Contents

1. Installation Instructions

1.1. Introduction

VisAD is a pure Java system for interactive and collaborative visualization and analysis of numerical data. It is described in detail in the *VisAD Java Class Library Developers Guide* available from the VisAD web page at http://www.ssec.wisc.edu/~billh/visad.html

1.2. Downloading the VisAD Source Code

To download the VisAD source code, first make sure the current directory is a directory in your CLASSPATH (which we will refer to as '/parent_dir' through the rest of this README file). Then get ftp://ftp.ssec.wisc.edu/pub/visad-2.0/visad_src-2.0.jar

If you have previously downloaded the VisAD source you should run 'make clear' in your visad directory to clear out the old source files before you unpack the new source.

Unpack the jar file by running:

```
jar xvf visad_src-2.0.jar
```

Unpacking VisAD will create the following sub-directories:

visad the core VisAD package

visad/ss VisAD Spread Sheet

visad/formula formula parser

visad/java3d Java3D displays for VisAD

visad/java2d Java2D displays for VisAD

visad/util VisAD UI utilities

visad/collab collaboration support

visad/cluster data and displays distributed on clusters

visad/python python scripts for VisAD

visad/browser connecting applets to VisAD servers

visad/math math (fft, histogram) operations

visad/matrix JAMA (matlab) matrix operations

visad/data VisAD data (file) format adapters

visad/data/units VisAD Units subsystem

visad/data/fits VisAD - FITS file adapter

visad/data/netcdf VisAD - netCDF file adapter

visad/data/netcdf/in netCDF file adapter input

visad/data/netcdf/out netCDF file adapter output

visad/data/netcdf/units units parser for netCDF adapter

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

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

visad/data/vis5d VisAD - Vis5D file adapter

visad/data/mcidas VisAD - McIDAS file adapter

visad/data/gif VisAD - GIF file adapter

visad/data/tiff VisAD - TIFF file adapter

visad/data/visad VisAD serialized object file adapter

visad/data/hdf5 VisAD - HDF-5 file adapter

visad/data/hdf5/hdf5objects VisAD - HDF-5 file adapter

visad/data/amanda VisAD - F2000 file adapter (neutrino events)

visad/data/text VisAD - text file adapter

visad/data/in VisAD - data input pipeline

visad/data/jai VisAD file adapter for images using JAI

visad/data/ij VisAD file adapter for images using ImageJ

visad/data/gis VisAD - ArcGrid and USGS DEM file adapters

visad/data/dods VisAD - DODS object adapter

visad/data/bio VisAD - Bio-Formats adapter

visad/install VisAD-in-a-box installer

visad/paoloa GoesCollaboration application

visad/paoloa/spline spline fitting application

visad/aune ShallowFluid application

visad/benjamin Galaxy application

visad/rabin Rainfall estimation spread sheet

visad/bom wind barb rendering for ABOM

visad/jmet JMet - Java Meteorology package

visad/aeri Aeri data visualization

visad/georef specialized earth coordinates

visad/meteorology meteorology

visad/gifts GIFTS

visad/sounder atmospheric sounding package

visad/examples small application examples

nom/tam/fits Java FITS file binding

nom/tam/util Java FITS file binding

nom/tam/test Java FITS file binding

ucar/multiarray Java netCDF file binding

ucar/util Java netCDF file binding

ucar/netcdf Java netCDF file binding

ucar/tests Java netCDF file binding

edu/wisc/ssec/mcidas Java McIDAS file binding

edu/wisc/ssec/mcidas/adde Java McIDAS file binding

ncsa/hdf/hdf5lib Java HDF-5 file binding

ncsa/hdf/hdf5lib/exceptions Java HDF-5 file binding

gnu/regexp GNU Regular Expressions for Java

gnu/regexp/util GNU Regular Expressions for Java

HTTPClient Jakarta Commons HttpClient

loci/formats LOCI Bio-Formats package

loci/formats/in Bio-Formats - read image formats

loci/formats/out Bio-Formats - write image formats

loci/formats/gui Bio-Formats - GUI components

loci/formats/codec Bio-Formats - codecs

These directories correspond to the packages in distributed with VisAD, except that the classes in `visad/examples` are in the default package (i.e., they do not include a package statement).

1.3. Building VisAD

We recommend you use Apache Ant, a cross platform Java-based build tool "kind of like make, without make's wrinkles."

Alternately, you can use Unix make or Windows NMAKE as follows.

Your CLASSPATH should include:

1. The parent directory of your visad directory.
2. The current directory.

Thus if VisAD is installed at /parent_dir/visad and you use csh, your .cshrc file should include:

```
setenv CLASSPATH /parent_dir:.
```

VisAD requires JDK 1.4 and Java3D. More information about these is available at:

```
http://java.sun.com/
```

On systems that support Unix make, you can simply run:

```
make debug
```

to compile the Java source code in all the directories unpacked from the source distribution, as well as native code in the `visad/data/hdfeos/hdfeosc` directory and certain application directories. If you want 'make debug' to compile native libraries, then you may need to change the line:

```
JAVADIR=/opt/java
```

in `visad/Makefile` if your java is installed in a directory other than `/opt/java`.

If you have NMAKE on Windows (2K, XP) you may run this from within the visad directory:

```
set CLASSPATH=c:\parent_dir;.\
nmake -f makefile.winnt debug
```

This does not compile native code. `parent_dir` is as defined above – the VisAD source code has been unpacked into `C:\parent_dir\visad`.

Note that using 'make debug' rather than 'make compile' will enable you to run using jdb in place of java in order to make error reports that include line numbers in stack dumps.

If you cannot use Apache Ant, Unix make or Windows NMAKE, you must invoke the Java compiler on the Java source files in all the directories unpacked from the source distribution. Note that the Java source code in the `visad/examples` directory has no package, so you must run `cd visad/examples` before you compile these Java source files.

If you do not use ant or make, then you must also run the rmic compiler on the following classes (after they are compiled by the javac compiler):

- visad.RemoteActionImpl
- visad.RemoteCellImpl
- visad.RemoteDataImpl
- visad.RemoteDataReferenceImpl
- visad.RemoteDisplayImpl
- visad.RemoteFieldImpl
- visad.RemoteFunctionImpl

- visad.RemoteReferenceLinkImpl
- visad.RemoteServerImpl
- visad.RemoteSlaveDisplayImpl
- visad.RemoteThingImpl
- visad.RemoteThingReferenceImpl
- visad.collab.RemoteDisplayMonitorImpl
- visad.collab.RemoteDisplaySyncImpl
- visad.collab.RemoteEventProviderImpl
- visad.cluster.RemoteAgentContactImpl
- visad.cluster.RemoteClientAgentImpl
- visad.cluster.RemoteClientDataImpl
- visad.cluster.RemoteClientFieldImpl
- visad.cluster.RemoteClientTupleImpl
- visad.cluster.RemoteClusterDataImpl
- visad.cluster.RemoteNodeDataImpl
- visad.cluster.RemoteNodeFieldImpl
- visad.cluster.RemoteNodePartitionedFieldImpl
- visad.cluster.RemoteNodeTupleImpl

1.4. Building Native Code for the HDF-EOS and HDF-5 File Adapters

Although VisAD is a pure Java system, it does require native code interfaces in its adapters for HDF-EOS and HDF-5 file formats. We believe that the need for these will disappear as the organizations supporting these file formats develop Java interfaces.

You can build the necessary libraries from source or on Sparc Solaris you can simply download `ftp://ftp.ssec.wisc.edu/pub/visad-2.0/libhdfeos.so` into your `visad/data/hdfeos/hdfeosc` directory, and download the appropriate file (for Sparc Solaris, Irix, Linux and Windows) from ftp://hdf.ncsa.uiuc.edu/HDF5/current/java-hdf5/JHI5_1_1_bin/lib/

into your `ncsa/hdf/hdf5lib` directory according to instructions available under *Download* at http://hdf.ncsa.uiuc.edu/java-hdf5-html/

The HDF-EOS and HDF-5 file adapters include native interfaces (JNI) to file interfaces written in C. To make the HDF-EOS VisAD native library on systems that support Unix make, change to the `visad/data/hdfeos/hdfeosc` directory and run `make all`.

Note that the native code in visad/data/hdfeos/hdfeosc does not include NASA/Hughes' HDF-EOS C file interface code; it only includes our C native code for creating a Java binding to their HDF-EOS C file interface. You must obtain the HDF-EOS C file interface code directly from NASA and NCSA. To do this, please follow the instructions in:

```
visad/data/hdfeos/README.hdfeos
```

We have successfully linked these libraries on Irix and Solaris.

You can also make the HDF-5 native libraries from source, according to instructions available from http://hdf.ncsa.uiuc.edu/java-hdf5-html/

Before you can run applications that use the HDF-EOS and HDF-5 file adapters, you must add

```
/parent_dir/visad/data/hdfeos/hdfeosc
```

and:

```
/parent_dir/ncsa/hdf/hdf5lib
```

to your `LD_LIBRARY_PATH`.

1.5. Building Native Code for Applications

Although VisAD is a pure Java system, applications of VisAD may include native code. The reality is that most science code is still written in Fortran.

The applications in `visad/paoloa`, `visad/paoloa/spline`, `visad/aune` and `visad/benjamin` also include native code in both C and Fortran.

Edit the Makefile in the visad/paoloa, visad/paoloa/spline, visad/aune and visad/benjamin to change the path:

```
JAVADIR=/opt/java
```

to point to the appopriate directory where you installed Java.

On systems that support Unix make, change to each of the directories `visad/paoloa`, `visad/paoloa/spline`, `visad/aune` and `visad/benjamin` run `make`. This will create the shared object files (i.e., file names ending in ".so") containing native code. To run these applications make sure that your `LD_LIBRARY_PATH` includes ".", change to one of these directories:

```
/parent_dir/visad/paoloa
/parent_dir/visad/paoloa/spline
/parent_dir/visad/aune
/parent_dir/visad/benjamin
```

and run the appropriate `java ...` command.

Note that the applications in `visad/paoloa` require data files available from ftp://ftp.ssec.wisc.edu/pub/visad-2.0/paoloa-files.tar.Z

1.6. Downloading VisAD Classes in Jar Files

If you want to write applications for VisAD but don't want to compile VisAD from source, you can dowload a jar file that includes the VisAD classes. This file is ftp://ftp.ssec.wisc.edu/pub/visad-2.0/visad.jar

Once you've got `visad.jar` simply add:

```
/parent_dir/visad.jar;.
```

to your CLASSPATH. Then you can compile and run applications that import the VisAD classes. However, if your application uses the HDF-EOS or HDF-5 file format adapters, then you will need to compile the native code as described in Section 4 of this README file. The visad.jar file includes the classes from these packages:

visad the core VisAD package

visad/ss VisAD Spread Sheet

visad/formula formula parser

visad/java3d Java3D displays for VisAD

visad/java2d Java2D displays for VisAD

visad/util VisAD UI utilities

visad/collab collaboration support

visad/cluster data and displays distributed on clusters

visad/python python scripts for VisAD

visad/browser connecting applets to VisAD servers

visad/math math (fft, histogram) operations

visad/matrix JAMA (matlab) matrix operations

visad/data VisAD data (file) format adapters

visad/data/units VisAD Units subsystem

visad/data/fits VisAD - FITS file adapter

visad/data/netcdf VisAD - netCDF file adapter

visad/data/netcdf/in netCDF file adapter input

visad/data/netcdf/out netCDF file adapter output

visad/data/netcdf/units units parser for netCDF adapter

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

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

visad/data/vis5d VisAD - Vis5D file adapter

visad/data/mcidas VisAD - McIDAS file adapter

visad/data/gif VisAD - GIF file adapter

visad/data/tiff VisAD - TIFF file adapter

visad/data/visad VisAD serialized object file adapter

visad/data/hdf5 VisAD - HDF-5 file adapter

visad/data/hdf5/hdf5objects VisAD - HDF-5 file adapter

visad/data/amanda VisAD - F2000 file adapter (neutrino events)

visad/data/text VisAD - text file adapter

visad/data/in VisAD - data input pipeline

visad/data/jai VisAD file adapter for images using JAI

visad/data/gis VisAD - ArcGrid and USGS DEM file adapters

visad/data/dods VisAD - DODS object adapter

visad/data/bio VisAD - Bio-Formats adapter

visad/install VisAD-in-a-box installer

visad/paoloa GoesCollaboration application

visad/paoloa/spline spline fitting application

visad/aune ShallowFluid application

visad/benjamin Galaxy application

visad/rabin Rainfall estimation spread sheet

visad/bom wind barb rendering for ABOM

visad/jmet JMet - Java Meteorology package

visad/aeri Aeri data visualization

visad/georef specialized earth coordinates

visad/meteorology meteorology

nom/tam/fits Java FITS file binding

nom/tam/util Java FITS file binding

nom/tam/test Java FITS file binding

ucar/multiarray Java netCDF file binding

ucar/util Java netCDF file binding

ucar/netcdf Java netCDF file binding

ucar/tests Java netCDF file binding

edu/wisc/ssec/mcidas Java McIDAS file binding

edu/wisc/ssec/mcidas/adde Java McIDAS file binding

ncsa/hdf/hdf5lib Java HDF-5 file binding

ncsa/hdf/hdf5lib/exceptions Java HDF-5 file binding

gnu/regexp GNU Regular Expressions for Java

gnu/regexp/util GNU Regular Expressions for Java

HTTPClient Jakarta Commons HttpClient

loci/formats LOCI Bio-Formats package

loci/formats/in Bio-Formats - read image formats

loci/formats/out Bio-Formats - write image formats

loci/formats/gui Bio-Formats - GUI components

loci/formats/codec Bio-Formats - codecs

In order to run the examples with visad.jar, download `ftp://ftp.ssec.wisc.edu/pub/visad-2.0/visad_examples.jar`
Unpack this jar file by running:

```
jar xvf visad_examples.jar
```

This will put `*.java` and `*.class` files into your `visad/examples` directory. Change to that directory and run the appropriate example application. Make sure that '.' is in your `CLASSPATH`.

1.7. Problems

If you have problems, send an email message to the VisAD mailing list at visad@unidata.ucar.edu
Join the list by sending an email message to majordomo@unidata.ucar.edu with:

```
subscribe visad
```

as the first line of the message body (not the subject line). Please include any compiler or run time error messages in the text of email messages to the mailing list.

Part I.

Ugo Taddei's VisAD Tutorial

This is the PDF Version of The VisAD Tutorial, originally written by Ugo Taddei and last updated on 19 august 2003. You can find the HTML Version of this tutorial on http://www.ssec.wisc.edu/~billh/tutorial/index.html

2. How to use this Tutorial

This tutorial introduces some basic features of VisAD in order to allow you to start programming with VisAD straight away. We assume no previous knowledge of the library itself, but an understanding of the Java® Programming Language is assumed. We shall not, however, need to go very deep into Java. In order to run the examples (and to later do your own development), you will need to install the VisAD package, and the Java2 and Java3D software (see the VisAD Prerequistes for more details).

Starting with a very simple example, we will explain how to create visualization programs for complex data structures. The reader should follow the basic tutorial steps, in order to maximize the understanding of VisAD and to learn how different displays can be created, as each step introduces a new feature.

The reader may, however, make use of the Index of Figures, where the program screenshots are listed and which serves as a visual reference guide on how to change display attributes and visualize data in different ways. The Index of Figures also includes links to the sections and to the program code, which is completely available.

The Table of Contents lists the sections and sub-sections and also is useful as an overview of both the tutorial and of VisAD capabilities.

3. Introduction to VisAD

This is the tutorial of VisAD, a Java Component Library for interactive analysis and visualization of numerical data. We will start by describing how to write a simple VisAD program to visualize some points as a single line and will, in section 2, continuously extend the program to show how to use some VisAD features. In section 3 we will turn our attention to 2-D `Sets`, which are the basis of images and 3-D surfaces.

3.1. Overview

A VisAD application generally starts with the definition of `Data` objects, which will represent your data in the application. VisAD's `Data` classes can represent simple numbers, such as a temperature, simple text strings, such as the name of a weather station, vectors of simple values, such as all of the data collected by a weather station (temperature, air moisture, precipitation, wind), and arrays of values, such as a times series of temperatures. In fact, VisAD `Data` objects can be assembled in complex hierarchies, known as `MathTypes`, to represent virtually any numerical and text data.

VisAD's Displays classes help you to construct displays of those data. These may be 2-D or 3-D, they may be animated, and they may be interactive.

VisAD defines classes for computational Cells that can also be linked to `Data` objects via `DataReference` objects (see below). Like Displays, Cells are updated whenever values of linked `Data` change. Cells take their name from spread sheet cells, because of the way spread sheet cells update when input values change.

VisAD also defines a variety of classes for User Interface objects. Many of these are based on the Java Swing® user interface toolkit, and they can all be easily embedded in a Swing GUI (Display objects are also easily embedded in Swing GUIs). The VisAD user interface classes are designed to help you design user interfaces for interactive control of VisAD Displays.

VisAD also provides a helper class, called `DataReference`, that is used for linking a `Data` object to a Display object. Once `Data` are linked to a Display object, the display will update whenever `Data` values change. In fact, Displays and Cells both extend Action, the general class for objects whose actions are triggered by changing `Data` values.

To summarize, VisAD applications are constructed with the following objects:

Data objects these range from simple real number values, text strings and vectors of real numbers, to complex hierarchies of data, which are referred to as `MathType`.

Display objects these generate interactive depiction of data. Display objects are linked to data objects through the use of `DataReference` objects (see below). Displays may be two- or three-dimensional, and provide extensive controls and direct manipulation.

Cell objects these are computations that are invoked whenever their input `Data` objects change value. Cells take their name from cells of spread sheets and are, like displays, linked to `Data` objects by means of a `DataReference` object.

User interface (UI) objects the user can use the Java Foundation Classes UI components as data input interfaces. Nevertheless, VisAD provides a few specialized UI components. The VisAD UI components may also be linked to `Data`, so that `Data` values may be changed by them. UI objects may also link to Actions so that they update whenever `Data` object values change.

DataReference objects these are pointers to `Data` objects. `DataReference` objects are necessary to represent variable data, just as "x" represents 3 in "x = 3".

Before we move on to our first application we need to consider the nature of the data that is to be visualized.

The VisAD data model defines a set of classes that can be used to build any hierachical numerical data structure. These complex hierarchical `Data` objects reflect the structure of the actual data. The primitive `Data` classes are the subclasses of `Scalar`: `Real` and `Text`, which contain a Java double and a text string, respectively. `Data` structure is achieved by using `Tuple`, `Set` and `Function` classes and their subclasses.

All `Data` objects have a `MathType`, which indicates the type of mathematical object that it approximates. Examples of `MathTypes` are: `ScalarType` (and its subclasses `RealType` or `TextType`), `TupleType` (and its subclass `RealTupleType`), `SetType`, and `FunctionType`.

Subclasses of `Data` are `Scalar`, `Tuple`, `Set` and `Function`. Subclasses of `MathType` are `ScalarType`, `TupleType`, `SetType`, and `FunctionType`. In a sense, the `Data` hierarchy reflects that of `MathType`.

Most applications include large `Data` objects that define some `RealTypes` as functions of other `RealTypes`. The starting point for any new application of VisAD is the definition of a set of `MathTypes`. For example, a simple function such as height = f(time), where time and height are `RealTypes`, is denoted in the VisAD documentation as

```
( time -> height )
```

and is defined with the `FunctionType ( time -> height )`. (Remeber: `FunctionType` is a subclass of `MathType`.)

A more complex data structure, like that of an image, might be defined with the `MathType`:

```
( ( row, column )  ->  ( red, green, blue ) )
```

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

```
( time -> ( (latitude, longitude, altitude)
-> (temperature, pressure, dew_point, wind_u, wind_v, wind _w ) ) )
```

So we move on to talk about a few aspects to consider when designing a `VisAD` application.

3.2. Designing a Typical VisAD Application

When writing a VisAD application there are three main steps to take:

1. Creation of the data you want to visualize,
2. Creation of the display and other visualization objects and
3. Adding interaction and functionality through the use of user interfaces, UI, or widgets.

Let us assume we have some data, and we want to build a VisAD application to visualize those data. For example, we have a cube, and we have calculated and/or measured the temperature inside it. So let us consider the above steps in more detail.

3.2.1. Creating Data

This is the first and most important step in designing a VisAD application. Although the display and its objects define to a great extent how data is to be drawn, the depiction also depends on the data structure. You might, for example, create a data structure, a `MathType`, to draw a one-dimensional function as a line:

```
(x -> y)
```

and then force the display to draw the individual points, rather than to connect them to make a line. On the other hand, your `MathType` might describe a set of (x,y) points indexed by some variable, or in VisAD notation:

```
( index -> (x,y) )
```

With the `MathType` above you're saying, that you have a set of disconnected points. In this case, there is no way to force the display to connect the points. You'd have to create a new `MathType`.

The Domain

The first step in creating data with VisAD `Data` Objects is to identify the basic quantities, or scalars. For example, if we have a cube, we identify three scalars: height, width and length. In VisAD, those would be `ScalarTypes`. As you know, `ScalarTypes` has two subclasses: `RealType` and `TextType`. The latter is for use with text, whereas the former is for use with "real" numbers ("real" in mathematical sense). So our three cube dimensions are `RealTypes`. `RealTypes` are static within a VisAD application. That means in practice, you can reuse them without the need to recreate them. They are generally constructed with a Java String, that is, their name, and with a VisAD unit. The units will be considered, for example, in calculations. Say, to create a `RealType` h, for "height", you do:

```
RealType height = RealType.getRealType("Height");
```

or you can do

```
RealType height = RealType.getRealType("Height", SI.meter, null);
```

to create a `RealType` with a unit, "meter". (Ignore the third argument for now; it defines the default `Set` of this `RealType`.) This static method of the `RealType` class will look for an already existing `RealType` called "height". If such `RealType` already exists, then you cannot use the constructors above. Use instead the static methods.

So we have identified our cube dimensions as `RealTypes`. Together, the three of them form a tuple, or, in VisAD, a `RealTupleType`. There are many ways to create a `RealTupleType`, and we are going to come across them later on in the tutorial. For now it suffices to say that we have created the basic "cube structure" with:

```
RealTupleType cubeTuple = new RealTupleType( height, width, length);
```

But what about the cube itself? How big is it? To answer these questions we have to know "how" you're defining your cube. Are you measuring height, width and length at constant intervals or not? Are you measuring the vertices only, or are you measuring some random points inside the cube? VisAD has a collection of data objects, `Sets`, to represent different kinds of samplings. We assume our cube is 1 meter high, 2 meters wide and 3 meters long. Furthermore we assume we measure height and width every 10 cm, but, for lazyness' sake, we measure length every 50 cm, only. And for freedom's sake, we decide to put the cube's width in the middle of our reference system. That is, the extreme values for the width are -1 and 1. (OK, to be precise about it, it's a parallelepiped rather than a cube, but let's call it "cube"). All this information, together with the tuple of the cube dimensions, are defined in a three-dimensional `Set`:

```
cubeSet = new Linear3DSet(cubeTuple,  // basic quantities given by height, width and length
  0.0,  1.0, 11,          // height starts at 0.0 m, ends at 1.0 m, and has 11 samples
  -1.0, 1.0, 21,          // width starts at -1.0 m, ends at 1.0 m, and has 21 samples
  0.0,  3.0,  7 );        // length starts at 0.0 m, ends at 3.0 m, and has 7 samples, one value every 50 cm
```

The word "Linear" means that sampling is regular. If the sampling is regular, and they occur at integer values, say, from 0 to N, than consider using an `"Integer" set`. If you were to sample the cube in a grid whose points are not regularly spaced, but they nevertheless form a grid, than you'd use a `"Gridded" Set`, and provide the `Set` with the individual height, width and length values. Should you know nothing about the topology of your sampling, that is, whether they form a grid or whether they are just randomly spread inside the cube, than you might want to use an "Irregular" `Set`, and let VisAD figure the topology out. you may have already guessed, that VisAD has 1-D and 2-D, as well an N-D and other `Sets`. Please refer to the VisAD Java Component Library Developers Guide for more details.

The Range

Ok, we've already got some data, and we could create a display and add the cube to it. But most certainly you are trying to visualize how some quantity (or quantities), a `RealType` (or a `RealTupleType`) vary according to some other quantity. Like in maths, you have one or more dependent variables as functions of independent variables. What we do in VisAD is precisely that. We create the independent variable(s), create the dependent variable(s) and then use an object to establish the mathematical function between them. In VisAD one often refers to the independent variables as "domain" and to the dependent variables as "range". So far, we've created the domain. Our domain is the cube given by the `RealTupleType` and its set is the `Linear3DSet`, which we call "domain set". So what about the range? We assume we are measuring the temperature inside the cube. We need to create the corresponding `RealType`:

```
RealType temperature = new RealType("Temperature");
```

Of course you might want to measure temperature and some other quantity, in which case you'd need another `RealType`, and you'd have a range composed by a `RealTupleType`. We will do this later in the tutorial, for now we want to show how you create a function:

```
cubeTempFunc = new FunctionType(cubeTuple, temperature);
```

That is, "temperature" is a function of height, width and length. You create a `FunctionType` with two `MathTypes` (remember, `MathType` is the superclass of `RealTupleType`, `RealTupleType`, `FunctionType`, etc). The first `MathType` is the domain, and the second, the range.

The `FunctionType` creates the relation between the `MathTypes` of the domain and the range, but it says nothing about "the data" itself. Furthermore, how do you link the cube given by the `Linear3DSet`, with the function given by the `FunctionType`, and those with the "temperature" values, which you are measuring and/or computing? The answer is a `Field` object, or more especifically a `FlatField`.

A `FlatField` is a subclass of `Field`, which is a subclass of `Function` (but not of `FunctionType`!), which is a subclass of `Data`, and thus a Visad data object. A `Field` represents a mathematical function. Inside it there's information about the domain, the domain set, the range and the range values. A `FlatField` is an extension of `Field`, and has been designed with computational efficiency in mind. Inside a `FlatField` you pack the `FunctionType`, the domain `Set` and then you "feed" it with with range ("temperature") values. The `FlatField` has quite a few useful methods and in the first few tutorial chapters we're going to make good use of it. A `FlatField` is created with:

```
FlatField tempInCube_ff = new FlatField( cubeTempFunc, cubeSet)
```

That is, the first parameter is the `FunctionType` and the second parameter is the domain `Set`. We have called our `FlatField` `tempInCubeFF`, note the "ff" at the end to denote its type. Of course, you don't need to do so, but it'll be done throughout this tutorial. As said, the `FlatField` holds not only the "temperature" values you'll provide, it also includes the `FunctionType`, with its domain and range types, and the domain `Set`. That is, quite a few things to fit in a short name, so therefore the "ff" at the end. In the tutorial, whenever you come across an object with "ff" at the end, remember, it's a `FlatField` and expect a lot from it.

Well, we are almost done with the creation of a not-so-simple data object. The only two things missing is to feed the `FlatField` with actual "temperature" values and to add the whole data to a display. Note that the `FlatField` above will be waiting for an array of floats (or doubles) with the shape float[range_dimension][

number_of_samples]. The first dimension corresponds to the dimension of the range, in our case it's 1, as we have "temperature", only. The second dimension is the total number of temperature values, which is 10 x 20 x 6, as given by the domain `Set`. You'd set the values with a call

```
FlatField.setSamples( float[][] temperValues );
```

Now we've got some complex data ready to be displayed, so we move on to consider the display of data.

