedge order); the return array is
organized as float[domain_dimension][number_of_samples] */
public float[][] getSamples() throws VisADException;

/** convert an array of indices to an array of sample values;
the return array is organized as
float[domain_dimension][indices.length] */
public float[][] indexToValue(int[] indices) throws VisADException;

/** convert an array of values to an array of indices of the nearest
samples; the values array is organized as
float[domain_dimension][number_of_values] */
public int[] valueToIndex(float[][] values) throws VisADException;

/** convert an array of indices to an array of double precision
sample values; this precision is currently only meaningful
for Linear1DSet and Gridded1DDoubleSet where it is intended
to represent date/time values as double precision seconds;
the return array is organized as
double[domain_dimension][indices.length] */
public double[][] indexToDouble(int[] indices) throws VisADException;

/** convert an array of double precision values to an array of
indices of the nearest samples; this precision is currently
only meaningfulful for Linear1DSet and Gridded1DDoubleSet
where it is intended to represent date/time values as double
precision seconds; the values array is organized as
```

```
double[domain_dimension][number_of_values] */
public int[] doubleToIndex(double[][] values) throws VisADException;
```

3.5.5 SimpleSet Methods

Listing 3.43: The SimpleSet methods

```
/** convert an array of values to arrays of indices and weights for
those indices, appropriate for interpolation; the values array is
organized as float[domain_dimension][number_of_values]; indices
and weights must be passed in as int[number_of_values][] and
float[number_of_values][]; on return, quantity( values[.][i] )
can be estimated as the sum over j of
weights[i][j] * quantity (sample at indices[i][j]);
no estimate possible if indices[i] and weights[i] are null */
public void valueToInterp(float[][] values, int[][] indices,
float[][] weights) throws VisADException;
```

3.5.6 Delaunay Constructors

The Delaunay class is serializable. A Delaunay object may only be local. The Delaunay classes include the following useful constructor:

Listing 3.44: The Delaunay constructors

```
/** the DelaunayCustom constructor allows applications to define
sampling topologies; the samples array is organized as
float[domain_dimension][number_of_samples] and the tris arrays
is organized as int[number_of_tris][manifold_dimension + 1];
each "tri" is a list of sample indices, and is a triangle,
tetrahedron, etc depending on manifold dimension */
public DelaunayCustom(float[][] samples, int[][] tris)
throws VisADException;
```

3.6 ErrorEstimates

The ErrorEstimate class contains an estimate of the variance of error associated with a value or a set of values. ErrorEstimates are included with individual Real values, and with each RealType component in the range of FlatFields. For example, one range component of a FlatField may consist of all temperature values in a model output grid, and these would be associated with a single average ErrorEstimate (see Section 3.9).

Data operations include options to propagate ErrorEstimates assuming that errors are distributed either independently or dependently, as well as an option to not propagate ErrorEstimates.

The VisAD ErrorEstimates are not a substitute for a detailed error analysis, but can provide a quick estimate of error magnitude and the possible need for detailed analysis.

3.6.1 ErrorEstimate Constructors

The ErrorEstimate class is serializable. An ErrorEstimate object may only be local. The ErrorEstimate class include the following constructors:

Listing 3.45: The ErrorEstimate constructors

```
/** construct an error distribution of number values with
given mean and variance, in Unit unit */
public ErrorEstimate(double variance, double mean,
long number, Unit unit);

