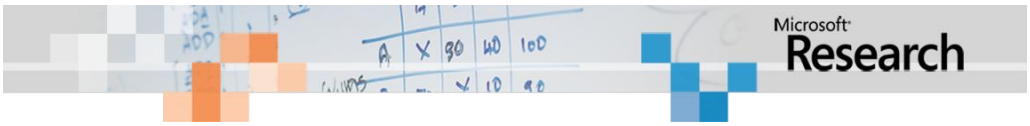


Introduction to Scientific DataSet

A managed library and viewer for scientific data

Version 1.2 – June 4, 2010

Abstract

Scientific DataSet is a managed library for reading, writing, and sharing array-oriented scientific data such as time series, matrices, satellite or medical imagery, and multidimensional numerical grids.

This guide is for C# programmers who want to use Scientific DataSet in their scientific computational programs. The introduction briefly describes the Scientific DataSet capabilities and data model and then presents a walkthrough that shows you how to:

- Read and write datasets in common formats.
- Switch from one type of data file to another without additional programming.
- Include rich descriptive metadata in your dataset to create self-descriptive data packages that can easily be shared with other programs.
- Use the DataSet Viewer to visualize data.

Note:

- For more information about Scientific DataSet and related projects, see "Resources" at the end of this document.
- For Scientific DataSet software, see the Microsoft Research Web site at http://research.microsoft.com/groups/science/software.aspx.
- To provide feedback about Scientific DataSet, send an e-mail message with your comments to mssds@microsoft.com.

Computational Science Laboratory
Microsoft Research Cambridge

Contents

The Challenge: A Common Data Model

Programmers who support scientific research often must create applications that support one or more specific data formats. Although scientific data—time series, satellite and medical imagery, and the like—are typically stored in arrays, each dataset is different. Scientific program code depends heavily on data format, and transferring data from one component to another can be difficult. Such problems hinder collaboration in the scientific community.

A single data model that supports multiple specific data formats makes it possible for programs to store and retrieve data without concern about formatting, thereby allowing the programs' users to focus on data analysis and computation rather than mundane input/output formats. The Unidata Common Data Model (CDM) implements such a data model for Java programs. However, a similar model has not been available for C#, managed C++, and Visual Basic® applications.

Scientific DataSet supports a data model that enables .NET Framework programs to benefit from an abstract view of data storage. By separating dataset access from the real work of scientific computation and visualization, Scientific DataSet makes it easier for researchers to collaborate and share data, and reduces the need for specialized programming for custom data formats. The Scientific DataSet data model builds upon the proven foundation of Unidata CDM and enhances it to provide greater interoperability and more robust data access.

Scientific DataSet was created by the Computational Science Laboratory at Microsoft Research in Cambridge, England, along with other tools for applying computational science principles in natural science research.

Introducing Scientific DataSet

Scientific DataSet is a managed library for reading, writing, and sharing array-oriented scientific data such as time series, matrices, satellite or medical imagery, and multidimensional numerical grids.

You can use Scientific DataSet with your scientific computational program so that:

- Your program is more interoperable, because Scientific DataSet can import and export data in different formats.
- Your program is more scalable, because Scientific DataSet can seamlessly switch from the human-readable text files that you might use in small-scale experiments and debugging to the high-performance binary data formats that might be used in production software.

Scientific DataSet includes an extensive class library for manipulating datasets in several formats. The class library can be used in any .NET language such as C#, Managed C++, or Visual Basic.

About this Document

This document is for C# programmers who want to start using Scientific DataSet in their scientific computational programs. It introduces methods for reading and writing datasets and shows how to use the DataSet Viewer to visualize data.

To take advantage of the capabilities described in this document, you should be familiar with:

- C# namespaces and classes.
- Microsoft .NET Framework.
- Windows® Presentation Foundation (WPF) applications.

Extensive programming experience is not required.

About Scientific DataSet

Scientific DataSet provides a rich set of features, including:

- Built-in support for several common data formats, such as comma-separated values (CSV), network common data form (NetCDF), and hierarchical data format (HDF5).

  You can also extend Scientific DataSet to support additional formats.
- A visualization tool that can run as a stand-alone utility or as a component of your program.
- The ability to create self-descriptive data packages by including rich metadata in your datasets.
- The ability to perform consistency checks and transactional updates.
- The ability to scale up from simple text files to multi-terabyte Windows Azure™ archives.

Data as Arrays. The Scientific DataSet library is optimized to handle data in the form of arrays, such as time series and tables, vectors and matrices, or multidimensional grids. Scientific DataSet bundles several related arrays and associated metadata in a single self-descriptive package, and it enforces certain constraints on the shapes of arrays to ensure data consistency.

Extensible, Loadable Data Providers. Scientific DataSet includes an extensible set of dynamically loadable data providers, so you can choose from different storage formats and different data access mechanisms. For example, different runs of the same computational program can read or write data differently by using text files in CSV format, binary NetCDF files, or other file format or communication mechanisms.

DataSet Viewer. DataSet Viewer can display the contents of your dataset in several visualizations. You can use DataSet Viewer as a stand-alone application or as a component of your own scientific program.

A key goal for Scientific DataSet is to enable concurrent access to data from multiple scientific applications in a distributed computing environment. As Microsoft Research continues to develop Scientific DataSet and related tools, your program can become

part of a sophisticated concurrent data flow system in which researchers collaborate to solve larger, more complicated problems.

Scientific DataSet Architecture and Data Model

The Scientific DataSet library is designed to work with your existing scientific analysis programs to read and write array-based datasets. The library includes data providers for the CSV and NetCDF formats, and you can extend it to support additional formats.

The Scientific DataSet library can read and write data in various formats and then supply that data to your programs, your data-fitting models, and to the DataSet Viewer for analysis and visualization, as Figure 1 shows.

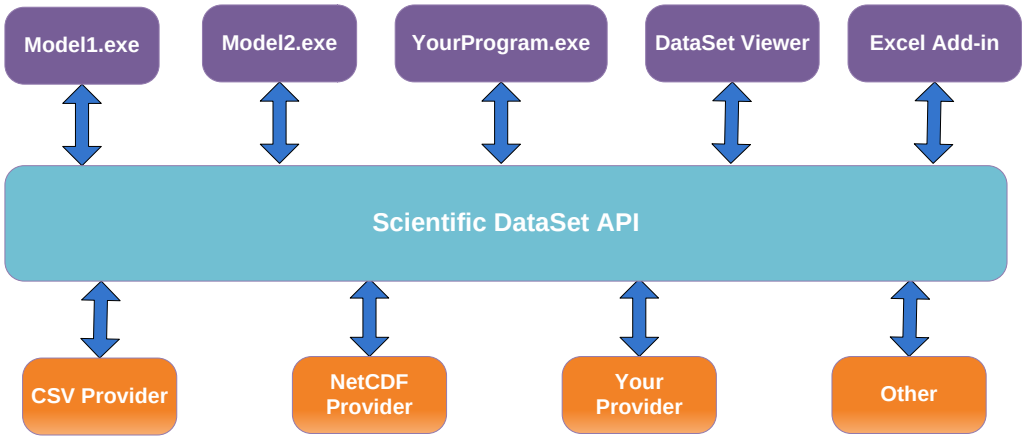


Figure 1. Scientific DataSet architecture

The Scientific DataSet application programming interface (API) supports the creation, access, and sharing of multidimensional array-oriented data. The dataset is self-describing: you can add metadata to identify the arrays, dimensions, units—or any other important information you want to archive or share. Figure 2 shows an example of the types and range of data and metadata that the Scientific DataSet API can handle.