3.2.2. Displaying Data

The first question you might ask yourself is whether to use of a 2D or of a 3D display. Luckily, in VisAD the choice of display is independent of the data. Whether it makes sense to use a 2D or 3D is up to you to decide. There are a variety of displays constructors and display renderers. In the tutorial, we will come across some of them. For now it's important to understand how VisAD displays data.

When building up the data structure, we identified basic quantities as `RealTypes`. When thinking of the cube of the previous example, it would be obvious to map each one of the dimensions to an axis of a 3D display. The object responsible for the mapping is a `ScalarMap`. When creating a `ScalarMap` you consider two things:

1. Which `ScalarType` will you map and
2. Where will you map it to.

The first point is clear, but remember that `ScalarType` is the superclass of `RealType` and of `TextType`. The "where will you map it to" implies not only the axes of a display, but also color, animation, iso-contours, text, shape and many others. These are known in VisAD as `DisplayRealTypes`, and they define how `RealTypes` are to be displayed. Let us look closer at such a `ScalarMap` by constructing a few:

```
ScalarMap heightZMap = new ScalarMap( height, Display.ZAxis );
ScalarMap widthXMap = new ScalarMap( width, Display.XAxis );
ScalarMap lengthYMap = new ScalarMap( length, Display.YAxis );
```

This should be pretty clear: we are mapping "height" to the z-axis, width to the x-axis and length to the y-axis. We haven't said anything yet about the display. If the display has such a map, then it'll map "height" data to the z-axis. Suppose we do the same for the other cube dimensions, then we have the whole cube in a 3D display.

To color the cube according to temperature values, you'd do

```
ScalarMap temperRgbMap = new ScalarMap( temperature , Display.RGB );
```

and you'd obtain a cube colored according to the "temperature" values. (The actual color table is predefined, but you can redefine it. See Section 4 for some examples of colored cubes und user-defined color-tables.)

`ScalarMaps` have a boring but helpful relative, the `ConstantMaps`. `ConstantMaps` extend `ScalarMaps`, but take no `ScalarType` as a parameter in the constructor. Instead, they take a (constant) Java double or a VisAD `Real` (`Real` is a subclass of `Data`). With a `ConstantMap` map you may add a constant shade of red, say 40

```
ConstantMap constRedMap = new ScalarMap( 0.4 , Display.Red );
```

or you can put some data at some constant place in a display and/or give it a constant color. For example, you could give a constant green color to a line, or put a surface at some z-value.

After choosing how to depict your data by choosing the right types of `ScalarMaps` and `ConstantMaps`, you add them to the display (you will see in the next section how this is done).

Having added all `ScalarMaps` of your choice, you have to tell the display which data to draw. For that, you use a `DataReference`. You feed a `DataReference` with the data you want, like, for example, with a `FlatField`, and then you add the `DataReference` to the display. At this step, you might add the data with an array of `ConstantMaps`, to give your data some different properties.

3.2.3. Interacting with Data

This step might not seem so important as the previous two, but, in fact, you only reach the desired usability of your application with a proper user interface. Apart from the standard Java UI, which you can use in your application, VisAD provides a number of special UIs. Interaction in VisAD generally occurs with the help of a Control. Control is a class which is implemented by `GraphicsModeControl`, `ColorControl`, `AnimationControl` and others. In particular, you'll find that most VisAD Controls have a corresponding UI. The choice of UIs depends not only on your data, but also on how interactive your application can and should be. One thing to notice, though, is that in VisAD the display is the main user interface. Not only does it provide the user with information about the data, but it supports interactive rotation, pan and zoom. It can also have, for example, `DirectManipulationRenderers`, so that user input occurs directly through the display. By using widgets not only can you change data depiction, but you can also change data values. This might trigger calculations, which, in turn, might change data. As we move along the tutorial, we'll get to know the VisAD widgets.

3.2.4. Summary

Before we start with our first VisAD application, we recap the main steps. When building the data structure, identify the basic quantities as `ScalarTypes`, that is `RealTypes` or `TextTypes`. Pack `RealTypes` in a `RealTupleType`. Use a `Set` (1D, 2D, 3D or N-D and Linear, Gridded, Integer, Irregular or other) as the domain `Set`. Build the range with the `RealTypes` identified as the independent variables. If there are more than one `RealType`, create a `RealTupleType` for the range. Create a `FunctionType` with the domain and the range. Create a `FlatField` based on the function and on the domain set. Put the range values in the `FlatField`.

For visualization, start with a display. Create the `ScalarMaps` you find necessary and add them to the display. Create a `DataReference`, feed it with data, add it to display. The display will be added to a Java Frame or other Java Component to be shown.

Use Controls to set parameters and customize your display. Refine your application with widgets.

We are then ready to write our first VisAD application, which will have a very simple `MathType` (just a `RealType`, called height, as a function of the `RealType` time, that is a `FunctionType ( time -> height )`), as well as a `Field` and a `Set`, a `DataReference` and a Display.

3.3. Our First VisAD Application

In this section we will plot a simple function, height = f(time), whose `MathType` reads:

```
( time -> height )
```

We assume time to be our independent variable and are given some values for height. We define time and height as `RealTypes`. `Data` for time is organized in an `Integer1DSet` (a subclass of `Set`, which is a subclass of `Data`). This `Set` is our domain `Set`. As the name says, this `Set` is a one-dimensional set of (consecutive) integers. We will also need a `FunctionType` (function our height = f(time)), a `FlatField` (another `Data` object), a `DataReference` (to link our `Data` to the display), a 2D display and two `ScalarMaps` to be included in the display.

`ScalarMaps` are objects which determine how `Data` objects are depicted. They define mappings from `RealTypes` (such as our time and height) to `DisplayRealTypes`, which are, for example, the x-, y- and z-axis, or the color components, or animation, etc. We will then use a Java Frame to show our display.

Listing 3.1: The source code of our first VisAD Application

```java
// Import needed classes

import visad.*;
import visad.java2d.DisplayImplJ2D;
import java.rmi.RemoteException;
import java.awt.*;
import javax.swing.*;

/**
Java Tutorial Example 1_01
The first tutorial example. A function height = f(time), represented by the
MathType ( time -> height ), is plotted as a simple line.
this function is actually the parabola height = 45 - 5 * time^2,
We have the height values and time is the continuous independent variable, with
data values given by a Set.
Run program with "java P1_01"
*/

public class P1_01{
  // Declare variables
  // The quantities to be displayed in x- and y-axis
  private RealType time, height;

  // The function height = f(time), represented by ( time -> height )
  private FunctionType func_time_height;

  // Our Data values for time are represented by the set
  private Set time_set;

  // The Data class FlatField, which will hold time and height data.
  // time data are implicitely given by the Set time_set
  private FlatField vals_ff;

  // The DataReference from the data to display
  private DataReferenceImpl data_ref;

  // The 2D display, and its the maps
  private DisplayImpl display;
  private ScalarMap timeMap, heightMap;

  // The constructor for our example class
  public P1_01 (String []args)
    throws RemoteException, VisADException {

    // Create the quantities
    // Use RealType(String name)
    time = new RealType("time");
    height = new RealType("height");

    // Create a FunctionType, that is the class which represents our function
    // This is the MathType ( time -> height )
    // Use FunctionType(MathType domain, MathType range)
    func_time_height = new FunctionType(time, height);


    // Create the time_set, with 5 integer values, ranging from 0 to 4.
    // That means, that there should be 5 values for height.
```

```java
    // Use Integer1DSet(MathType type, int length)
    time_set = new Integer1DSet(time, 5);

    // Those are our actual height values
    // Note the dimensions of the array:
    //   float[ number_of_range_components ][ number_of_range_samples]
    float[][] h_vals = new float[][]{{0.0f, 33.75f, 45.0f, 33.75f, 0.0f,} };

    // Create a FlatField, that is the class for the samples
    // Use FlatField(FunctionType type, Set domain_set)
    vals_ff = new FlatField( func_time_height, time_set);

    // and put the height values above in it
    vals_ff.setSamples( h_vals );

    // Create Display and its maps A 2D display
    display = new DisplayImplJ2D("display1");

    // Create the ScalarMaps: quantity time is to be displayed along x-axis
    // and height along y-axis
    // Use ScalarMap(ScalarType scalar, DisplayRealType display_scalar)
    timeMap = new ScalarMap( time, Display.XAxis );
    heightMap = new ScalarMap( height, Display.YAxis );

    // Add maps to display
    display.addMap( timeMap );
    display.addMap( heightMap );

    // Create a data reference and set the FlatField as our data
    data_ref = new DataReferenceImpl("data_ref");
    data_ref.setData( vals_ff );

    // Add reference to display
    display.addReference( data_ref );

    // Create application window, put display into it
    JFrame jframe = new JFrame("My first VisAD application");
    jframe.getContentPane().add(display.getComponent());

    // Set window size and make it visible
    jframe.setSize(300, 300);
    jframe.setVisible(true);
  }


  public static void main(String[] args)
    throws RemoteException, VisADException {
      new P1_01(args);
    }
}
```

The source code is available here.

By pressing and dragging with the left mouse button on the display you can move the graph around. By shift-clicking and moving the mouse up and down you can zoom in and out. Pressing and dragging the middle mouse button (on two-button mouse emulated by simultaneously clicking both buttons) shows a cross cursor that moves with the mouse. The values of the `RealTypes` at the cursor's position are shown on

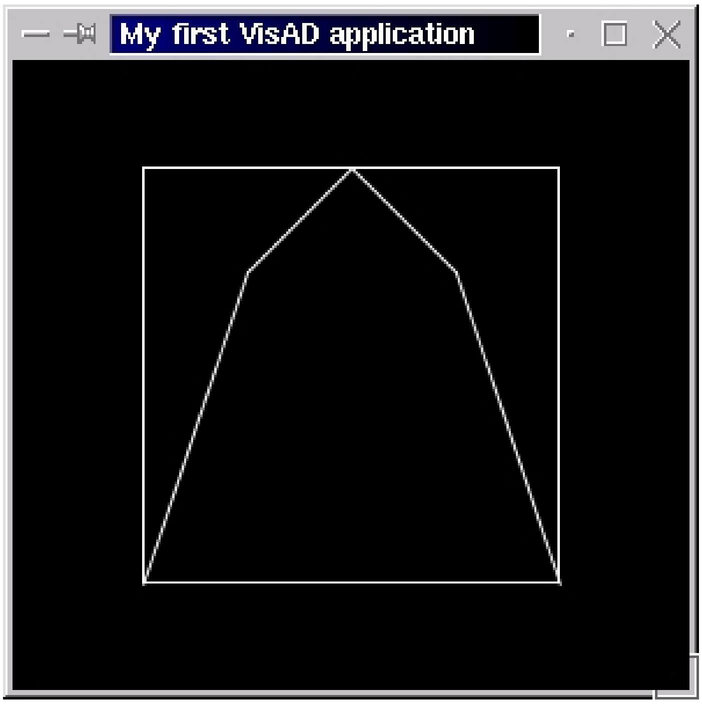

Figure 3.1.: If you compile the source code with `javac P1_01.java` and run with `java tutorial.s1.P1_01` you should be able to see the this window

the upper left corner of the display.

In the following section we will look further into 2D graphs and show how to control color, axes properties, and show how a different `MathType` structure can lead to a different rendering.

4. The Basics

4.1. Drawing scales and using units for RealType

In this example, we draw scales and both x- and y-axis. The example P2_01 is almost the same as the previous example. This time we define our `RealTypes` time and height with units:

```
time = new RealType("time", SI.second, null);
```

and

```
height = new RealType("height", SI.meter, null);
```

The first argument in the constructor is the name (a Java String) of the `RealType`. This name will by used to label the axes. You can get the name of a `RealType` with the method `RealType.getName()`. The method `RealType.getRealTypebyName( String name )` will return the `RealType` whose name is "name". Note that two `RealTypes` are equal if their names are equal. The second argument is the unit of the `RealType`. VisAD defines all SI units (ampere, candela, kelvin, kilogram, meter, second, mole and radian) and provides methods for defining your own units.

In section 2.7 we will create a new unit. By the way, you can get a `RealType`'s unit with the method `RealType.getDefaultUnit()`. The third argument in the constructor is the default set of the `RealType`. We shall ignore the set for the time being. The next addition we make to the first example is the call

```
GraphicsModeControl dispGMC = (GraphicsModeControl) display.↩
    getGraphicsModeControl();
```

that defines the variable `dispGMC` as display's `GraphicsModeControl`, and the subsequent call

```
dispGMC.setScaleEnable(true);
```

which specifies that scales should be drawn.

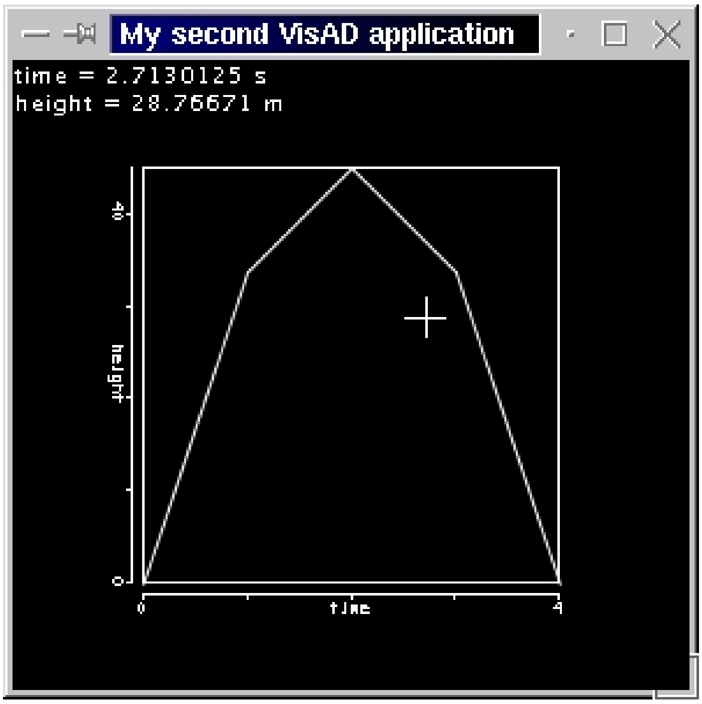

Figure 4.1.: Running program P2_01 generates a window like the shown. Note that the axes are now labelled, and the cursor's position (time and height) is correctly given in seconds and meters.

4.2. Scaling axes

You may have noticed that both axes were automatically scaled. In this section we will show how to manually scale the axes. Before we do that, we'll make another change in our program. We shall now use a different `Set` for the `RealType` time. In the previous examples, time `Data` was given by an `Integer1DSet`, the time_set. Now time_set will be a `Linear1DSet`. Note the arguments that this `Set` takes.

```
time_set = new Linear1DSet(time, -3.0, 3.0, 5);
```

We still use the same 5 height values, but now the parabola is correctly placed in the graph, that means time doesn't range from 0 to 4, because we use an adequate `Set`. Note that the parabola is given by $height = 45 - 5 \cdot time^2$. The `Integer1DSet` was used initially because we were not interested in the mathematical correctness, but only in having a set of 5 values.

After adding the heightMap to the display, we scale the y-axis (remember, heightMap has YAxis as `DisplayRealType`) with

```
heightMap.setRange( 0.0, 50.0);
```

The figure 4.2 is a screen shot of the example P2_02.

4.3. Plotting points by using a different MathType

We will now use a different `MathType` to organize our data in a different way, and see how the data structure gets depicted in a coherent way. Note that our previous `MathType` indicates a continuous function. Our new `MathType`, organized as

`( index -> (time, height) )`

suggests, on the other hand, a set of (time, height) points, which are indexed by an `Integer1DSet` (index_set). The difference is not a trivial one. A continuous line like that of the previous example might represent the theoretical values of a continuous function and therefore is plotted as such. The latter `MathType` might represent a set of values from an experiment which should, therefore, be plotted disconnected. As said, we are going to use an `Integer1DSet` for index. In order to organize time and height, we will use a `Tuple`:

```
private RealTupleType t_h_tuple;
t_h_tuple = new RealTupleType( time, height);
```

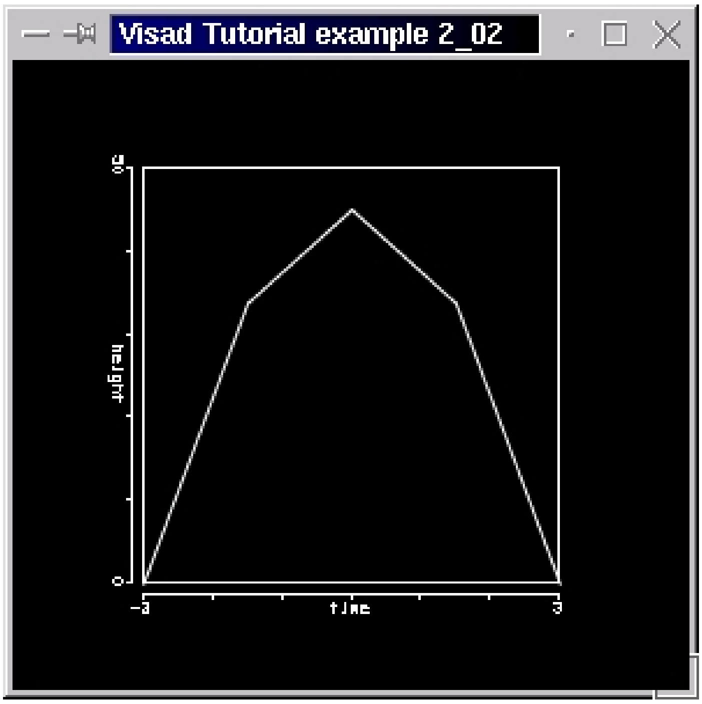

Figure 4.2.: Note that the y-axis is now scaled from 0 to 50.

The `FunctionType` now becomes:

```
func_i_tuple = new FunctionType( index, t_h_tuple);
```

And the `FlatField` is changed to include this:

```
vals_ff = new FlatField( func_i_tuple, index_set);
vals_ff.setSamples( point_vals );
```

The x-axis and the y-axis will be arbitrarily rescaled in the range from -4 to 4 and -10 to 50, respectively, using setRange().

The code for the complete example 2_03 is as follows:

Listing 4.1: The source code of our VisAD Example Application P2-03

```
// Import needed classes

import visad.*;
import visad.java2d.DisplayImplJ2D;
import java.rmi.RemoteException;
import java.awt.*;
import javax.swing.*;

/**
VisAD Tutorial example 2_03
Data are organized as MathType ( index -> ( time, height ) ) and
represent some points from the parabola of the previous example
Data are indexed (time, height) points and get depicted as such.
Run program with java P2_03
*/
public class P2_03{
  // Declare variables
  // The quantities to be displayed in x- and y-axes: time and height, respectively
  // Our index is alos a RealType
  private RealType time, height, index;

  // A Tuple, to pack time and height together
  private RealTupleType t_h_tuple;

  // The function ( time(i), height(i) ), where i = index,
  // represented by ( index -> ( time, height) )
  // ( time, height) are a Tuple, so we have a FunctionType
  // from index to a tuple
  private FunctionType func_i_tuple;

  // Our Data values, the points, are now indexed by the Set
  private Set index_set;

  // The Data class FlatField, which will hold time and height data.
  // time data are implicitely given by the Set time_set
  private FlatField vals_ff;
```

```java
  // The DataReference from the data to display
  private DataReferenceImpl data_ref;

  // The 2D display, and its the maps
  private DisplayImpl display;
  private ScalarMap timeMap, heightMap;

  public P2_03 (String []args)
    throws RemoteException, VisADException
  {
    // Create the quantities
    // x and y are measured in SI meters
    // Use RealType(String name, Unit u,  Set set), set is null
    time = new RealType("time", SI.second, null);
    height = new RealType("height", SI.meter, null);

    // Organize time and height in a Tuple
    t_h_tuple = new RealTupleType( time, height);

    // Index has no unit, just a name
    index = new RealType("index");

    // Create a FunctionType ( index -> ( time, height) )
    // Use FunctionType(MathType domain, MathType range)
    func_i_tuple = new FunctionType( index, t_h_tuple);

    // Create the x_set, with 5 values, but this time using a
    // Integer1DSet(MathType type, int length)
    index_set = new Integer1DSet(index, 5);

    // These are our actual data values for time and height
    // Note that these values correspond to the parabola of the
    // previous examples. The y (height) values are the same, but the x (↩
        time)
    // are now given given.
    float[][] point_vals = new float[][]{
      {-3.0f, -1.5f, 0.0f, 1.5f, 3.0f,},
      {0.0f, 33.75f, 45.0f, 33.75f, 0.0f,}
    };

    // Create a FlatField, that is the Data class for the samples
    // Use FlatField(FunctionType type, Set domain_set)
    vals_ff = new FlatField( func_i_tuple, index_set);

    // and put the height values above in it
    vals_ff.setSamples( point_vals );

    // Create Display and its maps
    // A 2D display
    display = new DisplayImplJ2D("display1");

    // Get display's graphic mode control and draw scales
    GraphicsModeControl dispGMC = (GraphicsModeControl) display.↩
        getGraphicsModeControl();
    dispGMC.setScaleEnable(true);

    // Create the ScalarMaps: quantity time is to be displayed along XAxis
    // and height along YAxis
    // Use ScalarMap(ScalarType scalar, DisplayRealType display_scalar)
    timeMap = new ScalarMap( time, Display.XAxis );
    heightMap = new ScalarMap( height, Display.YAxis );
```

```
    // Add maps to display
    display.addMap( timeMap );
    display.addMap( heightMap );

    // Scale heightMap. This will scale the y-axis, because heightMap has ↩
        DisplayRealType YAXIS
    // We simply choose the range from -4 to 4 for the x-axis
    // and -10.0 to 50.0 for
    timeMap.setRange( -4.0, 4.0);
    heightMap.setRange( -10.0, 50.0);

    // Create a data reference and set the FlatField as our data
    data_ref = new DataReferenceImpl("data_ref");
    data_ref.setData( vals_ff );

    // Add reference to display
    display.addReference( data_ref );

    // Create application window, put display into it
    JFrame jframe = new JFrame("VisAD Tutorial example 2_03");
    jframe.getContentPane().add(display.getComponent());

    // Set window size and make it visible
    jframe.setSize(300, 300);
    jframe.setVisible(true);
  }

  public static void main(String[] args)
    throws RemoteException, VisADException
  {
    new P2_03(args);
  }
}
```

The source code is available here, you should get an result like figure 4.3.

As expected, our data consisting of a set of points was plotted as such (on some high-resolution screens the points may be so small - a single pixel - that they are nearly invisible). Note that if you call

```
dispGMC.setPointMode(true);
```

where dispGMC is the `GraphicsModeControl` you can draw all lines in a display as points, without changing the `MathType`. The advantage is that you don't need to change the `MathType`, but the disadvantage is that you will only be able to draw your data as points.

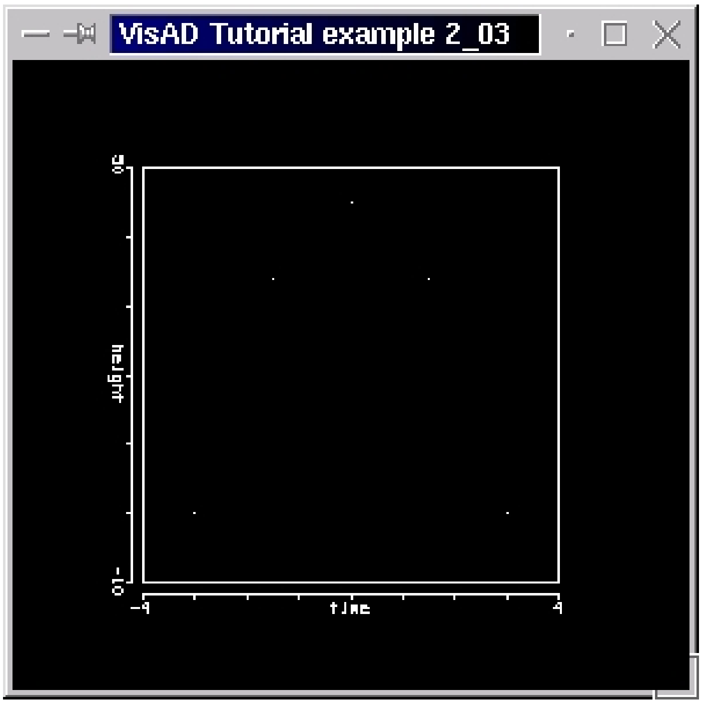

Figure 4.3.: If you compile the source and run it with java P2_03 you should be able to see the following window.

4.4. Using a ConstantMap to Change Data Depiction Attributes

In the above figure, our data were plotted as the disconnected points. On some displays the points are so small that they are difficult to see, so you might want them larger or you simply might want them plotted with some color. We will now change some basic attributes of those points by using color and points `ConstantMaps`. This is done by defining `ConstantMaps` like

```
ConstantMap[] pointsCMap = { new ConstantMap( 1.0f, Display.Red ),
new ConstantMap( 0.0f, Display.Green ),
new ConstantMap( 0.0f, Display.Blue ),
new ConstantMap( 3.50f, Display.PointSize )  };
```

Note that `Display.Red` is at its maximum (1.0f) and `Display.PointSize` defines the size of the points in pixels. The change is implemented by linking this `ConstantMap[]` with the display using the call:

```
display.addReference( data_ref, pointsCMap );
```

The source code is available here. If you compile it and run with java P2_04 you should be able to see the figure 4.4.

4.5. Using a SelectRange Map to limit plotting and adding two DataReferences to a display

We now combine examples 2_02 and 2_04 above to make a display that shows data plotted as both single points and as a line. To do this we will use one display object, but we will require two `FunctionTypes`, two `FlatFields`, and two `DataReferences`. First, create a `Linear1DSet` to define the sampling.

```
int LENGTH = 25;
time_set = new Linear1DSet(time, -3.0, 3.0, LENGTH );
```

Next, in addition to func_i_tuple created in the previous example, we need to create:

```
func_time_height = new FunctionType(time, height);
```

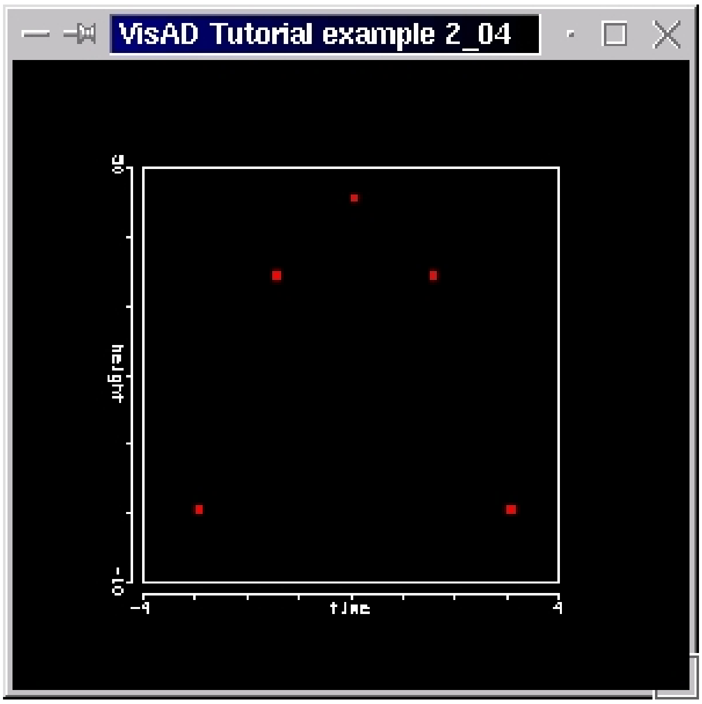

Figure 4.4.: The points are now displayed as red color-filled squares, 3 pixels on a side.

