IFrIT

User Guide

Reference Guide

by

Nick Gnedin and IFrIT

IFrIT Version 4.1.2

Ifrits in common mythology are jinn spirits that embody fire. They consider themselves superior to all races because they supposedly "came first," and they resent deeply that humans have found magical ways to take control over them.
Even when tasked, they show an ironic and malicious attitude, tending to subvert their masters' orders every time they can.

(Wikipedia.org)

Table of Contents

Table of Contents

Table of Contents

1 User Guide

1.1 Overview

1.1.1 What can IFrIT do?

IFrIT can visualize four different classes of data:

- **Scalar** data: several scalar variables in 3D space.
- **Vector field** data: a 3D field of vectors.
- **Tensor field** data: a symmetric 3x3 tensor in 3D space.
- **Particle** data: a set of particles (points) with several optional variables (numbers that distinguish particles from each other) per particle.

For the scalar data the following visualization objects are available:

- A two-dimensional surface (either an isosurface of a given variable, or a fixed geometric surface: a plane or a sphere). Several instances (copies) of each surface may co-exist (for example, isosurfaces at different levels of the same variable). Surfaces can be colored on the outside or inside by a value of another scalar variable, translated into color through a palette.
- An orthogonal cross section of a data cube, again several of them can be shown at a time.
- Volume rendering of one scalar variable.

The vector field data can be represented either as

- a "vector glyph" - a line that starts at each mesh point, points in the direction of the vector, and has a length proportional to the vector magnitude (sorry, no arrows in 3D), or
- a set of streamlines - lines along which the fluid would flow if the vector field is assumed to be a velocity field of some imaginary fluid. Streamlines can be colored by vector field properties (like magnitude, vorticity, etc), or by scalar variables, if the scalar data are loaded and have the same dimensions as the vector field. Streamlines also can be represented as tubes, with the tube diameter inversely proportional to the vector magnitude, or by ribbons with two neighborning streamlines being the ribbon boundaries.

The tensor field data are hard to visualize. At the moment, the only supported visualization mode is the "tensor glyphs" - ellipsiods with orientation and dimensions proportional to three tensor eigenvalues, placed at some of the vertices of the uniform mesh.

The particle data can be split into individual groups, and particles in each group can have various representations (dots, spheres, clouds of dots, etc), can be colored by the value of one of the variables, and can be sized with an arbittary sizing function by the value of another variable. Particles belonging to one group can be connected by a line - this is useful for, say, plotting trajectories.

Different modes of visualization are coexistent, they are activated/de-activated independently of each other. Several visualization windows can exist at the same time, each one having a full set of visualization objects. Some visualization windows can share the data between them, while other windows can be fully independent. Images from several visualization windows can be combined into one image file on the disk, tiling some windows together, and inserting reduced versions of some windows into larger other windows.

A large array of nifty features is also available, including highly advanced animation capabilities, a complex set of lights, markers to label various points in space, a capability to "pick" a point in the scene and retrive information about the data at this location, two scripting languages, etc.

1.1.2 General Structure of IFrIT

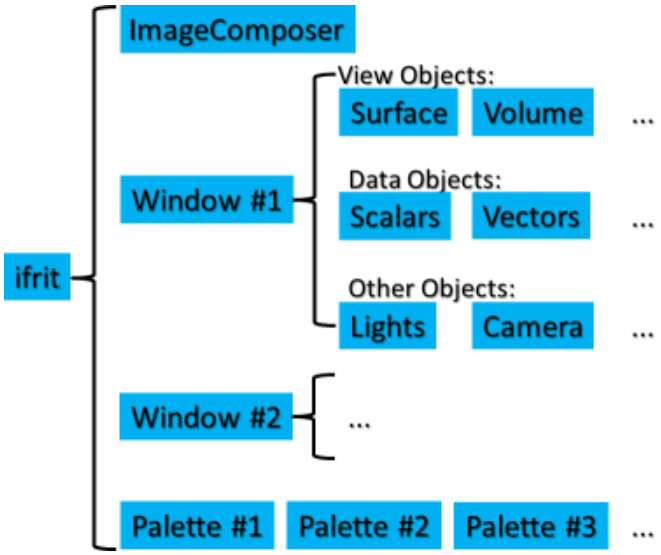
IFrIT consists of a set of components called **Objects**. Each object is designed to perform a certain function: just like in a computer various components - a processor, memory, a video card, etc - are designed to perform certain tasks. Various objects have diverse relations to each other: some objects are independent, while other objects may belong to different objects. You can think of the general structure of IFrIT as a directory hierarchy: objects may have "sub-objects" in them, these, in turn, may have "sub-sub-objects", etc.

When a user uses IFrIT, she/he interacts with various objects, causing them to perform various functions. The actual way in which a user interacts with IFrIT depends on a specific **shell** used, and two different shells are currently available. The simplest one is a **Command-line Shell**, that uses a command prompt to take typed commands from the user and forward them to other objects. Because typing commands is slow, this shell is only useful for driving IFrIT over a remote slow connection, when graphical user interface does not work well. The primary shell to use is GUI (Graphical User Interface) - it provides a user with a fast way of manipulating various objects via GUI elements (buttons, sliders, check boxes, etc).

Each invocation of the executable uses a specific shell, but more than one shell can be compiled into the executable, so that different shells can be used without re-compiling IFrIT. A specific shell can be invoked either by using Command-line Options, or if it specified as a default one. If the default shell is not specified, and none is requested in the command line, and more than one shell is compiled into the code, the GUI shell will be used. Only shells that have been compiled into IFrIT during the installation will be accessible, of course.

1.1.3 Objects and Properties

The root of the object hierarchy is the **ifrit** object; it serves as a container to other objects and performs a few other miscellaneous functions. The diagram below shows the hierarchy of various objects (the complete hierarchy is shown in Object Reference):



The **ifrit** object includes one or more **Window** objects. A **Window** object appears as an IFrIT visualization window on the screen; it forms an independent part of IFrIT that has its own set of other (sub-)objects and, often, its own data. Some of **Window** objects may share the data with another **Window** object - in that case the owner of the data is called a "parent" **Window** object, and sharing **Window** objects are called "clones". Irrespectively of whether a given **Window** object is a clone or a parent, it has a complete set of its own (sub-)objects.

Window objects use other objects for various functions. For example, some objects are responsible for a general setup of the visualization scene, such as lighting (**Lights** object), camera (**Camera** object), bounding box (**BoundingBox** object), clipping plane(**ClipPlane** object), and various accessories (such as rulers, labels, measuring boxes, etc).

Other objects perform operations on various components of the visualization scene. For example the **Surface** object represents a two-dimensional surface within the visualization scene that samples the three-dimensional scalar data (either as an isosurface of a particular scalar variable, or a specified geometric shape like a sphere or a plane), while the **Particles** object represents a set of particles (points).

Yet another group of objects perform various functions that are not directly represented in the visualization scene. For example, the **DataReader** object is responsible for loading data files into IFrIT, while the **Animator** object creates animations of your visualization scene in a diverse variety of ways.

Several objects do not belong to one of the **Window** objects, but rather communicate with all of them. An **ImageComposer** object is responsible for composing a snapshot or an animation image from several windows. An array of **Palette** objects represents palettes used by IFrIT. The complete hierarchy of all objects is presented in the Object Reference Guide.

All object are controlled by **properties**. You can think of a property as of a special variable: as soon as you assign a new value to the variable, something happens. For example, if you assign a value `true` to the property **BoundingBox.Visible**, the bounding box will appear in the visualization scene. Assign `false` to that property, and the bounding box disappears.

Properties can be controlled in two ways. A GUI shell uses GUI elements (widgets) to directly set or get a value of a property. Alternatively, a shell can use a scripting language and access object properties as script variables. In a command-driven shell a script is a must - otherwise, IFrIT cannot be controlled; in a GUI shell

a script is optional, but without a script animation capabilties will be rather limited. Currently only Python is supported as IFrIT scripting language (the primitive script of IFrIT 3 is no longer supported), but other languages may be added in the future (Python is popular, but it is kind of an overkill for animation scripting, and its syntax is too restrictive for efficient access to object properties).

In addition to properties-variables, some objects also have **methods** (C++ users will find terminology familiar) - script functions (rather than variables). For example, **Window.CloneOf()** method returns the index of the parent window if this **Window** object is a clone of another one, or invalid index if it is not a clone.

On a command line a given object property is accessed in a way consistent with the currently used scripting laguage syntax. Using Python as an example, the complete name of the property **Visible** of the object **BoundingBox** that belongs to the first **Window** object is addressed as **ifrit.Window[0].BoundingBox.Visible**. A **BoundingBox** belonging to the second window is accessed as **ifrit.Window[1].BoundingBox.Visible**, etc. Of course, in Python one can always assign a name to any object, so the assignment `b = ifrit.Window[0].BoundingBox` will let the user to access the property **Visible** simply as **b.Visible**. Never-the-less, full object names can be very long! For example, the type of a screen representation with which to display the particles that represent the built-in **Particles** data is fully referred to as **ifrit.Window[0].Particles["Particles"].Type** - such names will strain the patience of even most hardcore command-line afficionados.

In order to simplify naming objects and their properties, each object and property has a short form for its name as long as a full name. In short form **ifrit.Window[0].Particles["Particles"].Type** becomes **ifrit.w[0].p["p"].t**, which should satisfy even the strictest Linux guru.

1.1.4 Objects and Data

Six IFrIT objects do the actual visualizations: **CrossSection**, **Surface**, **Volume**, **Particles**, **VectorField**, **TensorField**. Therefore, these objects need the data to visualize. In the standard edition of IFrIT each of visualization objects uses just one type of data, but if extensions are installed, there may be more than one type of data for each object to use. Visualization objects create separate copies of themselves to deal with each type of data, hence all these data types can be configured and visualized independently. For example, **particles** may be visualized by points with the built-in **Particles** data type (**Particles.Type** = 0), while particles for the data type **Native VTK Poly Data** may be visualized with spheres (**Particles.Type** = 1). To distinguish these two choices, the name of the data type is used to qualify the name of the property, **ifrit.Window[0].Particles["Particles"].Type** = 0, but **ifrit.Window[0].Particles["NativeVTKPolyData"].Type** = 1 (in short form these two properties would appear as **ifrit.w[0].p["p"].t** and **ifrit.w[0].p["vtkp"].t**).

1.2 Running IFrIT

1.2.1 Starting and controlling IFrIT

The behaviour of IFrIT is determined by the shell it is starting in. Hence, specifying which shell you want to use is very important. This is done on the command line: the first command ine argument serves as a shell specification. Shell specifications, like all other command line options, begin with the dash (-) symbol. Currently, the following shells are available:

- **`-os`**: an off-screen shell, for running IFrIT in the background,
- **`-cl`**: a command-line shell,
- **`-qt`**: a GUI shell based on the Qt Graphical User Interface (`http://www.qt-project.org`) toolkit - this is the primnary way of using IFrIT.