/** construct an error distribution of 1 value with
given mean and variance, in Unit unit */
public ErrorEstimate(double mean, double variance, Unit unit);
```

3.7 AuditTrails

The AuditTrail class contains an ordered sequence of text strings documenting the history of a Data object, starting with external data sources (e.g., data files and URLs) and including Data operations. In order to conserve memory, AuditTrail objects are only associated with top-level Data objects (i.e., Data objects that are not components of Fields or Tuples).

The AuditTrail class is not yet implemented, so there is no constructor and method documentation.

3.8 Missing Data

Any Data object or primitive value may be marked as missing, meaning that its value is unknown or undefined. Missing values may be generated as the result of sensor failures, arithmetic failures (e.g., division by zero), or to mark incomplete data coverage (e.g., temperatures are not available for one time step of a model output). The NaN (Not a Number) value of the IEEE floating point standard is used to represent missing floats

and doubles in VisAD, since it has the correct arithmetic semantics (e.g., X .OP. NaN = NaN for any value X and any operation .OP.).

3.9 FlatFields - Data Operations and Efficiency

There is a natural trade-off between generality and efficiency, so the generality of the VisAD data model poses a challenge for efficiency. Efficiency is achieved by incorporating the following rule at all levels of the system:

Hint 2 (VisAD rule of efficiency) *Apply all data operations to arrays of values rather than individual values, and avoid methods that are invoked once per data value.*

The effectiveness of this rule was demonstrated in the C implementation of VisAD [8, 9], which had a general data model like the Java implementation.

The large Data objects in any application are Fields. Most array data in numerical programs are finite samplings of functions (for example, images are finite samplings of continuous radiance functions with a pixel for each sample) and these correspond to Fields. Even arrays that do not correspond to any obvious continuous function can be represented by Fields whose domains are sets of integers from 1 to N. The obvious way to implement the Field class is with an array of range sample objects, which would violate our rule because Field operations invoke methods on each range object. Thus the Field class is extended by FlatField, which simulates an array of range objects with arrays of Java primitive values. A FlatField can be used for a Field under the following two conditions:

1. The MathType of the Field range is a RealType, a RealTupleType, or a TupleType whose components are all RealTypes or RealTupleTypes (this allows subsets of a FlatField's range components to be grouped into RealTupleTypes to document CoordinateSystems).

2. All range samples have identical metadata, including Units, CoordinateSystems, shared ErrorEstimates, etc.

FlatFields are appropriate for images, multi-channel images, multi-variate grids, time series and many other types of numerical data arrays. Complex data may be implemented by Fields whose range samples are FlatFields. For example, a time sequence of images may be implemented by a Field whose domain is a set of time steps, and whose range samples are each images stored in FlatField.

In addition to computational efficiency, FlatFields also have better storage efficiency than Fields. Java primitive data require less storage than Java objects, shared metadata objects require less total space, and when possible function range values are stored in bytes, shorts or ints rather than floats. The FlatField constructor accepts range sampling Sets for each RealType component of its range. If the size of the sampling Set for a range component is 255, then values for that component are encoded as indices into that Set and stored in an array of bytes (the 256th code is used to represent missing values). Arrays of shorts or ints are used for larger set sizes, as appropriate. The default range sampling Sets are 1-D FloatSets, which cause range values to be stored as floats.

Numerical precision problems occur and can be very difficult to diagnose when they do. Thus developers may want to pass DoubleSets to the range sampling Sets argument of the FlatField constructor, in order to avoid precision problems.

Float.NaN and Double.NaN are used to represent missing float and double values. This avoids time-consuming explicit tests for missing values, since these IEEE NaNs have the right arithmetic semantics for missing values.

3.9.1 FlatField Constructors

FlatField is a subclass of FieldImpl. A FlatField object may only be local. The FlatField class include the following constructors:

Listing 3.46: The FlatField constructors

```
/** FlatField is a sampled function whose range is a Real,
a RealTuple, or a Tuple of Reals and RealTuples; if range
is a RealTuple, range_coordinate_system may be non-null
but must have the same Reference as RangeType default
CoordinateSystem; domain_set defines the domain sampling;
range_sets define samplings for range values - if range_set[i]
is null, the i-th range component values are stored as doubles;
if range_set[i] is non-null, the i-th range component values are
stored in bytes if range_sets[i].getLength() < 256, stored in
shorts if range_sets[i].getLength() < 65536, etc;
any argument but type may be null */
public FlatField(FunctionType type, Set domain_set,
CoordinateSystem range_coordinate_system,
Set[] range_sets, Unit[] units)
throws VisADException;

/** similar to the previous constructor, except that if
range_coordinate_systems[i] is non-null, then the i-th
component of the range type must be a RealTupleType whose
default CoordinateSystem has the same Reference */
public FlatField(FunctionType type, Set domain_set,
CoordinateSystem[] range_coordinate_systems,
Set[] range_sets, Unit[] units)
throws VisADException;
```

3.9.2 FlatField Methods

FlatField overrides many of the FieldImpl methods, plus it defines a number of methods for accessing range values as arrays of doubles and floats, and accessing range metadata (which are shared by all range samples).

Listing 3.47: The FlatField methods

```
/** convert FlatField to FieldImpl */
public Field convertToField()
throws VisADException, RemoteException;

/** return array of Units associated with each RealType
component of range; these may differ from default
Units of range RealTypes, but must be convertable */
public Unit[][] getRangeUnits();

/** return range CoordinateSystem assuming range type is
a RealTupleType (throws a TypeException if its not);
this may differ from default CoordinateSystem of
range RealTupleType, but must be convertable */
public CoordinateSystem[] getRangeCoordinateSystem();