Now, we need to create two arrays to hold our data values, and fill them appropriately:

```
float[][] d_vals = time_set.getSamples( true);
float[][] h_vals = new float[1][LENGTH];
```

We generate values for the line with a for-loop.

```
for(int i = 0; i < LENGTH; i++)
h_vals[0][i] =  45.0f - 5.0f * (float) (d_vals[0][i]*d_vals[0][i]);
```

This provides the data to make a curve of 25 connected points. Another new feature in this example is the use of a `ScalarMap` with a `SelectRange` as `DisplayRealType` to select the time range to be displayed. In section 2.2 we scaled the y-axis. If we had scaled it so that our data would be outside the display box, data would still have been displayed (but outside the box!). A `ScalarMap` with `SelectRange` as `DisplayRealType` will trim data depiction, so that only the data inside the valid range will be drawn. (You might want to combine a `ScalarMap.setRange()` call with a `ScalarMap` with `SelectRange` as `DisplayRealType`, as the former allows you to choose the range of the depicted `RealType` (thus scaling an axis) and the latter allows to choose the range in which the `RealType` can be draw (values ouside the range will be cut). You can also see section 2.11, where the difference between them should become clear.) We create such a `ScalarMap` with

```
timeRangeMap = new ScalarMap( time, Display.SelectRange );
```

add it to a display like we would do with any other `ScalarMap`:

```
display.addMap( timeRangeMap );
```

Then we get this `ScalarMap`'s `RangeControl`

```
RangeControl timeRangeControl = (RangeControl) timeRangeMap.getControl();
```

To make the data displayed or visible in the range from -2 to 4, define the range

```
float[] timeRange = { -2.0f, 4.0f };
```

and finally implement the changes by calling

```
timeRangeControl.setRange( timeRange );
```

The complete source code for this example is available here. If you compile it and run with `java P2_05` you should be able to see the window 4.5

4.6. Extending the MathType and using Display.RGB

In the previous examples, our line had the following `MathType`

`( time -> height )`

We mapped time to XAxis and height to YAxis. We will now extend this `MathType` to

`( time -> (height, speed) )`

where height and speed are `RealTypes` and form a `Tuple`:

`h_s_tuple = new RealTupleType(height, speed);`

Note that speed is the first derivative of height with respect to time. (You might like to know that the interface `Function` (subinterface of `Data`) actually includes a method to calculate the derivative of a `Function` with respect to a `RealType`. Remember that in VisAD a `FlatField` represents a mathematical function. It overrides the `Data.derivative()` method, with which you can calculate the derivative of a function, that is of a `FlatField`, with respect to a quantity, that is a `RealType`. But in this example we shall calculate the derivative in a for-loop. See section 4_07 for how to use `Data.derivative()`.) Our `FunctionType` is now a function

`func_t_tuple = new FunctionType(time, h_s_tuple);`

Note that the array which will hold the height and speed values is now dimensioned like `float[ number_of_range_components ][ number_of_range_samples]`, that is `float[2][LENGTH]`:

`float[][] h_s_vals = new float[2][LENGTH];`

We compute the values for height and speed with for-loop:

```
for(int i = 0; i < LENGTH; i++){

// height values...
h_s_vals[0][i] =  45.0f - 5.0f * (float) (t_vals[0][i]*t_vals[0][i]);

// ...and speed values: the derivative of the above function
h_s_vals[1][i] =  - 10.0f * (float) t_vals[0][i];
}
```

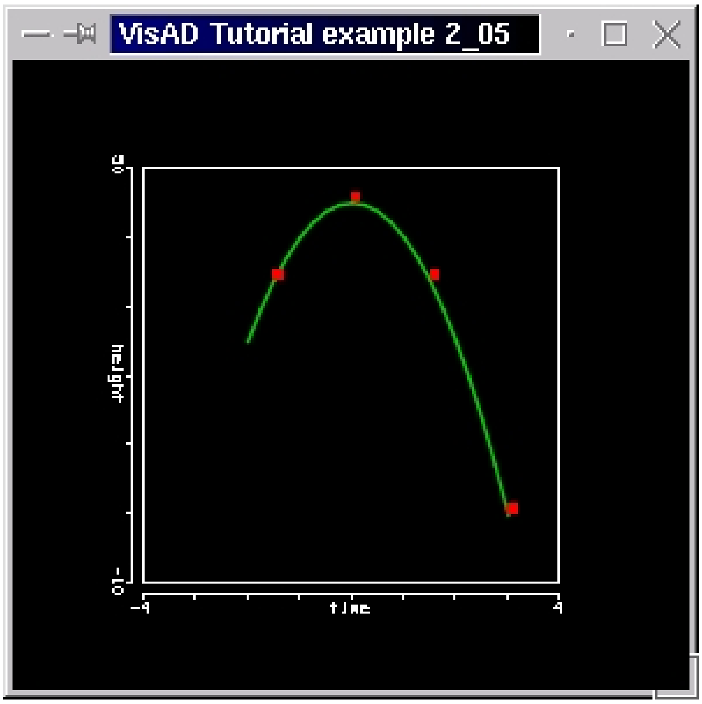

Figure 4.5.: Note that data outside the range from -2 to 4 is not shown. (Compare with the previous figures, in which both line and points existed in the range -3 to 3.) Note too the smoother curve, indicating the use of more data (for the line). Of course, the plots of the point value data and the data describing the line are completely independent (try changing their values in the code).

We create the `ScalarMap`

```
speedRGBMap = new ScalarMap( speed, Display.RGB );
```

to display speed in RGB color and then we add this `ScalarMap` to the display. The source code for this example is available here. The figure 4.6 is a screen shot of this program.

4.7. New Units, and changing line width with GraphicsModeControl

This example is almost the same as the previous example. This first little change regards the units of speed. We declare our new `Unit` with

```
Unit mps;
```

define it with

```
mps = SI.meter.divide( SI.second );
```

and refine speed with the new unit:

```
speed = new RealType("speed", mps, null);
```

That is, our new `Unit` is represented by mps and is simply `SI.meter` divided by `SI.second`, or simply meters per second. We define speed's units as mps. We also invert the `ScalarMaps` and map height to RGB and speed to YAxis, as shown below:

```
speedYMap = new ScalarMap( speed, Display.YAxis );
```

and

```
heightRGBMap = new ScalarMap( height, Display.RGB );
```

Just another little change is the call

```
dispGMC.setLineWidth( 3.0f );
```

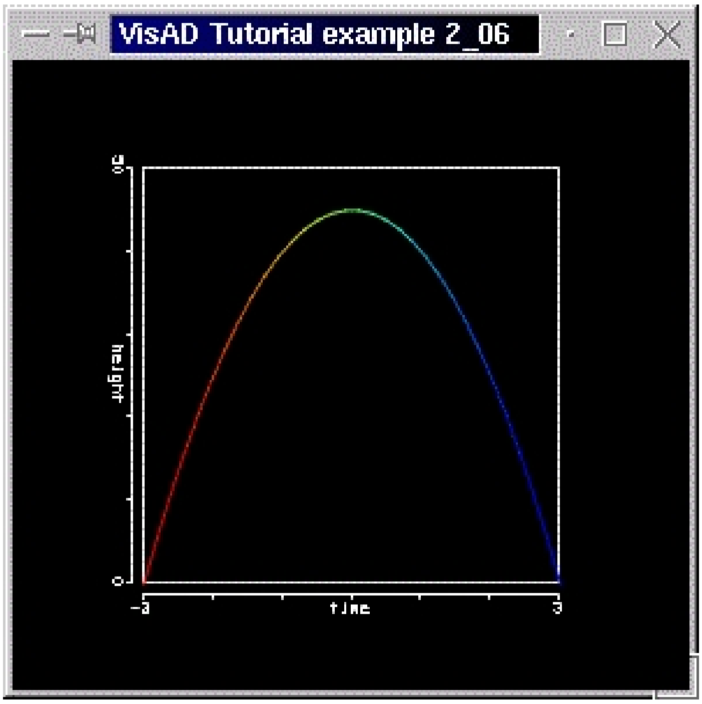

Figure 4.6.: Note that speed is signed, so positive (upward) values are colored red and negative (downward) values are colored blue. The color look-up table ranges from blue, through green to red. The color table is automatically adjusted to the `RealType` it's attached to, so that the minimum value of the `RealType` is mapped to the minimum value of the color table (blue) and the maximum value of the `RealType` corresponds to that of the color table (red). Intermediate values are linearly interpolated. In section 3 we will learn how to color a `RealType` using other `DisplayRealTypes` (like Red, Green and Blue, rather than RGB, which is attached to a pseudo color look-up table). In section 4 we will define and use a new color table.

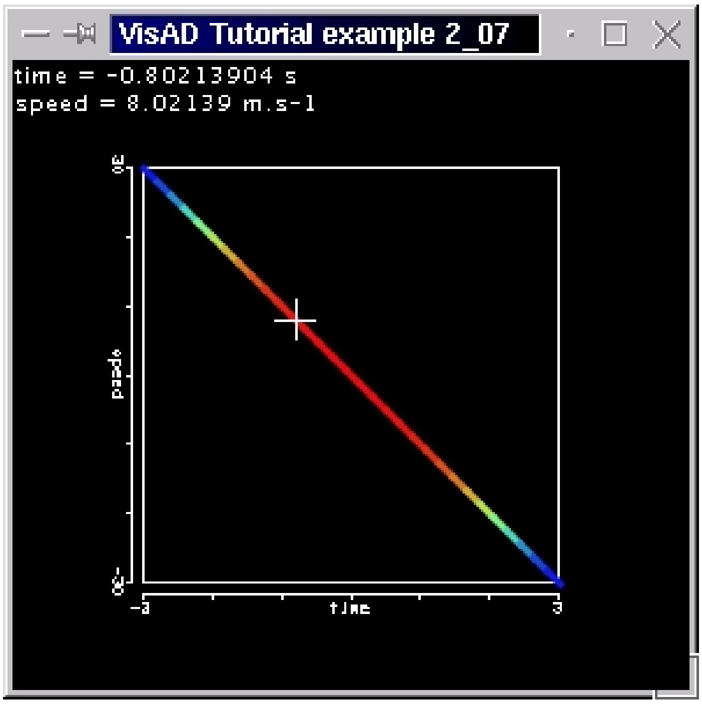

Figure 4.7.: Note that speed is displayed in the y-axis, and the values for time and speed at the cursor's location are correctly given in the proper units. Remember, not speed but height is mapped to color, so now we see red near time = zero, where the height is at the maximum, and blue at both ends where heights are near the minimum.

which results in thicker lines for the display. The source code for this example is available here and figure 4.7 is a screen shot of the program.

You might want to know, that the method `GraphicsModeControl.setPointWidth( float size )` changes the point size of a display (but is over-ridden by `ConstantMaps` to `Display.PointSize`, as used in section 2.4). Although the `GraphicsModeControl` provides you extensive controls over data depiction, VisAD provides a user interface for the it. In order to make full use of this interface, we shall only introduce it in section 3.8.

4.8. Plotting two quantities on same axis

In this section we restructure the `MathType` of the previous program

`( time -> (height, speed) )`

as

`( time -> height )`

and

`( time -> speed )`

We will need a `FunctionType` for each of the above functions

```
func_t_h = new FunctionType( time, height );
```

and

```
func_t_s = new FunctionType( time, speed );
```

as well as `FlatFields` and `DataReferences`. The `RealTypes` height and speed are both mapped to y-axis, and speed's `DataReference` include a yellow `ConstantMap`, to distinguish speed from height. The speed's axis is equally colored yellow, but with the call `ScalarMap.setScaleColor( float[] speedColor )`. The complete code is available here. The program generates a window like the figure 4.8.

You might also want to know that is possible to prevent an extra axis to be drawn, even though you might have added a corresponding `ScalarMap` (with XAxis, YAxis or ZAxis) to the display. This is achieved by calling

```
ScalarMap.setScaleEnable( false );
```

The code for this example already includes such a call. Just uncomment the line with `speedYMap.setScaleEnable(false)` to prevent the speed axis from being drawn.

4.9. Using a Gridded1DSet

So far we have used `Integer1DSet` and `Linear1DSet`, as our time (x-axis) domain. Both sets are finite arithmetic progression of values. Your data might not be in such a progression or you might, for whatever reason, want to use some values which are

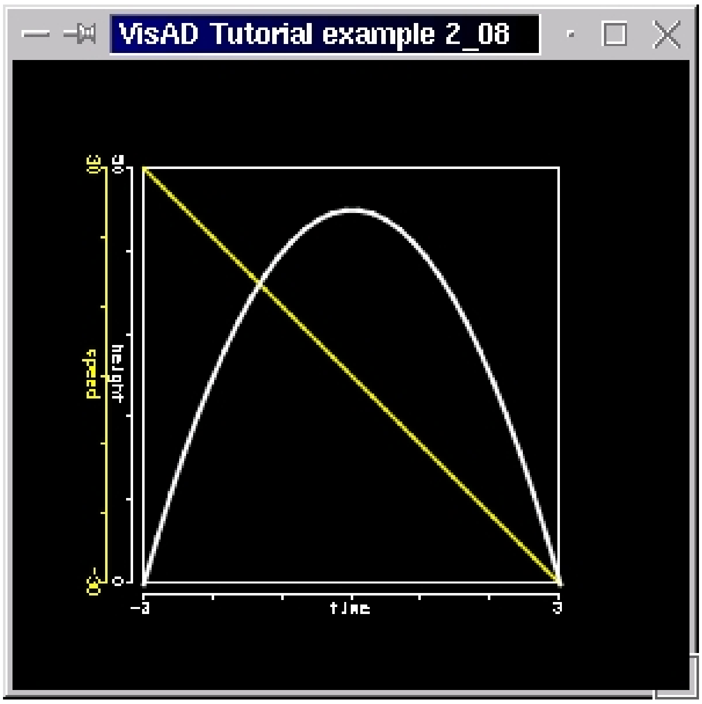

Figure 4.8.

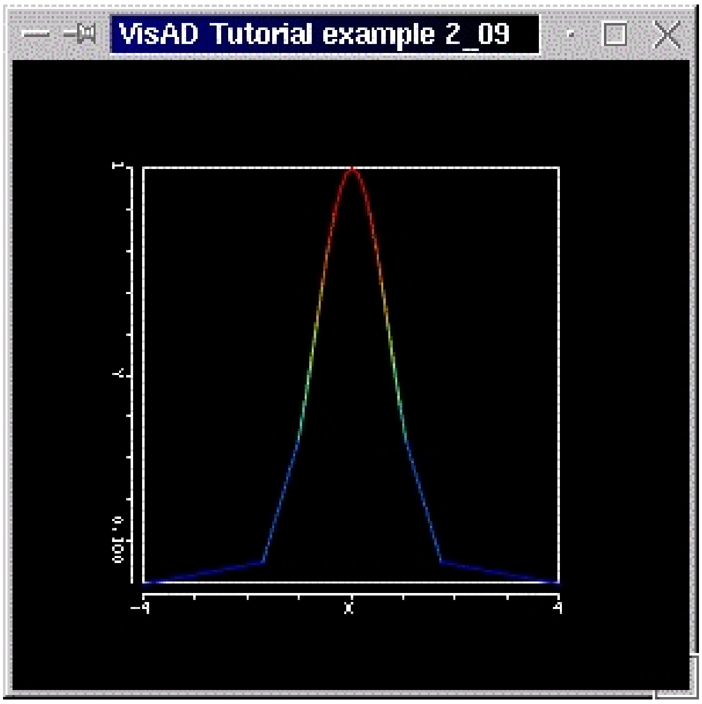

Figure 4.9.

more densely sampled in some region as it is in others. Our next example considers such a case.

We have a Gaussian distribution curve, but we are more concerned with the region around the maximum. We have our points concentrated around this region, and only a few samples at the base of the curve. Our `Gridded1DSet` is defined as follows

```
x_set = new Gridded1DSet(x, x_vals, LENGTH);
```

where x is the quantity to be displayed on the x-axis, x_vals is an array float[1][LENGTH] with the x values and LENGTH is simply the number of values. The corresponding values for y are given further down in the program. We also have a `ScalarMap` with `RealType` y mapped to `Display.RGB` added to our display. This will color the curve according to y. The source code for this example is available here. The figure 4.9 is a screen shot of this program.

Please note the varying sampling distances, indicating the use of the `Gridded1DSet`.

4.10. Using a RangeWidget

In this section we introduce the second VisAD User Interface (the first VisAD User Interface is a display!). We saw in section 2.2 how we can scale axes by calling `ScalarMap.setRange( double low, double hi )`. Although this method may suffice, in some cases you will want to be able to interactively change an axis range. To do this VisAD provides a user interface, the `RangeWidget`. We shall use example program P2_05 and add a `RangeWidget` to it.

To create a `RangeWidget` we simply declare one:

```
private RangeWidget ranWid;
```

Note that we have added the import statement

```
import visad.util.*;
```

In the directory `visad/util` you will find other useful user interfaces.

We actually create such a widget with

```
ranWid = new RangeWidget( timeMap );
```

Remeber, timeMap is the `ScalarMap` which of the `RealType` time and it's mapped to the x-axis. The final step it to add the widget to the window. You can see the code here. Running the program with `java tutorial.s2.P2_10` (and typing in the same range values given below) you should get a window like figure 4.10.

Note the range of the `RealType` time is between -2 and 4 (the curve only goes to 3 because our data, that is the domain set, is only defined between -3 and 3). By typing in a value in the text field and then pressing the "enter" key the display gets redrawn with the new range. You should run the example and try the widget out!

You should try to create a `RangeWidget` for the y-axis. You'd only need to create a widget for the heightMap and then add it to window.

4.11. Using a SelectRangeWidget

In this section we introduce the `SelectRangeWidget`, which allows you to select the range in which data will be drawn. In section 2.5 we used created a `ScalarMap`

```
timeRangeMap = new ScalarMap( time, Display.SelectRange );
```

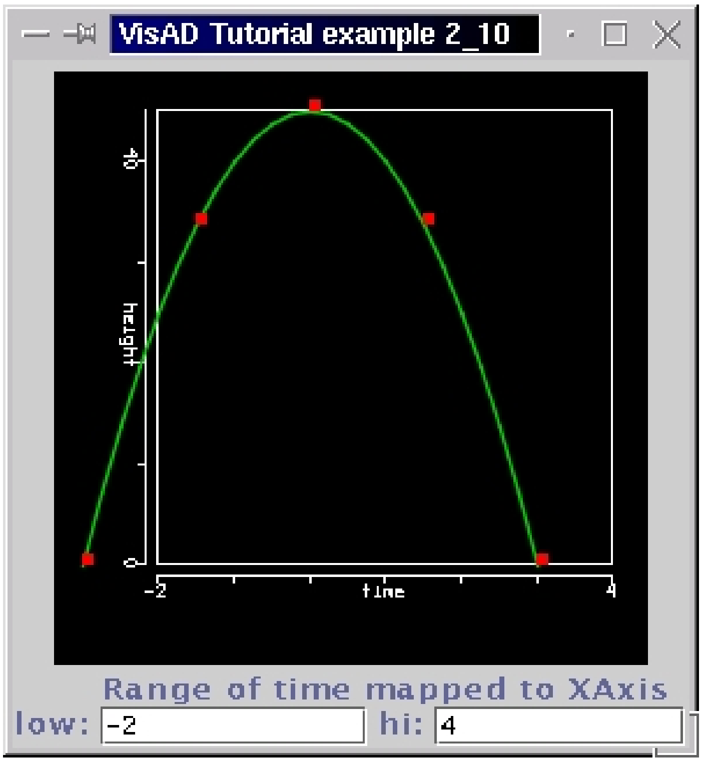
VisAD Tutorial example 2_10
height
time
-2
4
0
40
Range of time mapped to XAxis
low:
-2
hi:
4

Figure 4.10.

added it to the display, then we got its RangeControl

```
RangeControl timeRangeControl = (RangeControl) timeRangeMap.getControl();
```

defined a range

```
float[] timeRange = { -2.0f, 4.0f };
```

and finally implemented the changes by calling

```
timeRangeControl.setRange( timeRange );
```

This is fine in case that the selected range doesn't (or shouldn't) change during runtime. To interactively change Display.SelectRange bounds, VisAD provides the SelectRangeWidget. We shall use example program P2_10 to show this widget. In the program, we declare a SelectRangeWidget

```
SelectRangeWidget selRanWid;
```

We also need a map like

```
timeRangeMap = new ScalarMap( time, Display.SelectRange );
```

which determines that the RealType time will have a range which can be selected, as done in section 2.5. To create the widget we call

```
selRanWid = new SelectRangeWidget( timeRangeMap );
```

We then add this widget to the window as we have done with the previous widget. You can see the complete code here. Running the program with java tutorial.s2.P2_11you should get a window with a display and with the two widgets. See the screenshot 4.11.

Click and drag one of the yellow triangles to change one of the boundaries. Click and drag in the middle of the range to move both ends. If you are confused with the two widgets, then we urge you to run the example and try them out. The RangeWidget is responsible for "scaling the axis", whereas the SelectRangeWidget is responsible for selecting the range in which data can be drawn. We said in section 2.5 that you might want to combine both together in an application. Indeed, if by scaling the axis data gets drawn outside the display, you might use the SelectRangeWidget to avoid that. In the next section we make use of 2D Sets, and introduce images.

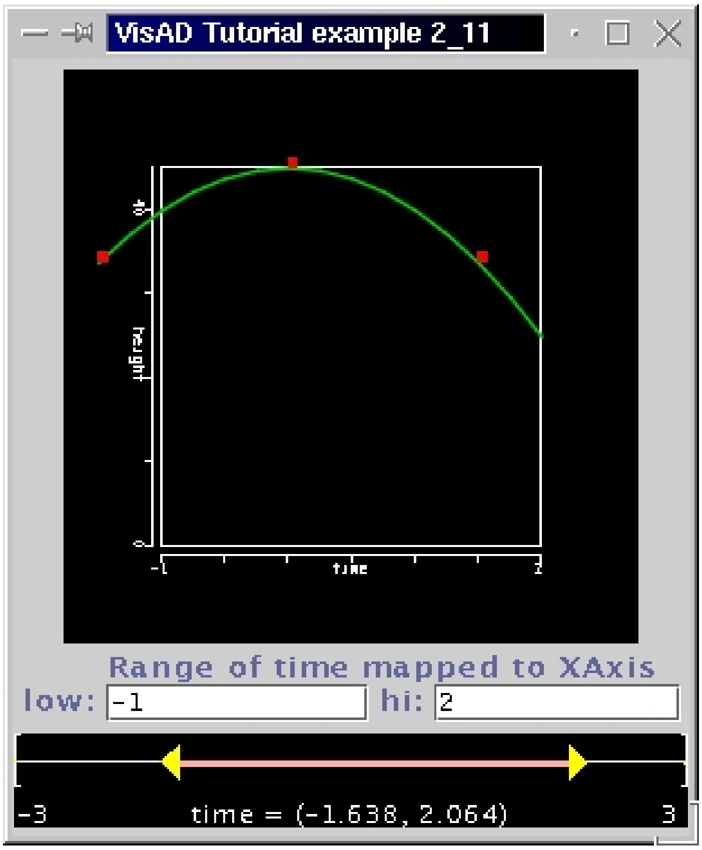
VisAD Tutorial example 2_11
height
time
Range of time mapped to XAxis
low: -1
hi: 2
-3
time = (-1.638, 2.064)
3

Figure 4.11.

5. Two-dimensional Sets

In this section we shall consider functions like `z = f(x,y)`. These are represented by the `MathType`

`( (x, y) -> range )`

The data for the quantities x and y will be given by a two-dimensional set. The first example is analog to our very first example (see sections 1.2 and 2.1), but with a two-dimensional domain. Our range might be interpreted as a surface (if range is composed of one `RealType`; range may of course be a `RealTupleType`) and it plays the role of the dependent variable. We shall, however start off with a 2D-display and shall map range to color.

5.1. Handling a 2-D array of data: using an Integer2DSet

Suppose we have a function represented by the `MathType`

`( (row, column) -> pixel )`

where row and column, and pixel are `RealTypes`. We organize (row, column) in a `RealTupleType` like the following

```
domain_tuple = new RealTupleType(row, column)
```

Our function, that is the pixel values for each row and column would be

```
func_dom_pix = new FunctionType( domain_tuple, pixel );
```

To define integer domain values (for the "domain_tuple") we use the class `Integer2DSet`:

```
domain_set = new Integer2DSet(domain_tuple, NROWS, NCOLS );
```

This means, define the domain by constructing a 2-dimensional set with values $0, 1, ..., NROWS - 1 \times 0, 1, ..., NCOLS - 1$. Note that these are integer values only.

We assume we have some $NROWS \times NCOLS$ pixel values in an array `float[NROWS][NCOLS]` (pixel values might be Java doubles, too). It is important to observe that the pixel samples are in raster order, with component values for the first dimension changing fastest than those for the second dimension. So, although the pixel values are in an array `float[NROWS][NCOLS]`, they will be stored in a `FlatField` like `float[1][NROWS * NCOLS]`. One reason for doing this is computational efficiency. Suppose we have an array with 6 rows and 5 columns, like

```
pixel_vals = {{0, 6, 12, 18, 24},
{1, 7, 12, 19, 25},
{2, 8, 14, 20, 26},
{3, 9, 15, 21, 27},
{4, 10, 16, 22, 28},
{5, 11, 17, 23, 29}  };
```

then these values should be ordered as the values above indicate.

