

# ContourIV Client/Server Documentation

Emilio Camahort

Center for Computational Visualization  
Texas Institute for Computational and Applied Mathematics  
The University of Texas at Austin  
Austin TX 78712–1188

April 9, 1999

## 1 Introduction

This document describes the design of the contourIV client/server implementation. It is intended to be a programmer's guide describing those details of the implementation that are difficult to infer from the actual source code. Such details include:

- ★ class hierarchy
- ★ object oriented design (objects and their relationships)
- ★ user interface design
- ★ structure of the scene graph(s)
- ★ client/server design issues:
  - server commands
  - data communication structures
  - client/server state
  - error handling

Hence, this documentation should be viewed as a complement to the actual source code of the application. It is strongly advised to read this documentation before reading or modifying the source code. Questions and comments about it should be addressed to: `ecamahor@cs.utexas.edu`

This document is divided into four sections. Section 2 provides some general information about contourIV and its implementation. Section 3 describes the design of the computing server. Section 4 describes the design of the client/server interface. Section 5 describes the design of the client's user interface.

## 2 Preliminaries

In the following description of the code, a module usually refers to a pair of `.h` and `.C` files, where the `.h` file defines a certain class, and the `.C` file contains its implementation. In some cases there may only be a `.h` or `.C` file. In all cases, however, we refer to them as modules, unless otherwise noted. Modules usually bear the same name as the class they define, spelled in lowercase.

A complete list of all ContourIV modules can be found in a separate file `modules`. The list includes the modules of the client/server version of the code, plus some obsolete modules of the original version. The relationships between modules depicted in figures 1 and 3.

There are basically three sets of modules that make up the application: (i) server modules, (ii) client modules, and (iii) common modules. The server modules contain code to read and store different types of data and extract information from them. The types supported by the application can be classified according to four different criteria:

- ★ single variable vs. multi-variate data,

- \* two-dimensional vs. three-dimensional data,
- \* unstructured vs. regular grid data, and
- \* single timestep vs. time-varying data.

Not all the functionality is currently supported for all possible combinations of these data types (see below). For a given dataset two types of information can be extracted: signature functions and isocontours. 3D regular data sets also allow slices to be extracted, although slice display is not yet supported at the client end. In order to extract isocontours a set of seeds is computed by the server upon reading a given data set. The seeds are stored at the server end, but can also be shipped to the client for display.

There are three source files shared by both the server and the client. Those files are: `data.h`, `commdata.h`, and `seed-cell.h`. Their functionality is described in section 4. Additionally, there is a different `server` module on each the server and the client that performs all the communication between both programs.

The client displays the contour spectrum and allows the user to extract and display isocontours of a given data set by moving a slider located on the spectrum's window. For a given isovalue the client requests an isocontour from the server, waits for its reception, and renders it. The client is a thin client, that is, it caches no information generated by the server, but the information that it is currently displaying. The server, on the other hand, uses lazy evaluation for all the client requests, that is, it does not precompute any information, but that necessary to fulfill the client's requests.

Finally, note that the functionality currently available is the same as the original ContourIV functionality for 3D regular meshes. Refer to appendix 6 for more details about the most recent release of the software.

### 3 Server

The server program consists of the following building blocks:

- \* the main program `main` and the `server` module, which contain all the server communication code;
- \* all the `data*` modules representing and containing dataset information;
- \* all the `conplot*` modules (except `conplot_p`), which implement isocontouring algorithms for the different types of data; and
- \* the `conplot_p` module and a large collection of support modules, which altogether implement dataset preprocessing and seed computation algorithms.

Their relationships are depicted in figure 1.

The main program simply sets up the table of server commands and invokes the Shastra communication substrate to start a server and listen at port number 8878 (unless otherwise specified in the command line). The `server` module implements all the server commands, as well as the necessary data structures to support concurrent access to multiple datasets and lazy evaluation. `main` and `server` are the **only** two modules that use, that is, include data communication structures and library routines.

The `data*` modules implement all possible dataset types handled by ContourIV. They form the class hierarchy depicted in figure 2. The base class `data` implements a single timestep of data. It has subclasses `dataslc`, `datavol`, `datareg2` and `datareg3`, one for each type of data handled by the server. The `data*` modules also include the necessary code to implement multivariate data, that is datasets whose elements have more than one component like, for example, a 3D vector field.