Only shells that have been compiled into IFrIT during the installation will be accessible. If you do not specify the shell in the command line, a default shell will be used. The default shell is the first compiled-in shell from the folowing order: **qt**, **cl**, **os**.

While there is just one way to run command-line and off-screen shells, a GUI shell can be used in two different regimes, depending on the roles IFrIT and Python play in their relation to each other - one is the master and another is a servant. Therefore, the GUI shell specification can be optionally appended with "/embed" or "/exten" (so that the full specifications is **`-qt/embed`** or **`-qt/exten`**) to explicitly request an *embedding* or *extending* mode. (Pythonians are familiar with the terminology of *extending* and *embedding*).

In the *embedding* mode IFrIT (a master) completely embeds and controls a Python interpreter (a servant) as its intergral part - this behaviour is equivalent to IFrIT 3 mode of operation. No command line is available in the console window, but a special Script Window provides a mini-IDE for the script, with a built-in editor, execution environment, and, optionally, a simple debugger (the latter is available if the script provides debugging hooks). This is the default mode for the GUI shell, as it provides the most control over the script execution.

In the *extending* mode IFrIT executable is starting a Python interpreter, which, in turns, starts a shell - i.e. Python becomes a master and IFrIT a servant. From the outside it appears as a Python command-line interpreter with IFrIT imported automatically. The only restriction is that when IFrIT exits, the Python interpreter exists as well. Extending mode is requested by appending the shell specification with "/exten".Thus, in the extending mode the command-line is available even in the GUI shell (except on Windows, where the console window is not automatically displayed by the operating system). This is the default mode for the command-line shell, but in the GUI shell the Script Window is not available - it is very cumbersome to run Python both as a master to IFrIT and as a servant to IFrIT which is itself a servant to the main Python interpreter.

Finally, one more method of running IFrIT is possible. In the *modular* mode IFrIT is running as a true Python module - i.e. you would start a Python interpreter, then call **import ifrit** to import the module, and then start IFrIT by calling **ifrit.start()** to launch a default shell or **ifrit.start(<shell>)** to start a specific shell, where <shell> is a Python string with the two-letter shell id ('cl' or 'qt'). IFrIT then starts in the extending mode, but it does not own the Python interpreter and stopping IFrIT will not close Python. In the modular mode IFrIT can either be stopped by exiting the main window in the GUI shell or by executing **ifrit.stop()** command in the Python interpreter. After IFrIT was stopped, it can or cannot be started again, depending on exact setup - Qt4 and Qt5 have a bug that prevents restarting a Qt application after it was closed, so if IFrIt was compiled with Qt4 or 5, and ran in the GUI mode, it could not be restarted again without restarting Python. This mode can be thought of Python being a king and IFrIT is just one of numerous subjects that does not interfere with the life of the king in any way but lives and dies as king commands.

The modular mode is available only when a shared library **ifrit.so** (**ifrit.pyd** on Windows) is compiled and placed in a location where Python can find it - check the installation instructions for details.

IFrIT can also function without any scripting language in a GUI shell. However, without a script only the built-in animation capabilities are available. The scripting language is, therefore, a useful tool to have around. To use it with IFrIT, it, obviously, must be installed on your system. Both Python 2.7 and Python 3 should be supported, but earlier versions may not work.

When IFrIT starts and while it is running, it can be controlled by several methods. The main method of controlling IFrIT is by using a shell to manipulate various objects. In a command-line shell that is achieved by issuing Python commands, like **ifrit.Window[0].Camera.Roll(10)** or **ifrit.w[0].bb.t = 2**, etc. In a GUI shell the same commands are issued by manipulating various GUI widgets with a mouse.

In addition, IFrIT can be controlled by using

- Mouse clicks and motions in the visualization window,
- Environment variables,
- Command-line options, and
- External files.

1.2.2 Mouse and Keyboard Controls

IFrIT reaction to mouse clicks and movements depends on which of three major mouse interaction modes it is in.

- **Display mode** is the default mouse interaction mode. In this mode the following binding of mouse and keyboard are available:

  Left button: rotates the camera around its focal point by moving the mouse.
  [Ctrl] key + **Left button**: rotates the camera around the Z-axis (axis perpendicular to the screen).
  Middle button: pans (moves in space) the camera.
  Right button: zooms the scene in and out.
  3 key: toggles into and out of stereo mode.
  P key: performs a pick operation (i.e. selects a point in the scene the mouse cursor is pointing to, and retrives information about the data at this location).
  R key: resets the camera view along the current view direction.
  S key: modifies the representation of all actors so that they are surfaces.
  W key: modifies the representation of all actors so that they are wireframe.

- **Fly-by mode** emulates flying an airplane through the current visualization scene.

  Left button: moves forward.
  Right button: moves backward.
  [Shift] key: accelerator in mouse and key modes.
  [Ctrl]+[Shift] keys: causes sidestep instead of steering.
  +/- keys: increases/deacreases normal speed.

- **Measuring box mode** becomes active when a measuring box is shown in the current visualization window. The box can be used to measure sizes of different features in the scene.

  Left button: rotates the measuring box around its origin.
  Middle button: translates the measuring box in/out of screen.
  Right button: translates the measuring box along scene axes.
  A/Z keys: scales the measuring box down/up.

S/X keys: adjusts box opacity.
C key: shifts the camera focal point to the box center.

- **Keyboard Interactor mode** uses keyboard to emulate all mouse interactions. The keyboard mode is slow, but precise and reproducible. In this mode the following keys are available:

  Left/Right (or **H/L**) keys: rotates the camera horizontaly around its focal point.
  Down/Up (or **J/K**) keys: rotates the camera vertically around its focal point.
  +/- keys: zoom the scene in/out.
  A/Z keys: slow down/speed up the interaction.
  Other key binding are the same as in the **Display** mode.

- **Common to all modes**:

  F key: switches in and out of the full screen mode (will not work in all shells).
  U key: dumps the current view of the scene into an image file on disk.

1.2.3 Environment Variables

IFrIT understands the following environment variables (all in capitals):

- **`IFRIT_DIR`**: The main IFrIT directory where you keep the state file(s) **ifrit.ini** and possibly other configuration files. If this variable is not set, IFrIT will check the environment variable HOME. If it exists, it will make $HOME/.ifrit the main directory (and will create that directory if it does not exist). If the environment variable HOME is not defined, IFrIT will try to create a main directory at /.ifrit. If everything else fails, IFrIT assumes that current directory is the main directory.
- **`IFRIT_SCRIPT_DIR`**: The directory where IFrIT keeps animation and control scripts. If not set, IFrIT will put scripts into its main directory.
- **`IFRIT_PALETTE_DIR`**: The directory where IFrIT keeps custom palettes. If not set, IFrIT will put palettes into its main directory.
- **`IFRIT_IMAGE_DIR`**: The directory where IFrIT keeps image and animation files. If not set, IFrIT will put image files into the current directory.
- **`IFRIT_DATA_DIR`**: The default directory for the data files. If not set, IFrIT will start searching for the data files in the current directory.
- **`IFRIT_SCALAR_FIELD_DATA_DIR`**: The default directory for the scalar data files. If not set, IFrIT will start searching for the scalar data in the default data directory.
- **`IFRIT_VECTOR_FIELD_DATA_DIR`**: The default directory for the vector field data files. If not set, IFrIT will start searching for the fector field data in the default data directory.
- **`IFRIT_TENSOR_FIELD_DATA_DIR`**: The default directory for the tensor field data files. If not set, IFrIT will start searching for the tensor field data in the default data directory.
- **`IFRIT_PARTICLE_SET_DATA_DIR`**: The default directory for the particle data files. If not set, IFrIT will start searching for the particle data in the default data directory.

1.2.4 Command-line Options

Several options can be specified in the command line when invoking IFrIT. Options begin with the dash (-) symbol, but the first option is special: it invokes IFrIT with a specific shell.

Different shells support different options, so for the full list of available options you need to refer to the documentation for the specific shell, or run ifrit with the `-help` option. However, all shells support the basic subset of command-line options:

- **`-h/-help/--help`**: shows the list of available command-line options, including those specific to a particular shell.
- **`-np/--num-procs`** *number*: sets the number of processors to *number* for parallel execution.
- **`-i/--init-file`** *filename*: loads the full internal state from the previously saved state file *filename*.
- **`-GPU/--force-GPU`**: forces IFrIT to try to use GPU even if VTK does not find it.
- **`-no-GPU/--force-no-GPU`**: forces IFrIT not to use GPU even if it is present on the system (useful for running IFrIT remotely).

If the last entry on the command line does not form any option, it is taken to be the name of directory; IFrIT will use this directory as his default DATA directory, overwriting the environment variable.

1.2.5 State File

IFrIT can remember its exact state and save it into the **ifrit.ini** file, which is placed into main IFrIT directory (see Environment Variables). This file is not intended to be modified by the user. When IFrIT starts, it looks for the **ifrit.ini** file in its main directory. If the file is found, IFrIT will read its internal configuration from it. Thus, if you spent a long time changing various settings and adjusting IFrIT to your needs, all you need to do is to save the state, and the next time you start IFrIT, all your settings will be restored automatically, and your hard work will not be wasted.

You can keep several of state files and use the appropriate one by specifying its name after **-i** option on the command line.

1.3 File Formats

1.3.1 Overview

IFrIT can visualize four different types of data:

- **Scalar** data: several scalar variables in 3D space.
- **Vector field** data: a 3D field of vectors.
- **Tensor field** data: a symmetric 3x3 tensor in 3D space.
- **Particle** data: a set of particles (points) with several optional variables (numbers that distinguish particles from each other) per particle.

Each class of data may contain more than one specific **type** of data. We use the word "type" here in the same

sense as it is used to describe different data types in computer language (integer, boolean, real, etc). Thus, each data **type** is a unique representation of a specific layout of data in space, and is associated with a specific format of a data file. For example, the built-in Scalars data type describes several scalar variables on a uniform rectangular mesh in space, loaded with a specific file format. What data types available for each class depends on what extensions of IFrIT are included in your installation.

Each data **type** has one-to-one correspondence with an identically (except for skipped white spaces) named data object. For example, the data object that corresponds to the built-in Scalars data type is named **Scalars**.