This can be done by creating a "linear" array float[1][number_of_samples] (here called "flat_samples"), and then by putting the original values in this array. A way of doing this is can be seen in the following loops:

```
for(int c = 0; c < NCOLS; c++)
  for(int r = 0; r < NROWS; r++)
    flat_samples[0][ c * NROWS + r ] = pixel_vals[r][c];
```

You can see how this is done in the code of example P3_01, as follows:

```
// Import needed classes
import visad.*;
import visad.java2d.DisplayImplJ2D;
import java.rmi.RemoteException;
import javax.swing.*;

/**
VisAD Tutorial example 3_01
A function pixel_value = f(row, column)
with MathType ( (row, column) -> pixel ) is plotted
The domain set is an Integer1DSet
Run program with "java tutorial.s3.P3_01"
*/
public class P3_01{
  // Declare variables

  // The quantities to be displayed in x- and y-axes: row and column
  // The quantity pixel will be mapped to RGB color
  private RealType row, column, pixel;

  // A Tuple, to pack row and column together, as the domain
  private RealTupleType domain_tuple;
```

```java
// The function ( (row, column) -> pixel )
// That is, (domain_tuple -> pixel )
private FunctionType func_dom_pix;

// Our Data values for the domain are represented by the Set
private Set domain_set;

// The Data class FlatField
private FlatField vals_ff;

// The DataReference from data to display
private DataReferenceImpl data_ref;

// The 2D display, and its the maps
private DisplayImpl display;
private ScalarMap rowMap, colMap, pixMap;

public P3_01(String []args)
  throws RemoteException, VisADException {
  // Create the quantities
  // Use RealType(String name);
  row = new RealType("ROW");
  column = new RealType("COLUMN");
  domain_tuple = new RealTupleType(row, column);
  pixel = new RealType("PIXEL");

  // Create a FunctionType (domain_tuple -> pixel )
  // Use FunctionType(MathType domain, MathType range)
  func_dom_pix = new FunctionType( domain_tuple, pixel);

  // Create the domain Set, with 5 columns and 6 rows, using an
  // Integer2DSet(MathType type, int lengthX, lengthY)
  int NCOLS = 5;
  int NROWS = 6;
  domain_set = new Integer2DSet(domain_tuple, NROWS, NCOLS );

  // Our pixel values, given as a float[6][5] array
  float[][] pixel_vals = new float[][]{
    {0, 6, 12, 18, 24},
    {1, 7, 12, 19, 25},
    {2, 8, 14, 20, 26},
    {3, 9, 15, 21, 27},
    {4, 10, 16, 22, 28},
    {5, 11, 17, 23, 29}
  };

  // We create another array, with the same number of elements of
  // pixel_vals[][], but organized as float[1][ number_of_samples ]
  float[][] flat_samples = new float[1][NCOLS * NROWS];

  // ...and then we fill our 'flat' array with the original values
  // Note that the pixel values indicate the order in which these values
  // are stored in flat_samples
  for(int c = 0; c < NCOLS; c++)
    for(int r = 0; r < NROWS; r++)
      flat_samples[0][ c * NROWS + r ] = pixel_vals[r][c];

  // Create a FlatField
  // Use FlatField(FunctionType type, Set domain_set)
  vals_ff = new FlatField( func_dom_pix, domain_set);
```

```java
    // ...and put the pixel values above into it
    vals_ff.setSamples( flat_samples );

    // Create Display and its maps
    // A 2D display
    display = new DisplayImplJ2D("display1");

    // Get display's graphics mode control and draw scales
    GraphicsModeControl dispGMC = (GraphicsModeControl) display.↩
        getGraphicsModeControl();
    dispGMC.setScaleEnable(true);

    // Create the ScalarMaps: column to XAxis, row to YAxis and pixel to RGB
    // Use ScalarMap(ScalarType scalar, DisplayRealType display_scalar)
    colMap = new ScalarMap( column, Display.XAxis );
    rowMap = new ScalarMap( row,    Display.YAxis );
    pixMap = new ScalarMap( pixel,  Display.RGB );

    // Add maps to display
    display.addMap( colMap );
    display.addMap( rowMap );
    display.addMap( pixMap );


    // Create a data reference and set the FlatField as our data
    data_ref = new DataReferenceImpl("data_ref");
    data_ref.setData( vals_ff );

    // Add reference to display
    display.addReference( data_ref );

    // Create application window and add display to window
    JFrame jframe = new JFrame("VisAD Tutorial example 3_01");
    jframe.getContentPane().add(display.getComponent());


    // Set window size and make it visible
    jframe.setSize(300, 300);
    jframe.setVisible(true);
  }

  public static void main(String[] args)
    throws RemoteException, VisADException {
      new P3_01(args);
  }
}
```

Running the program above (code available here) with "java tutorial.s3.P3_01" generates a window like the screen 5.1.

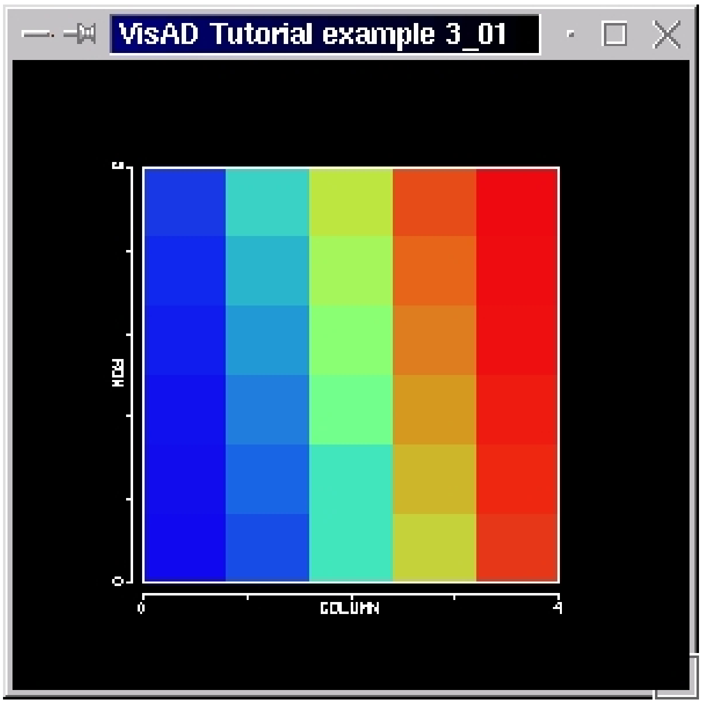

Figure 5.1.: Note again how the samples are organized. Remember that pixel values were mapped to RGB color. (Blue represents the smallest and red represents the largest.)

5.2. Continuous 2-D domain values: using a Linear2DSet

A `Linear2DSet` is a product of two `Linear1DSets`. They represent finite arithmetic progression of two different values. Our next example is almost like the previous example. We shall use, however, a `Linear2DSet`, which allows non-integer domain values, to define the 2-D domain set. First we rename the domain `RealTypes` "latitude" and "longitude", which are usually non-integer, since "row" and "column" suggest integer values (and an `IntegerSet` is indeed a sequence of consecutive integers).

In this example we shall consider the `MathType`

```
( (latitude, longitude) -> temperature )
```

with the `RealTypes` latitude, longitude and temperature. Our 2-D set will have the domain tuple:

```
domain_tuple = new RealTupleType(latitude, longitude);
```

Our set is

```
domain_set = new Linear2DSet(domain_tuple, 0.0, 6.0, NROWS,
0.0, 5.0, NCOLS);
```

Note that we define a first and a last value for both dimensions. This sets the domain values in NROWS (latitude) from 0.0 to 6.0 and in NCOLS (longitude) from 0.0 to 5.0. So latitude values progress like 0.0, 1.2, 2.4, 3.6, 4.8 and 6.0. Longitude values progress from 0.0 to 5.0 in 1.25 steps. You can get those values with the method `Linear2DSet.getSamples( boolean copy )`. The argument "copy" will make the method return a copy of the samples. Remember, the array is dimensioned `float[ domain_dimension ][NROWS * NCOLS]`, where `domain_dimension` equals 2.

If you compile the program P3_02 and run it with "java tutorial.s3.P3_02" you should see the window 5.2.

5.3. Color components: using different DisplayRealTypes

So far we have mapped a quantity, our dependent variable (temperature, in the previous example) to RGB color. Although you may define your own color table (see section 4), you might achieve satisfactory results by mapping one or many quantities to the proper `DisplayRealTypes`. Our next example illustrate this.

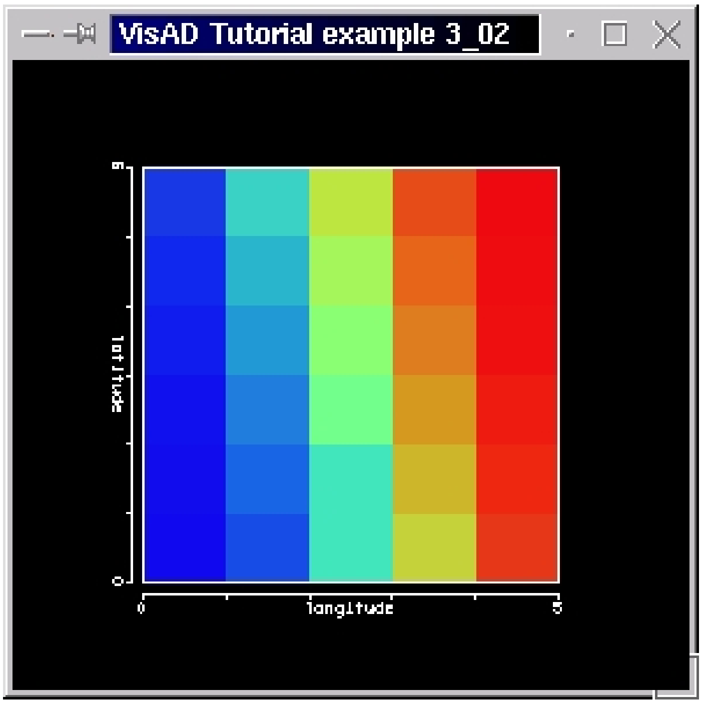

Figure 5.2.: P3-02

Before we change the DisplayRealType, we would like to draw attention to the parameters "first" and "last" of the previous example. We now define our set with

```
domain_set = new Linear2DSet(domain_tuple, 0.0, 12.0, NROWS,
0.0, 5.0, NCOLS);
```

The effect of changing 6.0 to 12.0 is to halve the resolution of latitude. The latitude range will be also changed to reflect this change (see screen shot below). As we have said, we will not map to RGB, bur instead to Display.Red, simply by defining the ScalarMap

```
tempMap = new ScalarMap( temperature, Display.Red );
```

We also create two ConstantMaps

```
double green = 0.0;
double blue = 0.0;

greenCMap = new ConstantMap( green, Display.Green );
blueCMap  = new ConstantMap( blue, Display.Blue );
```

and add them to the display. The reason for creating and adding these constant maps to the display is that the default values for green and blue is 1.0. (In fact, default values for red, green and blue are all 1.0, in order to create white graphics when color is not explicitely specified. See examples 1.1 and 2.3.) The result of the above changes can be seen in figure 5.3. The code is available here.

You could use Display.Cyan instead of Display.Red. This would result in a display with colors varying from red to black (rememeber, green and blue components are zero) like the display shown in the screenshot 5.4.

Note that the color varies from red to black. This is because in the RGB system cyan is defined as `cyan = white - red` which can be rewritten in component form `(red, green, blue)`:

```
(0,1,1) = (1,1,1) - (1,0,0)
```

So when temperature is at the maximum (red $=1$), cyan equals zero. Other subtractive color components are

```
magenta = white - green
```

and

```
yellow = white - blue
```

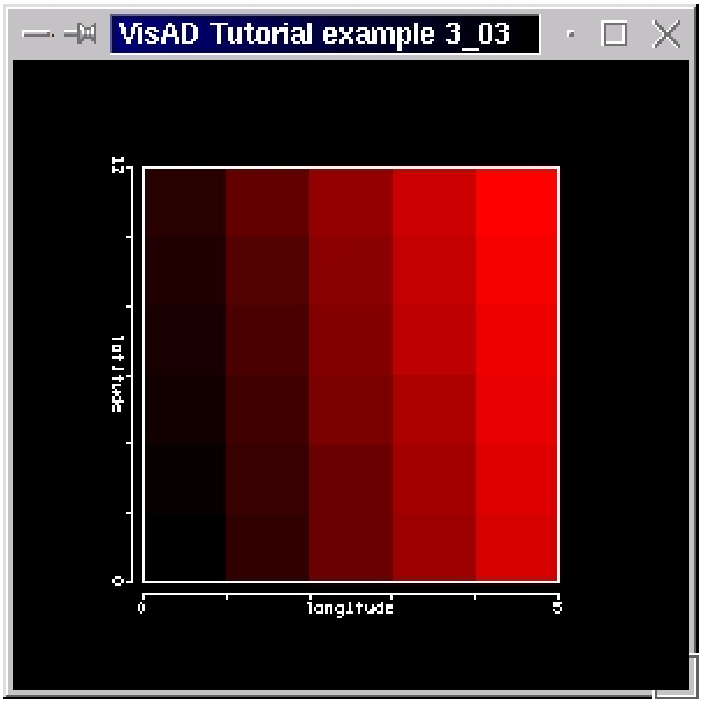

Figure 5.3.: P3-03

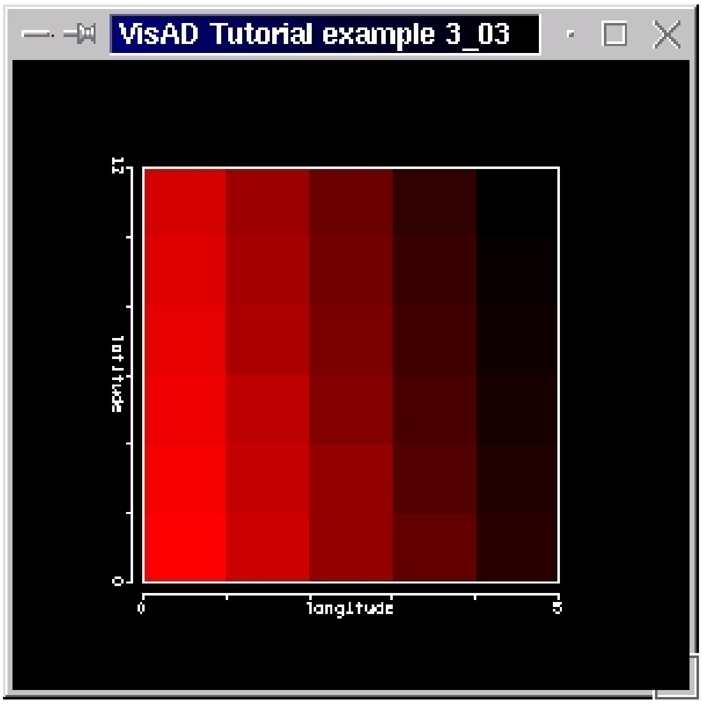

Figure 5.4.: P3-03a

You can try these out, just uncomment the appropriate line in the code of example P3_03. For example, we create the map

```
tempMap = new ScalarMap( temperature, Display.Green );
```

with the constant maps

```
double red = 0.0;
double blue = 0.4;

redCMap = new ConstantMap( green, Display.Red );
blueCMap  = new ConstantMap( blue, Display.Blue );
```

This will set a constant level of blue across the entire display. See the screen shot 5.5. Try decreasing the level of blue to 0.0 and see the change. (Remember in the RGB system adding red and green gives yellow, adding red and blue gives magenta and adding blue and green gives cyan (blue-green).) Note that there's no red, but some blue. One might use `Display.Cyan` instead of `Display.Red`. This would in a display with colors varying from red to black. (Actually, not quite totally black, because we have added a `ConstantMap` with some green.)

You should try out some other `DisplayRealTypes`. Just uncomment the appropriate lines in the code of example P3_03. Using `Display.CMY` (Cyan, Magenta, Yellow) results in figure 5.6, using `Display.Value` (or Brightness) results in figure 5.7.

This would be similar to creating and adding the maps

```
tempRedMap =   new ScalarMap( temperature, Display.Red  );
tempGreenMap = new ScalarMap( temperature, Display.Green  );
tempBlueMap =  new ScalarMap( temperature, Display.Blue );
```

without any other constant maps. In the beginning of this section we said you might achieve the coloring you want by using the right `DiplayRealTypes`. If you still don't get the colors you want, you might define your own RGB color table, and map a `RealType` to it (with the `DiplayRealType Display.RGB`. We will do that in section 4.)

5.4. Mapping quantities to different DisplayRealTypes

In the previous section we mapped a single quantity to different types of `DiplayRealTypes`. It's also possible to map different quantities to different `DisplayRealTypes`. We are going to map three quantities to the `Display.Red`, `Display.Green` and `Display.Blue`. (Remember, the RGB color system is adds the color components, so we expect black

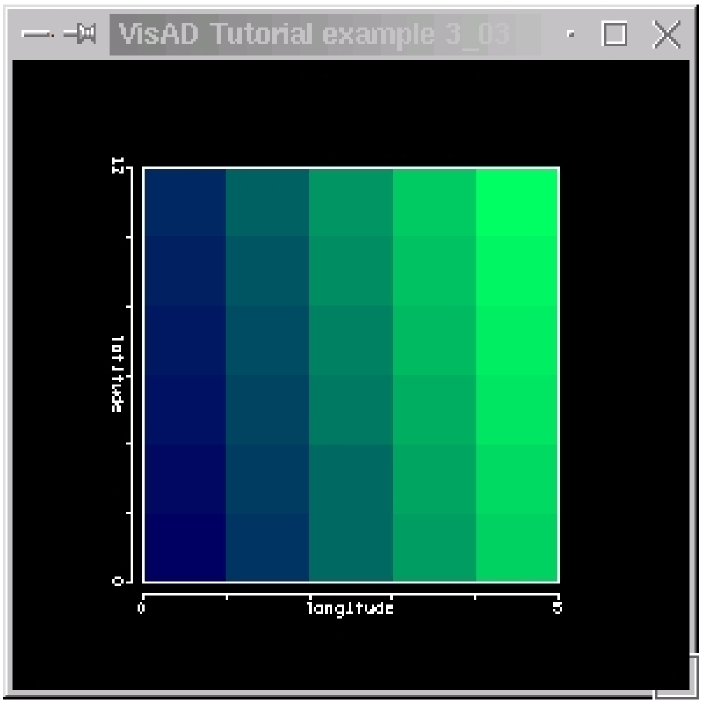

Figure 5.5.: P3-03b

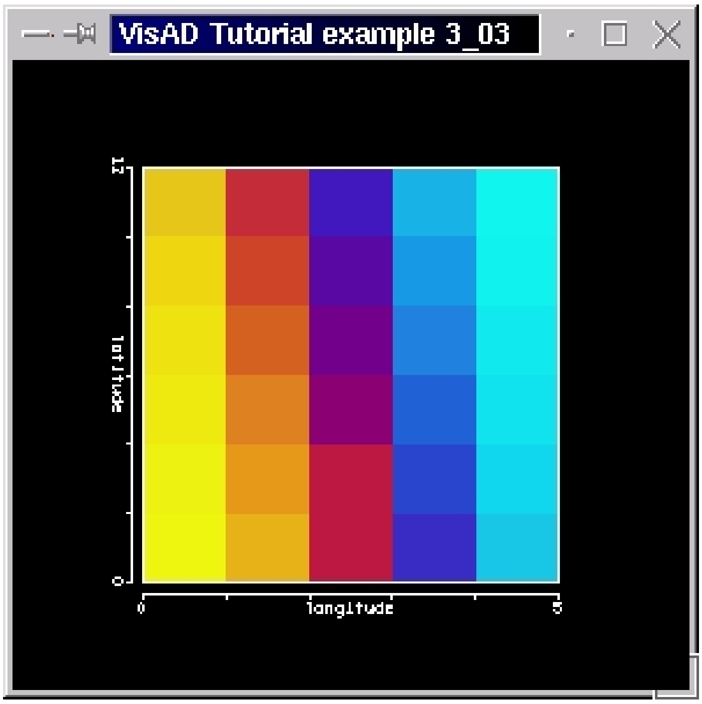

Figure 5.6.: P3-03c

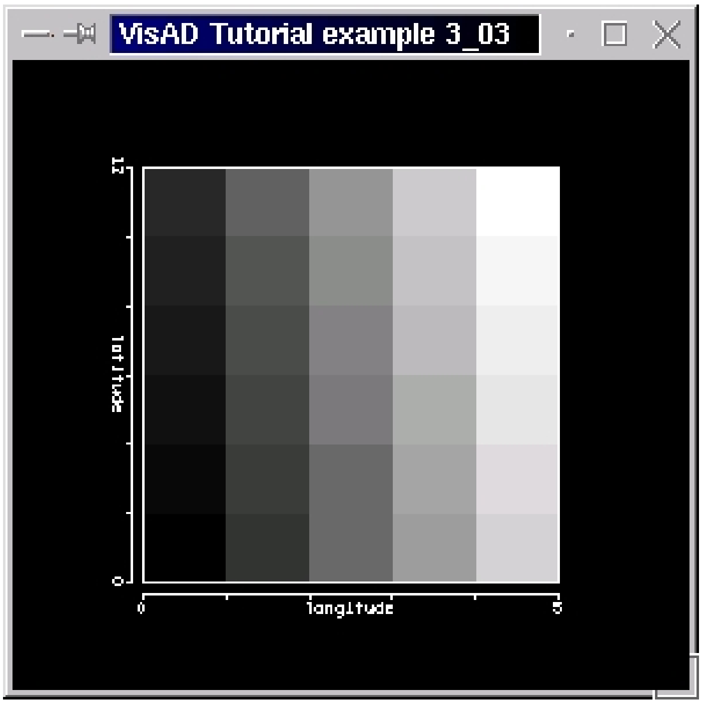

Figure 5.7.: P3-03d

where the quantities are minimum, and white where they are a their maxima.) To show this we extend our `MathType` from

```
( (latitude, longitude) -> temperature )
```

to

```
( (latitude, longitude) -> (temperature, pressure, precipitation) )
```

where the range (temperature, pressure, precipitation) is organized as a `RealTupleType`. We could use a constructor like

```
RealTupleType( RealType temperature, RealType pressure, RealType ↩
    precipitation);
```

for our `RealTupleType`. When the range has many `RealTypes` you might want to use a handier constructor:

```
RealTupleType( RealType[] my_realTypes);
```

That's how we create the `RealTupleType` for the range this example. We have the `RealTypes`

```
temperature = new RealType("temperature");
pressure = new RealType("pressure");
precipitation = new RealType("precipitation");
```

We create an array to `RealTypes` and then create the `RealTupleType` with this array:

```
RealType[] range = new RealType[]{ temperature, pressure, precipitation};

range_tuple = new RealTupleType( range );
```

Our function type is then

```
func_domain_range = new FunctionType( domain_tuple, range_tuple);
```

We use a `Linear2DSet` just like the one from the previous example (but with more samples and with different "first" and "last" values). We generate temperature, pressure and precipitation values in two for-loops and use some arbitrary functions (like sine, cosine, exponential). To set the sample values in the `FlatField`

```
vals_ff = new FlatField( func_domain_range , domain_set );
```

we need a "flat_samples" array of floats (although it might also be an array of doubles) just as float[number_of_domain_components][number_of_range_components], which is, in our case

```
flat_samples = new float[3][NCOLS * NROWS];
```

This time we call `FlatField.setSamples()` with an extra parameter

```
vals_ff.setSamples( flat_samples , false );
```

The argument "false" indicates that the array should not be copied. This is very important, since by telling the `FlatField` not to copy the array you might save some memory. As promised, we map temperature to red, pressure to green and precipitation to blue, as indicated by the following lines:

```
tempMap = new ScalarMap( temperature ,  Display.Red );
pressMap = new ScalarMap( pressure ,  Display.Green );
precipMap = new ScalarMap( precipitation ,  Display.Blue );
```

You can see the complete code here. Running the program will generate a window like the screen shot below:

Note that temperature (red) has a maximum at the top and a minimum at the bottom of the display. Pressure (green) has a maximum along longitude=0, and precipitation along latitude=0, both decreasing exponentially as one moves away from their maximum. Also note how the red, green and blue values are added, creating different colors. It is important to realize the difference between mapping a quantity to `Display.RGB` and mapping to `Display.Red`, `Display.Green` and `Display.Blue`. The former makes use of a user-definable color table and the latter maps the quantitiy to both red, green and blue, scaling these components between 0 (quantity's minimum) to 1.0 (quantity's minimum) and adding them.

5.5. Using IsoContour

In the following example we consider a `MathType` like

```
( (latitude , longitude) ->  temperature )
```

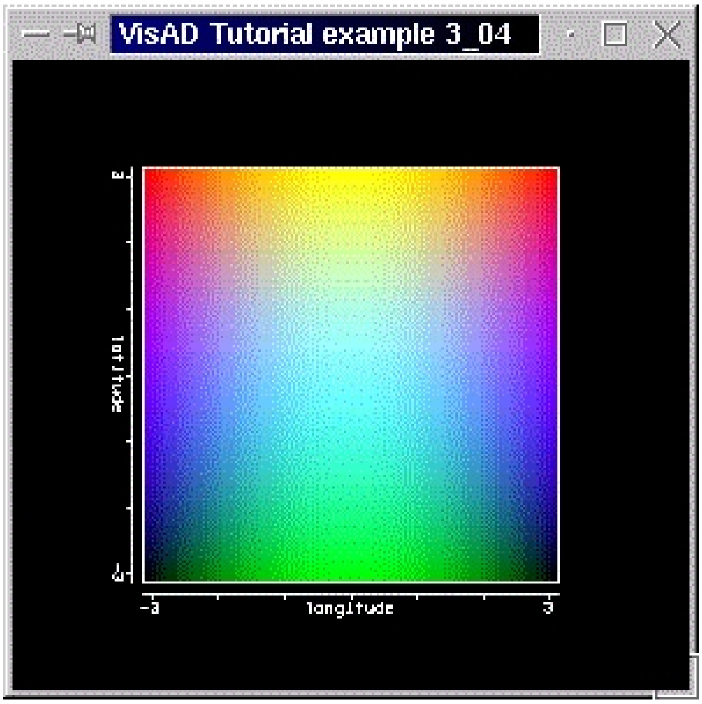

Figure 5.8.: P3-04

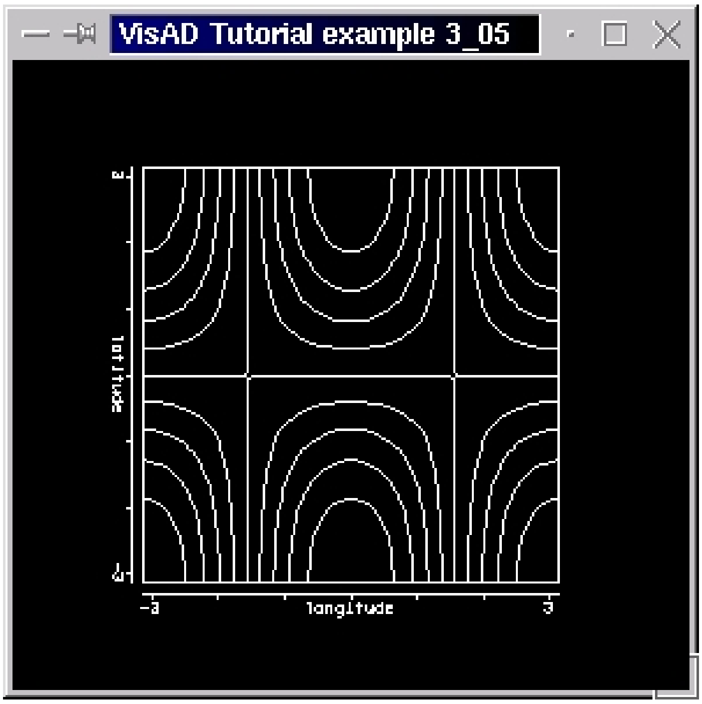

Figure 5.9.: P3-05

We will use a bigger `Linear2DSet` and will generate some values in the code. This is nothing really new. The (nice and) new feature of this example is the use of a new `DisplayRealType`:

```
tempIsoMap = new ScalarMap( temperature,  Display.IsoContour );
```

You might have already guessed: this `ScalarMap` will calculate the isocontours of the associated `RealType` (in this case, temperature). The result is a display with white isolines (the isotherms). The `IsoContour ScalarMap` is added to the display, as usual. You can see the complete code here.

Running the program will generate a window like the screen shot below:

If you want to colour the isolines according to the temperature, you simpy create and add the following map

```
tempRGBMap = new ScalarMap( temperature,  Display.RGB );
```

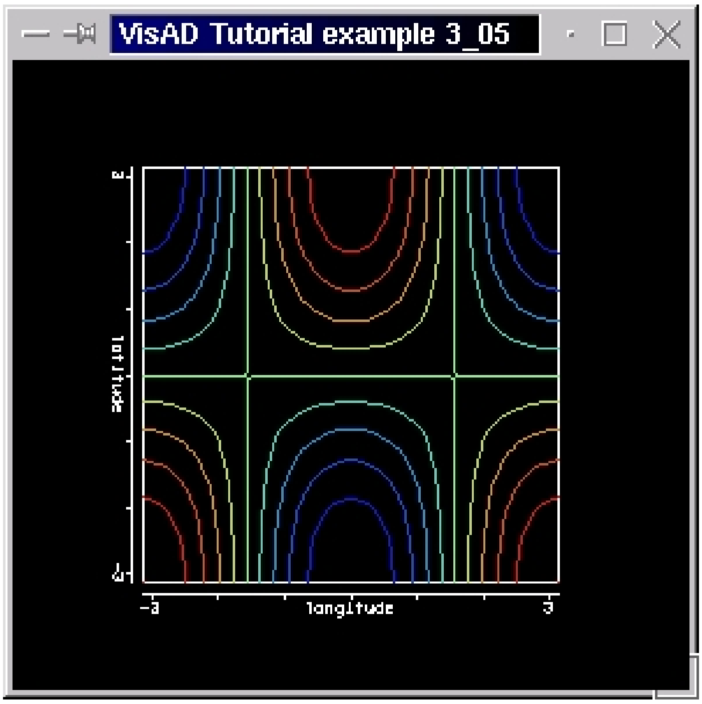

Figure 5.10.: Note that the isolines are now colored according to temperature.