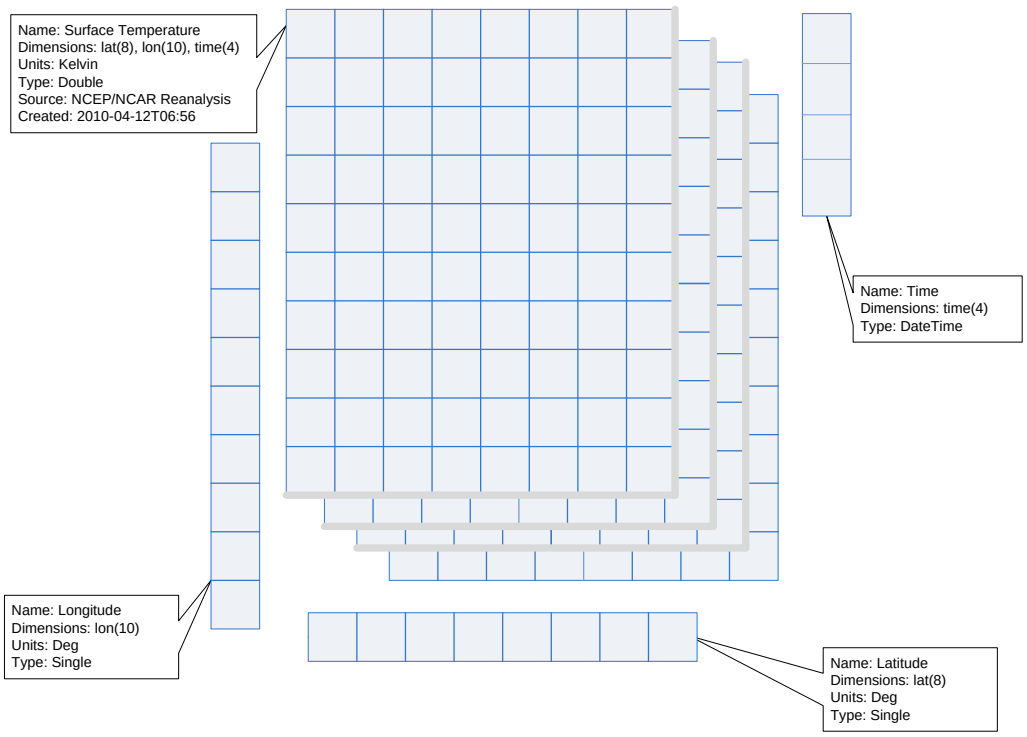


Figure 2. Sample Scientific DataSet

The Scientific DataSet library makes it easy for a program to append new data to a dataset, so you can add computed information to an array or extend a dataset with new types of information as additional measurement technologies become available. Such changes do not affect the ability of existing programs to read and write the dataset, so reprogramming is not required when a dataset changes.

For details about the data model and the object model, see the Scientific DataSet Reference documentation, which appears on your Windows Start menu after you install Scientific DataSet.

Installing the Scientific DataSet Package

The Scientific DataSet package is available for download from the Microsoft Research Web site, as listed in “Resources” at the end of this document.

Prerequisites

Scientific DataSet requires a computer that is running Windows XP or a later release, plus the software in the following list.

Software	Required for …
Microsoft .NET Framework 3.5 Service Pack 1 (SP1)	All Scientific DataSet applications
Microsoft Visual C#® 2008 Express Edition —OR— Any edition of Microsoft Visual Studio 2008 or later	Windows Presentation Foundation (WPF) applications that use DataSetViewerControl

For links to these software packages, see “Resources” at the end of this document.

Installation

To install the Scientific DataSet library, run the .MSI package provided on the Scientific DataSet Project Web site.

By default, the package installs Scientific DataSet in the C:\Program Files\Microsoft Research\Scientific DataSet 1.2 directory. The installation includes the following items:

- DataSet Viewer.exe application
- Sds.exe command-line utility
- Sds.h include file, with C++ class templates that simplify Scientific DataSet programming using managed C++
- DataSet Editor add-in installer for Microsoft Office Excel® 2007 and 2010
- Help file that describes the complete Scientific DataSet API

The installation package makes the following additional changes to your computer:

- Library assemblies are:
  - Placed in C:\Program Files\Reference Assemblies\ Microsoft Research\Scientific DataSet 1.2 directory so that they appear in Add Reference dialog box in Microsoft Visual Studio®.
  - Installed into the Global Assembly Cache (GAC).
- Four data providers are registered in the computer's Machine.config configuration file:

  CSV file format
  NetCDF file format
  In-memory storage
  Windows Communication Foundation (WCF)

After installation is complete, these providers are available to all programs that run on your computer.

A Walkthrough: Using Scientific DataSet in Your Programs

To introduce you to the Scientific DataSet library and tools, the following sections lead you on a walkthrough tour of the following Scientific DataSet features and capabilities:

Exercise 1: Add a Column to a CSV File
Exercise 2: Use the DataSet Viewer in Your Program
Exercise 3: Express Relationships between Variables as Shared Dimensions
Exercise 4: Store Descriptive Metadata for Variables and Datasets
Exercise 5: Perform Transactional Updates
Exercise 6: Use the NetCDF Provider with Large Datasets

Most methods used in this introduction are defined in the DataSetExtensions class from the Microsoft.Research.Science.Data.Imperative assembly. They are part of the Scientific DataSet imperative API.

Exercise 1: Add a Column to a CSV File

Suppose we need to process the following text file, which contains the result of an imaginary experiment.

```
X,Observation
17.84,1.628E-05
19.87,2.023E-05
22.22,2.060E-05
24.08,2.263E-05
25.98,2.333E-05
28.14,2.679E-05
29.8,2.771E-05
32.27,2.793E-05
34.25,3.079E-05
35.85,3.247E-05
```

Note: Unless otherwise noted, all the exercises in this document use this file as input, referred to as Tutorial.csv. We recommend that you save the original Tutorial.csv file so that you can start each exercise with a clean copy in the directory from which you run the exercises. If you run the program within the Visual C# or Visual Studio development environment, the file must be in the project's bin\debug or bin\release directory.

The Tutorial.csv text file is an example of a file in CSV format, which is a popular format for relatively small datasets. Such files can be read or written by many programs, including Microsoft Excel. In many programming languages, it is difficult to use standard file input and output functions to process CSV files. The standard functions read a file line by line, whereas in a CSV file the data is logically arranged in columns. Scientific DataSet can help in this situation, because it treats a dataset as a set of named variables.

From the point of view of Scientific DataSet, the Tutorial.csv file is a dataset that has two numeric variables named X and Observation, respectively. By using Scientific DataSet, your program can implement just one line of code to read a column of data.

The example console program in Listing 1 opens the data file Tutorial.csv and then proceeds as follows:

- Reads two columns of the data into arrays (lines 12–14).
- Computes coefficients of a linear model that approximates observations (lines 16–26).
- Evaluates predicted model values (lines 27–28).
- Adds those values to the dataset as a third column named Model (lines 30–31).