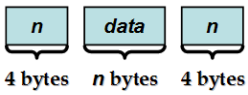
The standard edition of IFrIT includes four built-in data types:

- **Scalars** data file: uniformly spaced 3-dimensional mesh of data values with several scalar variables per file.
- **Vectors** data file: a 3D vector field specified on a uniformly spaced mesh.
- **Tensors** data file: a symmetric 3x3 tensor field specified on a uniformly spaced mesh.
- **Particles** data file: three positions and several optional variables for a set of particles (or points).

Extensions of IFrIT also include other data types, as listed in the Object Reference.

Binary Files

IFrIT can load both the plain text (ASCII) and binary data files. The binary files use **Fortran unformatted** file formats - these files can be created in many programming languages, including Fortran, C, and IDL (code examples are given in Appendix A). Fortran writes binary data into a file in **records**. Each record contains a 4-byte header, a body of the record, and a 4-byte footer, as shown in the image below:



Structure of Fortran record

The header and the footer are identical, and each contains a 4-byte integer number with the number of bytes in the body of the record. The body can contain any data. Because the length of the header and the footer are 4 bytes, one record cannot contain more bytes that can be described by a 4-byte signed integer number (2147483647). It is important that the header and the footer of the record be correct (i.e. contain the number of bytes in the body). IFrIT uses the header and the footer to verify the integrity of the data file and to deduce the endiness of the data.

File Sets

IFrIT can load several data files of different formats simultaneously as a "set". For example, if you are visualizing the built-in Scalars data and Particles data from a series of outputs from the same simulation, you can load files in pairs (or triplets if you add, for example, a Vectors file, etc). Only Animatable Files can be loaded as sets. For example, if you load a Particles file named part_1234.bin, and then load a Scalars file named scal_1234.bin (the same 4-digit record label), IFrIT will recognize these files as a set. The Particles file becomes a **set leader**, and you will be able to load files in sets: if you load a new leader (another Particles file, say, part_5678.bin), then the corresponding Scalars file (scal_5678.bin) will be loaded automatically. Sets can, of course, include data files of all 4 formats.

If one of the files in a set does not exist, or you load an individual file rather than a set, the set will be broken and IFrIT will treat files as unrelated.

Subdirectory `docs` of IFrIT source distribution contains three files: `writeIFRIT.f`, `writeIFRIT.c`, and `writeIFRIT.pro` that contain Fortran, C, and IDL code respectively for writing all four types of IFrIT data files. These files are also listed in Appendix A.

Cell vs Point Data

In order to understand the placement of the data in the scene, you need to know the difference between the cell and point data. Data can be specified on a uniform mesh in two distinct ways. If every cell of a mesh is represented as a cube, the data can be specified either at the center of a cube (**cell data**), or at the vertices of a cube (**point data**). Most of VTK classes can only handle point data, while most simulation codes place the data at cell centers. To handle this inconsistency, IFrIT allows to specify the placement of the data and converts the cell data into the point data automatically by appropriately shifting the data relative to the bounding box.

If the data in the data file is the point data, then nothing needs to be done and IFrIT can use it directly. In that case, though, you have to make sure that the data descriptions in IFrIT and in the file are consistent. For example, if you specify periodic boundary conditions, the data values on the opposite sides of the data cube should be identical. If they are not, then extending data periodically beyond the bounding box will create visualization artifacts.

If your file contains the cell data, IFrIT treats cell centers as vertices of a new uniform mesh; corners of this new mesh would not, in general, coincide with corners of the bounding box, but all viualization methods would work properly with your data.

You can thing about cell data as *filling the space completely*, so cell data provides a value for every spatial point, even those points that do not lie on the grid. Point data is, instead, *sampling the data* on a et of discreet point; interpolation is inevitable when using the point data.

1.3.2 Built-in Scalars Data

IFrIT Scalars data file contains uniformly spaced 3-dimensional mesh of data values. Both, plain text (ASCII) and binary (Fortran-type binary of any endiness) files are accepted.

Plain text file. This file should have an extension "txt" (as in "myfile.txt", lower-case or upper-case does not matter) and should contain in its first line three integer numbers: the sizes of the mesh in X, Y, and Z directions. These dimensions do not have to be the same. Each line after the first one should contain up to 10 floating point numbers as values for the physical variables at each cell of the mesh. The first dimension changes the fastest. For example, the following file:

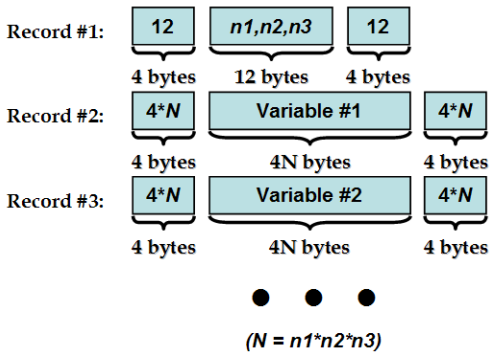
```
12 33 55
1.0456 4.56768 2.45e-30
0.9866 5.45890 3.07e-20
  ...
```

(12*33*55+1 lines altogether)

defines a 12 by 33 by 55 data mesh with three physical variables.

Binary file. This file should have an extension "bin" (as in "myfile.bin") and should be a **Fortran unformatted** binary file. The data in the file must be in **single precision** (Fortran `real*4`, C `float`). The file should contain at least 2 records. The first record should contain 12 bytes of data as 3 integer numbers: mesh sizes in three dimensions (`n1`, `n2`, and `n3`). The remaining records should contain `n1*n2*n3` floating point

numbers each: one scalar variable per record.



Structure of the IFrIT Scalars data binary file

Examples of the code that can write such files is given in Appendix A.

1.3.3 Built-in Vectors Data

IFrIT Vectors file format is identical to the built-in Scalars data file format with three vector components stored as three scalar variables.

1.3.4 Built-in Tensors Data

IFrIT (symmetric) Tensors data file format is identical to the built-in Scalars data file format with 6 tensor components stored as 6 scalar variables in the following order: 11, 12, 13, 22, 23, 33.

1.3.5 Built-in Particles Data

IFrIT Particles data file contains three positions and, optionally, several variables for a set of particles. Both, plain text (ASCII) and binary (Fortran-type binary of any endiness) files are accepted.

Plain ASCII file. This file should have an extension "txt" (as in "myfile.txt", lower-case or upper-case does not matter) and should contain in its first line one integer number: the total number of particles in the file. The second line should contain 6 numbers that determine how the particle positions relate to the bounding box. The first three numbers are X, Y, and Z coordinates of the lower-left-back corner of the bounding box in the units in which particle coordinates are given. The last three numbers are X, Y, and Z coordinates of the upper-right-front corner. For example, if you ran a simulation with a cubic box of 5 meters in size, and your particle positions are given in meters, and the left-lower-back corner of your box has coordinates of (0,0,0) meters, then the second line should be: `0.0 0.0 0.0 5.0 5.0 5.0 X-label Y-label Z-label`. Optionally, three strings are allowed at the end of the line to use as axis labels if the bounding box is displayed as 3-dimensional axes. This is useful for making 3D scatter plots.

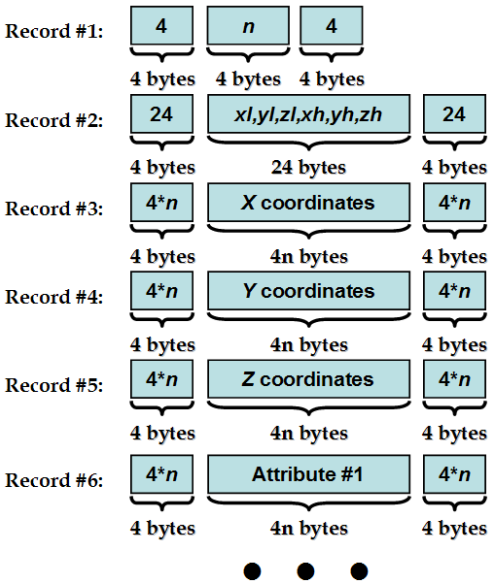
Each line after the second one should contain three floating point numbers as values for the three coordinates for each particle and, optionally, up to 10 more numbers as values of variables. The variables can be used to distinguish particles in a set. For example, the following file:

```
120
0.0 0.0 0.0 5.0 5.0 5.0
1.0456 4.56768 3.05678 2.45e-30 1.11e+10 0.555
0.9866 5.45890 -2.0568 3.07e-20 2.44e+11 -0.34
  ...
```

(120+2 lines altogether)

defines a set of 120 particles with three variable fields defined. Notice that the second particle is located outside the bounding box - there is nothing wrong with that.

Binary file. This file should have an extension "bin" (as in "myfile.bin") and should be a **Fortran unformatted** binary file. The file should contain at least 5 records. The first record should contain 4 bytes of data as 1 integer number: the number of particles `n`. The second record should contain 24 bytes as 6 floating point numbers for 6 values of the bounding box. Records from 3 to 5 contain `n` single or double precision floating point numbers (4*`n` or 8*`n` bytes) each, which are x, y, and z coordinates of particles (i.e. all x coordinates are stored in record 3, all y coordinates are stored in record 4, etc). Optional remaining records contain `n` single precision floating point numbers with particle variables (scalar values that characterize individual particles).



Structure of the IFrIT Particles data binary file

Examples of code that can write such files is given in Appendix A.

1.4 IFrIT Palettes

1.4.1 Built-in Palettes

IFrIT includes a set of pre-defined *palettes* (or *color-maps*), i.e. a one-dimensional sequences of colors that are used to map scalar values to a color of a point in the scene. Palettes identified by their numbers:

1: Rainbow

2: Isolum

3: Ametrine

4: Temperature

5:  Blue-white

6:  Prizm

7:  Starlight

8:  Green-white

9:  Blue-red

10:  Haze

11:  Stern

12:  Greyscale

13:  3 color

14:  4 color

Each palette can be used in its original form or "reversed", i.e. with colors changing from right to left. In addition, new palettes can be created using **Palette:New()** method.

Some objects also use a special palette called "brightness", which includes shades of different intensity for the current object color. For example, if you the color of the object is red, then using the brightness palette will color the object with shades of red of varied brightness. This palette can be useful, for example, for coloring particles with varied intensity of yellow color, to represent stars of different magnitudes.

1.5 Animation Support

1.5.1 Animation Mode

In the Animation mode IFrIT works automatically to generate an animation of your scene. You cannot control IFrIT interactively in this mode and you cannot change the scene - although IFrIT can change the scene for you automatically. By default, the output of an animation is a series of image files that show consequitive snapshots of the visualization scene. Animation images have **frame** as the root of their names and they are distinguished by a 5-digit sequential number; each files for a given animation go into the subdirectory **anim**, i.e. the results of an animation will be called:

```
anim/frame-00001.jpg
anim/frame-00002.jpg
anim/frame-00003.jpg
...
```

(up to 99999 images can be created). In this case it remains your task to convert the set of images into an animation format of your choice (many image viewers will also have a slide show feature, so you can simply display the set of images as a slide show too). VTK also has a capability has been added to create MPEG (if it was linked with the MPEG library) and AVI (on some platforms) movies - the choice is controlled by the **ImageWriter.MovieOutput** property.