In the code of example P3_05 the `ScalarMap` above has been created but not added to the display. Uncomment the line

```
display.addMap( tempRGBMap );
```

to add a RGB map, which will color the isolines like in the figure 5.10.

5.6. Controlling contour properties: using ContourControl

In this section we get to know another of VisAD's Control classes, the `ContourControl`. Most of the code of this example is exactly like the previous. The only difference is in the declaration and creation of a `ContourControl` object:

```
ContourControl isoControl = (ContourControl) tempIsoMap.getControl();
```

Note that we get the `ContourControl` from an `IsoContour ScalarMap`. Now that we have the `ContourControl` in our hands, we do something useful with it. We set the contour intervals to be "interval", to be drawn only between the minimum and maximum values, "lowValue" and "highValue", respectively, and to start drawing the contours at "base" value:

```
float interval = 0.1250f;  // interval between lines
float lowValue = -0.50f;   // lowest value
float highValue = 1.0f;    // highest value
float base = -1.0f;        //  starting at this base value
```

by calling the method

```
isoControl.setContourInterval(interval, lowValue, highValue, base);
```

While we still in control of the contours, we draw the contour labels too:

```
isoControl.enableLabels(true);
```

The result can be seen in the screen shot below. The code is available here.

Note that we have denser isolines (due to the "interval"), which are drawn from -0.5 (lowValue) to 1.0 (highValue). Also note that the base lies below the lowValue. (It's possible to draw dashed lines below the base.) In the figure you can also see the labels showing the value at some isolines. Although the `ContourControl` provides the control for how isolines should be depicted, you might not want to have to set those parameters in your code. To avoid that, VisAD also provides a user interface, the `ContourWidget` (please see section 4.2), which is the interface for the controls mentioned above and for a few more. Before we carry on to combine a flat surface with the respective iso-contours a few comments. You can also use the `ContourControl` to fill-in between the contours. This is achieved by calling

```
isoControl.setContourFill(true);
```

This requires the `RealType` that is mapped to `Display.IsoContour` to be also mapped to `Display.RGB`, otherwise it won't work properly. I the next section, however, we draw contours on top of the surface by other means.

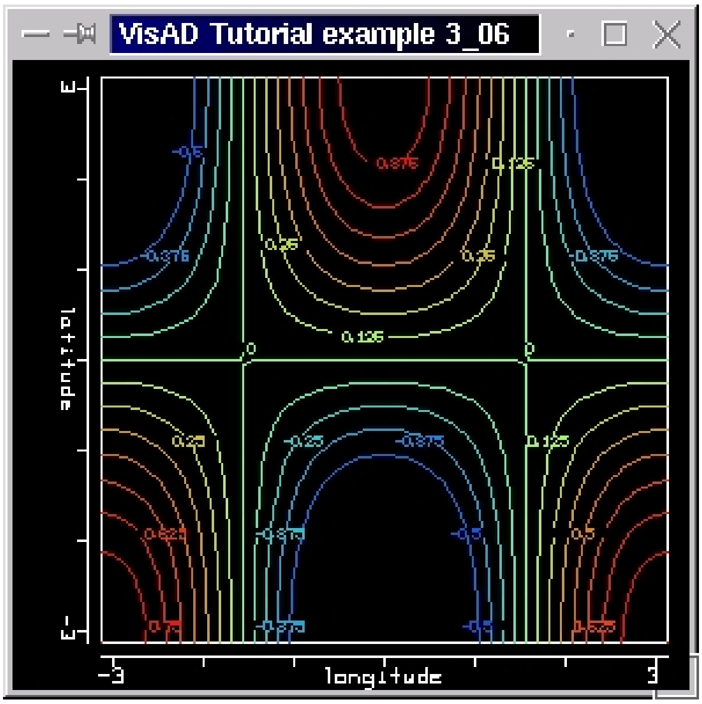

Figure 5.11.: P3-06

5.7. IsoContours over image

In this section we will draw the isolines over the colored surface. You might be tempted to think that you only need to add an `IsoContour` map and an RGB map, and then you get an image with the corresponding contours on top. From section 3.5 it should be clear that's not what happens. The behavior of the default `DataRenderers` is to render either filled colors or contours (or flow vectors or shapes or text) for a single `FlatField`. So we will need another `FlatField`, but we can (and should) use the same values, thus we neither need to generate values again nor do we need to copy those values to another array (thus saving memory). Although we will be plotting the same temperature values, as a colored image and as isocontours, we shall need another `RealType`, because we must discern which of those `RealTypes` ("color" temperature or "isocontour" temperature) should be mapped to which `DisplayRealType`. (We will want "color" temperature to be mapped to RGB and "isocontour" temperature to be mapped to `IsoContour`.) Sure we need a new `FunctionType` (isocontour temperature as function of 2-D domain) and finally a new reference, for the isocontour temperature (remeber, no need to copy the values, we shall use the same).

We start with our previous example and add the new `RealType`, `FunctionType`, `FlatField` and `DataReference`

```
RealType isoTemperature;
FunctionType func_domain_isoTemp;
FlatField isoVals_ff;
DataReferenceImpl iso_data_ref;
```

The `RealType` isoTemperature is defined as

```
isoTemperature = new RealType("isoTemperature", SI.kelvin, null);
```

Note that we have used the `SI` units kelvin, just as we have for the `RealType` temperature. (This is optional. Had we defined the `RealTypes` without units, the visual result would have been the same.) The `FunctionType` is

```
func_domain_isoTemp = new FunctionType( domain_tuple, isoTemperature);
```

where the domain tuple is the `RealTupleType` formed by latitude and longitude. After creating an extra `FlatField` (for isoTemperature)

```
iso_vals_ff = new FlatField( func_domain_isoTemp, domain_set);
```

we use the method `FlatField.getFloats(boolean copy)` to get the (float) temperature values (using copy = false in order not to copy the values).

```
float [][] flat_isoVals = vals_ff.getFloats(false);
```

We then set the isocontours `FlatField`'s samples with

```
iso_vals_ff.setSamples( flat_isoVals , false );
```

Again using an argument `copy = false`, to avoid copying the array. Please note the we have created a "temporary" array `float[][] flat_isoVals`, but just for clarity's sake. We could have called

```
iso_vals_ff.setSamples( vals_ff.getFloats(false) , false );
```

which does the same, but which is not very adequate for showing what is returned with the call `FlatField.getFloats(boolean copy)`. The next steps are the creation of the `ScalarMaps`

```
tempIsoMap = new ScalarMap( isoTemperature,  Display.IsoContour );
tempRGBMap = new ScalarMap( temperature,  Display.RGB );
```

and their addition to the display, as usual. We also create a `DataReference`, set its data and add to the display

```
iso_data_ref = new DataReferenceImpl("iso_data_ref");
iso_data_ref.setData( iso_vals_ff );
display.addReference( iso_data_ref );
```

The result can be seen in the screen shot below. The code is available here.

Note that the contours are drawn in white and they have the same interval, minimum and maximum value of the previous example. If you want contour lines of a quantity, e.g. temperature, drawn over the colored field of another quantity, e.g. pressure, than you'd only need to set pressure's `FlatFields` with pressure values (rather than copy the values, as we've done). We have also drawn contour labels. Remeber, using an array of `ConstantMaps` you can set the isolines colors, as shown in section 2.4. We shall do that in the next example.

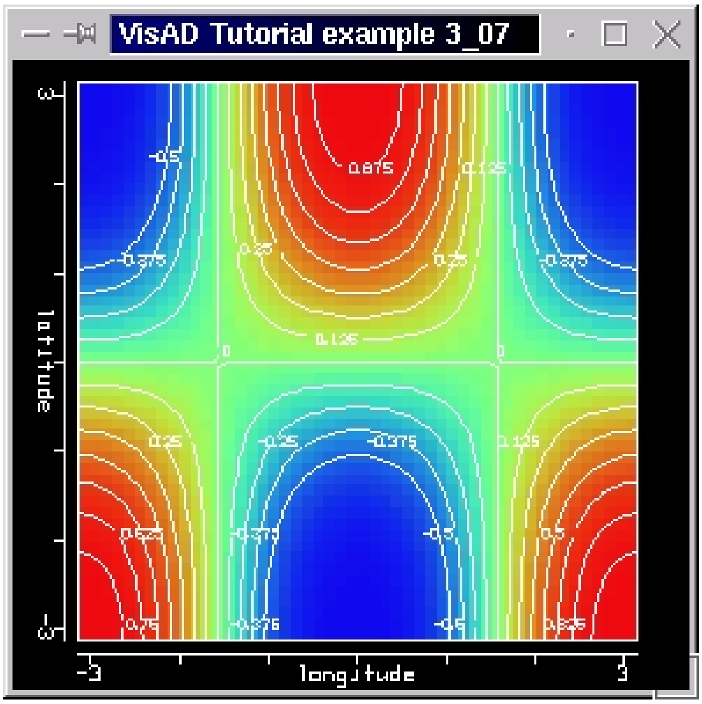

Figure 5.12.: P3-07

5.8. Using the GraphicsModeControlWidget

The `GraphicsModeControlWidget`, or `GMCWidget`, provides a user interface for users to interactively change parameters of `GraphicsModeControl`, for example line width as seen in section 2.7. To create `GMCWidget` we simply do

```
gmcWidget = new GMCWidget( dispGMC );
```

where `dispGMC` is simply the `GraphicsModeControl` that was already available. We have chosen to add the `GMCWidget` to the same `JFrame` of the display (you may, of course, create a new `JFrame` for it). As promised, we color the contours we a constant (and dull) gray (75% of each red, green and blue component), by the means of an array of `ConstantMaps`:

```
ConstantMap[] isolinesCMap = { new ConstantMap( 0.75f, Display.Red ),
new ConstantMap( 0.75f, Display.Green ),
new ConstantMap( 0.75f, Display.Blue ) };
```

and the call

```
display.addReference( iso_data_ref, isolinesCMap );
```

The complete code for example P3_08 is available here. Running the program with "java tutorial.s3.P3_08" will generate a window like the screen shot below.

As you can see in the screen shot, the `GMCWidget` allows you to change line width and point size as well as select whether you want scales to be drawn, whether data should be rendered as points and whether you would like texture mapping. You should run example program P3_08 and try it out! Note that we could have created a `ScalarMap` like

```
isoTempRGBMap = new ScalarMap( isoTemperature,  Display.RGB );
```

and have it added it to display to color the isolines. The necessary line are all available in the code for you to try out (although you won't see much of the isolines, as they have exactly the same color as the background; try changing their width and/or changing the colored field to point mode). don't forget to call

```
display.addReference( iso_data_ref, null );
```

instead of calling it with the `ConstantMaps`.

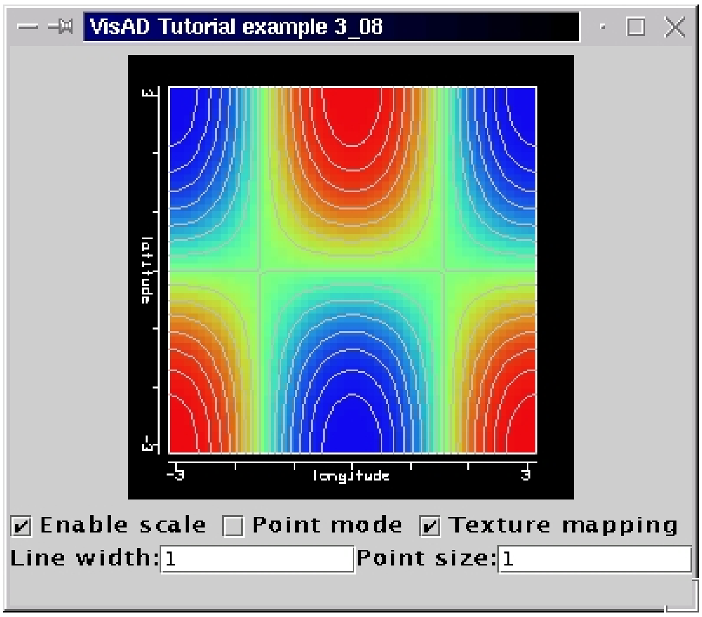

Figure 5.13.: P3-08

5.9. Combining color and isocontour in an extended MathType

The following example does not have any "new" feature. It's just a combination of the topics treated so far. We shal extend the `MathType` of the previous example and shall draw data in an "unconventional" way. Our new `MathType` is

```
( (latitude , longitude) -> (altitude , temperature) )
```

which is almost like the previous one, only that the range (altitude, temperature) has now two `RealTypes`. We shall map the first `RealType` to `IsoContour` and the other to RGB. The result should be a display with the isolines (altitude contours), colored according to the temperature. As want to map altitude to `IsoContour` we create the `ScalarMap`

```
altIsoMap = new ScalarMap( altitude ,  Display.IsoContour );
```

and the isolines are going to be colored according to the temperature because of the `ScalarMap`

```
tempRGBMap = new ScalarMap( temperature ,  Display.RGB );
```

The two `ScalarMaps` are then added to the display, as usual. You can see the complete code here. Running the program will generate a window like the screen shot 5.14:

Note that the altitude isolines are colored according to temperature. (The altitude curve has a peak around the point (longitude, latitude) = (0, 0), otherwise the curve tends to zero. The color pattern is just like that of the screenshot in section 3.6.) If you want to draw the isolines over the surface, then you have to split the `MathType` into two:

```
( (latitude, longitude) -> altitude )
```

and

```
( (latitude, longitude) -> temperature )
```

This is just what we did in section 2.7. We need two `FunctionTypes` as well as two `FlatFields` (and two float[1][NCOLS * NROWS] samples arrays) and two `DataReferences`, one for altitude and the other for temperature.

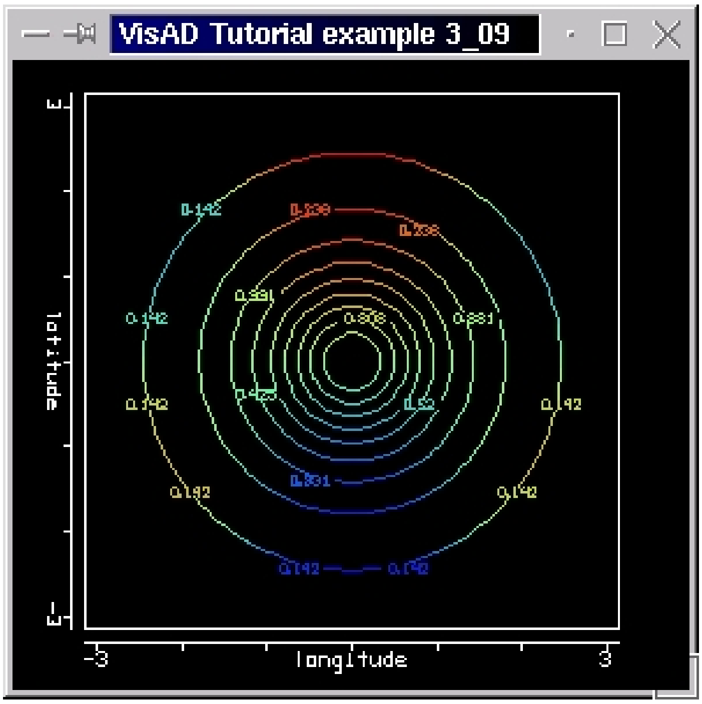

Figure 5.14.: P3-09

Although the screen shot above is unusual in the sense that one does not often color contours according to a quantity other than the contour quantity itself (because it's a difficult thing to implement?), in VisAD you're not bound to such "traditions". You are free to try out different data depictions, and for that it generally suffices to change the `ScalarMaps` (although not every choice of `DisplayRealType` is legal). We do encourage you to try changing the `ScalarMaps` and `DisplayRealTypes` (if you haven't done it so far!) and for that we have provided some extra lines of code, which you only have to uncomment, compile and run.

Looking at the screen shot again you might think it'd be better to map temperature to color and altitude to the z-axis. Indeed it is! So by now you should be asking yourself how you actually create a 3-D display and how you map some `RealType` to the z-axis. This is now a trivial issue: just create a 3-D display (rather than a 2-D one) and map your `RealType` to `Display.ZAxis`. We'll do that in the next section.

6. Three-dimensional Displays

7. Animation

8. Interaction

Part II.

Other VisAD Tutorials for Java Programmers

9. The VisAD DataModel Tutorial

This tutorial is about using the VisAD Data Model in Everyday Programming and has been last updated in March, 2000

9.1. Introduction

Fundamental to VisAD is the Data Model. The Data Model is a collection of VisAD interfaces and classes that allow the programmer to describe the data to be used in their program: the associated units and error estimates, how is it organized, related to other data, and so on. Some Data Model class objects contain no "real" data, but may contain *meta data* (data about the data, such as the units) or show how different data sets should be grouped for making a display.

While there are many extensions to this collection of classes and interfaces, we shall only introduce the fundamental Data object types and illustrate the parallels with quantities you may already be familiar with. Each will have a source code example to allow you to experiment on your own.

If you are more comfortable with `Python`, then we've made a Python (Jython) version of this tutorial that shows the examples in that language.

9.2. Scalars

In VisAD the values of a `Scalar` may be either `Real` which is used for numeric quantities, or `Text` for strings of characters. In this discussion, we deal only with `Real` objects.

9.2.1. Real (actual) numbers

No doubt about it, if you've written Java code, you are already familiar with doing something like:

Listing 9.1: Very simple Java code example

```
double a,b,c;
a = 10.;
b = 255.;
```

```
c = a + b;
System.out.println("sum = "+c);
```

If you are new to Java, but come from a Fortran background, then this might be more familiar:

Listing 9.2: Very simple Java code example

```
REAL A,B,C
A = 10
B = 255
C = A + B
WRITE(*,*) 'sum = ',C
```

Well, if you want to use the VisAD Data model, here's what you'd say (the complete program is provided):

```
import visad.*;

public class dataex1 {
public static void main (String arg[]) {
  try {
    Real a,b,c;
    a = new Real(10.);
    b = new Real(255.);
    c = (Real) a.add(b);
    System.out.println("sum = "+c);
  } catch (Exception e) {
    System.out.println(e);}
  }
}
```

When you run this example, you get:

```
sum = 265.0
```

By doing this, of course, you are not going to be convinced that there is any advantage to using the VisAD Data model. So, let's explore another form of constructor for `visad.Real`.

9.2.2. Estimating Errors

The form of constructor for Real `Real(double value, double error)` allows us to provide an error estimate for the value. This estimate can then be propagated through mathematical operations to provide an error estimate.

```
Real a,b,c;
```

```
a = new Real(10.,1.);
b = new Real(255.,10.);
c = (Real) a.add(b);
System.out.println("sum = "+c);
```

When you run this example, you still get:

```
sum = 265.0
```

The VisAD default for most math operations, however, is to not propagate errors, so to make use of this, we must explicitly indicate how we want error estimate to propagate. This is done by using an alternate signature of the `add` method (and all other math operators):

```
Real a,b,c;
a = new Real(10.,1.);
b = new Real(255.,10.);
c = (Real) a.add(b, Data.NEAREST_NEIGHBOR, Data.INDEPENDENT);
System.out.println("sum = "+c);
System.out.println("error of sum is="+c.getError().getErrorValue());
```

When you run this example, you get:

```
sum = 265.0
error of sum is=10.04987562112089
```

The constants supplied to the `add` method are the `type of interpolation` and the `type of error propagation`. In this simple case, the *type of interpolation* is not really relevant, but as you will see later, VisAD Data types may contain finite approximations to continuous functions and when these are combined mathematically, may need to be resampled in order to match domains.

9.2.3. Using Units

Another powerful feature of the VisAD Data model is that it may handle units. If your quantities are physical and have associated Units, then you might prefer to create a VisAD `MathType` that explicitly defines the metadata characteristics of your quantities. A `MathType` is used to define the kind of mathematical object that the Data object approximates.

Hint 1 (MathType is mandatory) *Every Data object in VisAD must have a* `MathType`*.*

In the previous examples, a default `MathType` with the name *Generic* was implicitly used for our `Real` objects. In the simplest form for dealing with Units, the constructor for a `MathType` which defines Real values is:

```
RealType(String name, Unit u, Set s)
```

which allows you to assign a unique `name` to this `MathType`, a `Unit` for this, and define a default `Set`. In practice, the `Set` is seldom used and should just be passed as `null` in most cases. To make use of this, we modify the program to read as follows:

```
Real t1,t2,sum;

RealType k = new RealType("kelvin",SI.kelvin,null);
t2 = new Real(k,273.);
t1 = new Real(k,255.);

sum = (Real) t1.add(t2);

System.out.println("sum = "+sum+" "+sum.getUnit());
```

When you run this example, you get:

```
sum = 528.0 K
```

In this example, we were able to use an SI Unit (ampere, candela, kelvin, kilogram, meter, second, mole, radian, steradian). Note that we constructed two variables with the same `MathType`, that is the same name, Unit, and Set. The only thing that is different is the numeric value. If you are using some other unit, VisAD provides mechanisms for making up Units for those. As an example, you can use the `Parser.parse()` method from the `visad.data.netcdf.units` package to create a *VisAD Unit* from a String name.

```
Real t1,t2,sum;

Unit degC = visad.data.netcdf.units.Parser.parse("degC");

RealType tc = new RealType("tempsC",degC,null);
t2 = new Real(tc,10.);

RealType k = new RealType("kelvin",SI.kelvin,null);
t1 = new Real(k,255.);

sum = (Real) t1.add(t2);

System.out.println("sum = "+sum+" "+sum.getUnit());
```

When you run this example, you get:

```
sum = 538.15 K
```

Observe that although we defined the value of variable `y` to be in degree Celsius, when we added the two variables together, the value of `y` was automatically converted to degrees Kelvin. As long as the units are transformable, VisAD handles this. If you attempt to combine quantities with incompatible units, an Exception is thrown. If you'd like to get the value listing in Celsius, then change the println to read:

```
System.out.println("sum = "+sum.getValue(degC)+" "+degC);
```

When doing arithmetic on `Real` objects, you may need at some point to use a constant value for something. As with all VisAD `Data` objects, in order to perform these operations, the `Units` must all match. When you use the simplest form of constructor for `Real` to define a numeric value, VisAD sets its `Unit` to a default value which can then be used to do arithmetic with any other Real. To illustrate, let's modify the previous example to compute the average of the two temperature values:

```
Real t1,t2,average;

Unit degC = visad.data.netcdf.units.Parser.parse("degC");

RealType tc = new RealType("tempsC",degC,null);
t2 = new Real(tc,10.);

RealType k = new RealType("kelvin",SI.kelvin,null);
t1 = new Real(k,255.);

Real two = new Real(2.0);

average = (Real) t1.add(t2).divide(two);

System.out.println("average = "+average+" "+average.getUnit());
```

When you run this program, you get:

```
average = 269.075 K
```

9.3. Tuples

A Tuple object contains a collection of Data objects whose number, sequence and type are defined by the `MathType` of the Tuple. There is also a subclass of `Tuple` named `RealTuple` which is a collection of `Real` objects, but again whose number and sequence are fixed by the `RealTupleType` associated with the

`RealTuple`. This object is like a fixed-length vector, like (x, y, z). Contrast this with a Java `array` which is a set of identical objects of that particular type. You will, in fact, find a constructor for VisAD's `RealTuple` where the data values are passed in as an array of `s`.

9.3.1. Making the MathTypes

Let's suppose we want to have a single object that keeps a collection of values of temperature, wind speed, and time together. The approach is to first define the MathTypes for each of these quantities. For example:

```
RealType temperature, speed, time;

Unit degC = visad.data.netcdf.units.Parser.parse("degC");

temperature = new RealType("temperature", degC, null);

Unit kts = visad.data.netcdf.units.Parser.parse("kts");
speed = new RealType("speed", kts, null);

Unit sec = visad.data.netcdf.units.Parser.parse("seconds");
time = new RealType("time", sec, null);

RealTupleType mydata = new RealTupleType(time, speed, temperature);
```

9.3.2. Using numbers

Now that we've defined the MathTypes, let's see how this works with some "real" data. Add the following lines of code to the above fragment:

```
double obsTemp = 32.;
double obsSpeed = 15.;
double obsTime = 4096.;

double[] values = {obsTime, obsSpeed, obsTemp};

RealTuple obs = new RealTuple(mydata, values);

System.out.println("obs = "+obs);
```

When you run all this now, you get:

```
obs = (4096.0, 15.0, 32.0)
```

9.3.3. Arithmetic with Tuples

Let us now suppose we have a second set of observed data and add this code onto the end of our example:

```
double obsTemp2 = -10.;
double obsSpeed2 = 7.;
double obsTime2 = 1234.;
```

```java
double[] values2 = {obsTime2, obsSpeed2, obsTemp2};

RealTuple obs2 = new RealTuple(mydata, values2);
System.out.println("obs2 = "+obs2);
```

When you run this addition, you get:

```
obs = (4096.0, 15.0, 32.0)
obs2 = (1234.0, 7.0, -10.0)
```

Our main purpose is to average all the values together. Again we need to define a `Real` for our constant, and then do just what we did previously:

```java
Real two = new Real(2.0);
RealTuple avg = (RealTuple) obs.add(obs2).divide(two);
System.out.println("avg = "+avg);
```

Finally when you run the complete example, you get:

```
obs = (4096.0, 15.0, 32.0)
obs2 = (1234.0, 7.0, -10.0)
avg = (2665.0, 11.0, 284.15)
```

Note that the temperature was converted to the base unit of kelvin.

Hint 2 (arithmetic capability of VisAD applies in consistent manner)
Most important - as you can see the arithmetic capability of VisAD applies to all types of Data objects in the same manner.

Although we have used a VisAD `RealTuple`, it is of course just a specific kind of a `Tuple` that only contains `Reals`. `Tuples` can be used to collect together all types of VisAD Data objects, including `Sets` and `Functions`.

9.4. Sets

As shown in the next section, `Set` objects are most often used to define the finite sampling of the domain of `Field` Objects (which approximates a function by interpolating its values at a finite subset of its domain).

In VisAD, the `Set` class has many sub-classes for different ways of defining finite subsets of the `Set's` domain. Near the top of the list is the

DoubleSet which includes all the double precision values that can be represented in the computer's 64 bit word... it is a finite, but very large Set. Farther down the hierarchy, for example, is the Linear1DSet which consists of n values of a simple arithmetic progression between two specified values.

VisAD comes with lots of implementations of the Set interface, in order to represent lots of common topologies. In this introduction, however, we'll deal only with the Linear1DSet.

The Set object also defines the CoordinateSystem of the Field's domain and the Units of the domain's RealType components. The following section will deal more explicitly with Fields.