/** return range CoordinateSystem associated with
RealTupleType that is index-th component of range
TupleType; this may differ from default
CoordinateSystem of RealTupleType component of
range TupleType, but must be convertable */
public CoordinateSystem[] getRangeCoordinateSystem(int index);

/** return array of ErrorEstimates associated with each
RealType component of range; each ErrorEstimate is a
mean error for all samples of a range RealType
component */
public ErrorEstimates[] getRangeErrors();

/** set ErrorEstimates associated with each RealType
component of range */
public void setRangeErrors(ErrorEstimates[] errors)
throws VisADException;

/** set range array as range values of this FlatField;
the array is dimensioned
double[number_of_range_components][number_of_range_samples];
copy array if copy flag is true */
public void setSamples(double[][] range, boolean copy)
throws VisADException, RemoteException;

/** set range array as range values of this FlatField;
the array is dimensioned
double[number_of_range_components][number_of_range_samples];
copy array if copy flag is true */
public void setSamples(float[][] range, boolean copy)
throws VisADException, RemoteException;

/** get this FlatField's range values in their default range
Units (as defined by the range of the FlatField's
FunctionType); the return array is dimensioned
```

```
double[number_of_range_components][number_of_range_samples] */
public double[][][] getValues()
throws VisADException, RemoteException;
```

3.10 Immutable Data

Most Data classes and metadata classes are immutable, in order to ensure the thread-safeness of VisAD applications in distributed computing environments. The only exceptions are Field and its sub-classes. Field metadata cannot change, but the values of Field and FlatField range samples can change (as well as the ErrorEstimates associated with FlatField range samples). Fields are mutable since they may be very large and it would be inefficient to have to copy them to change individual range values.

3.11 DataReferences

Since the only way to change the value of an immutable Data object is to replace it with a different Data object, there is a need for a class to represent variable Data. Thus the DataReference class defines mutable references to Data objects. In an application, for example, the variable `current_time` may be represented by a DataReference object that refers to a succession of immutable Real objects.

3.11.1 DataReference Constructors

DataReference is an interface that may apply to both local and remote DataReference objects. The DataReferenceImpl class applies only to local DataReference objects, while the RemoteDataReference interface and RemoteDataReferenceImpl class apply only to remote DataReference objects (see Section 6 for more information). The DataReference classes include the following constructors:

Listing 3.48: The ImmutableData constructors

```
/** construct a DataReferenceImpl object with the given name */
public DataReferenceImpl(String name) throws VisADException;

/** construct a RemoteDataReferenceImpl object to provide remote
access to reference */
public RemoteDataReferenceImpl(DataReferenceImpl reference)
throws RemoteException;
```

3.11.2 DataReference Methods

Generally useful DataReference methods include:

Listing 3.49: The DataReference methods

```
/** get MathType of referenced Data object, or null if none;
this is more efficient than getData().getType() for
RemoteDataReferences */
public MathType getType() throws VisADException, RemoteException;

/** get referenced Data object, or null if none */
public Data getData() throws VisADException, RemoteException;

/** set reference to data, replacing any currently referenced
Data object; if this is local (i.e., an instance of
DataReferenceImpl) then the data argument must also be
local (i.e., an instance of DataImpl);
if this is Remote (i.e., an instance of RemoteDataReference)
then a local data argument (i.e., an instance of DataImpl)
will be passed by copy and a remote data argument (i.e., an
instance of RemoteData) will be passed by remote reference */
public void setData(Data data) throws VisADException, RemoteException;
```

3.12 Application Example: Arrays versus VisAD Functions

In order to understand how to write numerical applications with VisAD, it is useful to compare VisAD with C. VisAD and C both allow applications to define complex data structures from basic primitives. For example, a multi-spectral image can be defined in C using a structure and an array:

Listing 3.50: a multi-spectral image defined in C

```
struct pixel {
  float ir_radiance;
  float vis_radiance;
};
struct pixel image[nlines][nelements];
```

A similar multi-spectral image can be defined in VisAD using RealTupleTypes and a FunctionType:

Listing 3.51: a multi-spectral image defined in VisAD

```
RealTupleType location = new RealTupleType(new RealType("line"), new ↩
    RealType("element"));
```