Listing 1. Adding a column to a CSV file

```
1  using System.Text;
2  using sds = Microsoft.Research.Science.Data;
3  using Microsoft.Research.Science.Data.Imperative;
4
5  namespace Tutorial1
6  {
7      class Program
8      {
9          static void Main(string[] args)
```

```
10          {
11              // read input data
12              var dataset = sds.DataSet.Open("Tutorial.csv");
13              var x = dataset.GetData<double[]>("X");
14              var y = dataset.GetData<double[]>("Observation");
15              // compute model parameters
16              var xm = x.Sum() / x.Length;
17              var ym = y.Sum() / y.Length;
18              double xy = 0;
19              for (int i = 0; i < x.Length; i++)
20                  xy += (x[i] - xm) * (y[i] - ym);
21              double xx = 0;
22              for (int i = 0; i < x.Length; i++)
23                  xx += (x[i] - xm) * (x[i] - xm);
24              var a = xy / xx;
25              var b = ym - a * xm;
26              var model = new double[x.Length];
27              for (int i = 0; i < x.Length; i++)
28                  model[i] =  a * x[i] + b;
29              // write output data
30              dataset.Add<double[]>("Model");
31              dataset.PutData<double[]>("Model", model);
32          }
33      }
34  }
```

The program in Listing 1 uses methods of the DataSetExtensions class, which is part of the Scientific DataSet Imperative API. Therefore:

- The example project must reference the following two assemblies:

  Microsoft.Research.Science.Data
  Microsoft.Research.Science.Data.Imperative

- The source code must include the **using** statement as shown on line 3:

```
using Microsoft.Research.Science.Data.Imperative;
```

Supplying a Variable Name and Type of Data

Line 13 in Listing 1 reads the entire column headed X by calling the **GetData** method. In this example project, we call **GetData** as follows:

GetData <*type*> (*variablename*)

where:

- *Type* specifies the expected type of data.
- *Variablename* is a string that specifies the name of the variable.

When you call Scientific DataSet methods in strongly-typed languages such as C#, the Scientific DataSet library does not coerce data types. The data type in a dataset and the type of data that you specify as a *type* parameter to the **GetData** method must match exactly.

In its simple form, a CSV file does not have explicit typing of data. In that case, Scientific DataSet uses the following heuristics:

- If all values in a column can be interpreted as numbers, true/false, or date/time values, then the column takes the type **Double**, **Boolean**, or **DateTime**, correspondingly.
- Otherwise, the column has type **String**.
- Metadata or the **inferInts=true** provider parameter can change this default behavior, as described in "Exercise 2: Use the DataSet Viewer Control in Your Program" later in this paper.

Line 31 of Listing 1 calls the **PutData** method to store the computed Model value in the dataset. However, **PutData** stores data in existing variables only. Therefore, we must first create a variable to store the computed values.

Creating Variables by Calling the Add Method

If the dataset does not contain a variable for the model data, you must call the **Add** method (line 30) to create one. In its simplest form, **Add** takes the following parameters:

Add <*type*> (*variablename*)

where:

- *Type* specifies the type and rank of the data in the variable. The parameter can be:
  - A simple scalar type. Scientific DataSet supports all standard integers, floating point numbers, **Boolean**, **DateTime**, and **String**.
  - A one-dimensional array of that type or an array of higher rank to store vectors, matrices, grids, and other multi-dimensional data.
- *Variablename* can be any string, although we strongly recommend that you follow general rules for program identifiers: start with a letter followed by letters, digits, and underscore symbols.

  Note: Variable names are for convenience only. A variable can have no name at all, or several variables in a dataset can have the same name. However, handling datasets that contain duplicate variable names is rather inconvenient.

You should be careful when using the **Add** method. For example, if we run the program in Listing 1 a second time on the output Tutorial.csv file from the first run of the program, the **Add** method will create a fourth column of Model data and the program will fail at **PutData**, because **PutData** cannot uniquely identify which Model variable to output data to. For this reason, it is better to put the **Add** method in a conditional clause.

To use the Add method in a conditional clause

- Use a statement such as the following at line 30:

```
if (!dataset.Any(v => v.Name == "Model"))
    dataset.Add<double[]>("Model");
```

Now the model data will always be output in the same column, as shown in the following example:

```
X,Observation,Model
17.84,1.628E-05,1.72831547908291E-05
19.87,2.023E-05,1.89603556368157E-05
22.22,2.06E-05,2.09019428230564E-05
24.08,2.263E-05,2.2438688425783E-05
25.98,2.333E-05,2.40084823210414E-05
28.14,2.679E-05,2.57930901177562E-05
29.8,2.771E-05,2.71645942578241E-05
32.27,2.793E-05,2.920532632166E-05
34.25,3.079E-05,3.08412168019819E-05
35.85,3.247E-05,3.21631485032522E-05
```

Using Variable IDs Instead of Variable Names

To avoid any possible ambiguity, you can use variable IDs instead of variable names. The variable ID is an integer that uniquely identifies a variable within a dataset. Variable IDs are valid only until the dataset is disposed.

Scientific DataSet does not store variable IDs; instead, it assigns them to variables at the time your program opens a dataset. In general, referencing a variable by its ID is more reliable and efficient than referencing it by name.

The **Add** method returns the variable it creates. In the program shown in Listing 1, we ignore that fact.

To use the variable ID in the example project

- Replace lines 30 and 31 in Listing 1 with the following:

```
int varid = dataset.Add<double[]>("Model").ID;
dataset.PutData<double[]>(varid, model);
```

Now, if we run the program multiple times, it will successfully create multiple identical columns.

To use variable IDs with a conditional clause

- Revise the program in Listing 1 as follows to create a file with only three columns:

```
int varid = dataset.Any(v => v.Name == "Model") ?
    dataset["Model"].ID :
    dataset.Add<double[]>("Model").ID;
dataset.PutData<double[]>(varid, model);
```

Reading a Modified DataSet File

Datasets typically grow and become more complicated over time as research continues. Existing C or C++ programs typically require modification to adapt to the changed dataset. With Scientific DataSet, however, no such modifications are required.

Even though you have added a column to the dataset, you can successfully run the program that appears in Listing 1 against an updated dataset that contains an additional column.

For example, assume that you have updated the dataset in Tutorial.csv to contain an additional observation parameter named StdError, as follows:

```
X,Observation,StdError
17.84,1.628E-05,1.0E-07
19.87,2.023E-05,1.0E-07
22.22,2.06E-05,2.0E-07
24.08,2.263E-05,2.0E-07
25.98,2.333E-05,2.0E-07
28.14,2.679E-05,2.0E-07
29.8,2.771E-05,2.0E-07
32.27,2.793E-05,3.0E-07
34.25,3.079E-05,3.0E-07
35.85,3.247E-05,3.0E-07
```

If you run the program shown earlier in Listing 1 on this modified dataset, the output dataset contains the following:

```
X,Observation,StdError,Model
17.84,1.628E-05,1.0E-07,1.72831547908291E-05
19.87,2.023E-05,1.0E-07,1.89603556368157E-05
22.22,2.06E-05,2.0E-07,2.09019428230564E-05
24.08,2.263E-05,2.0E-07,2.2438688425783E-05
25.98,2.333E-05,2.0E-07,2.40084823210414E-05
28.14,2.679E-05,2.0E-07,2.57930901177562E-05
29.8,2.771E-05,2.0E-07,2.71645942578241E-05
32.27,2.793E-05,3.0E-07,2.920532632166E-05
34.25,3.079E-05,3.0E-07,3.08412168019819E-05
35.85,3.247E-05,3.0E-07,3.21631485032522E-05
```

Exercise 2: Use the DataSet Viewer in Your Program

When you install Scientific DataSet, the DataSet Viewer application is added to your computer. The DataSet Viewer can display the contents of a dataset by using several visualizations:

- A table of values
- A line/markers chart
- A color map
- A contour line plot

You can reuse DataSet Viewer functionality in your own programs. In this exercise, you create a WPF application by using Microsoft Visual C#® 2008 Express Edition, which is available for free from the Microsoft Web site, as listed in “Resources” at the end of this paper. You can also use any edition of Visual C++ 2008 or later.