To avoid overwriting already created files, subsequent animations from a single IFrIT run will go into subdirectories **anim2**, **anim3**, etc.

Animation mode is only available if the name of the data file has a specific form. Even if you are going to use just one file for your animation (for example, for a fly-by through a fixed scene), you still need to name your data file in a standard way so that IFrIT can understand it as belonging to the animation series - i.e. as an animatable file.

1.5.2 Animatable Files

Animations are normally made from a series of files. IFrIT will automatically read new files in order to create an animation. The file names **must** be in the following format for IFrIT to find them:

```
[any_string_as_prefix]NNNN.suf
```

where `suf` is `txt` for plain text files or `bin` for binary files (see Supported Files Formats), NNNN is a 4-digit number (from 0000 to 9999) called **record number**, and the file name can contain any string before that as a prefix. For example, the following file names will be recognized by IFrIT as series:

```
myfile0345.bin
var_9988.bin
0564.txt
```

whereas the following file names are valid names for a single data file, but cannot be used in making an animation:

```
myfile345.bin
var_9988a.bin
0564var.txt.
```

Files that belong to a series are called **animatable**.

If IFrIT recognizes the data file as a member of a series, it will first complete all the operations requested for the current file, and then will automatically load the next file in the series. The files in the series do not have to be numbered sequentially. For example, if the series contains only two files, `myfile0345.bin` and `myfile0817.bin`, IFrIT will load `myfile0817.bin` right after `myfile0345.bin`, even if many numbers in between are missing. IFrIT will not leave the Animation mode until all the files in the series are visualized.

1.5.3 Running Animator

Animator object runs in the following way: for each animatable file (each record) it modifies the displayed scene based on its internal settings (for example, if the **Animator.Style** property is set to 1 (rotate/scale, the scene will be rotated and/or scaled based on the values of **Animator.Phi**, **Animator.Theta**, **Animator.Roll**, and **Animator.Zoom** properties) for a number of steps (the actual number is set by the **Animator.NumberOfFrames** property). After that number of frames is produced, the next animatable file (i.e. next record) is loaded. If the record is absent, the animation stops.The built-in animation support is rather limited, hence one needs to use a script to achieve a complete control of IFrIT objects. While there are many ways the **Animator** object can be run from a script, it offers a convenient function **Animator.Continue()** that implements the main animation driver.

The built-in behaviour is equivalent to a trivial loop over this function (using Python as the scripting language):

```
while ifrit.Window[0].Animator.Continue(): pass
```

but the trivial loop can be, of course, modified to an arbitrary complexity. For example, the following Python segment

```
win = ifrit.Window[0]
win.Animator.Reset()
level = 0.1
while win.Animator.Continue():
    win.Surface[0].IsoSurfaceLevel = level
    level = level*1.01
```

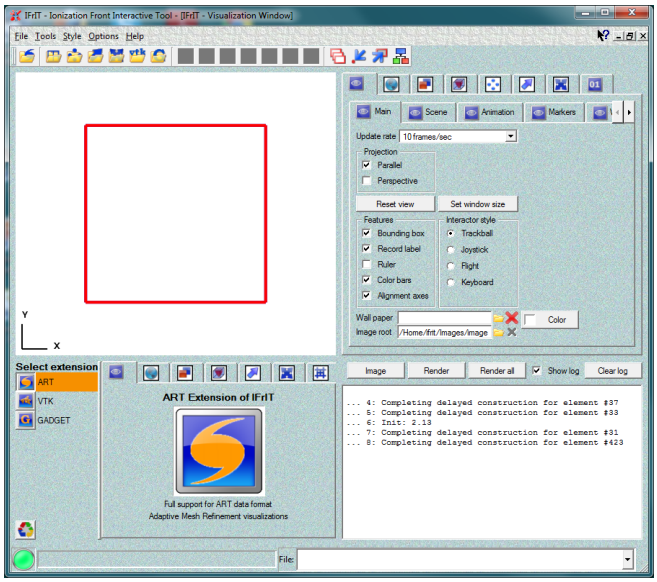
first resets all internal **Animator** settings, and then modifies the level of the isosurface for every subsequent animation frame.

Two convenience functions **Animator.CurrentFrame()** and **Animator.CurrentRecord()** can be used to query the values of the current record and the current frame in the animation loop.

2 Shell Reference

2.1 GUI Shell Reference

2.1.1 Graphical User Interface (GUI) Shell



The GUI shell provides a fast access to all object properties through a set of GUI **widgets** (on-screen control elements). All widgets have a dynamic help feature: pressing [Cntr] and F1 keys together while a mouse cursor points on a particular GUI widget will pop-up a small help window for that widget.

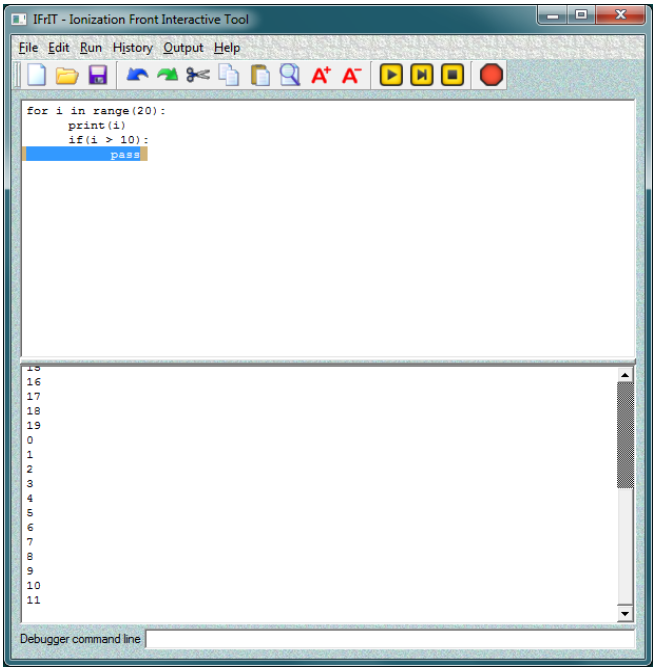
A GUI shell also accepts several additional command-line options:

- **`-d`**: start IFrIT with all windows docked together; this is equivalent to starting IFrIT and then docking windows using the corresponding menu option.
- **`-fs <number>`**: increase (if **`number`** > 0) or decrease (if **`number`** < 0) the size of font in GUI windows; this also increases/decreases window sizes.
- **`-nf`**: do not show a splash window at start-up.
- **`-om/--old-style-window-manager`**: this option instructs IFrIT not to expect that the window manager is modern and has a close window button on top of a window; if this option is specified, IFrIT will place a close button on dialog windows.
- **`-sd/--small-desktop`**: this option tells IFrIT that the size of the desktop is small, so that it should not arrange windows on the desktop; by default, IFrIT assumes that the desktop is small if its height is less than 900 pixels and its width is less than 1200 pixels.
- **`-sc/--slow-connection`**: this option tells IFrIT that it is being run remotely and that the network connection is slow; in that case rendering of the scene is not automatic, instead either a **Render** button (that appears at the bottom of the main window) needs to be pressed or the visualization window must be clicked on to render the current scene.
- **`-ddi/--disable-delayed-initialization`**: by default IFrIT starts with minimal initializations, postponing expensive operations of creating various dialogs and pages only when these dialogs and pages are actually used. This feature is experimental; if segfaults occur during the operation of the

GUI, this option may be used to force complete initialization of all widgets before at startup.

In addition, the GUI shell includes several dialogs that provide additional functionality, as described below.

2.1.2 Script Window



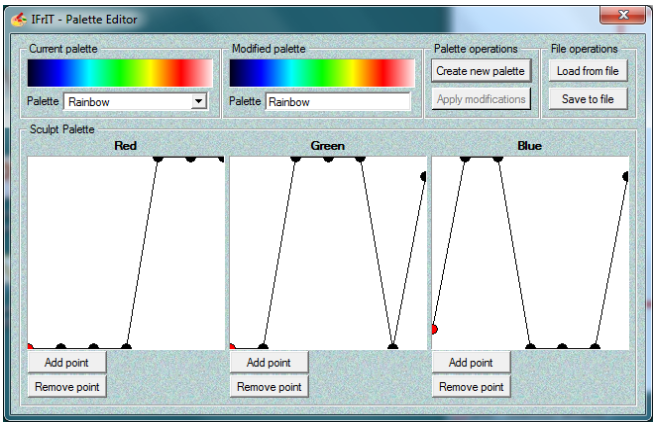
Script Window is a tool designed to run and debugging scripts. It includes a standard editor with usual **New**, **Open**, **Save**, etc file manipulation functions and **Undo**, **Cut**, **Paste** etc text editing functions for editing the script. Script is run by using the **Run** menu or a yellow "play" button. If **Run:Debug script** menu item is checked, minimal script debugging capabilities become available: a break point can be set on a particular script line, and after the script is stopped at the breakpoint, the script can be executed one line at a time. During the execution, the current script line is highlighted as if it was selected in the editor mode, and selection follows the script as it is being executed.

The bottom part of the Script Window is an area where the script stdout and stderr are directed (stderr output will be printed in red).

Finally, in the debugging mode a single editor line appears at the bottom, which can be used to execute simple script commands while the main script is running, like printing variable values or changing the values of variables.

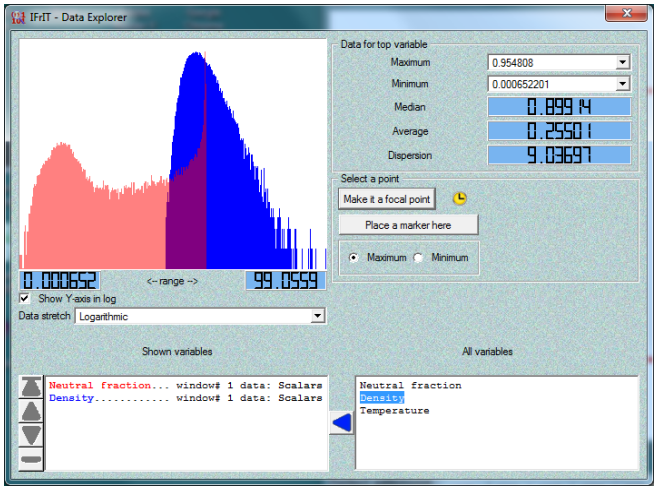
The script segment in the editor window is saved into a history buffer after it is executed. If the segment is replaced with a new segment, the new one will be saved as well, etc, a-la a notebook. The history buffer can be loaded back into the editor from the main menu.