The subclasses `dataslc` and `datavol` represent 2D and 3D unstructured data, respectively. 2D data is organized as a planar graph with vertices and edges. 3D data is organized as a 3D triangular mesh. The subclasses `datareg2` and `datareg3` represent 2D and 3D regular grids of data, respectively. They are represented as 2D or 3D arrays.

The classes `dataset2d`, `data3d`, `datareg2`, and `datareg3` are container classes that represent time-dependent versions of `dataslc`, `datavol`, `datareg2`, and `datareg3`, respectively. They are all subclasses of `dataset` and they contain one object per timestep. All objects are stored in an array indexed by time.

The `conplot*` set of modules encapsulates all the algorithms related to isocontouring. The modules `conplot2d`, `conplot3d`, `conplotreg2` and `conplotreg3` contain the algorithms that generate isocontours for the above four types of datasets. Modules `conplot2d` and `conplotreg2` generate objects of class `contour2d`, while modules `conplot3d` and `conplotreg3` generate objects of class `contour3d`. These contain the actual isocontours computed by the `conplot*`

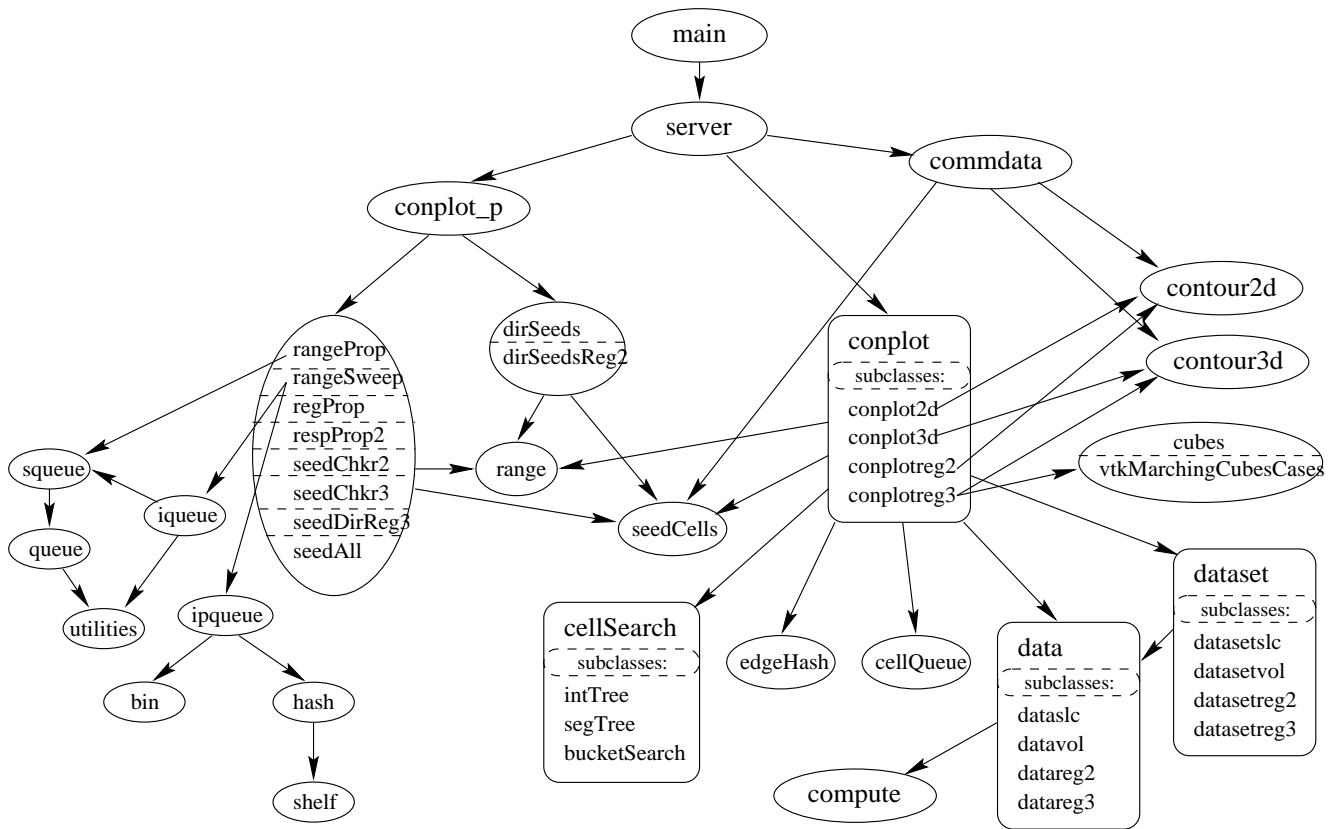


Figure 1: Server modules: the arrows indicate dependencies between modules.

set of modules. Finally, `conplotreg3` uses the modules `cubes.h` and `vtkMarchingCubesCases.h` to generate the isocontours. These are modules that contain data necessary to apply the marching cubes algorithm to a 3D regular dataset.

The algorithms in the `conplot*` modules use a set of seeds for fast isocontour computation. Seed sets are computed on demand, one per timestep and variable (when handling multivariate data). The module `conplot.p.c` has the code that computes the seed sets, also called sets of seed cells. The modules that it relies on are depicted in figure 1. Two of those modules, `range` for range arithmetic and `seedCells` for storing the seed cells, are