Note: If you are unfamiliar with Visual C# 2008 Express Edition, see the Visual C# Developer Center on MSDN®, which is listed in “Resources.”

To reuse the components of DataSet Viewer in your own program

1. Run Visual C# 2008 Express Edition, and create a new project that uses the WPF Application project template.
2. Add references to assemblies and libraries, as follows:
   - On the **Project** menu, click **Add Reference**.
   - On the **.NET** tab, include the following references:

Microsoft.Research.Science.Data
Microsoft.Research.Science.Data.Imperative

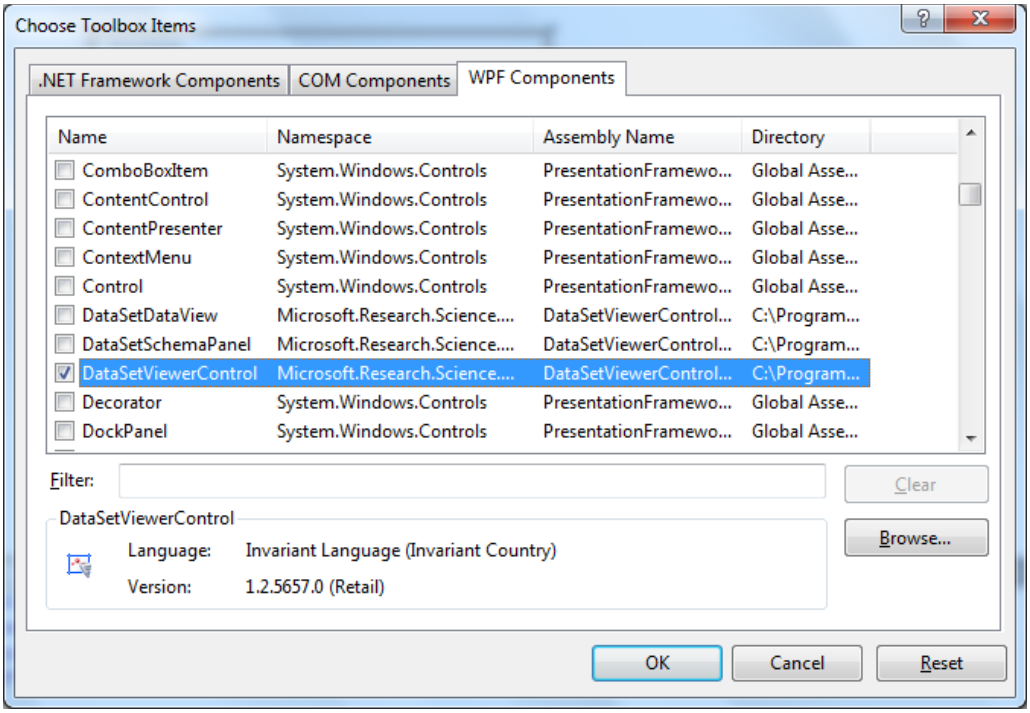
- On the **Browse** tab, navigate to the Scientific DataSet installation folder. Add references to the following dynamic-link libraries (DLLs):

  DataSetViewerControls.dll
  DataSetViewerCore.dll

- Click **OK.**

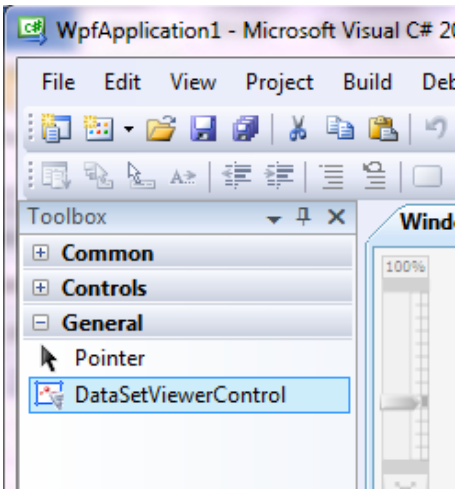
3. Add the DataSetViewerControl to the toolbox, as follows:

   - Click the Toolbox icon and then right-click an empty space in the Toolbox window.
   - On the context-sensitive menu that appears, click **Choose Items**.
   - In the **Choose Toolbox Items** dialog box, go to the **WPF Components** tab and click **Browse**.
   - Navigate to the Scientific DataSet 1.2 installation directory, click DataSetViewerControls.dll, and then click **Open**.
   - Finally, in the **ChooseToolboxItems** box, click DataSetViewerControl, as the following figure shows, and click **OK.**



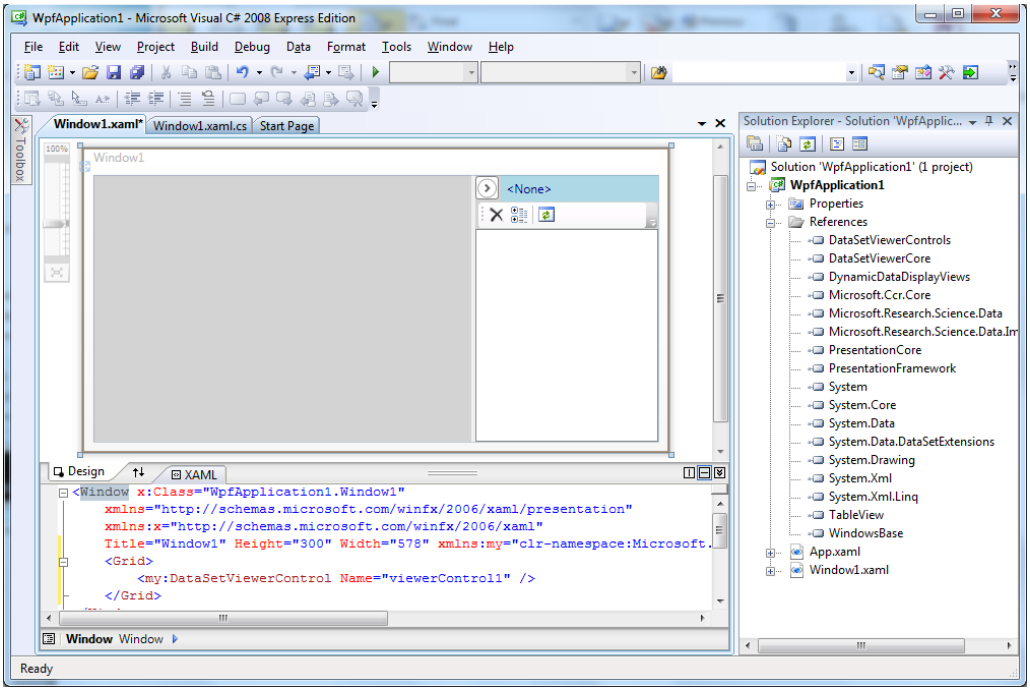
Adding DataSetViewerControl to the toolbox

DataSetViewerControl should now appear on the **Toolbox** menu, as in the following figure.



DataSetViewerControl on the Toolbox menu

4. Double-click **DataSetViewerControl** to add it to the main application window.
5. Resize the control so that it fills the entire window, as shown in the following figure.