9.4.1. Making a Set

Working with Sets is deceptively easy. For example, a program that contains these two lines of code:

```
Linear1DSet s = new Linear1DSet(-33., 33., 5);
System.out.println("set s = "+s);
```

will produce this output when run:

```
Set s = Linear1DSet: Length = 5 Range = -33.0 to 33.0
```

The Linear1DSet defined in this case has associated MathType of *Generic*. There is an alternate form of the constructor for a Linear1DSet that allows you to define the MathType for this set of numbers, as well.

9.4.2. Set methods

There are several usage methods available for working with Sets. For example, you may need an enumeration of the values of our little Linear1DSet. If you add the code:

```
float[][] sam = s.getSamples();

for (int i=0; i&lt;sam[0].length; i++) {
  System.out.println("i = "+i+"  sample = "+sam[0][i]);
}
```

the program would produce the following output:

```
set s = Linear1DSet: Length = 5 Range = -33.0 to 33.0

i = 0  sample = -33.0
```

```
i = 1  sample = -16.5
i = 2  sample = 0.0
i = 3  sample = 16.5
i = 4  sample = 33.0
```

Other methods worth checking out include `indexToDouble()` which returns an array of `doubles` given an array of indices. There is also an inverse method, `doubleToIndex()`.

9.5. Functions

A VisAD `Function` Data object represents a function from a domain to values of some specific type (a range). `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). This object is really the heart of VisAD's `Data` model when working with many forms of geophysical data, which tend to be samples of continuous fields.

In order to use `Function` objects, it is necessary to define the sampling using a `Set` of some kind for the domain and to supply appropriate samples for the corresponding range values. Let's make a small example. In this case, I want to define a `Function` that can convert temperatures from degrees Fahrenheit to Kelvin.

The MathTypes

First, we must define the appropriate `MathTypes`:

Listing 9.3: The MathTypes

```java
RealType domain = new RealType("temp_F");
RealType range = new RealType("temp_Kelvin");

FunctionType convertTemp = new FunctionType(domain, range);
```

Here, we have defined the `RealType` for our domain to represent degrees F, and for the range for degrees K. The `FunctionType` defines the mapping from the `domain` to the `range`.

The samples

Now we need to define and set the values for the samples of the `Function`. Let's say that we know the values of temperature in both units at -40F and 212F:

```java
Set domain_set = new Linear1DSet(-40., 212., 2);
```

We use a 1D Set because we are only defining a scalar at each point in the range (rather than a vector).

The FlatField object

Finally, we need to construct the VisAD `Data` object that will provide for the desired finite sampling. The `FlatField`, which is a subclass of `Field` designed for use with the Java primitive type `double`, provides just such a representation and functionality. So we can say:

```
FlatField convertData = new FlatField(convertTemp, domain_set);
```

Which constructs a `FlatField` which is defined by a `FunctionType` defined over the values of the `domain_set`. First, create an array that contains the numeric values of the range samples at the two points in the `domain_set`:

```
double[][] values = new double[1][2];
values[0][0] = 233.15; // = -40F

values[0][1] = 373.15; // = 212F
```

Then put the range samples into the FlatField using:

```
convertData.setSamples(values);
```

Evaluating Functions

Okay, so now let's test our `Function` by providing a domain value (that is, a temperature in degrees F) that we want to convert:

```
Real e = new Real(14.0);
Data v = convertData.evaluate(e);

System.out.println("value for 14.0F = "+v);

double vf = (((Real)v).getValue() - 273.15)*9./5. + 32;
System.out.println("           or (doing the math) = "+vf);
```

When you run this whole example, you get:

```
value for 14.0F = 263.1499938964844
or (doing the math) = 13.999989013671915
```

9.5.1. Sampling modes

By default, the `evaluate()` method in a `Function` uses the sampling mode called `Data.WEIGHTED_AVERAGE`. You may also sample using `Data.NEAREST_NEIGHBOR`, which in this case would give a different result, since the domain value of 14.0 would be closest to the domain sample at -40.0, which then means that 233.15 is the associated range value. If we change the `evalute()` call to read:

```
double v = convertData.evaluate(e, Data.NEAREST_NEIGHBOR, Data.NO_ERRORS);
```

then results in this output when you run the program now:

```
value for 14.0F = 233.14999389648438
or (doing the math) = -40.00001098632808
```

9.6. Parting points...

1. The main purpose of this section of the `VisAD Tutorial` is to encourage the use of the Data model whether or not you are planning on using anything else in VisAD (of course, the use of lots of other VisAD software is encouraged ;-)
2. This has been a really brief discussion of topics that can become quite complex. For example, a `Function` with a domain of "time" and a range of radar images forms the basis for animations.
3. Don't take the names of anything too seriously.
4. I'd like the thank Ugo Taddei, Bill Hibbard, Don Murray, and Stuart Weir for their input to this tutorial (some of which were taken verbatim).

10. The VisAD DataRenderer Tutorial

This is an initial version of the VisAD `DataRenderer` Tutorial. This is a very complex topic. The tutorial starts with an overview and general theory of operation of `DataRenderers`, then considers the specific design of some example `DataRenderers`. Please send any suggestions for how it can be improved to hibbard@facstaff.wisc.edu.

10.1. Overview of DataRenderers

In the VisAD system, `visad.DataRenderer` is an abstract class of objects that transform `Data` objects into depictions in Display objects, and in some cases transform user gestures back into changes in `Data` objects. Whenever a `visad.Data` object is linked to a `visad.Display` object, via a `visad.DataReference` object, an object of some concrete subsclass of `visad.DataRenderer` is part of the linkage. The current hierarchy of `DataRenderer` subclasses distributed with VisAD is:

```
visad.DataRenderer
visad.java2d.RendererJ2D
visad.java2d.DefaultRendererJ2D
visad.bom.BarbRendererJ2D
visad.java2d.DirectManipulationRendererJ2D
visad.bom.BarbManipulationRendererJ2D
visad.java3d.RendererJ3D
visad.java3d.DefaultRendererJ3D
visad.java3d.AnimationRendererJ3D
visad.bom.BarbRendererJ3D
visad.bom.SwellRendererJ3D
visad.bom.ImageRendererJ3D
visad.bom.TextureFillRendererJ3D
visad.cluster.ClientRendererJ3D
visad.cluster.NodeRendererJ3D
visad.java3d.DirectManipulationRendererJ3D
visad.bom.BarbManipulationRendererJ3D
```

```
visad.bom.SwellManipulationRendererJ3D
visad.bom.CurveManipulationRendererJ3D
visad.bom.PickManipulationRendererJ3D
visad.bom.PointManipulationRendererJ3D
visad.bom.RubberBandBoxRendererJ3D
visad.bom.RubberBandLineRendererJ3D
```

The major division of this class hierarchy is between the two graphics APIs currently used to implement VisAD `Displays`: `Java2D` and `Java3D`. The classes under `visad.java3d.RendererJ3D` generate `Data` object depictions as `Java3D` scene graphs. The classes under `visad.java2d.RendererJ2D` generate `Data` object depictions as VisAD internal scene graphs, using subclasses of `visad.VisADSceneGraphObject`, which are rendered using Java2D.

When applications link a `DataReference` to a `DisplayImpl` by invoking the `DisplayImpl`'s `addReference()` or `addReferences()` method, they can optionally pass an object of class `DataRenderer`. If they do not pass such an object then a default is constructed by the `DisplayImpl`. This default is a `visad.java2d.DefaultRendererJ2D` object for a `DisplayImplJ2D` and a `visad.java3d.DefaultRendererJ3D` object for a `DisplayImplJ3D`. These default `DataRenderers` implement logic for generating depictions of virtually any VisAD `Data` object according to virtually any set of `ScalarMaps`. This generality is necessary for the default `DataRenderers` in order for applications to be able to visualize virtually any data in any way, without needing to define their own non-default `DataRenderers`. However, non-default `DataRenderers` can be selective about which `Data` objects and which sets of `ScalarMaps` they accept.

10.1.1. Reasons for Non-Default DataRenderers

Non-default `DataRenderers` exist for the following reasons:

1. To produce `Data` object depictions more efficiently (i.e., faster or using less memory) than the default `DataRenderers`.
2. To produce depictions with appearances different than the default appearances.
3. To interpret user gestures as changes to `Data` object values. These are known as direct manipulation `DataRenderers`.
4. To combine multiple `Data` objects in a single depiction.
5. Or a limitless list of more radical reasons. For example, NodeRendererJ3Ds on a set of cluster nodes make Serializable scene graph depictions for parts of a `Data` object distributed over the cluster, and a ClientRendererJ3D collects and merges these into a unified depiction in a DisplayImplJ3D on a client machine.

We will give examples of `DataRenderer` subclasses that exist for each of the first three reasons. The `visad.bom.ImageRendererJ3D` class is designed to generate depictions of rectangular images and image sequences more efficiently than the defaults. The `Data` object linked via a `visad.bom.ImageRendererJ3D` object must have a `MathType` that conforms to one of the patterns:

```
((x, y) -> z)
(t -> ((x, y) -> z))
((x, y) -> (r, g, b))
(t -> ((x, y) -> (r, g, b)))
```

and the `Display` object must have `ScalarMaps` (actually a subset of these as needed for the `RealTypes` in the `MathType`):

```
t -> Animation
x -> a spatial axis
y -> a different spatial axis
z -> RGB
r -> Red
g -> Green
b -> Blue
```

Further, the domain `Set` of the `Field` with `MathType ((x, y) -> ...)` must be a `GriddedSet`. The `visad.bom.BarbRendererJ3D` class is designed to render identically to the default except that flows are rendered by wind barbs rather than arrows. Thus its linked `Data` object and `Display` object can have the same broad range of `MathTypes` and sets of `ScalarMaps` allowed by the default `DataRenderers`. The `visad.java3d.DirectManipulationRendererJ3D` class is designed to interpret user gestures as data changes for a variety of simple `MathTypes` and sets of `ScalarMaps`. The `Data` object linked via a `visad.java3d.DirectManipulationRendererJ3D` object must have a `MathType` that conforms to one of the patterns:

```
x
(..., x, ...)
(..., x, ..., y, ...)
(..., x, ..., y, ..., z, ...)
(x -> (..., y, ...)
(x -> (..., y, ..., z, ...)
```

The exact criteria on `ScalarMaps` of the Display object are complex. In the case of a `RealType` 'x' or a `RealTupleType` (..., x, ...), there must be a `ScalarMap` of x to a spatial axis, with other `ScalarMaps` of x allowed. In the case of a `RealTupleType`

(..., x, ..., y, ...) or (..., x, ..., y, ..., z, ...) there must be a ScalarMap of at least one of the component RealTypes to a spatial axis. In the case of a FunctionType (x -> (..., y, ...) or (x -> (..., y, ..., z, ...) there must be a ScalarMap of x to a spatial axis and a ScalarMap of at least one of y or z to a spatial axis. These ScalarMap criteria are designed so that there is a way to interpret spatial gestures as unambiguous modifications of at least some values in the Data object.

There is no DataRenderer subclass currently part of VisAD that exists for the fourth reason: to combine multiple Data objects in a single depiction. However, a user did crea such a DataRenderer almost immediately after the system's initial release in early 1998 (a heroic effort). The purpose was to texture map one image Data object onto a surface defined by another Data object, where the two Data objects had different spatial sampling resolution. Technically this DataRenderer should have been accompanied by new instances of DisplayRealType to be used in ScalarMaps defining what RealType values would be used to compute pixel coordinates in the texture image Data object. However, the DataRenderer used the existing DisplayRealTypes Red and Green for that purpose (somewhat of a hack solution, but effective).

If you are defining a new subclass of DataRenderer, it should be for one of the five reasons listed in this section. You will need to think about what restrictions your DataRenderer will place on the MathType of its Data object and the set of ScalarMaps in its Display object. You may also need new instances of DisplayRealType to define various parameters of novel rendering techniques.

10.1.2. How to Avoid Writing Non-Default DataRenderers

If you are contemplating writing your own subclass of DataRenderer, a key question is whether there is some other way to accomplish your goals. The alternative to a custom DataRenderer is often a network of Data and Cell (i.e., computational) components, possibly including existing non-default DataRenderers. For example, applications can alter the appearance of Data depictions by replacing the simple network:

```
Data -> DisplayImpl
```

with:

```
Data -> CellImpl -> Data -> DisplayImpl
```

The idea is that the CellImpl computes a new Data object (or perhaps several Data objects) whose depiction generated by existing DataRenderers will have the desired appearance for the original Data object. For example, the derived Data object or objects may include new RealType vaues mapped to Shape, that can be used to "draw" virtually any depiction.

Complex manipulation of `Data` objects can sometimes be accomplished by linking auxilliary `Data` objects to the `DisplayImpl` via existing direct manipulation `DataRenderers`, with `CellImpl` components that compute new values for the original `Data` object based on the user's manipulation of the auxilliary `Data` objects. For example, `RealTuple` data objects, draggable via `DirectManipulationRendererJ3D`, can be placed at vertices of the depiction of a complex `Data` object, with a `CellImpl` that moves the corresponding "vertex" of complex `Data` object.

The `visad.bom.FrontDrawer` class is a good example. It enables users to draw weather fronts. It includes a `Set` object linked to the `DisplayImpl` via a `CurveManipulationRendererJ3D`. When the user finishes drawing the `Set`, a `CellImpl` is executed that smooths the curve represented by the `Set`, and uses it to derive a complex `FieldImpl` whose depiction is a repeating frontal shape along the smoothed curve.

It is generally true that most goals can be met with clever networks of existing VisAD components and `DataRenderers`, allowing programmers to avoid creating new `DataRenderer` subclasses.

10.1.3. DataRenderer Constructors

Your new subclass of `DataRenderer` will be a subclass of `visad.java2d.RendererJ2D` or `visad.java3d.RendererJ3D`, unless you are implementing VisAD displays for a new graphics API or doing something equally radical. In fact, your new `DataRenderer` will probably be a subclass of `visad. java2d. DefaultRendererJ2D` or `visad. java3d. DefaultRendererJ3D` if it does not interpret user gestures as `Data` object changes, and a subclass of `visad. java2d. DirectManipulationRendererJ2D` or `visad. java3d. DirectManipulationRendererJ3D` if it does. All of these classes have constructors with no arguments, so your new `DataRenderer` subclass does not need an explicit constructor unless it needs special arguments from the constructor. For example, the `visad. bom. CurveManipulationRendererJ3D` is a subclass of `visad. java3d. DirectManipulationRendererJ3D` that allows users to define `UnionsSets` of `Gridded2Dets` with manifold dimension = 1 (typically used to define map outlines) by free hand drawing. Its constructors define arguments for defining conditions on the shift and control keys for enabling user drawing, and a boolean argument to restrict the `UnionSet` to a single `Gridded2Dset`.

10.1.4. ShadowTypes

The real work of generating depictions of `Data` objects is done by subclasses of `visad.ShadowType`. Every `Data` object has a `MathType`, which is really a tree structure of various subclasses of `MathType`. For example, the shorthand `MathType` notation:

```
(hour -> ((line, element) -> brightness))
```

actually represents the tree structure:

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

Recursive algorithms that traverse this tree structure are used to generate depictions of `Data` objects with this `MathType`. These recursive algorithms need to be able to store temporary information in the nodes of the tree structure. However, since a `Data` object may be linked to many Display objects, each with their own `DataRenderer`, using the `MathType` objects for temporary storage would lead to conflicts. Furthermore, the recursive algorithms may vary between different `DataRenderers`. Hence another class hierarchy is needed for building tree structures that "shadow" the `MathType` tree structure. This is the class hierarchy under `visad.ShadowType`. A tree structure of `ShadowTypes` is created for each link between a `Data` object and a Display object, and different subclasses of `ShadowType` can be used to define different algorithms for generating `Data` depictions. The `ShadowType` class hierarchy includes one set of classes that are independent of graphics API, a set for each graphics API (Java2D and Java3D), and others as needed for non-default `DataRenderers`. The hierarchy independent of graphics API is:

```
ShadowType
ShadowScalarType
ShadowRealType
ShadowTextType
ShadowTupleType
ShadowRealTupleType
ShadowFunctionOrSetType
ShadowFunctionType
ShadowSetType
```

Note the neat correspondence of this hierarchy to the `MathType` hierarchy, except for the addition of `ShadowFunctionOrSetType`. This exists because the visualization algorithms for `Set` and Function objects are essentially identical (Sets are treated as the domain Sets of Fields without any range values), and common code for `ShadowFunctionType` and `ShadowSetType` can go in `ShadowFunctionOrSetType`.

The classes visad.java2d.ShadowTypeJ2D and visad.java3d.ShadowTypeJ3D are subclasses of visad.ShadowType, and these have subclass hierarchies:

```
ShadowTypeJ2D
ShadowScalarTypeJ2D
ShadowRealTypeJ2D
ShadowTextTypeJ2D
ShadowTupleTypeJ2D
ShadowRealTupleTypeJ2D
ShadowFunctionOrSetTypeJ2D
ShadowFunctionTypeJ2D
ShadowSetTypeJ2D
```

```
ShadowTypeJ3D
ShadowScalarTypeJ3D
ShadowRealTypeJ3D
ShadowTextTypeJ3D
ShadowTupleTypeJ3D
ShadowRealTupleTypeJ3D
ShadowFunctionOrSetTypeJ3D
ShadowFunctionTypeJ3D
ShadowSetTypeJ3D
```

Because Java does not allow multiple inheritance, objects of these classes adapt objects of the corresponding graphics-API-independent classes in order to have access to their methods. For example, the ShadowTypeJ3D class includes the variable:

```
ShadowType adaptedShadowType;
```

and the ShadowRealTupleTypeJ3D class includes the method:

```
public ShadowRealTupleType getReference() {
  return ((ShadowRealTupleType) adaptedShadowType).getReference();
}
```

ShadowRealTupleTypeJ3D includes similar implementations for every other method it needs to "inherit" from ShadowRealTupleType, and other graphics-API-dependent classes include similar sets of method implementations invoked via adaptedShadowType.

Understanding logic in the ShadowType classes can be a bit tricky, because it moves between methods in the graphics-API-independent classes and methods in the graphics-API-dependent classes. Much of the logic of generating depictions is done in the graphics-API-independent classes which construct VisAD's internal scene graphs (subclasses of visad.VisADSceneGraphObject). These are either converted to Java3D scene graphs by subclasses of ShadowTypeJ3D, or left as is by subclasses of ShadowTypeJ2D (for later rendering using Java2D). Throughout the rest of this tutorial, we will use the notation ShadowTypeJ*D to indicate any graphics-API-dependent analog of ShadowType, and similarly ShadowFunctionTypeJ*D and so on for graphics-API-dependent analogs of subclasses of ShadowType.

A subclass of DataRenderer defines the subclasses of ShadowType it will use to generate Data depiction by implementing a set of factory methods. Here are the implementations of these methods in visad.java3d.RendererJ3D:

```java
public ShadowType makeShadowFunctionType(
  FunctionType type, DataDisplayLink link, ShadowType parent)
  throws VisADException, RemoteException {
    return new ShadowFunctionTypeJ3D(type, link, parent);
}

public ShadowType makeShadowRealTupleType(
  RealTupleType type, DataDisplayLink link, ShadowType parent)
  throws VisADException, RemoteException {
    return new ShadowRealTupleTypeJ3D(type, link, parent);
}

public ShadowType makeShadowRealType(
  RealType type, DataDisplayLink link, ShadowType parent)
  throws VisADException, RemoteException {
    return new ShadowRealTypeJ3D(type, link, parent);
}

public ShadowType makeShadowSetType(
  SetType type, DataDisplayLink link, ShadowType parent)
  throws VisADException, RemoteException {
    return new ShadowSetTypeJ3D(type, link, parent);
}

public ShadowType makeShadowTextType(
  TextType type, DataDisplayLink link, ShadowType parent)
  throws VisADException, RemoteException {
    return new ShadowTextTypeJ3D(type, link, parent);
}

public ShadowType makeShadowTupleType(
  TupleType type, DataDisplayLink link, ShadowType parent)
  throws VisADException, RemoteException {
    return new ShadowTupleTypeJ3D(type, link, parent);
}
```

In each of these method signatures, the 'type' argument is the corresponding object from the tree structure of MathTypes, the 'link' argument is the DataDisplayLink object that defines the link between a Data object and a Display object, and the 'parent' argument is the parent ShadowType in the tree structure (or null if this ShadowType is the root of the tree).

It is possible that a non-default DataRenderer would consist solely of implementations of some of these factory methods, defining alternate subclasses of ShadowType for generating Data depictions.

10.1.5. DisplayRealTypes

Instances of the visad.DisplayRealType class define various types of values used by algorithms for generating Data depictions. These include display spatial axes (e.g., XAxis, YAxis, ZAxis), color components (e.g., Red, Green, Blue), Animation, IsoContour, flow components (e.g., Flow1X, Flow1Y, Flow1Z), etc. Applications do not define subclasses of DisplayRealType. Instead they define new instances of DisplayRealType.

New DisplayRealType instances may imply new rendering algorithms and hence require new subclasses of DataRenderer and ShadowType. However, the default DataRenderers can detect and interpret new instances of DisplayRealType for new spatial, color or flow coordinates, as long as they are components of a DisplayTupleType with a CoordinateSystem whose reference is Display.DisplaySpatialCartesianTuple = (XAxis, Yaxis, Zaxis), Display.DisplayRGBTuple = (Red, Green, Blue), Display.DisplayFlow1Tuple = (Flow1X, Flow1Y, Flow1Z), or Display.DisplayFlow2Tuple = (Flow2X, Flow2Y, Flow2Z). This enables applications to define new spatial, color and flow coordinates without defining new DataRenderers.

10.1.6. General DataRenderer Theory of Operation

A Display is either a local DisplayImpl or a RemoteDisplayImpl, which adapts a local DisplayImpl. Methods of RemoteDisplayImpl simply invoke the corresponding methods of the adpated DisplayImpl, so we only need to understand DisplayImpl. Its doAction() method is invoked when one of its linked Data or DataReference objects changes value, or when some other event such as a Control change occurs, that may require the scene graph for any linked Data object to be recomputed. The doAction() method invokes the prepareAction() method of each linked DataRenderer, which determines if recomputation of the scene graph is required for this DataRenderer, and computes the ranges of RealType values in the linked Data object for autoscaling, if requested by the DisplayImpl (this will happen if this is the first attempt to display any linked data, if the application calls a method of DisplayImpl requesting autoscaling, or if a previous autoscaling request failed to establish a value range for some RealType because of null or missing data).

No current subclass of DataRenderer overrides the implementation of prepareAction() in DataRenderer. This invokes the DataDisplayLink. prepareData() method, which computes default values for DisplayRealTypes, analyzes the ScalarMaps linked to the DisplayImpl via calls to the ShadowType. checkIndices() method, and calls the DataRenderer. checkDirect() method to determine whether this DataRenderer supports direct manipulation for the linked Data object and set of ScalarMaps. The checkIndices() recursively calls itself down the tree structure of ShadowTypes to determine which ScalarMaps are relevant to each subtree of the MathType tree structure, including especially which are relevant to each RealType and TextType (these are the leaves of the MathType tree). The checkIndices() method determines whether the combination of MathType and ScalarMaps are feasible for rendering (e.g., a ScalarMap to Animation is illegal for a RealType occurring in a Function range) and generates an appropriate Exception if not. The checkIndices() method also precomputes lots of information useful for generating Data depictions and saves it in the ShadowTypes.

You probably do not need to override the prepareAction() method in your new DataRenderer. If you need new instances of DisplayRealType that are not new spatial, color or flow coordinates, then you probably do need to override the checkIndices() methods of your new ShadowTypes. The default implementations of checkIndices() are complex in order to deal with arbitrary MathTypes and sets of ScalarMaps. However, most custom DataRenderers deal with restricted MathTypes and ScalarMaps and hence can have much simpler implementations of checkIndices() (and other methods that you may need to override).

After DisplayImpl. doAction() invokes the prepareAction() method for each linked DataRenderer, it uses the RealType range data to autoscale the ScalarMaps if autoscaling is requested, then invokes the doAction() method for each linked DataRenderer. This method has implementations in visad. java2d.RendererJ2D and visad.java3d.RendererJ3D, which invoke the DataRenderer.doTransform() method if the prepareAction() method determined that the scene graph for the linked Data needs to be recomputed. These RendererJ2D and RendererJ3D implementations of doAction() manage the attachment and de-attachment of the scene graph depicting their Data objects to and from the overall scene graph for the DisplayImpl.

You probably do not need to override the doAction() method in your new DataRenderer. None of the DataRenderer subclasses distributed with VisAD override the implementations in visad.java2d.RendererJ2D and visad.java3d.RendererJ3D.

A number of non-default DataRenderers override the implementation of the doTransform() method. This method returns a scene graph depicting the linked Data and has different signatures for different graphics APIs. Hence doTransform() is not declared as a method of the abstract DataRenderer class, but is rather a method name reused in similar ways by DataRenderer's subclasses for different graphics APIs. The implementations of doTransform() in visad.java2d.DefaultRendererJ2D and visad.java3d.DefaultRendererJ3D are quite similar. They both construct a scene graph group node to serve as the parent for the scene graph depicting the linked Data object (a javax.media.j3d.BranchNode for DefaultRendererJ3D and a visad.VisADGroup for DefaultRendererJ2D). They get the root ShadowTypeJ*D for the linked Data object, which will be the root of a ShadowType tree containing results computed by DataRenderer.prepareAction(). They get the linked Data object and catch any RemoteException indicating failure to access a remote Data object (if the Data object is null, they simply return a null value for the parent group of the scene graph which will trigger a "data is null" message in the display). The real work of DataRenderer.doTransform() is done by the call to doTransform() method of the root ShadowTypeJ*D (note that doTransform() is not a method of ShadowType, but is a method of its subclasses). This doTransform() call is bracketed by calls to the preProcess() and postProcess() methods of ShadowType. These were designed into the system as a way to accumulate information during the ShadowType.doTransform() call that is only assembled into a scene graph after it is all accumulated, but this feature has never been used. Your DataRenderer can probably ignore the preProcess() and postProcess() methods. The doTransform() method in both DefaultRendererJ2D and DefaultRendererJ3D calls the DataDisplayLink.clearData() method. The reference to Data is cached in the DataDisplayLink during a DisplayImpl.doAction() cycle, in order to maintain consistency in case Data changes during the process, and in order to avoid multiple retrievals of remote Data. The clearData() method clears this cache. The DataRenderer.doTransform() method also initializes some arrays passed to the doTransform() method of the root ShadowType - more about these later.

The visad. java2d. DirectManipulationRendererJ2D and visad. java3d. DirectManipulationRendererJ3D classes define their own implementations of doTransform() in order to add a test for whether direct manipulation is supported for the linked Data and Displays (this test is primarily on the Data's MathType and the Display's ScalarMaps).

10.1.7. General ShadowType Theory of Operation (KEY SECTION)

This is a key section because the tree structure under a root ShadowType provides the basis for the way that a scene graph is constructed to depict a Data object. In fact, there are four related tree structures involved:

1. The Data object's tree structure, with Real, Text and Set objects as leaves, and Tuple and Field objects as non-leaf nodes.

2. The Data object's MathType also forms a tree structure, similar to the Data tree structure except that there is a single range MathType under a FunctionType, but may be many range Data objects under the corresponding Field. There is a diagram of a MathType tree structure at the start of Section 1.4.

3. The ShadowType tree structure is derived from and identical to the MathType tree structure. The doTransform() method is called recursively down this tree structure. A recursive call is made for each range Data value under a Field, rather than just once for the single range ShadowType of the ShadowFunctionTypeJ*D. Also, no recursive call is made in certain cases.