2.1.3 Palette Editor



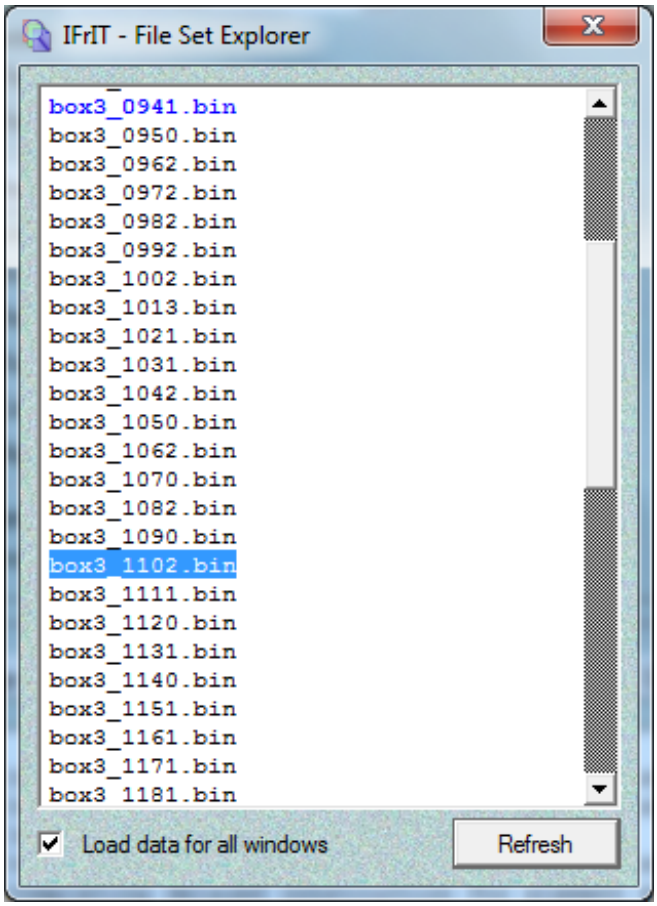
Palette Editor allows to modify existing palettes, create new ones, save or load custom palettes to/from a file, and remove palettes from the list. Editing of a palette is performed using three windows for sculpting a color component for all three colors. Other functions can be invoked by pressing the respective buttons. **Palette Editor** saves palettes into files with extensions ".ipf" (IFrIT Palette File).

2.1.4 Data Explorer



The **Data Explorer** dialog can be used to obtain information about the currently loaded data. The available variables from the "All variables" window can be moved to the "Shown variables" window with a click of a button. The histograms of all shown variables are displayed in the main window, and the information about the top shown variables is listed on the right.

2.1.5 File Set Explorer



File Set Explorer can be used to load an arbitrary member of a file set. The main part of the window is a list of members in the current file set. Clicking with the left mouse button on any member of the set will highlight it. Using the mouse with the left button pressed, several consequitive members can be highlighted at once. Pressing the **`Enter`** key will load selected members of the set one after another. Double clicking on a given member will load this one member too. In addition to using the mouse, a user can also use keyboard navigation (**`up`** and **`down`** keys, **`Page Up`** and **`Page Down`**, and **`Home`** and **`End`** keys) to move along the displayed list. If the **`Control`** key is pressed together with other navigation keys, the selection will change instead. If the **`Load data for all visualization windows`** box is checked, IFrIT will try to load corresponding members (with the same record number) for all sets displayed in various visualization windows. For example, if window #1 shows the data from a file `aaa_0010.bin`, and window #2 shows the data from a file `bbb_0015.bin`, then loading the file `aaa_0020.bin` in window #1 will also cause loading of the file `bbb_0020.bin` in window #2 (if such a file exists).

2.1.6 Image Composer

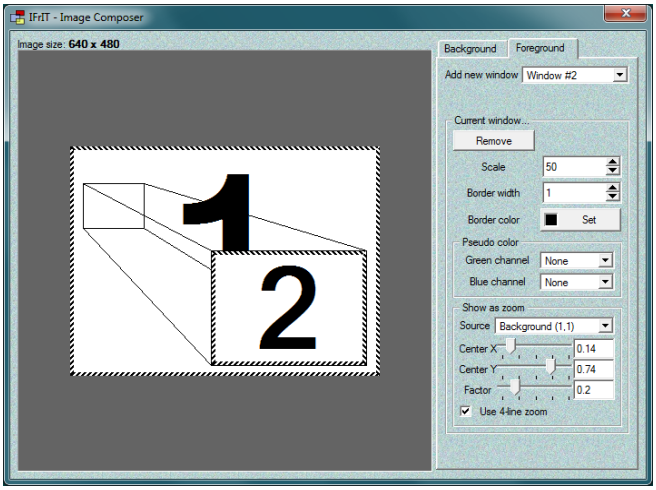
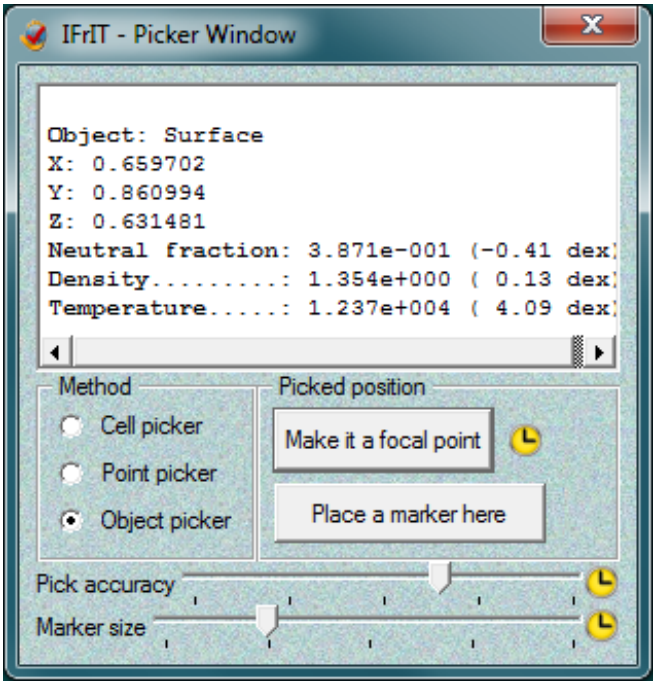


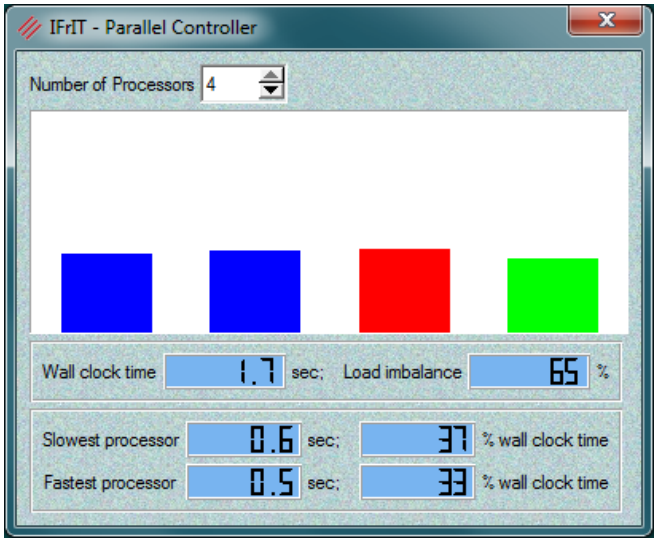
Image Composer dialog is a tool that allows you to organize several visualization windows into one image (both as a single image and as a part of an animation). **Image Composer** has two layers: a background and a foreground. A small book (tab) widget to the right of the drawing area gives you controls for these two layers. Background layer consists of regularly tiled images from one or several visualization windows. You van change the number of tiles in the horizontal and vertical directions, assign a given visualization window to a given tile (or not assign any window at all), add a border to the full image with a selectable color, and specify whether the border only surrounds the whole image or each individual tile as well. Tiles that have no visualization window attached to them can be filled with a background image that should be read from an external file. Usual IFrIT image formats are understood (PNG, JPEG, PNM, BMP, and TIFF). Foreground layer consists of freely floating "inserts" that must be associated with a visualization window. Inserts can be moved around by clicking on them with the left mouse button and dragging them around. They can be scaled to a fraction of their original size and added a border of a predefined color. Check the Reference Guide for the **Image Composer** object for more information.

2.1.7 Picker Window



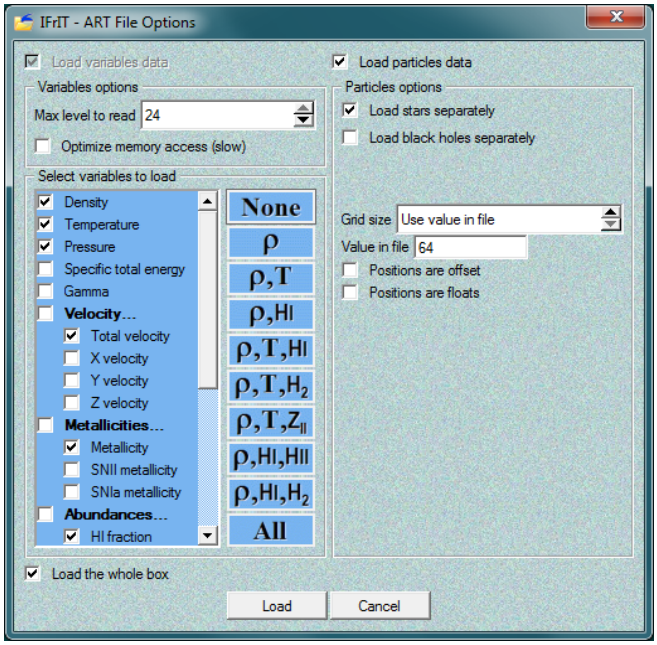
Picker Window is brought up during the pick operation - when you press the "P" key in the visualization window. If anything is picked in your scene, the **Picker Window** will appear, and a small marker sphere will be placed at the picked position. The **Picker Window** has a slider to control the size of the matker and a check box to select hardware vs software picking. The hardware picking is much faster, but may not work properly when translucent objects are present in the scene.