Resizing DataSetViewerControl

6. To create the **Window_Loaded** event handler, double-click the Window1 title bar in the Design tab to display the event handler template, and copy the computation code from Listing 2 to the event handler.
7. Add the following **using** statements:

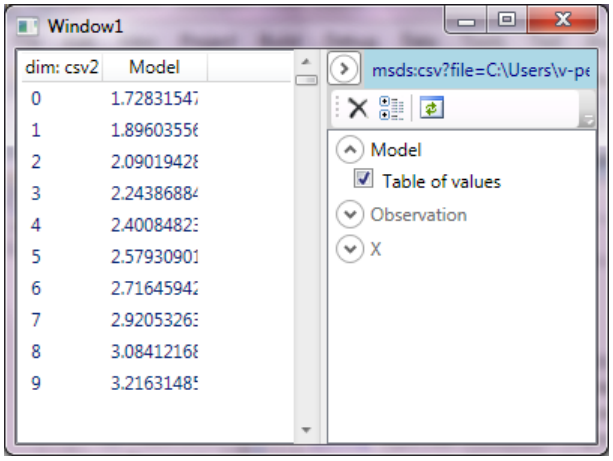
```
using sds = Microsoft.Research.Science.Data;
using Microsoft.Research.Science.Data.Imperative;
```

Listing 2. Event handler of a WPF program that plots data using the DataSetViewerControl

```
1  private void Window_Loaded(object sender, RoutedEventArgs e)
2  {
3      // read input data
4      var dataset = sds.DataSet.Open("Tutorial.csv");
5      if (!dataset.Any(var => var.Name == "Model"))
6      {
7          var x = dataset.GetData<double[]>("X");
8          var y = dataset.GetData<double[]>("Observation");
9          // compute model parameters
10         var xm = x.Sum() / x.Length;
11         var ym = y.Sum() / y.Length;
12         double xy = 0;
13         for (int i = 0; i < x.Length; i++)
14             xy += (x[i] - xm) * (y[i] - ym);
15         double xx = 0;
16         for (int i = 0; i < x.Length; i++)
17             xx += (x[i] - xm) * (x[i] - xm);
18         var a = xy / xx;
19         var b = ym - a * xm;
20         var model = new double[x.Length];
21         for (int i = 0; i < x.Length; i++)
22             model[i] =  a * x[i] + b;
23         // write output data
24         var varid = dataset.Add<double[]>("Model").ID;
25         dataset.PutData<double[]>(varid, model);
26     }
27     viewerControl1.DataSet = dataset;
28 }
```

Important: Before you run the program, ensure that the project's bin\release or bin\debug directory contains a clean copy of the original data file named Tutorial.csv.

8. The last line of the **Window_Loaded** method (line 27) passes the dataset to the DataSetViewerControl for processing. When you run the program in Listing 2, you see the following window:



Application window with DataSetViewerControl

The left pane displays the current visualization—in this case, a table of variable values. The right pane lists all the variables in the dataset. Under each variable, the

DataSet Viewer shows visualizations that are compatible with current visualization in the left pane. No other visualizations are compatible with the table of Model variable values, so all other variables in the left pane are grayed out.

To explore the full list of available options in DataSet Viewer

- Uncheck the **Table of values** visualization and click **Model**, **Observation**, and **X**.

Figure 3 shows the resulting list.

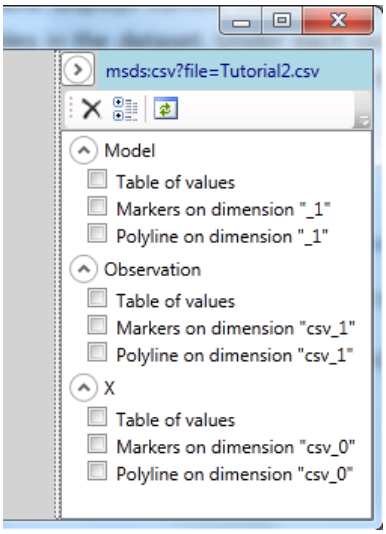


Figure 3. Visualizations for three unrelated columns

All variables in this example have the same list of visualizations: Table, Markers, and Polyline. You can select any of them. You can even select both Markers and Polyline for the same variable. However, you cannot plot Model against X, and you cannot select Polyline for Model and Markers for Observation. These visualizations are incompatible with each other, because the dataset does not indicate that these variables somehow relate to each other.

To plot one variable against another, the variables must share a dimension.

Exercise 3: Express Relationships between Variables as Shared Dimensions

In Scientific DataSet, relationships between variables are expressed using “shared dimensions.” A dataset dimension is an index space with a unique name.

In our example in Listing 2, each variable has its own index space. Their names are automatically chosen by the Scientific DataSet library. When Scientific DataSet reads the variables X and Observation from the file, it names their dimensions csv_0 and csv_1, respectively. The **Add** method automatically chose the dimension _1 for Model. You can clearly see this in the list of visualizations shown earlier in Figure 3.

To enable richer metadata in the dataset and consequently richer behavior of components that read this dataset, we can make all our variables share the same index space. For example, saying that variable Observation shares the dimension with variable X, we mean that Observation[i] relates somehow to X[i] for all indices i in the shared index space. This also introduces a constraint on the dataset:

> Two variables that share an index space must always be the same size along the shared dimension.

For example, a matrix can share its first dimension—the number of rows—with a vector. According to this constraint, the number of rows in the matrix must match the length of the vector. However, the matrix can have any width because it does not share the column index with the vector.

The only metadata in our sample CSV file Tutorial.csv is the header line. However, when we open the file, we can ask Scientific DataSet to assign a single shared dimension to all the columns that have the same height.

To assign a single shared dimension to all the columns of the same height

- Change line 4 of the sample code in Listing 2 to the following:

```
var dataset = sds.DataSet.Open("Tutorial.csv?inferDims=true");
```

- Replace the modified Tutorial.csv file with the original file, and then run the program again.

As you can see from this example code, you can put a question mark and provider parameters after the file name. Figure 4 shows that DataSetViewerControl can now plot—for example—Observation against X.

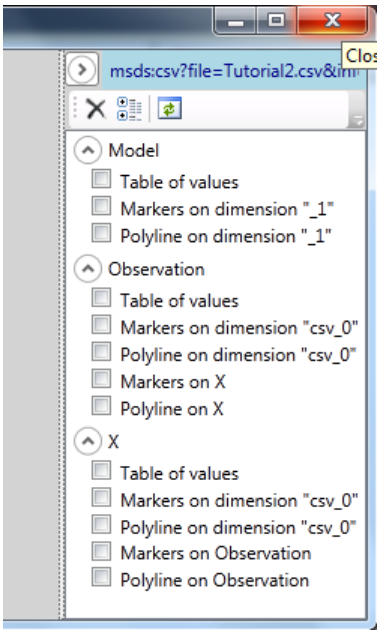


Figure 4. List of possible visualizations when Observation and X share the dimension

The Model variable is still in a separate index space. To tell Scientific DataSet that Model is related to other variables, you must explicitly specify the dimension name in the call to the **Add** method.

To specify the shared dimension name in the example code

- Add the dimension name to the **Add** call at line 24 in Listing 2:

```
var varid = dataset.Add<double[]>("Model",
     dataset.Dimensions[0].Name).ID;
```

We mentioned earlier that variables that share dimensions must be the same size. The **Add** method creates a variable but does not commit it to the dataset, as described in "Exercise 5: Perform Transactional Updates" later in this paper. This statement creates the Model variable, but Model does not contain any data; that is, its size is 0. Scientific Dataset will commit the variable to the dataset when all the variables that share the same dimension have the same size.