4. The scene graph depiction of a `Data` object has a tree structure roughly similar to the `Data` tree structure. This scene graph tree structure is assembled via the scene graph groups returned by the recursive calls to `doTransform()`.

As noted, the `doTransform()` recursive calls do not always descend all the way to the leaf nodes in the `ShadowType` tree structure. Rather, the analysis in the recursive `ShadowType.checkIndices()` calls determines that certain nodes in the `ShadowType` tree structure are designated as "terminal" nodes, meaning that `doTransform()` is not called recursively to the children of these nodes. A `ShadowFunctionTypeJ*D` node is terminal if it is "flat" (i.e., the range of its `FunctionType` is a `RealType`, a `RealTupleTyple`, or a `TupleType` of `RealTypes` and `RealTupleTypes`). A `ShadowSetTypeJ*D` is terminal. A "flat" `ShadowTupleTypeJ*D` (including any `ShadowRealTupleTypeJ*D`), `ShadowRealTypeJ*D` or `ShadowTextTypeJ*D` is terminal if it is not the child or descendant of a terminal `ShadowType`.

The `ShadowType` tree structure plays one more key role in the way scene graphs are constructed: `DisplayRealType` values are passed down the tree structure and used to determine the locations, colors and other graphical attributes of scene graph nodes. Any Real values in a non-terminal Tuple (i.e., a Tuple corresponding to a non-terminal `ShadowTupleTypeJ*D`) are converted to `DisplayRealType` values via any applicable `ScalarMaps` and passed down to any non-Real components of the Tuple. Similarly, Real values from the domain of a non-terminal Field are converted to `DisplayRealType` values via any applicable `ScalarMaps` and passed down to the corresponding range `Data` objects. These values are passed down in the 'value_array' argument of the `doTransform()` method. At terminal nodes, these "passed down" `DisplayRealType` values are combined with `DisplayRealType` values computed from the corresponding `Data` object to create scene graph nodes.

The key method of the `ShadowType` subclasses is `doTransform()`. It has one signature in the graphics-API-dependent subclasses and a different signature in the graphics-API-independent subclasses. Specifically, the signature in the graphics-API-independent subclasses is:

```
public boolean doTransform(Object group, Data data, float[] value_array,
  float[] default_values, DataRenderer renderer, ShadowType shadow_api)
  throws VisADException, RemoteException
```

and the signature in the graphics-API-dependent subclasses is:

```
public boolean doTransform(Object group, Data data, float[] value_array,
  float[] default_values, DataRenderer renderer)
  throws VisADException, RemoteException
```

The reason for these different signatures is that the recursive calls to `doTransform()` are made on the tree of graphics-API-dependent `ShadowTypes`, but these generally delegate their work by calling `doTransform()` for their adapted graphics-API-independent `ShadowType` and that call has an extra 'shadow_api' argument where the graphics-API-dependent `ShadowType` can pass a 'this' reference to itself. The `doTransform()` method of the graphics-API-independent `ShadowType` can use this to invoke methods that require graphics-API-dependent logic (for example, adding geometry and appearance informaiton to a scene graph group).

The arguments to `doTransform()` are:

Object group parent scene graph group for any scene graph subtrees generated by this `doTransform()`.

float[] value_array array of `DisplayRealType` values passed down `ShadowType` tree in recursive `doTransform()` calls. For any `ScalarMap` 'map' the index of the value of its `DisplayRealType` in `value_array` is returned by `map.getValueIndex()`.

float[] default_values array of default values for `DisplayRealTypes` (to be used if no values is determined by a `ScalarMap`), passed down `ShadowType` tree in recursive `doTransform()` calls. For any `ScalarMap` 'map' the index of the value of its `DisplayRealType` in `default_values` is returned by `map.getDisplayScalarIndex()`.

DataRenderer renderer the `DataRenderer` that made the top-level call to `doTransform()`.

ShadowType shadow_api for the graphics-API-independent `ShadowType` subclasses only, this is the corresponding graphics-API-dependent `ShadowType`.

Different `ShadowType` subclasses have different implementations of `doTransform()`, but they all work in two basic stages:

1. converting data values into `DisplayRealType` values via `ScalarMaps`, and
2. for non-terminal `ShadowTypes` passing the `DisplayRealType` values to recursive calls to `doTransform()`, and for terminal `ShadowTypes` using the `DisplayRealType` values to construct scene graph nodes.

The `doTransform()` method of a graphics-API-dependent `ShadowType` typically just invokes the `doTransform()` method of its adapted graphics-API-independent `ShadowType`. This starts by checking for null data and other error conditions, then getting a Vector of `ScalarMaps` and associated housekeeping information. It constructs an array `float[][] display_values` for accumulating `DisplayRealType` values converted via `ScalarMaps` from data values. Then it fills the 'display_values' array with any `DisplayRealType` values in the 'float[] value_array' argument passed down the tree, as determined from the inherited_values array computed by the `checkIndices()` method during the `prepareAction()` phase.

The way that 'display_values' is filled with data values varies for different `MathTypes`. A Real object only has one value, but its `RealType` may occur in multiple `ScalarMaps` and so it may fill multiple entries in 'display_values'. A `RealTuple` object or terminal Tuple object has multiple Real values, each used to fill entries in 'display_values' according to relevant `ScalarMaps`. A non-terminal Tuple object similarly accumulates entries into its 'display_values' array, then passes this as the 'value_array' argument in recursive `doTransform()` calls for each Tuple component that is not a Real or a `RealTuple`.

`Field` and `Set` objects will have multiple values for the same `RealType`, one for each sample of the `Field` or `Set`. Thus `float[][] display_values` is doubly indexed to permit an array of multiple values for some of its entries. Note that `RealTuple`, `Set` and `Field` objects may have `CoordinateSystems` with reference `RealTuples` whose Real values may occur in `ScalarMaps`: these must also be converted to `DisplayRealType` values and put into the 'display_values' array. Finally, Text values are handled specially. They are not passed as arguments down the recursive `doTransform()` calls, but are stored in variables of one `ShadowType` and then retrieved by its child nodes in the `ShadowType` tree. This works because it only makes sense to have a single Text value at any terminal node in the `ShadowType` tree.

Once all data values have been converted to `DisplayRealType` values in the 'display_values' array, they are either passed to recursive calls to `doTransform()` or in terminal `ShadowTypes` they are used to construct scene graph nodes. In a non-terminal `ShadowType` the scene graphs returned by the recursive `doTransform()` calls are all made children of a scene graph group. In a non-terminal Field `ShadowType` whose domain is a single `RealType` mapped to Animation or `SelectValue`, the scene graph group is a Switch, which is linked into the `AnimationControl` or `ValueControl` which selects a scene graph child based on Animation or `SelectValue` behavior.

In a terminal `ShadowType`, a sequence of calls are made to methods that assemble various kinds of graphical information from appropriate `DisplayRealType` values in the `display_values` array. These methods are implemented in `ShadowType` and are:

assembleSelect() assembles boolean flags from values for `SelectRange`. This information is altered when other `assemble*()` methods find missing values.

assembleColor() assembles red, green, blue and alpha byte values from values for Red, Green, Blue, Alpha, RGB, RGBA and any other `DisplayRealTypes` in a color coordinate system with reference (Red, Green, Blue).

assembleFlow() assembles Cartesian Flow1 and Flow2 values from values for any flow `DisplayRealTypes`.

assembleSpatial() assembles Cartesian spatial coordinates from values for XAxis, YAxis, ZAxis and any other `DisplayRealTypes` in a spatial coordinate systems with reference (XAxis, YAxis, ZAxis). If needed for filled rendering (e.g., lines, triangles, textures) or contours, this also constructs a spatial `Set` object to supply a topology for rendering. The spatial `Set` will have domain dimension = 3 for (XAxis, YAxis, ZAxis) but may have manifold dimension <= 3.

assembleShape() assembles an array of `VisADGeometryArrays` from values for Shape.

If there are any `DisplayRealType` values for `Shape`, `Text`, `Flow` or `IsoContour` these are handled specially. Each of these results in data being depicted by some specialized "shape" other than a 0-D, 1-D, 2-D or 3-D "graph" of the data. There are methods in `ShadowType` named `makeFlow()`, `makeText()` and `makeContour()` which make these various specialized shapes. These methods can be over-ridden in extensions of `ShadowType` to change the appearance of data depictions.

If there are no `DisplayRealType` values for `Shape`, `Text`, `Flow` or `IsoContour` then data are depicted directly via a 0-D, 1-D, 2-D or 3-D "graph". The `makePointGeometry()` method in `ShadowType` depicts data as isolated points, eliminating `NaN` values (i.e., missing values). This is used for data that have manifold dimension = 0, and as a "punt" for data with manifold dimension = 3 but where volume rendering is not done because the topology in (XAxis, YAxis, ZAxis) coordinates is not a `LinearSet`. Otherwise data are depicted by a 1-D, 2-D or 3-D graph. The only 3-D graph option is volume rendering, which is done in `visad.ShadowOrFunctionSetType.doTransform()` via 3-D textures (actually a stack of 2-D textures because 3-D texture mapping is not implemented on Windows NT) when the boolean `isTexture3D = true`.

2-D graphs may be implemented by shaded triangles or by 2-D texture mapping in `visad.ShadowOrFunctionSetType.doTransform()` when either of the booleans `isTextureMap` or `curvedTexture = true`. Note `isTextureMap` is true only if the topology in (XAxis, YAxis, ZAxis) coordinates is a `LinearSet`. If `curvedTexture = true` then the data texture is laid on a sub-sampled surface (for efficiency) and hence rendering is not an exactly accurate depiction. The degree of sub-sampling is controlled by the `curvedSize` variable in `GraphicsModeControl`.

For 2-D or 3-D linear textures, missing data (including data not selected in `SelectRange`) is depicted as either black or transparent, depending in the `missingTransparent` flag in `GraphicsModeControl`. For `curvedTexture` and for non-texture 1-D and 2-D graphs, missing data is handled by removing missing points from the geometry via the `VisADGeometryArray.removeMissing()` method (with different implementations for different sub-classes). Note this is preceded by a call to `Set.cram_missing()`, which sets `NaNs` in the spatial `Set` (normally `NaNs` are illegal as `Set` coordinates) to be detected later by `removeMissing()`. Map projection discontinuities are removed from 1-D and 2-D geometries via calls to `VisADGeometryArray.adjustLongitude()` and

`VisADGeometryArray. adjustSeam(). adJustLongitude()` detects and removes lines and triangles crossing a longitude seam (often at the 180 degree date line, but not always). `adjustSeam()` detects and removes lines and triangles crossing any map projection seam (it is not always accurate in detecting seams, since it uses a heuristic method based on derivatives of `DisplayTupleType CoordinateSystem` transforms).

10.1.8. Direct Manipulation Theory of Operation

Direct manipulation `DataRenderers` translate user mouse or wand gestures (generally with the right mouse button held down) as changes to `Data` values. The `visad.DataRenderer` class defines a context for doing this, as a set of methods that direct manipulation `DataRenderers` need to implement (these methods have non-abstract implementation in `DataRenderer`, which must be over-ridden for a direct manipulation `DataRenderer` to function correctly). Their signatures are:

```
// determine if the MathType and ScalarMap are valid for direct manipulation
public void checkDirect()
  throws VisADException, RemoteException

// return reason why direct manipulation is invalid
public String getWhyNotDirect()

// save array of spatial locations for manipulation "grab points"
public synchronized void setSpatialValues(float[][] spatial_values)

// return minimum distance from mouse ray to a "grab point"
public synchronized float checkClose(double[] origin, double[] direction)

// interpret mouse ray as a manipulation of data
public synchronized void drag_direct(VisADRay ray, boolean first, int ↩
    mouseModifiers)
```

Other methods that direct manipulation `DataRenderers` may implement (but are not required to) include:

```
// may be called by drag_direct() for temporary scene graph change
public void addPoint(float[] x)
  throws VisADException

// called when mouse button is released, ending manipulation
public synchronized void release_direct()

// may be called by applications to stop manipulation
public void stop_direct()

// return the index of the "grab point" closest to the mouse ray
public int getCloseIndex()
```

Note that some direct manipulation `DataRenderers` include implementations of the `doTransform()` method (with signature appropriate for their graphics API). For example, `visad.bom.PointManipulationRendererJ3D`, `visad.bom.RubberBandBoxRendererJ3D` and `visad.bom.RubberBandLineRendererJ3D` all include implementations that return empty `BranchGroups`, since none of them actually creates `Data` depictions.

The `checkDirect()` method is called by `DataDisplayLink.prepareData()` and decides whether this `DataRenderer` supports the `Data`'s `MathType` and the `Display`'s `ScalarMaps`. Rather than returning a boolean, it records its decision by a call to:

```
public void setIsDirectManipulation(boolean b)
```

If `checkDirect()` decides that it doesn't support the `MathType` and `ScalarMaps`, it records a reason in a String to be returned by a call to `getWhyNotDirect()`.

The `setSpatialValues()` method is called by `doTransform()` (or by the methods it invokes) to record the "grab points" of the `Data` depiction in 3-D graphics coordinates. Note that its spatial_values argument array is organized float[3][number_of_points]. If the `Data` object is a `Real` or `RealTuple`, then number_of_points will be 1, but if the `Data` object is a `Field` or `Set` then the depiction will be a curve and there will be many grab points along that curve. Note that for `visad.bom.BarbManipulationRendererJ2D` and `visad.bom.BarbManipulationRendererJ3D` the grab point location is head of the wind barb, whereas the (latitude, longitude) location of the wind determines the location of the barb's tail. However, for most direct manipulation `DataRenderers` the grab point locations coincide with `Data` spatial locations.

The `MouseBehavior` invokes the `checkClose()` and `drag_direct()` methods when the user holds down the right mouse button (the choice of mouse button can of course be changed by custom `MouseBehavior` subclasses, and note a wand is substituted for the mouse by `visad.java3d.WandBehaviorJ3D`). Mouse locations define rays in 3-D space (for 2-D graphics the ray is simply into the screen, i.e., parallel to the Z axis). The ray is passed to `checkClose()` as an origin and direct, but passed to `drag_direct()` as a `VisADRay` (these are equivalent).

The `checkClose()` method returns the minimum distance from the ray to the grab points passed to the `DataRenderer` via `setSpatialValues()`. When the right mouse button is first pressed, the `MouseBehavior` compares the distances it gets from each direct manipulation `DataRenderer` linked to the `Display`. All subsequent mouse motion events with the right button pressed generate calls to the `drag_direct()` method of the `DataRenderer` whose `checkClose()` returned the least distance.

The `checkClose()` method computes the perpendicular distance from the ray to each grab point. For the closest grab point it determines the closest point on the ray and stores a 3-D vector offset (in variables named offsetx, offsety and offsetz) from the closest point to the grab point. This offset vector is used in `drag_direct()` to avoid having the data values "snap" to the cursor, if the application has called the `DataRenderer` method:

```java
public void setPickCrawlToCursor(boolean b)
```

with `b = true`. In this case, the `Data` value gradually "crawls" toward the mouse location.

Some `DataRenderers`, such as `visad.bom.CurveManipulationRendererJ3D`, allow the user to draw new `Data` depictions even where no depiction exists. In such circumstances their `checkClose()` implementations sometimes return 0.0f as a way to assert their claim to the manipulation. In order to avoid such `DataRenderers` monopolizing all manipulations, their constructors have arguments where application can specify conditions on SHIFT and CTRL key states under which they are active. The `checkClose()` methods of such `DataRenderers` can call the `DataRenderer` method:

```java
public int getLastMouseModifiers()
```

to get the SHIFT and CTRL key states when the right mouse button was pressed.

When a `DataRenderer` may have multiple grab points, they may implement the `getCloseIndex()` method to allow application to retrieve the index of the closest grab point as determined by `checkClose()`. For example, `getCloseIndex()` is implemented by `visad.bom.PickManipulationRendererJ3D` to enable applications to discover which point along a curve (and hence which Field or `Set` sample) was picked by the user.

The `drag_direct()` method does the real work of a direct manipulation `DataRenderer`. It determines a 3-D graphical location from the cursor ray (this involves picking a point along the cursor ray, which is a bit subtle - more about this below), converts this back through any applicable display spatial `CoordinateSystem`, then back through applicable `ScalarMaps`, to get up to 3 visad.Real values. These are used to update Real sub-objects of the `Data` object being manipulated. The Real values are also used to generate Strings passed to the `DisplayRenderer.setCursorStringVector()` method (to be displayed as a cursor location in the upper left corner of the Display window unless the application has disabled the cursor location display).

The default implementation in `DataRenderer.drag_direct()` illustrates the functions required of any implementation of this method. First, it checks to make sure that critical information is available (non-null). Then it checks whether the applications has called `stop_direct()`. Then it extracts the origin and direction of its `VisADRay` argument and, if `pickCrawlToCursor` has been set, adds a decreasing fraction of the pick offset to the origin. If it is the first call to `drag_direct()` after the right mouse click, it gets the grabbed spatialValues location in point_x, point_y and point_z.

Next comes the subtle problem of determining unique new `RealType` values, which requires a point in 3-D (or 2-D), whereas a mouse location defines a ray consisting of an infinite numbers of points. In the default implementation in `DataRenderer.drag_direct()`, this ambiguity is resolved in one of two ways. If only one or two `ScalarMaps` of `RealTypes` are relevant for the `MathType` of the linked `Data`, then these determine a one- or two-dimensional sub-manifold of display space (a line or a plane). In this case the ambiguity is resolved by finding the intersection of the cursor ray with the plane or finding its closest point to the line. Note that the default implementation of `DataRenderer.drag_direct()` requires that spatial `ScalarMaps` are to the Cartesian spatial `DisplayRealTypes` (i.e., XAxis,YAxis and ZAxis) rather than through display `CoordinateSystems`, just so these one- and two-dimensional sub-manifolds are lines and planes rather than curved. If three `ScalarMaps` of `RealTypes` are relevant, then the ambiguity is resolved by intersecting the ray with the plane perpendicular with the ray and containing (point_x, point_y, point_z).

Some non-default implementations of `drag_direct()` resolve this ambiguity in other ways. For example, `visad.bom.CurveManipulationRendererJ3D.drag_direct()` allows `ScalarMaps` to be to non-Cartesian spatial `DisplayRealTypes`, and resolves the ambiguity by using Newton's method to find the intersection of the cursor ray with curved two-dimensional sub-manifolds in display space. This enables users to draw curves on the surfaces of spheres, for example.

Once a `drag_direct()` implementation has determined unique new `RealType` values, it must use them to appropriately modify `Data` objects. The default implementation in `DataRenderer.drag_direct()` provides a nice example of doing this in cases when the linked `Data` is a `Real`, a `RealTuple` and a `FlatField`.

Part III.

The VisAD Cookbock

11. Curtis Rueden's example apps

Curtis Rueden's wrote some additional VisAD examples and little apps that are presented here originally.

11.1. Additional VisAD examples

I have coded several small VisAD programs over the years to demonstrate various VisAD concepts. I thought it might be nice to provide them all from a web site, as one more VisAD resource. Enjoy! :-)

11.1.1. AnchoredPoint

A VisAD display containing a fixed-width line with one manipulable endpoint, and one fixed endpoint. This example should be useful for learning about VisAD's direct manipulation and computational cell (CellImpl) logic.

Listing 11.1: AnchoredPoint Example

```java
// AnchoredPoint.java

/*
This application demonstrates a fixed-length line with one manipulable
endpoint (the other endpoint is fixed at the display's center).
*/

import visad.*;
import visad.java3d.*;
import visad.util.Util;

import java.awt.event.*;
import java.rmi.RemoteException;

import javax.swing.*;

public class AnchoredPoint {

  private static final float LENGTH = 5;
  private static final float END_X = 2;
  private static final float END_Y = 3;

  public static void main(String[] args) throws Exception {
    // math types
```

```java
    RealType x = RealType.getRealType("x");
    RealType y = RealType.getRealType("y");
    final RealTupleType xy = new RealTupleType(x, y);

    // mappings
    ScalarMap xmap = new ScalarMap(x, Display.XAxis);
    ScalarMap ymap = new ScalarMap(y, Display.YAxis);
    xmap.setRange(END_X - LENGTH, END_X + LENGTH);
    ymap.setRange(END_Y - LENGTH, END_Y + LENGTH);

    // display
    DisplayImpl display = new DisplayImplJ3D("display",
      new TwoDDisplayRendererJ3D());
    display.disableAction();
    display.addMap(xmap);
    display.addMap(ymap);
    GraphicsModeControl gmc = display.getGraphicsModeControl();
    gmc.setScaleEnable(true);
    gmc.setPointSize(5.0f);

    // data references
    final DataReferenceImpl line_ref = new DataReferenceImpl("line");
    final DataReferenceImpl pt_ref = new DataReferenceImpl("point");
    display.addReference(line_ref);
    display.addReferences(new DirectManipulationRendererJ3D(), pt_ref, null);

    // data objects
    doPoint(xy, 0, 0, pt_ref);
    doLine(xy, 0, 0, line_ref);

    // computational cell
    CellImpl cell = new CellImpl() {
      public void doAction() {
        // get point coordinates
        RealTuple tuple = (RealTuple) pt_ref.getData();
        if (tuple == null) return;
        double[] vals = tuple.getValues();
        float xval = (float) vals[0];
        float yval = (float) vals[1];

        // adjust point coordinates
        float xlen = END_X - xval;
        float ylen = END_Y - yval;
        float len = (float) Math.sqrt(xlen * xlen + ylen * ylen);
        if (!Util.isApproximatelyEqual(len, LENGTH)) {
          double lamda = LENGTH / len;
          xval = (float) (END_X + lamda * (xval - END_X));
          yval = (float) (END_Y + lamda * (yval - END_Y));
          try { doPoint(xy, xval, yval, pt_ref); }
          catch (Exception exc) { exc.printStackTrace(); }
          return; // point change will retrigger cell
        }

        // update line
        try { doLine(xy, xval, yval, line_ref); }
        catch (Exception exc) { exc.printStackTrace(); }
      }
    };
    cell.addReference(pt_ref);
    display.enableAction();
```

```
    // show display onscreen
    JFrame frame = new JFrame("Fixed-length line with one anchored point");
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
    p.add(display.getComponent());
    frame.setContentPane(p);
    frame.setSize(400, 400);
    Util.centerWindow(frame);
    frame.show();
  }

  private static void doLine(RealTupleType rtt, float x, float y,
    DataReferenceImpl line_ref) throws VisADException, RemoteException
  {
    float[][] samples = { {x, END_X}, {y, END_Y} };
    Gridded2DSet set = new Gridded2DSet(rtt, samples, 2);
    line_ref.setData(set);
  }

  private static void doPoint(RealTupleType rtt, float x, float y,
    DataReferenceImpl pt_ref) throws VisADException, RemoteException
  {
    pt_ref.setData(new RealTuple(rtt, new double[] {x, y}));
  }

}
```

Download code: AnchoredPoint.java

11.1.2. CursorSSCell

A VisAD SpreadSheet cell extension that prints the cursor coordinates to the console as they change. This example should be useful for learning how to write your own SpreadSheet cell extensions, for defining custom SpreadSheet behaviors.

Listing 11.2: CursorSSCell Example

```
//
// CursorSSCell.java
//

/*
Below is a simple extension of visad.ss.FancySSCell that prints range values
to the console window whenever the cursor is being displayed.  It shouldn't
be hard to modify this code to display the range values in a JLabel or other
such GUI component.

You should be able to follow this pattern to extend FancySSCell in any way
you desire, producing any number of different custom spreadsheet cell
behaviors.
*/
```

```java
import java.awt.Frame;
import java.rmi.RemoteException;
import java.util.Vector;
import visad.*;
import visad.formula.FormulaManager;
import visad.ss.*;

public class CursorSSCell extends FancySSCell {

  public CursorSSCell(String name, FormulaManager fman, RemoteServer rs,
    boolean slave, String save, Frame parent)
    throws VisADException, RemoteException
  {
    super(name, fman, rs, slave, save, parent);

    addDisplayListener(new DisplayListener() {
      public void displayChanged(DisplayEvent e) {
        // get cursor value
        double[] scale_offset = new double[2];
        double[] dum_1 = new double[2];
        double[] dum_2 = new double[2];
        DisplayRenderer renderer = VDisplay.getDisplayRenderer();
        double[] cur = renderer.getCursor();
        Vector cursorStringVector = renderer.getCursorStringVector();
        if (cursorStringVector == null || cursorStringVector.size() == 0 ||
          cur == null || cur.length == 0 || cur[0] != cur[0])
        {
          return;
        }

        // locate x and y mappings
        ScalarMap[] maps = getMaps();
        ScalarMap map_x = null, map_y = null;
        for (int i=0; i<maps.length && (map_x==null || map_y==null); i++) {
          if (maps[i].getDisplayScalar().equals(Display.XAxis)) {
            map_x = maps[i];
          }
          else if (maps[i].getDisplayScalar().equals(Display.YAxis)) {
            map_y = maps[i];
          }
        }
        if (map_x == null || map_y == null) return;

        // get scale
        map_x.getScale(scale_offset, dum_1, dum_2);
        double value_x = (cur[0] - scale_offset[1]) / scale_offset[0];
        map_y.getScale(scale_offset, dum_1, dum_2);
        double value_y = (cur[1] - scale_offset[1]) / scale_offset[0];
        RealTuple tuple = null;
        try {
          tuple = new RealTuple(new Real[] {
            new Real((RealType) map_x.getScalar(), value_x),
            new Real((RealType) map_y.getScalar(), value_y)});
        }
        catch (VisADException exc) { exc.printStackTrace(); }
        catch (RemoteException exc) { exc.printStackTrace(); }

        // check each data object in the cell
        Data[] data = getData();
        for (int i=0; i<data.length; i++) {
```

```
          if (data[i] instanceof FlatField) {
            // get range values
            FlatField ff = (FlatField) data[i];
            double[] range_values = null;
            try {
              Data d = ff.evaluate(tuple);
              if (d instanceof Real) {
                Real r = (Real) d;
                range_values = new double[1];
                range_values[0] = r.getValue();
              }
              else if (d instanceof RealTuple) {
                RealTuple rt = (RealTuple) d;
                int dim = rt.getDimension();
                range_values = new double[dim];
                for (int j=0; j<dim; j++) {
                  Real r = (Real) rt.getComponent(j);
                  range_values[j] = r.getValue();
                }
              }
            }
            catch (VisADException exc) { exc.printStackTrace(); }
            catch (RemoteException exc) { exc.printStackTrace(); }

            // display range values somehow; e.g.:
            System.out.print("data #" + i + ": " +
              "(" + value_x + ", " + value_y + "): ");
            if (range_values == null) System.out.println("null");
            else {
              if (range_values.length == 1) {
                System.out.println(range_values[0]);
              }
              else {
                System.out.print("(" + range_values[0]);
                for (int j=1; j<range_values.length; j++) {
                  System.out.print(", " + range_values[j]);
                }
                System.out.println(")");
              }
            }
          }
        }
      }
    });
  }

  public static void main(String[] args) {
    SpreadSheet.setSSCellClass(CursorSSCell.class);
    SpreadSheet.main(args);
  }

}
```

Download code: CursorSSCell.java

11.1.3. FormulaEval

A command-line application that demonstrates the visad.formula package by evaluating simple formulas. This example should be useful for deciphering VisAD's formula package. Note, however, that the visad.formula package is somewhat obsolete now, since VisAD is integrated so well with Jython, which provides similar but much more advanced functionality.