shared by the `conplot*` modules. `seedCells` is also used by the server module and the client application. It defines the seed cell data structure and a container that stores a set of seeds. Additionally, the `conplot*` modules use a set of seed search structures. `cellQueue` implements a queue of cell identifiers, and `cellSearch` defines an abstract search structure for seed cells, that can be instantiated to be an interval tree: `intTree`, a segment tree: `segTree`, or an array of buckets: `bucketSearch`.

## 4 Communications Interface

This section describes how the communications interface is designed on both the server and the client side. A server can attend to multiple clients each accessing the same or a different dataset. A given client can only be working with one server and one dataset at a time. A server allows two or more different clients to share the same dataset in order to avoid replicating data in memory. A dataset requested by a client is stored in an object and assigned a dataset identifier. Later requests for data are then evaluated lazily, as they are issued by the clients to the server.

By data we mean seed cells, signature functions, signature values, data slices and isocontours. Once computed, seed cells and signature functions are kept locally at the server. They are indexed in order by dataset identifier, variable and timestep. That way they do not need to be recomputed when requested by the same or another client. Data associated to a dataset is kept until there are no clients using it. The server keeps track of how many clients are currently accessing a dataset, and deletes it from main memory when all the clients are done using it.

The client is a thin client with barely any local state. It sits on a communications substrate that encapsulates all the interaction

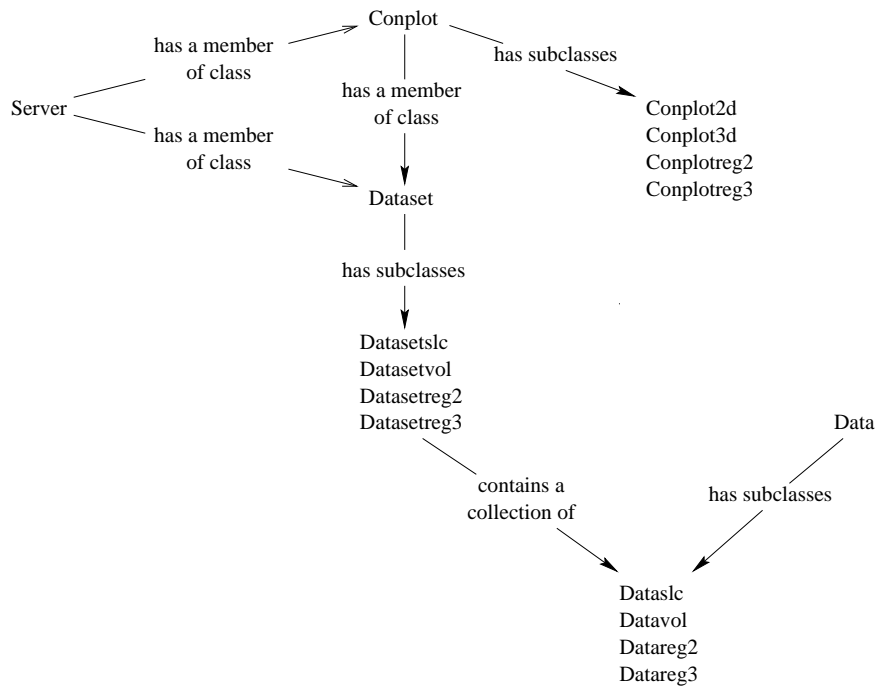


Figure 2: The data\* classes and their relationships.