It would be a mistake to reference this variable in **PutData** by name in this case; instead, you must use its variable ID. The variable name does not become part of the dataset until the variable has been committed to the dataset. However, you can directly reference the variable by using its variable ID, as discussed in "Exercise 1: Add a Column to a CSV File" earlier in this walkthrough.

The list of available visualizations now grows considerably, as Figure 5 shows. We can now draw Model and Observation on the same plot.

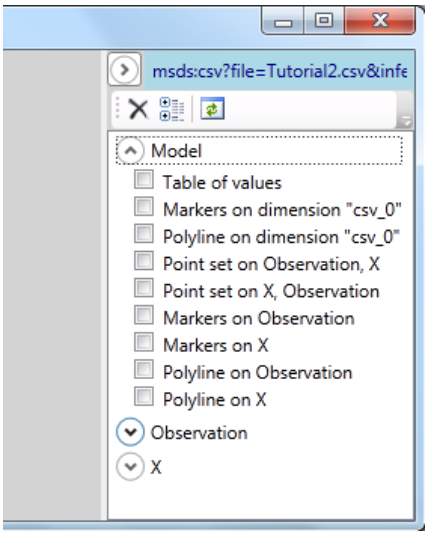


Figure 5. List of possible visualizations when three variables share the same dimension

To draw Model and Observation on the same plot in the example

- Select **Polyline on X** under Model, and select **Markers on X** under Observation.

Figure 6 shows the resulting visualization.

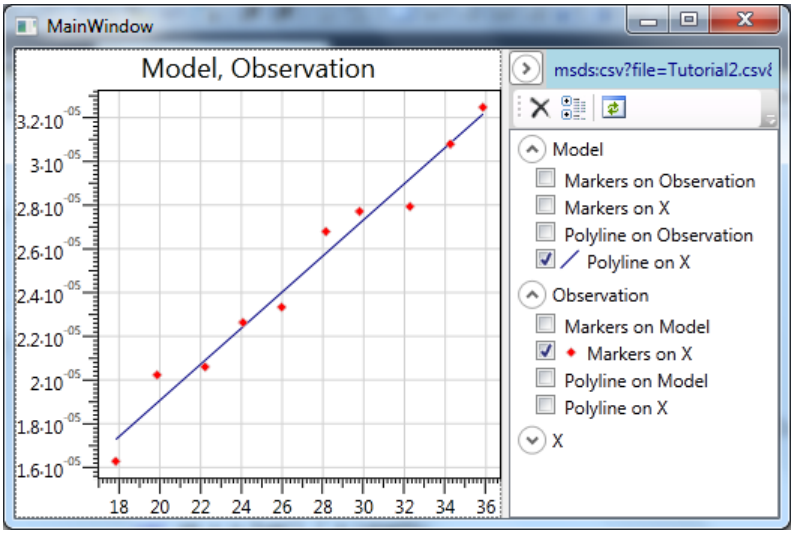


Figure 6. Drawing two variables on the same plot

Exercise 4: Store Descriptive Metadata for Variables and Datasets

In addition to using Scientific DataSet to store variable names and dimension names, you can:

- Store other descriptive metadata in the form of a (*key*, *value*) dictionary.
- Associate metadata with each variable individually or with the dataset as a whole.

The metadata can contain longer descriptions, units of measurements, a valid range of values, a value that denotes the absence of a value (missing value), and so on. By storing metadata in your dataset, you can provide a self-descriptive data package that includes information for use by other programs.

To programmatically add metadata

- To add metadata to the dataset from within your program, use the **PutAttr** method. In this exercise, we call **PutAttr** as follows:

  PutAttr (*variable*, *key*, *value*)

  where:

  - *Variable* can be the variable name or variable ID.
  - *Key* is a string that supplies the name of a metadata key.
  - *Value* is the metadata value. The value can be of any scalar type that Scientific DataSet supports or can be a one-dimensional array of such a type.

For example, DataSetViewerControl uses the VisualHints metadata value to select the default visualization for the dataset. You can assign a string for this metadata as follows:

```
dataset.PutAttr(0, "VisualHints", "Model(X) "
    + "Style:Polyline;Stroke:Navy;;"
    + "Observation(X) Style:Markers;Color:Red");
```

The VisualHints metadata value can contain several hints separated by two semicolons. A hint:

- Tells the control which variables to draw and which visualization style to use.
- Lists specific visualization parameters, separated by single semicolons.

The preceding example contains two hints: one for the Model variable and one for the Observation variable. Each hint contains two visualization parameters. With the addition of the VisualHints metadata value, the DataSetViewerControl displays a plot similar to that shown earlier in Figure 6 without additional user intervention.

Metadata that we add programmatically to our dataset does not persist in the CSV file because the original file does not have metadata entries. This behavior ensures better compatibility with programs that expect to find a plain table in the CSV file, but is undesirable in our case. For example, if we open the resulting file by using the stand-alone DataSet Viewer application, we must still:

- Specify additional constructor parameters.
- Compose the plot by using the user interface.

To append metadata to the CSV file

- To append metadata to the Tutorial.csv file—thus overriding the default behavior of Scientific DataSet—change line 4 to include the **appendMetadata** provider parameter as follows:

```
var dataset =
sds.DataSet.Open("Tutorial.csv?inferDims=true&appendMetadata=true");
```

Every time Scientific DataSet modifies Tutorial2.csv, it now adds metadata to the end of the file, as in the following:

```
X,Observation,Model
17.84,1.628E-05,1.72831547908291E-05
19.87,2.023E-05,1.89603556368157E-05
22.22,2.06E-05,2.09019428230564E-05
24.08,2.263E-05,2.2438688425783E-05
25.98,2.333E-05,2.40084823210414E-05
28.14,2.679E-05,2.57930901177562E-05
29.8,2.771E-05,2.71645942578241E-05
32.27,2.793E-05,2.920532632166E-05
34.25,3.079E-05,3.08412168019819E-05
35.85,3.247E-05,3.21631485032522E-05

ID,Column,Variable Name,Data Type,Rank,Missing Value,Dimensions
1,A,X,Double,1,,csv_0:10
2,B,Observation,Double,1,,csv_0:10
5,C,Model,Double,1,,csv_0:10

Variable,Key,Type,Value
0,VisualHints,String,Model(X) Style:Polyline;Stroke:Navy;;Observation(X)
Style:Markers;Color:Red
```

A blank line always separates data and metadata so that programs such as Excel can distinguish them easily.

Exercise 5: Perform Transactional Updates

In previous examples, we modified an existing dataset by adding more data to it. A disadvantage of this technique is the need to deal carefully with repetitive program runs. A more traditional and safer approach is to read one file and write another one. In this exercise, we use this approach to perform transactional updates to the dataset.

Consider the program shown in Listing 3.