Listing 11.3: Formula Evaluation Example

```java
//
// FormulaEval.java
//

/*
This program evaluates a simple formula using VisAD's formula package.
To run it, type "java FormulaEval 3.8 4.5 x+2*y" at the command line, where
"3.8" is a possible value for x, "4.5" is a possible value for y, and "x+2*y"
is the desired formula to evaluate.
*/

import java.rmi.RemoteException;
import visad.*;
import visad.formula.*;

public class FormulaEval {

  public static void main(String[] argv)
    throws VisADException, RemoteException
  {
    // get arguments from command line
    if (argv.length < 3) {
      System.out.println("Please enter three arguments: " +
        "two numbers and a formula.");
      System.exit(1);
    }
    double d1 = 0;
    double d2 = 0;
    try {
      d1 = Double.parseDouble(argv[0]);
      d2 = Double.parseDouble(argv[1]);
    }
    catch (NumberFormatException exc) {
      System.out.println("First two arguments must be numbers.");
      System.exit(2);
    }
    String formula = argv[2];

    // create two VisAD Data objects that store floating point values
    Real x = new Real(d1);
    Real y = new Real(d2);

    // create formula manager
    FormulaManager fman = FormulaUtil.createStandardManager();

    // register Data objects with formula manager
```

```
    fman.setThing("x", x);
    fman.setThing("y", y);

    // assign formula to new variable
    fman.assignFormula("f", formula);

    // wait for formula to finish computing, just to be safe
    fman.waitForFormula("f");

    // get value of function from formula manager
    Real f = (Real) fman.getThing("f");

    // print out results
    System.out.println("x = " + x.getValue() + ", y = " + y.getValue());
    System.out.println("f(x,y) = " + formula + " = " + f.getValue());

    // kill threads
    try { Thread.sleep(500); }
    catch (InterruptedException exc) { }
    ActionImpl.stopThreadPool();
  }

}
```

Download code: FormulaEval.java

11.1.4. IrregularRenderTest

An example of how to do volume rendering when your data is not evenly spaced. This program is very similar to LinearRenderTest, except that the domain set is an Irregular3DSet, which must be resampled to a Linear3DSet before VisAD can display the data as a volume rendering.

Listing 11.4: IrregularRenderTest Example

```
// IrregularRenderTest.java

import java.rmi.RemoteException;

import javax.swing.*;

import visad.*;
import visad.java3d.DisplayImplJ3D;

public class IrregularRenderTest {

  public static void main(String[] args)
    throws VisADException, RemoteException
  {
    // create types
    RealType x = RealType.getRealType("x");
    RealType y = RealType.getRealType("y");
    RealType z = RealType.getRealType("z");
    RealTupleType xyz = new RealTupleType(x, y, z);
```

```java
    RealType value = RealType.getRealType("value");

    // generate some irregular (random) samples
    int count = 512;
    float[][] samples = new float[3][count];
    for (int i=0; i<count; i++) for (int j=0; j<3; j++) {
      samples[j][i] = (float) (1000 * Math.random());
    }
    Irregular3DSet iset = new Irregular3DSet(xyz,
      samples, null, null, null, null, false);

    // build field
    FunctionType ftype = new FunctionType(xyz, value);
    FlatField field = new FlatField(ftype, iset);
    float[][] values = new float[1][count];
    for (int i=0; i<count; i++) {
      values[0][i] = 1500 - (Math.abs(samples[0][i] - 500) +
        Math.abs(samples[1][i] - 500) + Math.abs(samples[2][i] - 500));
    }
    field.setSamples(values, false);

    // resample field to regular grid
    int size = 32;
    count = size * size * size;
    Linear3DSet set = new Linear3DSet(xyz,
      0, 1000, size, 0, 1000, size, 0, 1000, size);
    field = (FlatField)
      field.resample(set, Data.WEIGHTED_AVERAGE, Data.NO_ERRORS);

    // create display
    DisplayImpl display = new DisplayImplJ3D("display");
    display.getGraphicsModeControl().setPointSize(5.0f);
    display.addMap(new ScalarMap(x, Display.XAxis));
    display.addMap(new ScalarMap(y, Display.YAxis));
    display.addMap(new ScalarMap(z, Display.ZAxis));
    ScalarMap color = new ScalarMap(value, Display.RGBA);
    display.addMap(color);

    // assign alpha channel
    BaseColorControl cc = (BaseColorControl) color.getControl();
    cc.setTable(tweakAlpha(cc.getTable()));

    // add data to display
    DataReferenceImpl ref = new DataReferenceImpl("ref");
    ref.setData(field);
    display.addReference(ref);

    // show display onscreen
    JFrame frame = new JFrame("Irregular rendering test");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(display.getComponent());
    frame.setBounds(200, 200, 400, 400);
    frame.show();
  }

  private static float[][] tweakAlpha(float[][] table) {
    int pow = 2;
    int len = table[3].length;
    for (int i=0; i<len; i++) {
      table[3][i] = (float) Math.pow((double) i / len, pow);
    }
```

```
    return table;
  }

}
```

Download code: IrregularRenderTest.java

11.1.5. LinearRenderTest

A simple demonstration of VisAD's volume rendering capabilities.

Listing 11.5: LinearRenderTest Example

```
// LinearRenderTest.java

import java.rmi.RemoteException;

import javax.swing.*;

import visad.*;
import visad.java3d.DisplayImplJ3D;

public class LinearRenderTest {

  public static void main(String[] args)
    throws VisADException, RemoteException
  {
    // create types
    RealType x = RealType.getRealType("x");
    RealType y = RealType.getRealType("y");
    RealType z = RealType.getRealType("z");
    RealTupleType xyz = new RealTupleType(x, y, z);
    RealType value = RealType.getRealType("value");

    // generate some regular samples
    int size = 32;
    int count = size * size * size;
    Linear3DSet set = new Linear3DSet(xyz,
      0, 1000, size, 0, 1000, size, 0, 1000, size);
    float[][] samples = set.getSamples(false);

    // build field
    FunctionType ftype = new FunctionType(xyz, value);
    FlatField field = new FlatField(ftype, set);
    float[][] values = new float[1][count];
    for (int i=0; i<count; i++) {
      values[0][i] = 1500 - (Math.abs(samples[0][i] - 500) +
        Math.abs(samples[1][i] - 500) + Math.abs(samples[2][i] - 500));
    }
    field.setSamples(values, false);

    // create display
    DisplayImpl display = new DisplayImplJ3D("display");
    display.getGraphicsModeControl().setPointSize(5.0f);
    display.addMap(new ScalarMap(x, Display.XAxis));
    display.addMap(new ScalarMap(y, Display.YAxis));
```

```java
    display.addMap(new ScalarMap(z, Display.ZAxis));
    ScalarMap color = new ScalarMap(value, Display.RGBA);
    display.addMap(color);

    // assign alpha channel
    BaseColorControl cc = (BaseColorControl) color.getControl();
    cc.setTable(tweakAlpha(cc.getTable()));

    // add data to display
    DataReferenceImpl ref = new DataReferenceImpl("ref");
    ref.setData(field);
    display.addReference(ref);

    // show display onscreen
    JFrame frame = new JFrame("Linear rendering test");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(display.getComponent());
    frame.setBounds(200, 200, 400, 400);
    frame.show();
  }

  private static float[][] tweakAlpha(float[][] table) {
    int pow = 2;
    int len = table[3].length;
    for (int i=0; i<len; i++) {
      table[3][i] = (float) Math.pow((double) i / len, pow);
    }
    return table;
  }

}
```

Download code: LinearRenderTest.java

11.1.6. MiniDataServer

An example that serves a data object on an RMI server. The object can then be viewed remotely with the VisAD SpreadSheet. This example should be useful for deciphering the VisAD SpreadSheet's RMI support.

Listing 11.6: MiniDataServer Example

```java
// MiniDataServer.java

import java.awt.event.*;
import java.net.*;
import java.rmi.*;
import javax.swing.*;
import visad.*;
import visad.data.*;
import visad.java2d.DisplayImplJ2D;

/*
This example creates a RemoteServer and loads a data object into it.
Start it up by typing:
```

```
    java MiniDataServer ServerName DataName dataFile

where ServerName is the desired name for the RMI server, DataName is
the desired name for the data reference, and dataFile is the name of
the data file to load up and serve.  Be sure you start up rmiregistry
before running MiniDataServer.

Then, load up the SpreadSheet and try:
    rmi://ip.address/ServerName/DataName
(where ip.address is your machine's IP address) and you should see
the data in the SpreadSheet cell.
*/

public class MiniDataServer {

  public static void main(String[] args) throws Exception {
    if (args.length < 3) {
      System.err.println("Please specify three command line arguments:");
      System.err.println("  - Server name (e.g., MyServer)");
      System.err.println("  - Cell name (e.g., A1)");
      System.err.println("  - Data file (e.g., mydata.nc)");
      System.exit(-1);
    }
    String server = args[0];
    String cell = args[1];
    String file = args[2];

    // load data
    System.out.println("Loading " + file + "...");
    DefaultFamily loader = new DefaultFamily("loader");
    Data data = loader.open(file);

    // set up display
    System.out.println("Setting up display...");
    ScalarMap[] maps = data.getType().guessMaps(false);
    DisplayImplJ2D display = new DisplayImplJ2D("MiniDataServer");
    for (int i=0; i<maps.length; i++) display.addMap(maps[i]);
    DataReferenceImpl ref = new DataReferenceImpl(cell);
    ref.setData(data);
    display.addReference(ref);

    // start up remote server
    System.out.println("Starting remote server...");
    RemoteServerImpl rsi = null;
    try {
      rsi = new RemoteServerImpl();
      Naming.rebind("///" + server, rsi);
    }
    catch (java.rmi.ConnectException exc) {
      System.err.println("Please run rmiregistry first.");
      System.exit(-2);
    }
    catch (MalformedURLException exc) {
      System.err.println("Error binding server; try a different name.");
      System.exit(-3);
    }
    catch (RemoteException exc) {
      System.err.println("Error binding server:");
      exc.printStackTrace();
      System.exit(-4);
    }
```

```
    rsi.addDataReference(ref);

    // set up GUI
    System.out.println("Bringing up display...");
    JFrame frame = new JFrame("Mini data server");
    JPanel pane = new JPanel();
    pane.setLayout(new BoxLayout(pane, BoxLayout.X_AXIS));
    frame.setContentPane(pane);
    pane.add(display.getComponent());
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    frame.pack();
    frame.show();
  }

}
```

Download code: MiniDataServer.java

11.1.7. RadialLine

Similar to AnchoredPoint, but uses manual picking instead of a VisAD direct manipulation renderer. This implementation allows the user to drag the line around no matter where on the line it is clicked. This method is a bit more work than AnchoredPoint, but offers more control.

Listing 11.7: RadialLine Example

```
// RadialLine.java

/*
This application demonstrates a fixed-length line that is manipulable
through manual picking (i.e., not with a direct manipulation renderer).
*/

import visad.*;
import visad.java3d.*;
import visad.util.Util;

import java.awt.event.*;
import java.rmi.RemoteException;
import java.util.Vector;

import javax.swing.*;

public class RadialLine {

  private static final float LENGTH = 5;
  private static final float END_X = 2;
  private static final float END_Y = 3;
  private static final double THRESHOLD = 0.1;

  public static void main(String[] args) throws Exception {
```

```
    // math types
    RealType x = RealType.getRealType("x");
    RealType y = RealType.getRealType("y");
    final RealTupleType xy = new RealTupleType(x, y);

    // mappings
    ScalarMap xmap = new ScalarMap(x, Display.XAxis);
    ScalarMap ymap = new ScalarMap(y, Display.YAxis);
    xmap.setRange(END_X - LENGTH, END_X + LENGTH);
    ymap.setRange(END_Y - LENGTH, END_Y + LENGTH);

    // display
    final DisplayImpl display = new DisplayImplJ3D("display",
      new TwoDDisplayRendererJ3D());
    display.disableAction();
    display.addMap(xmap);
    display.addMap(ymap);
    GraphicsModeControl gmc = display.getGraphicsModeControl();
    gmc.setScaleEnable(true);
    gmc.setPointSize(5.0f);

    // data references
    final DataReferenceImpl lineRef = new DataReferenceImpl("line");
    display.addReference(lineRef);

    // data objects
    doLine(xy, 0, 0, lineRef);

    display.enableEvent(DisplayEvent.MOUSE_DRAGGED);
    display.addDisplayListener(new DisplayListener() {
      private boolean isDragging = false;
      public void displayChanged(DisplayEvent e) {
        // verify mouse press or drag
        int id = e.getId();
        boolean press = id == DisplayEvent.MOUSE_PRESSED;
        boolean drag = id == DisplayEvent.MOUSE_DRAGGED;
        boolean release = id == DisplayEvent.MOUSE_RELEASED;
        if (!press && !drag && !release) return;

        // verify right mouse button only
        MouseEvent mouse = (MouseEvent) e.getInputEvent();
        if (!SwingUtilities.isRightMouseButton(mouse)) return;

        if (release) {
          isDragging = false;
          return; // done dragging
        }

        // get point coordinates
        int x = e.getX();
        int y = e.getY();
        double[] vals = pixelToDomain(display, x, y);

        // verify coordinates are close enough to the line
        if (press) {
          try {
            Gridded2DSet set = (Gridded2DSet) lineRef.getData();
            float[][] samps = set.getSamples(false);
            double[] ep1 = {samps[0][0], samps[1][0]};
            double[] ep2 = {samps[0][1], samps[1][1]};
            double dist = getDistance(ep1, ep2, vals, true);
```

```java
            if (dist > THRESHOLD) return; // click is too far away
            isDragging = true;
          }
          catch (VisADException exc) { exc.printStackTrace(); }
        }
        if (!isDragging) return;

        // adjust point coordinates
        float xval = (float) vals[0];
        float yval = (float) vals[1];
        float xlen = END_X - xval;
        float ylen = END_Y - yval;
        float len = (float) Math.sqrt(xlen * xlen + ylen * ylen);
        if (!Util.isApproximatelyEqual(len, LENGTH)) {
          double lamda = LENGTH / len;
          xval = (float) (END_X + lamda * (xval - END_X));
          yval = (float) (END_Y + lamda * (yval - END_Y));
        }

        // update line
        try { doLine(xy, xval, yval, lineRef); }
        catch (Exception exc) { exc.printStackTrace(); }
      }
    });

    display.enableAction();

    // show display onscreen
    JFrame frame = new JFrame("Radial line with manual picking");
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
    p.add(display.getComponent());
    frame.setContentPane(p);
    frame.setSize(400, 400);
    Util.centerWindow(frame);
    frame.show();
  }

  private static void doLine(RealTupleType rtt, float x, float y,
    DataReferenceImpl lineRef) throws VisADException, RemoteException
  {
    float[][] samples = { {x, END_X}, {y, END_Y} };
    Gridded2DSet set = new Gridded2DSet(rtt, samples, 2);
    lineRef.setData(set);
  }


  // -- Utility methods --

  /** Converts the given cursor coordinates to domain coordinates. */
  public static double[] cursorToDomain(DisplayImpl d, double[] cursor) {
    return cursorToDomain(d, null, cursor);
  }

  /** Converts the given cursor coordinates to domain coordinates. */
  public static double[] cursorToDomain(DisplayImpl d,
    RealType[] types, double[] cursor)
  {
```

```
    // locate x, y and z mappings
    Vector maps = d.getMapVector();
    int numMaps = maps.size();
    ScalarMap mapX = null, mapY = null, mapZ = null;
    for (int i=0; i<numMaps; i++) {
      if (mapX != null && mapY != null && mapZ != null) break;
      ScalarMap map = (ScalarMap) maps.elementAt(i);
      if (types == null) {
        DisplayRealType drt = map.getDisplayScalar();
        if (drt.equals(Display.XAxis)) mapX = map;
        else if (drt.equals(Display.YAxis)) mapY = map;
        else if (drt.equals(Display.ZAxis)) mapZ = map;
      }
      else {
        ScalarType st = map.getScalar();
        if (st.equals(types[0])) mapX = map;
        if (st.equals(types[1])) mapY = map;
        if (st.equals(types[2])) mapZ = map;
      }
    }

    // adjust for scale
    double[] scaleOffset = new double[2];
    double[] dummy = new double[2];
    double[] values = new double[3];
    if (mapX == null) values[0] = Double.NaN;
    else {
      mapX.getScale(scaleOffset, dummy, dummy);
      values[0] = (cursor[0] - scaleOffset[1]) / scaleOffset[0];
    }
    if (mapY == null) values[1] = Double.NaN;
    else {
      mapY.getScale(scaleOffset, dummy, dummy);
      values[1] = (cursor[1] - scaleOffset[1]) / scaleOffset[0];
    }
    if (mapZ == null) values[2] = Double.NaN;
    else {
      mapZ.getScale(scaleOffset, dummy, dummy);
      values[2] = (cursor[2] - scaleOffset[1]) / scaleOffset[0];
    }

    return values;
  }

  /** Converts the given pixel coordinates to cursor coordinates. */
  public static double[] pixelToCursor(DisplayImpl d, int x, int y) {
    MouseBehavior mb = d.getDisplayRenderer().getMouseBehavior();
    VisADRay ray = mb.findRay(x, y);
    return ray.position;
  }

  /** Converts the given pixel coordinates to domain coordinates. */
  public static double[] pixelToDomain(DisplayImpl d, int x, int y) {
    return cursorToDomain(d, pixelToCursor(d, x, y));
  }

  /**
   * Computes the minimum distance between the point v and the line a-b.
   *
   * @param a Coordinates of the line's first endpoint
   * @param b Coordinates of the line's second endpoint
```

```java
   * @param v Coordinates of the standalone endpoint
   * @param segment Whether distance computation should be
   *                constrained to the given line segment
   */
  public static double getDistance(double[] a, double[] b, double[] v,
    boolean segment)
  {
    int len = a.length;

    // vectors
    double[] ab = new double[len];
    double[] va = new double[len];
    for (int i=0; i<len; i++) {
      ab[i] = a[i] - b[i];
      va[i] = v[i] - a[i];
    }

    // project v onto (a, b)
    double numer = 0;
    double denom = 0;
    for (int i=0; i<len; i++) {
      numer += va[i] * ab[i];
      denom += ab[i] * ab[i];
    }
    double c = numer / denom;
    double[] p = new double[len];
    for (int i=0; i<len; i++) p[i] = c * ab[i] + a[i];

    // determine which point (a, b or p) to use in distance computation
    int flag = 0;
    if (segment) {
      for (int i=0; i<len; i++) {
        if (p[i] > a[i] && p[i] > b[i]) flag = a[i] > b[i] ? 1 : 2;
        else if (p[i] < a[i] && p[i] < b[i]) flag = a[i] < b[i] ? 1 : 2;
        else continue;
        break;
      }
    }

    double sum = 0;
    for (int i=0; i<len; i++) {
      double q;
      if (flag == 0) q = p[i] - v[i]; // use p
      else if (flag == 1) q = a[i] - v[i]; // use a
      else q = b[i] - v[i]; // flag == 2, use b
      sum += q * q;
    }

    return Math.sqrt(sum);
  }

}
```

Download code: RadialLine.java

11.1.8. RiversColor

This program is just like `visad/examples/Rivers.java`, except that the line segments are different colors instead of plain white. (This scenario requires a more complex MathType.)

Listing 11.8: RiversColor Example

```java
// RiversColor.java

/*
This application demonstrates using UnionSets and FieldImpls
to create a collection of colored line segments.
*/

import visad.*;
import visad.java2d.*;

import java.awt.BorderLayout;
import java.awt.event.*;
import java.rmi.RemoteException;

import javax.swing.*;

/** RiversColor is based on visad/examples/Rivers.java. */
public class RiversColor {

  public static void main(String args[])
    throws VisADException, RemoteException
  {
    RealTupleType earth =
      new RealTupleType(RealType.Latitude, RealType.Longitude);

    // construct straight south flowing river1
    float[][] points1 = {{3.0f, 2.0f, 1.0f, 0.0f},
                         {0.0f, 0.0f, 0.0f, 0.0f}};
    Gridded2DSet river1 = new Gridded2DSet(earth, points1, 4);

    // construct east feeder river2
    float[][] points2 = {{3.0f, 2.0f, 1.0f},
                         {2.0f, 1.0f, 0.0f}};
    Gridded2DSet river2 = new Gridded2DSet(earth, points2, 3);

    // construct west feeder river3
    float[][] points3 = {{4.0f, 3.0f, 2.0f},
                         {-2.0f, -1.0f, 0.0f}};
    Gridded2DSet river3 = new Gridded2DSet(earth, points3, 3);

    // construct river system set
    Gridded2DSet[] riverSystem = {river1, river2, river3};
    UnionSet riversSet = new UnionSet(earth, riverSystem);

    // construct river field for coloring rivers
    RealType red = RealType.getRealType("red");
    RealType green = RealType.getRealType("green");
    RealType blue = RealType.getRealType("blue");
    RealTupleType rgb = new RealTupleType(red, green, blue);
    FunctionType ftype = new FunctionType(earth, rgb);
```

```java
    FlatField riversField = new FlatField(ftype, riversSet);
    float[][] samples = new float[][] { // 4+3+3=10 sample points total
      {1, 1, 1, 1, 1, 1, 1, 0, 0, 0}, // red
      {1, 1, 1, 1, 0, 0, 0, 1, 1, 1}, // green
      {0, 0, 0, 0, 1, 1, 1, 1, 1, 1}  // blue
    };
    riversField.setSamples(samples, false);

    // create a DataReference for river system
    final DataReference riversRef = new DataReferenceImpl("rivers");
    riversRef.setData(riversField);

    // create a Display using Java2D
    DisplayImpl display = new DisplayImplJ2D("image display");

    // map earth coordinates to display coordinates
    display.addMap(new ScalarMap(RealType.Longitude, Display.XAxis));
    display.addMap(new ScalarMap(RealType.Latitude, Display.YAxis));

    // map color components to color space
    ScalarMap redMap = new ScalarMap(red, Display.Red);
    ScalarMap greenMap = new ScalarMap(green, Display.Green);
    ScalarMap blueMap = new ScalarMap(blue, Display.Blue);
    redMap.setRange(0, 1);
    greenMap.setRange(0, 1);
    blueMap.setRange(0, 1);
    display.addMap(redMap);
    display.addMap(greenMap);
    display.addMap(blueMap);

    // link the Display to riversRef
    display.addReference(riversRef);
    riversRef.setData(riversField);

    // create JFrame (i.e., a window) for display and slider
    JFrame frame = new JFrame("RiversColor 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 BorderLayout());
    frame.setContentPane(panel);

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

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

}
```

Download code: RiversColor.java

11.1.9. SurfaceAnimation

Illustrates a curved 2D surface embedded in a 3D display, whose color values animate over time, using MathType `(time -> ((x, y) -> (z, value)))`

Listing 11.9: SurfaceAnimation Example

```java
// SurfaceAnimation.java

import java.awt.event.*;
import java.rmi.RemoteException;
import javax.swing.JFrame;

import visad.*;
import visad.java3d.*;

/** Constructs a surface whose colors animate over time. */
public class SurfaceAnimation {

  public static void main(String[] args)
    throws VisADException, RemoteException
  {
    int numTimePoints = 10;
    int xLen = 32, yLen = 32;
    int len = xLen * yLen;

    // construct data types
    RealType tType = RealType.getRealType("time");
    RealType xType = RealType.getRealType("x");
    RealType yType = RealType.getRealType("y");
    RealType zType = RealType.getRealType("z");
    RealType vType = RealType.getRealType("value");
    RealTupleType xy = new RealTupleType(xType, yType);
    RealTupleType zv = new RealTupleType(zType, vType);
    FunctionType surfaceType = new FunctionType(xy, zv);
    FunctionType animType = new FunctionType(tType, surfaceType);
    Integer2DSet surfaceSet = new Integer2DSet(xy, xLen, yLen);
    Integer1DSet animSet = new Integer1DSet(tType, numTimePoints);

    // generate surface values
    float[] surface = new float[len];
    for (int y=0; y<yLen; y++) {
      for (int x=0; x<xLen; x++) {
        // a nice, rounded surface
        float xn = (float) xLen / 2 - x;
        float yn = (float) yLen / 2 - y;
        surface[y * xLen + x] = xn * xn + yn * yn;
      }
    }

    // generate color values
    FieldImpl data = new FieldImpl(animType, animSet);
    for (int t=0; t<numTimePoints; t++) {
      FlatField field = new FlatField(surfaceType, surfaceSet);
      float[] values = new float[len];
      // a linear progression of color values
      for (int i=0; i<len; i++) values[i] = len * t + i;
      float[][] samples = {surface, values};
```

```
      field.setSamples(samples, false);
      data.setSample(t, field);
    }

    // create display
    DisplayImpl display = new DisplayImplJ3D("display");
    DataReferenceImpl ref = new DataReferenceImpl("ref");
    ref.setData(data);
    display.addMap(new ScalarMap(tType, Display.Animation));
    display.addMap(new ScalarMap(xType, Display.XAxis));
    display.addMap(new ScalarMap(yType, Display.YAxis));
    display.addMap(new ScalarMap(zType, Display.ZAxis));
    display.addMap(new ScalarMap(vType, Display.RGB));
    display.addReference(ref);

    // start animation
    AnimationControl animControl = (AnimationControl)
      display.getControl(AnimationControl.class);
    animControl.setOn(true);

    // show display onscreen
    JFrame frame = new JFrame("Surface animation");
    frame.getContentPane().add(display.getComponent());
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    frame.pack();
    frame.show();
  }

}
```

Download code: SurfaceAnimation.java

11.1.10. WhiteSSCell

A VisAD SpreadSheet cell extension with white cell backgrounds instead of black ones. This example should be useful for learning how to write your own SpreadSheet cell extensions, for defining custom SpreadSheet behaviors.

Listing 11.10: WhiteSSCell Example

```
//
// WhiteSSCell.java
//

/*
You can get the DisplayImpl from a BasicSSCell (a spreadsheet cell)
by calling BasicSSCell.getDisplay(). However, whenever a BasicSSCell
switches dimensions, a new DisplayImpl must be initialized. So, the
best way to ensure the spreadsheet always has cells with white
backgrounds is to make a simple extension.

Below is an extension of visad.ss.FancySSCell that does the trick.
*/
```

```java
import java.io.*;
import java.awt.Frame;
import java.rmi.*;
import visad.*;
import visad.formula.*;
import visad.ss.*;

/** An extension of visad.ss.FancySSCell for cells with white backgrounds. */
public class WhiteSSCell extends FancySSCell {

  /** Extended from visad.ss.FancySSCell. */
  public WhiteSSCell(String name, FormulaManager fman, RemoteServer rs,
    boolean slave, String save, Frame parent) throws VisADException,
    RemoteException
  {
    super(name, fman, rs, slave, save, parent);
  }

  /**
   * Extends visad.ss.FancySSCell.constructDisplay so that whenever a
   * new DisplayImpl is constructed, its background is set to white.
   */
  public synchronized boolean constructDisplay() {
    boolean success = super.constructDisplay();
    if (success) {
      DisplayRenderer dRenderer = VDisplay.getDisplayRenderer();
      try {
        // set background color
        dRenderer.setBackgroundColor(1.0f, 1.0f, 1.0f); // white
        dRenderer.setBoxColor(0.0f, 0.0f, 0.0f); // black
      }
      catch (VisADException exc) { exc.printStackTrace(); }
      catch (RemoteException exc) { exc.printStackTrace(); }
    }
    return success;
  }

  public static void main(String[] args) {
    SpreadSheet.setSSCellClass(WhiteSSCell.class);
    SpreadSheet.main(args);
  }

}
```

Download code: WhiteSSCell.java

Part IV.

Other helpful stuff