between the client and the server. The substrate keeps track of the information pertaining to the client/server connection, including:

- \* the connection's socket number,
- \* the dataset's identifier and some associated info, like extents, etc,
- \* the current set of seed cells (if any) and its parameters,
- \* the current set of signature functions and its parameters,
- \* the current sets of slices (if any) and its parameters, and
- \* the current isocontour and its parameters.

The communications substrate makes sure that requests are only sent to the server when data is not already stored locally. It also guarantees that no memory-to-memory copies are performed on the client side (except for signature functions). To do so the seed, slice, and contour data is only allocated and freed by the `xdr` routines, and hence data should never be freed by other modules than the communications substrate. It is also recommended that they not be copied to keep the code efficient.

The client/server communications substrate is defined by seven modules. Three of those modules are shared by both the client and the server. `commdata.h` defines all the data structures sent by the server to the client. It includes modules `data.h` and `seedCells.h` for basic data and seed cell definitions, respectively. All three modules are kept with the server distribution with a link from the client's distribution directory.

The data structures in `commdata.h` define the client/server data interface. They are defined as `struct`'s instead of classes, because they are read only and they have no methods associated, that is they endure no changes or processing at the client side. Using `struct`'s greatly simplifies building, sending, receiving and accessing these data structures. There are nine data structures:

<code>DatasetParams</code>	basic information about a given dataset,
<code>VariableNames</code>	variable names and parameters,
<code>SeedData</code>	seed information,
<code>SliceData</code>	a data slice,
<code>Signature</code>	data associated to a signature function,
<code>SignatureData</code>	all signature functions of a dataset,
<code>SignatureValues</code>	signature values for a given isovalue,

Contour2dData      a 2d isocontour, and  
 Contour3dData      a 3d isocontour.

The other four communications modules contain the communications routines used by the server and the client. On the server side the modules are `main`, the main program, and `server`, which implements the server commands listed below. On the client side, the communications related modules are `contour`, the main program, and `server`, which encapsulates all the client/server communications routines.

Finally, the client/server interface is described by the following static routines defined in `server.h`:

```
connectToServer(hostname, portnr)
loadDataset(datatype, meshtype, nrvars, nrtimes, files)
getDatasetParameters(data_id)
getVariableNames(data_id)
getSeedCells(data_id, variable, time)
getSignatureFunctions(data_id, variable, time)
getSignatureValues(data_id, variable, time, isovalue)
getSlice(data_id, variable, time, axis, value)
getContour2d(data_id, variable, time, isovalue)
getContour3d(data_id, variable, time, isovalue, colorvar)
saveContour2d(data_id, variable, time, isovalue, filename)
saveContour3d(data_id, variable, time, isovalue, colorvar, filename)
clearDataset(data_id)
```

where

`hostname` is the hostname running the server,  
`portnr` is the number of the port the server is listening to,  
`datatype` is the dataset's base type from the command line,  
`meshtype` is the dataset's type from the command line,  
`nrvars` is the number of variables in a single data element, also from the command line,  
`nrtimes` is the number of timesteps or, equivalently, files that made up the dataset,  
`files` are the filenames containing the dataset,  
`data_id` is an identifier allocated to the dataset by the server,  
`variable` is one of the variables of the dataset  
`colorvar` is one of the variables of the dataset used to color an isosurface,  
`time` is a timestep of the dataset  
`axis` is a one of the three coordinate axis  $x, y$  and  $z$ ,  
`value` is a timestep of the dataset, and  
`filename` is a filename, including a possible directory path.