Listing 3. Using separate datasets for input and output

```
static void Main(string[] args)
{
    if (args.Length != 2)
        throw new ArgumentException("I expect 2 command line"
             + "parameters.");
    // open input dataset. Set 'read only' mode by default
    var uri = sds.DataSetUri.Create(args[0]);
    if (!uri.ContainsParameter("openMode"))
        uri.OpenMode = sds.ResourceOpenMode.ReadOnly;
    var input = sds.DataSet.Open(uri);
    Console.WriteLine(input);
    // open output dataset. Set 'create' mode by default
    uri = sds.DataSetUri.Create(args[1]);
    if (!uri.ContainsParameter("openMode"))
        uri.OpenMode = sds.ResourceOpenMode.Create;
    var output = sds.DataSet.Open(uri);
    // read input data
    var x = input.GetData<double[]>("X");
    var y = input.GetData<double[]>("Observation");
    if (x.Length != y.Length)
        throw new ArgumentException("X and Observation"
              + "must have equal length");
    // compute model parameters
    var xm = x.Sum() / x.Length;
    var ym = y.Sum() / y.Length;
    double xy = 0;
    for (int i = 0; i < x.Length; i++)
         xy += (x[i] - xm) * (y[i] - ym);
    double xx = 0;
    for (int i = 0; i < x.Length; i++)
        xx += (x[i] - xm) * (x[i] - xm);
    var a = xy / xx;
    var b = ym - a * xm;
    // write output data
    int x_id = output.Add<double[]>("X", "table1").ID;
    int y_id = output.Add<double[]>("Observation", "table1").ID;
    int m_id = output.Add<double[]>("Model", "table1").ID;
    output.PutAttr(m_id, "long_name",
         "linear fit to Observation");
    output.PutAttr(m_id, "Model_A", a);
    output.PutAttr(m_id, "Model_B", b);
    output.PutAttr(0, "VisualHints",
        "Model(X) Style:Polyline;Stroke:Navy;;"
        + "Observation(X) Style:Markers;Color:Red");
    for (int i = 0; i < x.Length; i++)
    {
        output.Append(x_id, x[i]);
```

```
48          output.Append(y_id, y[i]);
49          output.Append(m_id, a * x[i] + b);
50      }
51      Console.WriteLine(output);
52  }
```

If you compile this program into Tutorial3.exe, you can run it from the command prompt or directly from Visual Studio.

To run this program from the command prompt

1. Change your directory to the folder that contains Tutorial3.exe and supply two command-line parameters:

   tutorial3 tutorial3.csv results.csv

The program will read data from the first file and output results to the second file.

To run the program directly from Visual Studio

- Specify command-line parameters on the **Debug** tab in the project's Properties window.

Opening a Dataset by Using a URI

Earlier, in "Exercise 1: Add a Column to a CSV File," we opened datasets by using a string argument with a file name and optional provider parameters. Lines 6–16 in Listing 3 show an alternative solution:

- First, we create a specialized uniform resource identifier (URI) object from a string.

  This object has a set of typed properties that are specific to Scientific DataSet providers. You can programmatically set the typed properties to change the behavior of Scientific DataSet.

- Next, we test whether a user supplied the **openMode** provider parameter in the command-line arguments.

  If not, the example program applies default **openMode** values for both the parameters. The default value for the input file is **readOnly** and the default for the output file is **create**. As a result of the defaults, the program cannot change the input dataset and always gets a new empty dataset for output, deleting the existing file if necessary.

The URI appears in a summary of the dataset. Line 11 of Listing 3 shows the fastest way to display such a summary

To display a summary of dataset contents

- Use the **Console.Writeline** method, as shown in Line 11 of Listing 3:

  ```
  Console.WriteLine(input);
  ```

This is what you should see in program output:

```
msds:csv?file=Tutorial3.csv&openMode=readOnly
[1]
DSID: df730881-6724-4b97-b6f3-4359a72083cb
[2] Observation of type Double (csv_1:10)
```

```
[1] X of type Double (csv_0:10)
```

In this output:

- The first line shows the standard dataset URI, which consists of the mandatory URI schema “msds” and the Scientific DataSet provider name “csv” followed by optional provider parameters.
- The second line shows the dataset version number—1—as an integer in square brackets.

  Scientific DataSet increments the version number each time it commits changes to the dataset.
- The third line shows the unique dataset identifier (DSID).
- A list of variables follows the DSID. For each variable, the summary shows the variable ID, name, data type, and shape:
  - Variable shape includes one dimension for vectors, two dimensions for matrices, and so forth.
  - The variable summary shows both the name and the length for each of the dimensions.

Updating the File

The computation in lines 21–33 of Listing 3 is the same as what we’ve seen earlier in the paper. Let’s now turn to the output section starting at line 35. This section of code writes data to the file as transactions—that is, it saves proposed changes, maintains a version number that tracks the number of changes, and commits the changes to the file only when all the related variables are ready to write.

In lines 35-37, we compose the dataset schema:

- We call the **Add** method to create the new dataset variables X, Observation, and Model. Each variable stores a one-dimensional array of doubles.
- We specify “table1” as the dimension name for all three variables because they share this dimension.

Several related columns that can have different data types but have the same height constitute what is commonly called a table. A dataset can contain several such tables, possibly of different heights. For example, to store a graph in a dataset you can create a table of node properties with one row per node and a table of edges with two columns that have source and destination node numbers.

So far, the example creates “table1”, which consists of three columns.

Lines 38–44 add metadata to the Model variable:

- The metadata includes numeric values for the coefficients that will be used to generate model values.
- The resulting file becomes a self-descriptive package that contains both the data and information about how the data was produced.

In lines 45–50, the program writes the data itself to the dataset in a loop, row by row, by using the **Append** method. **Append** adds data to an existing variable. The following shows the final output dataset summary:

```
msds:csv?file=Results.csv&openMode=create
[18]
DSID: bb322951-7ae5-40db-a314-b1377b12b9b3
[3] Model of type Double (table1:10)
[2] Observation of type Double (table1:10)
[1] X of type Double (table1:10)
```

You can see that all three variables have the length of 10. What is more interesting is the version number of 18. This needs more explanation.

Version Numbers

We have already mentioned that Scientific DataSet increments the dataset version number each time the dataset changes. For example, the call to the **Add** method in line 35 of Listing 3 increments it by one, as do the calls to **Add** and **PutAttr** in lines 36–44. Therefore, by the start of the output loop, the version number is 8. The loop repeats 10 times, and each iteration contains three calls to the **Append** method.

Why does the version increase to 18 and not to 38? Consider the first iteration of the loop:

- At the beginning all three variables are empty and their lengths equal zero.
- Now we call **Append** for variable X. Thus, we propose to increase its length to one.
- Scientific DataSet receives this proposal but does not make the change, because the shared dimension constraint requires that variables X, Observation, and Model have equal length.

Increasing the length of variable X without increasing the length of Observation and Model would result in lengths of 1, 0, and 0, respectively. This is a normal situation when working with datasets and not an exception. Scientific DataSet does not change the dataset immediately; instead, it keeps our proposal in its proposed changes list.

When we call the **Append** method for the second time:

- Scientific DataSet considers our second proposed change together with the previously saved one.
- Now the lengths of the variables would be 1, 1, and 0, so the shared dimension constraint is still not satisfied.

After the third call to the **Append** method in Listing 3, the proposed set of changes results in a length of 1 for each of the three variables. So Scientific DataSet finally commits the changes and increments the version number. Thus, the program makes only one commit per iteration.

Checking for Uncommitted Changes

What happens if by mistake you do not fill the whole table row?

To experiment with not filling the whole table row

- Comment out line 49 in Listing 3 and then run the program again.

You will see the following summary for the output dataset, and the file itself will contain no data:

```
msds:csv?file=Results.csv&openMode=create
[8]*
DSID: 5b92850c-2dc3-4d57-9981-1bcca3110ef7
[3]* Model of type Double (table1:0)
[2]* Observation of type Double (table1:10)
[1]* X of type Double (table1:10)
```

The asterisk after the version number indicates that the dataset has uncommitted proposed changes, and an asterisk after a variable ID indicates that the same is true for that variable. It is a good practice to check whether all changes have been committed before program exit. Uncommitted changes usually indicate some error in your program.