2.1.8 Parallel Controller



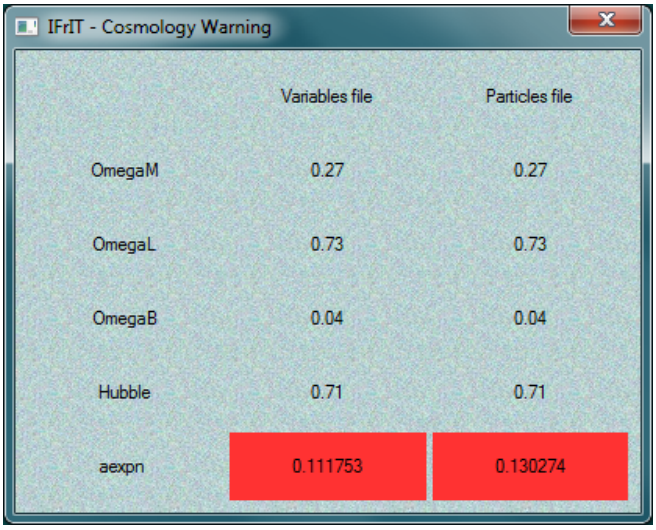
Parallel Controller can be used to monitor the parallel performance. Most of IFrIT operations can be done in parallel if more than one processor is available. The maximum possible number of processors that IFrIT will use is set by `-np` command-line option.

2.1.9 ART File Options



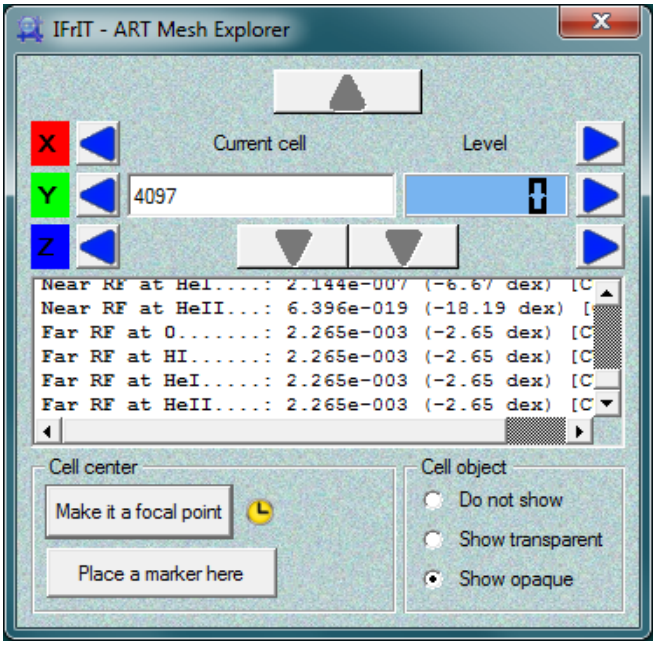
When opening ART data files, several specialized options are available via the ART **File Options** dialog: limiting the number of mesh levels read, loading the particle data together with the mesh data (IFrIT will recognize both old - PMcrd*.DAT - and new - *.dph - file names), selecting a subset of mesh variables to load, optimizing the placement of the data in memory for better cache access (the latter takes extra time and temporarily uses about 20% more memory; the cache optimization option only makes sense to use if the user plans to do detailed analysis of a given data file), and overwriting the particle parameter NGRIDC set in the particle data file (this parameter may not be set correctly, but IFrIT uses it to scale particle coordinates to OpenGL coordinates).

2.1.10 ART Cosmology Warning



The ART **Cosmology Warning** dialog reports a discrepancy between the cosmological parameters in the Variables and Particles data files.

2.1.11 ART Mesh Explorer



The ART **Mesh Explorer** dialog allows to traverse the ART mesh cell-by-cell horizontally (in space) as well as vertically (over levels). It provides information about a given cell, and is complimentary to the **Picker** object, which provides information about a specific spatial location.

2.2 Non-GUI Shell Reference

2.2.1 Command-line Shell

The command-line shell is simply an interface to the underlying script interpreter. For convenience, an additional function `inc(file)` is defined for the interpreter, that loads and executes the file `file`. It is introduced primarily for portability, and is an analog of `execfile` in Python 2, which Python 3 unfortunately

is misisng. As a bonus, if the argument `file` starts with `+`, it is prepended with the value of IFRIT_DIR environment variable (if set).

2.2.2 Off-screen Shell

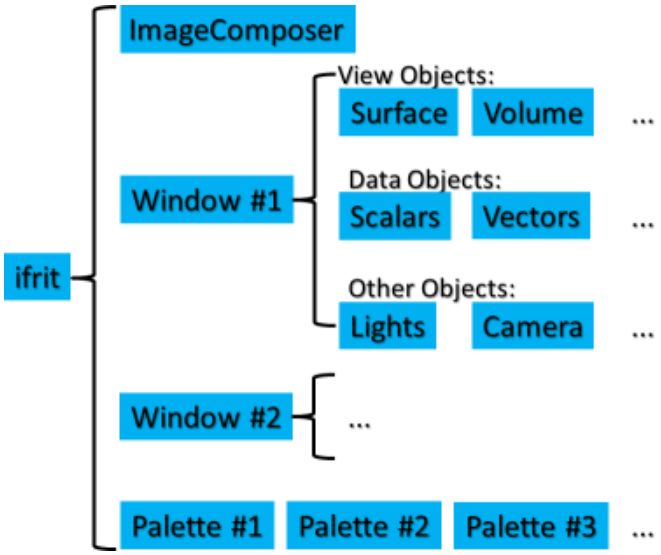
The off-screen shells runs IFrIT in the background, and, hence, does not offer any direct interaction with visualization objects. In order to make visualization, a file with the script should be supplied as the first non-option argument (i.e. the first argument not starting with the dash -). Be aware, however, that on Unix it still requires a valid X11 display to be present - i.e. you can use that shell to run IFrIT in the background of an X-terminal, but not on a "headless" worker node of a cluster that does not have an OpenGL installation. For the latter mode, you will need to (i) install Mesa software rendering library (and be praised if you succeed), (ii) compile VTK with Mesa rather than your native system OpenGL library, and (iii) link IFrIT to that VTK installation. Only then the off-screen shell will work without an active X11 display. Here is some info on how this is achieved for ParaView, which also uses VTK:
`http://www.paraview.org/Wiki/ParaView/ParaView_And_Mesa_3D`, but beyond that you are entirely on your own.

3 Object Reference

3.1 Overview

3.1.1 Objects and Properties

The root of the object hierarchy is the **ifrit** object; it serves as a container to other objects and performs a few other miscellaneous functions. The diagram below shows the hierarchy of various objects (the complete hierarchy is shown in Object Reference):



The **ifrit** object includes one or more **Window** objects. A **Window** object appears as an IFrIT visualization window on the screen; it forms an independent part of IFrIT that has its own set of other (sub-)objects and, often, its own data. Some of **Window** objects may share the data with another **Window** object - in that case the owner of the data is called a "parent" **Window** object, and sharing **Window** objects are called "clones". Irrespectively of whether a given **Window** object is a clone or a parent, it has a complete set of its own (sub-)objects.

Window objects use other objects for various functions. For example, some objects are responsible for a general setup of the visualization scene, such as lighting (**Lights** object), camera (**Camera** object), bounding box (**BoundingBox** object), clipping plane(**ClipPlane** object), and various accessories (such as rulers, labels, measuring boxes, etc).

Other objects perform operations on various components of the visualization scene. For example the **Surface** object represents a two-dimensional surface within the visualization scene that samples the three-dimensional scalar data (either as an isosurface of a particular scalar variable, or a specified geometric shape like a sphere or a plane), while the **Particles** object represents a set of particles (points).

Yet another group of objects perform various functions that are not directly represented in the visualization scene. For example, the **DataReader** object is responsible for loading data files into IFrIT, while the **Animator** object creates animations of your visualization scene in a diverse variety of ways.

Several objects do not belong to one of the **Window** objects, but rather communicate with all of them. An **ImageComposer** object is responsible for composing a snapshot or an animation image from several windows. An array of **Palette** objects represents palettes used by IFrIT. The complete hierarchy of all objects is presented in the Object Reference Guide.

All object are controlled by **properties**. You can think of a property as of a special variable: as soon as you assign a new value to the variable, something happens. For example, if you assign a value `true` to the property **BoundingBox.Visible**, the bounding box will appear in the visualization scene. Assign `false` to that property, and the bounding box disappears.

Properties can be controlled in two ways. A GUI shell uses GUI elements (widgets) to directly set or get a value of a property. Alternatively, a shell can use a scripting language and access object properties as script variables. In a command-driven shell a script is a must - otherwise, IFrIT cannot be controlled; in a GUI shell a script is optional, but without a script animation capabilties will be rather limited. Currently only Python is supported as IFrIT scripting language (the primitive script of IFrIT 3 is no longer supported), but other languages may be added in the future (Python is popular, but it is kind of an overkill for animation scripting, and its syntax is too restrictive for efficient access to object properties).

In addition to properties-variables, some objects also have **methods** (C++ users will find terminology familiar) - script functions (rather than variables). For example, **Window.CloneOf()** method returns the index of the parent window if this **Window** object is a clone of another one, or invalid index if it is not a clone.

On a command line a given object property is accessed in a way consistent with the currently used scripting laguage syntax. Using Python as an example, the complete name of the property **Visible** of the object **BoundingBox** that belongs to the first **Window** object is addressed as **ifrit.Window[0].BoundingBox.Visible**. A **BoundingBox** belonging to the second window is accessed as **ifrit.Window[1].BoundingBox.Visible**, etc. Of course, in Python one can always assign a name to any object, so the assignment `b = ifrit.Window[0].BoundingBox` will let the user to access the property **Visible** simply as **b.Visible**. Never-the-less, full object names can be very long! For example, the type of a screen representation with which to display the particles that represent the built-in **Particles** data is fully referred to as **ifrit.Window[0].Particles["Particles"].Type** - such names will strain the patience of even most hardcore command-line afficionados.

In order to simplify naming objects and their properties, each object and property has a short form for its name as long as a full name. In short form **ifrit.Window[0].Particles["Particles"].Type** becomes **ifrit.w[0].p["p"].t**, which should satisfy even the strictest Linux guru.