```
RealTupleType pixel = new RealTupleType(new RealType("ir_radiance"), new ↩
    RealType("vis_radiance"));
FunctionType image_type = new FunctionType(location, pixel);
Set location_set = new Integer2DSet(nlines, nelements);
FlatField image = new FlatField(image_type, location_set);
```

In general, we can list the following analogies between C and VisAD data structuring tools:

C	VisAD
float, double, int	RealType
char string[]	TextType
struct	TupleType, RealTupleType
array	FunctionType

In these analogies, C and VisAD syntax differ considerably. However, that kind of difference should be familiar to programmers with experience in several programming languages. The important similarities and differences relate to the meanings of these data structuring tools. Most differences involve metadata integrated into the meanings of data. For example, VisAD Reals and C floats implement the same set of operations, but operations on VisAD Reals may invoke Unit conversions and propogate ErrorEstimates and missing data indicators (some C implementations also propogate missing data indicators in the form of IEEE NaNs). C structs and VisAD Tuples have very similar meanings - they are both fixed length lists of other data structures. However, VisAD RealTuples may include CoordinateSystems and operations on RealTuples may invoke coordinate transforms.

The most complex differences exist for the analogy between C arrays and VisAD Functions, because of the variety of metadata integrated into VisAD Functions. The rest of this section of the Developers Guide is dedicated to explaining the relation between arrays and Functions in a series of program examples. With the proper understanding, you can use Functions anywhere you can use arrays, but Functions also allow you to express some very complex operations simply.

3.12.1 Subtracting Images as Pixel Arrays in C

The following C code could be used to compute the difference between two multispectral images:

Listing 3.52: Subtracting images as pixel arrays in C

```
#define nlines 256
#define nelements 256

struct pixel {
```

```
  float ir_radiance;
  float vis_radiance;
};

image_difference(image1, image2)
struct pixel image1[nlines][nelements];
struct pixel image2[nlines][nelements];
{
  int i, j;
  for (i=0; i<nlines; i++) {
    for (j=0; j<nelements; j++) {
      image1[i][j].ir_radiance -= image2[i][j].ir_radiance;
      image1[i][j].vis_radiance -= image2[i][j].vis_radiance;
    }
  }
}
```

This code assumes a fixed size for its image arguments, but that would not be hard to generalize. It also assumes a fixed set of spectral bands for its image arguments, that both images have the same size, that their pixel locations are aligned, and that image radiance values have the same units and calibration.

3.12.2 Subtracting Images as Pixel Arrays in VisAD

The following Java / VisAD code could be used to compute the difference between two multi-spectral images, in a pixel-by-pixel manner similar to the C code in Section 3.12.1:

Listing 3.53: Subtracting images as pixel arrays in VisAD

```
void image_difference(FlatField image1, FlatField image2)
  throws VisADException, RemoteException {
  // extract pixel radiance values from images
  double[][] pixels1 = image1.getValues();
  double[][] pixels2 = image2.getValues();
  // loop over spectral bands in image1
  for (int i=0; i<pixels1.length; i++) {
    // loop over pixels in one spectral band
    for(int j=0; j<pixels1[i].length; j++) {
      pixels1[i][j] -= pixels2[i][j];
    }
  }
  // set pixel radiance values in image1
  image1.setSamples(pixels1);
}
```

This code does not assume a fixed size for its image arguments, and does not assume that they have only two spectral bands. However, it does assume that both images have the same size and the same set of spectral bands, that their pixel locations are aligned, and that image radiance values have the same units and calibration.

This code example demonstrates that it is easy to treat VisAD Functions like simple arrays, extracting their values into ordinary arrays using the getValues method and setting values from ordinary arrays using the setSamples method.

3.12.3 Subtracting Images as Functions in VisAD

The following Java / VisAD code computes the difference between two multi- spectral images at a high level, which allows VisAD to integrate all their metadata into the operation:

Listing 3.54: Subtracting images as functions in VisAD

```
FlatField image_difference(FlatField image1, FlatField image2)
  throws VisADException, RemoteException {
  return (FlatField) image1.subtract(image2);
}
```

This code only assumes that the two images have the same set of spectral bands. If necessary it will resample the locations of image2 to the locations of image1, transform locations from one coordinate system to another and convert location units, convert radiance units and transform between radiance calibration coordinate systems, and propogate error estimates and missing data indicators.

This code example demonstrates that Functions can be manipulated at a high level, similar to array operations in some high-level languages (such as IDL) but integrating a variety of metadata in those operations. High-level operations on Functions include basic arithmetic such as add and multiply with other Functions or with Reals, as well as derivative, resampling, and display.