The **HasChanges** property for the output dataset is true if uncommitted changes are present. You can add the following to the sample to throw an exception if uncommitted changes are present:

```
if (output.HasChanges)
    throw new sds.ConstraintsFailedException("Uncommitted data "
        + "remain in output dataset");
```

Disabling Automatic Commits

Automatic commit of all the proposed changes sometimes introduces a significant performance penalty. For example, with each change in a CSV file, Scientific DataSet creates a new file, writes all the data and metadata in the file, and then deletes the previous version. This is the most robust way to change the file, but it can take considerable time if the file is large or requires many small changes.

You can easily disable automatic commits.

To disable automatic commits

- Use the following statement in your program:

```
output.IsAutocommitEnabled = false;
```

Starting from the point at which the statement appears, Scientific DataSet keeps all the proposed changes in memory and tries to implement them in the dataset when you call the **Commit** method.

To call the Commit method

- Use the following statement in your program:

```
output.Commit();
```

Before you call the **Commit** method, you must ensure that your proposed changes collectively satisfy all shared dimension constraints. If not, the method throws an exception.

Exercise 6: Use the NetCDF Provider with Large Datasets

The Scientific DataSet library can read and write data in multiple data formats without any change to program code. The following example shows how to ask Scientific DataSet to write to the NetCDF format.

To write results from the example in Listing 3 into a binary NetCDF file

- In Visual Studio, change the parameters on the Debug tab to specify an output file that has the .nc extension.

  —OR—

- Enter the following command at the command prompt:

  tutorial3 tutorial3.csv results.nc

The summary of the output dataset is similar to the one for a CSV file, except that the provider name portion of the dataset URI specifies “nc” instead of “csv”:

```
msds:nc?file=Results.nc&openMode=create
[18]
DSID: 8050a43d-97cd-498f-a284-787787167106
[3] Model of type Double (table1:10)
[2] Observation of type Double (table1:10)
[1] X of type Double (table1:10)
```

The NetCDF portable binary format has a significant overhead for small datasets, but it is very efficient for storing larger datasets. For more information about the NetCDF format, see the Unidata Web site, which is listed in “Resources” at the end of this paper.

We will illustrate the use of the NetCDF provider by using the file air.mon.mean.nc as an example. This 126-MB file is an output of the U.S. National Centers for Environmental Protection-Department of Energy (NCEP-DOE) Reanalysis 2 project. The file is available from the NCEP-DOE project data server. For more information about NCEP-DOE or to download the file, see “Resources” at the end of this paper.

Let’s first explore the contents of the file by using the **Sds** command-line utility, which is provided with Scientific DataSet.

To display a summary of a file’s contents

- Run **Sds** from the command prompt and specify the target NetCDF file, as follows:

  Sds air.mon.mean.nc

The command displays the following output:

```
[6] air of type Int16 (time:372) (level:17) (lat:73) (lon:144)
[5] time_bnds of type Double (time:372) (nbnds:2)
[4] time of type Double (time:372)
[3] lon of type Single (lon:144)
[2] lat of type Single (lat:73)
[1] level of type Single (level:17)
```

The **Sds** utility takes a Scientific DataSet file path or URI as a parameter and displays a list of the variables in the file. In the example output, you can see that the file contains the 4-dimensional variable **air**, which shares its dimensions with five other variables in the dataset.

To get more information about the contents of the air.mon.mean.nc file

- Print the metadata for the whole dataset, using the following command:

  Sds meta air.mon.mean.nc

  The command displays the following output:

```
                Name = air.mon.mean.nc
         Conventions = CF-1.0
               title = Monthly NCEP/DOE Reanalysis 2
             history = created 2002/03 by Hoop (netCDF2.3)
            comments = Data is from
NCEP/DOE AMIP-II Reanalysis (Reanalysis-2)
(4x/day).  It consists of most variables interpolated to
pressure surfaces from model (sigma) surfaces.
            platform = Model
              source = NCEP/DOE AMIP-II Reanalysis (Reanalysis-2) Model
         institution = National Centers for Environmental Prediction
          references = http://wesley.wwb.noaa.gov/reanalysis2/
http://www.cdc.noaa.gov/cdc/data.reanalysis2.html
```

- Print the metadata for an individual variable by using the following command:

  Sds meta air.mon.mean.nc air

  This command displays the following output:

```
[6] air of type Int16 (time:372) (level:17) (lat:73) (lon:144)
                Name = air
           long_name = Monthly Air Temperature on Pressure Levels
         valid_range = -32765 -10260
unpacked_valid_range = 137.5 362.5
        actual_range = 179.4077 315.7219
               units = degK
          add_offset = 465.15
        scale_factor = 0.01
       missing_value = 32766
          _FillValue = -32767
           precision = 2
least_significant_digit = 1
             GRIB_id = 11
           GRIB_name = TMP
            var_desc = Air temperature
             dataset = NCEP/DOE AMIP-II Reanalysis (Reanalysis-2) Monthly Averages
          level_desc = Pressure Levels
           statistic = Individual Obs
         parent_stat = Other
```

Next Steps

Now that you have a sense of what Scientific DataSet can do and how to use it, you can start to use it in your computational programs. For example, using your own data files:

- Add computed data to a CSV or NetCDF file.
- Create a visualization by using the DataSet Viewer.
- Add VisualHints metadata to a CSV file to describe the appropriate visualizations for your data.
- Perform iterative dataset writes using the **Append** method.

The current release of Scientific DataSet supports the following features:

- Virtualized access to heterogeneous scientific data sources

- CSV and NETCDF data provider
- Ability to extend Scientific DataSet with your own providers
- Dataset Viewer
- Add-in data editor for Microsoft Excel

As we continue to develop Scientific DataSet, we are investigating the following additional features:

- The ability to work within a distributed environment
- Inspectable datasets
- More providers

For updates, tools, and discussion, see the Scientific DataSet project Web site, which is listed in “Resources.”

For more details about Scientific DataSet capabilities, see the Scientific DataSet Help.chm file.

Resources

This section provides links to software and additional information.

Scientific Dataset Library, Tools, Documentation and Discussion

Scientific DataSet Project Site
http://research.microsoft.com/projects/sds

Software and Tools for Computational Science
http://research.microsoft.com/groups/science/software.aspx

Software

The following software packages are available to download at no charge from Microsoft:

Microsoft .NET Framework 3.5 Service Pack 1
http://www.microsoft.com/downloads/details.aspx?FamilyId=AB99342F-5D1A-413D-8319-81DA479AB0D7

Microsoft Visual C# 2008 Express Edition
http://www.microsoft.com/express/Windows/

See the Visual C# Developer Center on MSDN at
http://msdn.microsoft.com/vcsharp/

Data Formats and Providers

NCEP-DOE Project data server
The data file air.mon.mean.nc is available at the following address:
ftp://ftp.cdc.noaa.gov/Datasets/ncep.reanalysis2.derived/pressure/air.mon.mean.nc

NCEP-DOE Reanalysis 2 Summary
http://www.esrl.noaa.gov/psd/data/gridded/data.ncep.reanalysis2.html

NetCDF on the Unidata Web Site
http://www.unidata.ucar.edu/software/netcdf/

Unidata Common Data Model
http://www.unidata.ucar.edu/software/netcdf-java/CDM/