3.1.2 Object Hierarchy

- **ifrit**
    - **ImageComposer**
    - **Palette**
    - **Window**
        - ♦ **ARTMesh**
            - ◊ **Material**

- **Animator**
- **BoundingBox**
- **Camera**
- **ClipPlane**
- **ColorBar**
- **CrossSection**
  - **Material**
- **Data**
  - **ARTBlackHoleParticles**
  - **ARTEddingtonTensorField**
  - **ARTParticles**
  - **ARTStellarParticles**
  - **ARTVariables**
  - **ARTVelocityField**
  - **GADGETBoundaryParticles**
  - **GADGETBulgeParticles**
  - **GADGETDiskParticles**
  - **GADGETGasParticles**
  - **GADGETHaloParticles**
  - **GADGETStellarParticles**
  - **NativeVTKPolyData**
  - **NativeVTKScalars**
  - **NativeVTKTensors**
  - **NativeVTKVectors**
  - **Particles**
  - **Scalars**
  - **Tensors**
  - **Vectors**
- **DataReader**
- **ImageWriter**
- **Label**
- **Lights**
- **Marker**
  - **Material**
- **MeasuringBox**
- **Particles**
  - **Material**
- **Picker**
- **Ruler**
- **Surface**
  - **Material**
- **TensorField**
  - **Material**
- **VectorField**
  - **Material**
- **Volume**
  - **Material**

3.2 Regular (Non-Data) Objects

3.2.1 ARTMesh object

Short form: **am**

ARTMesh is a view object that shows a representation of the Adaptive (AMR) Mesh under the ART extension. The mesh can be shown either as a collection of dots representing centers of computational cells, or as a mesh frame consisting of edges of all or a subset of computational cells.

Available properties:

- **Color** (short form: **c**); `color[*]`
  Properties **Color** and **Opacity** set these two properties for visualization objects, which are represented as solid surfaces (OpenGL polygonal mesh) - which are all visualization objects except the **Volume** object. These two properties can be either scalar (1-component) or arrays (multi-component), depending on the type of the visualization objects they belong to (check descriptions for specific objects for more details). The **Color** property takes a 3-component RGB value `int,int,int` (like 255,0,0 for red). The **Opacity** properties takes a floating point number between 0 and 1.
- **LevelColor** (short form: **lc**); `color[25]`
  The color of a given level of the mesh.
- **LineWidth** (short form: **lw**); `int[*]`
  The width of a line representing the mesh cell of a given level in grid mode.
- **MaxLevel** (short form: **maxl**); `int`
  MaxLevel and **MinLevel** properties specify the range of mesh levels to show.
- **Method** (short form: **m**); `int[*]`
  A method for visualizing the AMR mesh. Two values are currently supported:
  - **Method=0** (point mode) will display the mesh as a set of points, each representing a center of a single cell,
  - **Method=1** (grid mode) will show the outline of each cell, like a mesh skeleton.
- **MinLevel** (short form: **minl**); `int`
  See MaxLevel.
- **Opacity** (short form: **o**); `float[*]`
  See Color.
- **PointSize** (short form: **ps**); `int[*]`
  The size of a point representing the mesh cell of a given level in point mode.
- **SampleRate** (short form: **sr**); `int`
  A factor by which the mesh is rarefied. Because a realistic computational mesh is very dense, it would not show well on a limited resolution screen. This factor can be used to show only a subset of all cells, thus making the scene legible.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

Child objects:

- **Material**

3.2.2 Animator object

Short form: **a**

Animator creates a series of snapshots ("animation"). It can modify the scene between the subsequent snapshots in a variery of ways, from plain rotation to following a camera path, and load subsequent data files at specified intervals to follow the evolution of the scene, as explained in Animation Support. The animation can start with a title page, and a small logo image can be displayed in every animation frame.

Available properties:

- **Continue** (short form: **c**); **`bool( )`**
  This function is the main driver for the **Animator** object. Each call advances the animation by one frame; the function returns `true` if the animaton is to continue, or `false` otherwise. It is designed to be used in a script in a `while` loop, like this (for Python script):
  ```
  while Animator.Continue():
  -- make your changes between frames --
  ```
- **CurrentFrame** (short form: **cf**); **`int( )`**
  A query (script function) that returns the current animation frame.
- **CurrentRecord** (short form: **cr**); **`int( )`**
  A query (script function) that returns the current file record.
- **Debug** (short form: **d**); **`int`**
  A boolean property that toggles a debugging mode in which the animator replaces the actual scene (which may be expensive to render) with a simple object, which is fast to render while the script is being debugged.
- **LogoFile** (short form: **lf**); **`string`**
  The name of the file with the image to be used as a logo shown on every animation snapshot image. The image must be in one of the formats understood by VTK (JPEG, PNM, BMP, TIFF, and PNG). Logo images are usually small, if the image in the file exceeds 20% of the animation snapshot image, it will be scaled down. The logo image can be placed in one of the four corners of the animation image with the LogoLocation property, and its opacity (transparency) can be controlled with the LogoOpacity property.
- **LogoLocation** (short form: **ll**); **`int`**
  The position of the logo image on the screen. Integer numbers from 0 to 3 are accepted, as follows:
  - ◊ upper-left corner: LogoLocation=0
  - ◊ upper-right corner: LogoLocation=1
  - ◊ lower-left corner: LogoLocation=2
  - ◊ lower-right corner: LogoLocation=3
- **LogoOpacity** (short form: **lo**); **`float`**
  The opacity of the logo image, given as a floating point number from 0 to 1. Zero opacity makes the logo transparent (invisible), while opacity of 1 makes it solid (non-transparent).
- **NumberOfBlendedFrames** (short form: **nb**); **`int`**
  The number of consequitive frames that are blended together in each animation snapshot. Blending frames creates an impression of gradual transition from one frame to another.
- **NumberOfFrames** (short form: **nf**); **`int`**
  The number of frames for each input data file. This option can be used in several ways. For example, if you have only one file in a series, using this option you can create an animation of rotation or fly-by through your data series. If you have a whole series of data files, by setting the number of frames per file you can control the speed with which the animation is played.

For example, if you have just 30 data files, they will be played in 1 second with a regular MPEG stream. By setting the number of frames per file to 10, you extend your animation for 10 seconds (with 3 frames per second it will still look ok if the differences between subsequent data files are not great).

- **NumberOfTitlePageBlendedFrames** (short form: **tbf**); `int`
  The number of frames during which the title page is blended with the first image of the visualization scene. Blending creates an effect of a title page slowly dissolving to reveal the visualization scene.
- **NumberOfTitlePageFrames** (short form: **tnf**); `int`
  The number of frames during which the title page is displayed at the beginning of the animation.
- **NumberOfTransitionFrames** (short form: **nt**); `int`
  The number of consequitive frames that are blended together when a new data file is read. This property is similar to **NumBlendedFrames**, but only blends frame after a file read, other frames are saved into the image files as they are.
- **Phi** (short form: **dp**); `float`
  Properties **Phi**, **Theta**, and **Roll** control the angles (in degrees) by which the scene is rotated between the two consequitive frames. **Phi** and **Theta** are usual spherical angles, i.e. phi direction is horizontal and theta direction is vertical. **Roll** angle is the angle around the axis parallel to the axis of the camera (an axis perpendicular to the plane of the screen). In the **tumble** mode the direction of rotation changes, so only the total angle of rotation matters.
- **Reset** (short form: **r**); `bool( )`
  An action that resets all internal setting to the default state. This command may be useful when the **Animator** is driven from a script, for ensuring that the script does not accidently inherit any previous settings.
- **Roll** (short form: **dr**); `float`
  See Phi.
- **Style** (short form: **s**); `int`
  A method of transformation of a scene between the two subsequent frames. The first frame of the animation is produced from the current scene. After the first frame, IFrIT can transform the scene for the next frame in one of the four ways:
  - **static (Style=0)**: the scene does not change from a frame to frame. The scene will change only if new data files are loaded.
  - **rotate/scale (Style=1)**: the scene rotates (and optionally scales) during the animation by a predefined amount (set with **Phi**, **Theta**, **Roll**, and **Zoom** properties).
  - **tumble (Style=2)**: the scene tumbles in a random fashion during the animation, i.e. it rotates (and optionally scales) by a predefined amount in the direction that slowly changes by a random amount. This is a nice feature for making animations that show the whole scene all the time.
  - **fly-by (Style=3)**: the camera flies through the visualization scene along a randomly precessing trajectory. This feature allows to focus on the detail, but the whole scene may not be visible for most of the time. IFrIT however takes care to insure that the camera always points toward the central part of the scene. The starting position of the camera depends on the projection mode: in the perspective projection mode the camera starts from the current position, in the parallel projection mode the camera moves closer to the center of the scene before starting a fly-by.
- **Theta** (short form: **dt**); `float`
  See Phi.
- **TitlePageFile** (short form: **tf**); `string`
  The name of the file with the image that should be used as title page for the animation. The image must be in one of the formats understood by VTK (JPEG, PNM, BMP, TIFF, and

PNG). If the size of the title page image is different than the size of the animation snapshot image, the title page image will be smoothly stretched to match the animation image. The title page will be displayed for a number of frames set by the NumTitlePageFrames property. After that, a number of frames specified by the NumTitlePageBlendedFrames property will be blended with the first image of the visualization scene, producing an effect of a gradually dissolving title page.

- **Zoom** (short form: **dz**); `float`
  A zooming factor by which the scene is scaled between the two consequitive frames.

3.2.3 BoundingBox object

Short form: **bb**

BoundingBox object represents the frame that encloses (but not truncates) the main visualizatin scene. The bounding box serves as a useful reference frames for placing other objects in relation to each other.

Available properties:

- **AxesLabels** (short form: **l**); `string[3]`
  When the **Bounding Box** is displayed as axes, properties **AxesLabels** and **AxesRanges** specify labels and ranges for the three axes respectively. The former property takes 3 string-values arguments, while the latter takes 6 floating point numbers as `Xmin`, `Xmax`, `Ymin`, etc.
- **AxesRanges** (short form: **r**); `float[6]`
  See AxesLabels.
- **Type** (short form: **t**); `int`
  The type of the **Bounding Box**, as follows:
  - ◊ **0**: the default IFrIT-style bounding box (red-blended-into-blue);
  - ◊ **1**: the classic bounding box (red in X-direction, green in Y, blue in Z);
  - ◊ **2**: the hair-thin bounding box (one-pixel wide lines);
  - ◊ **3**: the classic bounding box with X, Y, and Z axes shown as coordinate axes, with arrows at the end, labels, and ranges. This type is useful for showing 3D scatter plots.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

3.2.4 Camera object

Short form: **c**

Camera object represents the current camera that observes the scene. The camera supports two projection modes: parallel (orthogonal) and perspective. The orientation of the camera is set by its **Position**, a **FocalPoint** (the point in space the camera is directly looking at), and direction in space that the camera considers to be "up" (**ViewUp** vector). In perspective projection that's all one needs to fully specify the camera. In parallel projection the distance between the focal point and camera position is inrrelevant (the focal point is at infinity), so the scale of the view is set by the **ParallelScale** property.