For the exact types of the above parameters see the file `server.h` in the client's distribution. The following table shows the return values of each of the above routines:

<code>connectToServer</code>	connection socket (or error)
<code>loadDataSet</code>	dataset identifier
<code>getDatasetParameters</code>	dataset parameters
<code>getVariableNames</code>	variable names and parameters
<code>getSeedCells</code>	set of seedcells and associated parameters
<code>getSignatureFunctions</code>	set of signature functions
<code>getSignatureValues</code>	an array of signatures values
<code>getSlice</code>	slice and associated parameters
<code>getContour2d</code>	contour2d and associated parameters
<code>getContour3d</code>	contour3d and associated parameters
<code>saveContour2d</code>	error (if any)
<code>saveContour3d</code>	error (if any)
<code>clearDataset</code>	error (if any)

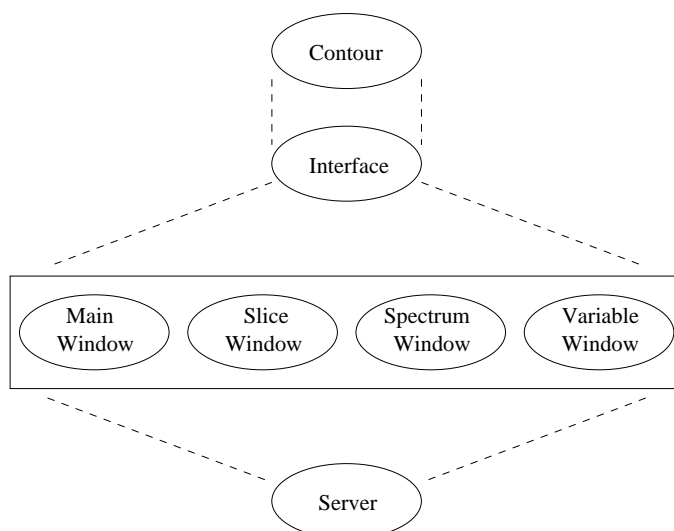


Figure 3: Client basic structure.

## 5 Client

The ContourIV client application is logically organized in a four-level structure depicted in figure 3. The main contour program, `contour.C`, parses the command parameters, establishes a connection to a remote server, and starts the user interface. The user interface module, `interface`, manages the global state of the program, and encapsulates the four windows of the client application: the main window, the slice control window, the contour spectrum window, and the variable control window. The server module, `server`, provides the thin, (apparently) stateless interface to the remote server. All modules of the client application use the `server` module. `contour.C` uses the `interface` and `server` modules, but none of the window modules.

The five top-level classes of the user interface provide a window-system independent wrapper. The design allows each window class to be modified independently from the global state of the interface and the rest of window classes. There is one class for each window in the user interface:

<code>MainWindow</code>	manages the main window,
<code>SliceWindow</code>	controls the slice selection,
<code>SpectrumWindow</code>	manages the contour spectrum, and
<code>VariableWindow</code>	for variable selection control.

The `SliceWindow` module is not implemented, yet. There is one object for each of the above classes. Window objects and classes have the same name, except for their initials: lower-case for objects and upper-case for classes. Window objects have non-redundant independent state. Hence, they can be modified separately without side effects. The global application variables, namely the current dataset id, variable, timestep, etc. are stored in the interface object. They are accessed through interface `get/set` methods. State managed by each specific window, like the scene graph for the main window, and the signature functions for the spectrum window are stored in the window's object. Window objects can be updated from other windows by using `update` methods.

Window classes are window-system dependent, but in principle they can use different windowing or rendering capabilities. Windows can thus be modified to use newer APIs, say OpenGL Optimizer, without affecting the other windows. Currently all windows rely on legacy OpenInventor code. Specifically,

- ★ `MainWindow` uses:
  - `SoContour2d`, an Inventor node class for 2D isocontours, and
  - `SoContour3d`, an Inventor node class for 3D isocontours,
- ★ `SpectrumWindow` uses:

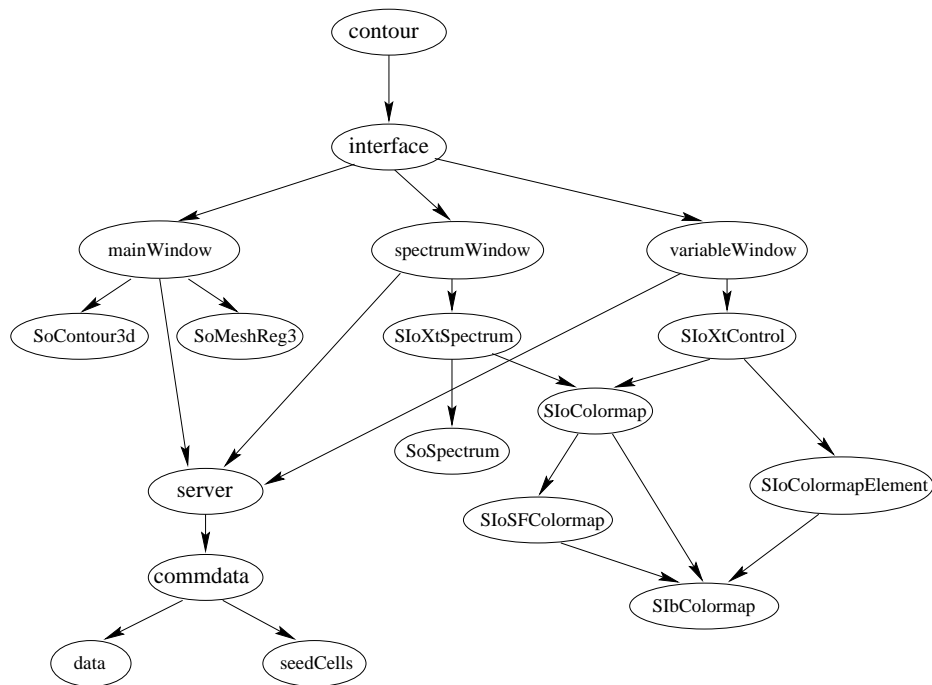


Figure 4: Client modules: the arrows indicate dependencies between modules.

- `SIoXtSpectrum` which manages the entire window, and

\* `VariableWindow` uses:

- `SIoXtControl` which manages the entire window

`SpectrumWindow` and `VariableWindow` are mere wrappers around `SIoXtSpectrum` and `SIoXtControl`, respectively. They are provided to facilitate future migration to a different window or user interface API. The overall structure of the client distribution is depicted in figure 4.

## 6 Demos

The `contouriv` demo directory is in: `roberto: ecamahor/ccv/demo`. It mimics Valerio's demo directory in: `roberto:/ccv/ccv/demo`. In order to start a remote server (on `roberto` or `scantling`) you MUST have `.rhosts` files in either machine, since the scripts that start the servers use `rsh`. Here's is a sample `.rhosts` for doing demos on `rishi`:

```
rishi.ticam.utexas.edu <login_name>
```

Once you have the `.rhosts` files setup, `cd` to `ecamahor/ccv/demo` and run: `RUN-DEMO`. Start a server and then any client application. The client applications will know where the server is, but you need to start a server first!!

You can start as many clients as you want. Also, you can start a new server on a different machine, and from that point on you can start another group of clients. They will now use the new server (as opposed to the first one you started).

Clients terminate their server connections gracefully when you press the quit button. Servers may need to be killed explicitly. Killing `RUN-DEMO` with the delete key is the ONLY way of killing all servers, but you may want to double-check by doing:

```
ps -fu <login_name> | grep server
```

on both the local and the remote machine. Please do kill all servers after running a demo, since they all listen to the same port, and leaving one behind prevents everybody else from running their own server.

## Appendix: Release Notes

### January 1999 Release Notes

The current distribution has the same functionality available as the original functionality of ContourIV for 3d regular meshes. Slice and 2D isocontour extraction routines are provided in the client/server interface, but they require client user interface support to be tested and stabilized. Otherwise, all the client/server interface routines are stable (or so do I think). Newly added functionality includes:

- \* server can handle multiple connections from multiple clients,
- \* server can load multiple datasets, one for each client,
- \* server loads each dataset only once, hence, two clients will share the same dataset at the server end, and
- \* all data is computed lazily at the server end.

Source code is located in: `roberto: ecamahor/ccv/src/ivclient|server` Scattered documentation files, including the sources for this document, can be found in: `roberto: ecamahor/ccv/doc`. See section 6 for information on how to run demos.

### April 1999 Release Notes

This new release incorporates the following changes to the distribution:

**Source directory change.** The Inventor client sources are now located in `roberto: ecamahor/ccv/src/ivclient`.

**Saving isocontours to a file.** It is possible now to save an isocontour on the server side. A button, labelled F has been added to the client's user interface in order to support this feature. The isocontour is saved to a file in `iPoly` format. The name of the file is chosen by the client as follows:

```
<pathname>/<filename>.<variable>.<timestep>.<isovalue>.ipoly
```

where `<pathname>` and `<filename>` are the path and file names of the original dataset and `<variable>`, `<timestep>` and `<isovalue>` are the parameters defining the isocontour. If the original dataset is comprised of multiple time-dependent files, then the first one is chosen for `<pathname>` and `<filename>`. Two new functions `saveContour2d` and `saveContour3d` have been added to the communications interface to provide this functionality. A description of these functions can be found in section 4 of this document.

**Support for variable names.** Support for variable names has been added in order to allow variable names in the variable control window. A file with the same name as the dataset's file, but extension `.var` should be placed in the same directory as the dataset. For example, for `pot-2eti-glucose-3fields.raw` the file would be `pot-2eti-glucose-3fields.var` and its contents:

```
van der Waals  
electrostatic  
total energy
```

that is, one line per variable name in the same order as the variables appear in the dataset file. If the dataset is comprised of multiple time-dependent files, then the first one is chosen and its extension substituted by `var`. A new function `getVariableNames` has been added to the communications interface to provide this functionality. A description of this function can be found in section 4 of this document. Similarly, a new data structure `VariableNames` has been added to the communications data interface in `datacomm.h`. Refer to this file for more information.

**The libcontour library.** The new library `libcontour` provides the same functionality as the ContourIV server, that is, it allows loading a dataset from disk and extracting isocontours, slices, etc. using library routines instead of a compute server. Additionally, it supports processing of data passed as a parameter, so you do not need to obtain the data from a file. Instead, you can simply pass it to the library stored in one or more arrays.

The library distribution is located in `roberto: ecamahor/ccv/src/library`. This directory contains the sources, the Makefile, a README file with some additional information, and the library file `libcontour.a` and header file `contour.h` that make the distribution. There are two sample test programs located in `roberto: ecamahor/ccv/libtest/disk` and `roberto: ecamahor/ccv/libtest/memory`. They contain code to load a data set, extract an isocontour, and

display it. The first directory contains code that uses the library to read the data directly from disk. The second directory contains code that reads the data from disk on its own and then calls library routines to create a dataset and extract isocontours.

## Appendix: Files with Inventor Includes

Following is a list of the modules of the original ContourIV program that contained Inventor code in them. Some of them have been removed from the distribution(s). The list is useful for historical purposes, as well as to aide in the conversion of the client code to a different window system, like `gtWidgets`.

```
SibColormap.h
SioColormap.C
SioColormap.h
SioColormapElement.C
SioColormapElement.h
SioSFColormap.h
SioXtControl.C
SioXtControl.h
SioXtSpectrum.C
SioXtSpectrum.h
SoContour2d.C
SoContour2d.h
SoContour3d.C
SoContour3d.h
SoControl.C
SoControl.h
SoMesh2d.C
SoMesh2d.h
SoMesh3d.C
SoMesh3d.h
SoMeshReg2.C
SoMeshReg2.h
SoMeshReg3.C
SoMeshReg3.h
SoSpectrum.C
SoSpectrum.h

conplot.C
conplot.h
conplot2d.C
conplot3d.C
conplotreg2.C
conplotreg3.C

contour.C
window.C
```

## Appendix: List of modules

The list of modules of the client/server ContourIV application is located in the file `modules`. Each module is followed by a brief description and some other related information. Modules are organized in groups. A letter in the first column indicates whether the module is part of the client (C) or the server (S) distribution. Modules with no letter are modules of the original ContourIV distribution that are either obsolete or pending the implementation of related functionality.