Available properties:

- **Azimuth** (short form: **a**); `bool( double )`
  Rotate the camera horizontally (about the view up vector) centered at the focal point. This is a method that specifies the angle of rotation in degrees.
- **ClippingRange** (short form: **cr**); `pair`
  This property sets the location of the near and far clipping planes along the direction of projection (in OpenGL coordinates). Both of these values must be positive. How the clipping planes are set can have a large impact on how well z-buffering works. In particular the front clipping plane can make a very big difference. Setting it to 0.01 when it really could be 1.0 can have a big impact on your z-buffer resolution farther away.
- **ClippingRangeAuto** (short form: **cra**); `bool`
  Switch the automatic adjustment of the **Clipping Range** on and off. In almost all cases this property should be set.
- **Elevation** (short form: **e**); `bool( double )`
  Rotate the camera vertically (about the cross product of the direction of projection and the view up vector) centered on the focal point. This is a method that specifies the angle of rotation in degrees.
- **EyeAngle** (short form: **ea**); `double`
  Separation between eyes (in degrees) in the stereo mode.
- **FocalPoint** (short form: **f**); `vector`
  The focal point of the camera.
- **OrthogonalizeView** (short form: **ov**); `bool( )`
  Re-orient the camera for the nearest coordinate axis to align along the direction of viewing, and other two directions to align horizontally and vertically.
- **ParallelProjection** (short form: **pp**); `bool`
  The camera projection mode to parallel (if `true`), or perspective (if `false`).
- **ParallelScale** (short form: **ps**); `double`
  The scaling used for a parallel projection, i.e. the height of the viewport in OpenGL distances. Note that the **ParallelScale** parameter works as an "inverse scale" - larger numbers produce smaller images. This method has no effect in perspective projection mode.
- **Pitch** (short form: **p**); `bool( double )`
  Rotate the focal point vertically (about the cross product of the view up vector and the direction of projection, centered at the camera's position. This is a method that specifies the angle of rotation in degrees.
- **Position** (short form: **x**); `vector`
  The position of the camera in space (a 3D vector).
- **Reset** (short form: **rs**); `bool( )`
  A method that resets the camera so that the nearest side of the bounding box faces the screen.
- **Roll** (short form: **r**); `bool( double )`
  Rotate the camera about the direction of projection. This is a method that specifies the angle of rotation in degrees
- **ViewAngle** (short form: **va**); `double`
  The width (in degrees) of the field of view.
- **ViewAngleVertical** (short form: **vav**); `bool`
  A boolen property that specifies whether the **ViewAngle** is measured vertically or horizontally.
- **ViewUp** (short form: **u**); `vector`
  The direction that camera considers to be "up".
- **Yaw** (short form: **y**); `bool( double )`
  Rotate the focal point horizontally (about the view up vector centered at the camera's

position). This is a method that specifies the angle of rotation in degrees.

- **Zoom** (short form: **z**); `bool( double )`
  In perspective mode, decrease the view angle by the specified factor. In parallel mode, decrease the parallel scale by the specified factor. A value greater than 1 is a zoom-in, a value less than 1 is a zoom-out.

3.2.5 ClipPlane object

Short form: **cp**

A clipping plane cuts off a portion of the scene, thus allowing to look inside a visuaized object. This objects controls up to 6 clipping planes; each plane is manipulated by **Direction** and **Position** properties.

Available properties:

- **Delete** (short form: **dlt**); `bool( int )`
  A method (script function) to delete a given clipping plane.
- **Direction** (short form: **d**); `vector`
  The direction of the normal to the clipping plane - this is a vector with each component corresponding to one clipping plane.
- **New** (short form: **new**); `bool( )`
  A method (script function) to create a new clipping plane. The newly created plane will be located at the rigin and oriented perpendcular to the Z axis.
- **Number** (short form: **num**); `int`
  The number of clipping planes (uo to 6). **Direction** and **Position** properties become **Number**-sized arrays, and individual properties of separate clipping planes can be set independently as different components of each property array.
- **Position** (short form: **x**); `vector`
  The position of the center of the plane - this is a vector with each component corresponding to one clipping plane.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

3.2.6 ColorBar object

Short form: **cb**

ColorBar shows a correspondence between the color of a point in space and the value of scalar variable which this color represents. The object displays two bars with text captions on the left and right side of the visualization window. Normally, IFrIT will display color bars automatically, but they can also be manipulated manually.

Available properties:

- **Automatic** (short form: **a**); `bool`
  A switch specifying whether the **ColorBar** is in the automatic mode or not. In the automatic mode, IFrIT determines what variables to display at what color bar. In the manual mode the

contents of each color bar is determined by the user.

- **Color** (short form: **c**); `color`
  The color of the text legend. In properties colors are specified as 3-component RGB values `int,int,int` (like 255,0,0 for red).
- **DataType** (short form: **dt**); `data[2]`
  The data type for the right (`DataType[0]`) and left (`DataType[1]`) color bar.
- **Palette** (short form: **p**); `int[2]`
  The palette index for the right (`Palette[0]`) and left (`Palette[1]`) color bar. The palette id is the same quantity used to specify a palette for visualization objects (palette index if >0 and -palette index if
- **SideOffset** (short form: **so**); `float`
  A distance by which the color bars are offset from the side of the visualization window, in the fraction of the window width.
- **Size** (short form: **s**); `float`
  A relative size of the color bar as a fraction of the window size.
- **Var** (short form: **v**); `int[2]`
  The index of a variable for the right (`Index[0]`) and left (`Index[1]`) color bar.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

3.2.7 CrossSection object

Short form: **x**

A **Cross Section** object shows a section (a slice) of the visualization scene. The slice is orthogonal, i.e. it is always perpendicular to either X, or Y, or Z axis. Use a **Surface** object to show an arbitrary oriented slice. Because of the restricted orientation, the **Cross Section** object is significantly faster to visualize than a **Surface** object. Multiple slices can exists simulatenously. An important feature of the **Cross Section** object is that it treats point and cell data differently. If you want to show your data as it is, without any interpolation, set the **VoxelLocation** property of the **DataReader** object to 1 (cell data) and use no interpolation in this object. This is the only object that can show your data without any interpolation.

Available properties:

- **Delete** (short form: **dlt**); `bool( int )`
  A method (script function) to delete a given instance of this object.
- **Direction** (short form: **d**); `int[*]`
  The orientation of the **Cross Section** object. Takes one of the values 0, 1, or 2 (the slice is perpendicular to the X, Y, or Z axis respectively).
- **InterpolateData** (short form: **id**); `bool[*]`
  A boolean property that specifies whether the data are linearly interpolated on the cross section.
- **LevelToMark** (short form: **ltm**); `int[*]`
  The level of the ART mesh to mark.
- **Location** (short form: **l**); `double[*]`
  The current location of the **Cross Section** along its direction.
- **MarkOneLevel** (short form: **mol**); `bool[*]`
  A boolean property that toggles showing crosses (x) over all cells of a specified level, to

identify which cells and their level on the cross section.

- **MaxLevel** (short form: **maxl**); `int`
  The maximum level of the mesh to show on the cross section.
- **MeshColor** (short form: **mc**); `color[*]`
  The line color for plotting the ART mesh on top of the cross section.
- **MeshLineWidth** (short form: **lw**); `int[*]`
  The line width for plotting the ART mesh on top of the cross section.
- **MeshVisible** (short form: **mvis**); `bool[*]`
  A boolean property that toggles showing the ART mesh overlaid on the cross section.
- **Method** (short form: **m**); `int[*]`
  One of the available methods to render a cross section: **polygons** (0) or **texture** (1). The textute method is usually faster and gives the best quality, but it may not be always available.
- **New** (short form: **new**); `bool( )`
  A method (script function) to create a new instances of this object. The newly created instance will be initially identical to the last one.
- **Number** (short form: **num**); `int`
  The number of instances (independent screen representations) of this object. All properties of the object become **Number**-sized arrays, and individual properties of separate instances can be set independently as different components of each property array.
- **PaintVar** (short form: **pv**); `int[*]`
  The index of a variable to paint the object with. If the object does not visualize scalar data (for examle, a vector field), the scalar data should be compatible with the object data. The **Palette** property sets the respective palette to be used on the color bar.
- **Palette** (short form: **p**); `int[*]`
  Palette property takes an integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of -1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.
- **SampleRate** (short form: **sr**); `int[*]`
  The factor by which the data are downsampled while the **Cross Section** moves across the box. This property is only meaningful in the GUI shell.
- **StepLevel** (short form: **sl**); `int[*]`
  The level the cell size of which is used for stepping the cross section. Only used in GUI shell.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.
- **WidthAtLevel** (short form: **wl**); `int[*]`
  In ART extension the width of the cross sections does not have to be infinitely small. Properties **WidthInCells** and **WidthAtLevel** specify the cross section width as a fixed number of cells (**WidthInCells**) of a given level (**WidthAtLevel**). For example, setting this properties to **WidthInCells**=3 and **WidthAtLevel**=5 will make the cross section width equal to the 3 cell sizes of level 5. The value shown on the finite width cross section depends on the **WidthWeightingMode** property.
- **WidthInCells** (short form: **wc**); `int[*]`
  In ART extension the width of the cross sections does not have to be infinitely small. Properties **WidthInCells** and **WidthAtLevel** specify the cross section width as a fixed

number of cells (**WidthInCells**) of a given level (**WidthAtLevel**). For example, setting this properties to **WidthInCells**=3 and **WidthAtLevel**=5 will make the cross section width equal to the 3 cell sizes of level 5. The value shown on the finite width cross section depends on the **WidthWeightingMode** property.

- **WidthWeightingMode** (short form: **wm**); `int[*]`
  This property specifies how the data are averaged if the cross section **Width** is more than zero. The value of 0 of this property specifies that there is no weighting, and the data are simply averaged over the width of the cross section. If **WidthWeightingMode**>0, the value of this property is taken to be the index of the loaded ART variable which is used for weighting the average. It is important to note that the index refers to the **loaded** variable, so the same variable may have different indecies in different data files.

Child objects:

- **Material**

3.2.8 Data object

Short form: **d**

Data is a simple collection for all supported types of data. It exists for mere convenience and to avoid name clashes between new user-defined types and other IFrIT objects.

Available properties:

(This object has no properties.)

Child objects:

- **ARTBlackHoleParticles**
- **ARTEddingtonTensorField**
- **ARTParticles**
- **ARTStellarParticles**
- **ARTVariables**
- **ARTVelocityField**
- **GADGETBoundaryParticles**
- **GADGETBulgeParticles**
- **GADGETDiskParticles**
- **GADGETGasParticles**
- **GADGETHaloParticles**
- **GADGETStellarParticles**
- **NativeVTKPolyData**
- **NativeVTKScalars**
- **NativeVTKTensors**
- **NativeVTKVectors**
- **Particles**
- **Scalars**
- **Tensors**

- **Vectors**

3.2.9 DataReader object

Short form: **dr**

DataReader is responsible for loading data files of all types from the disk. It can also perform calculations on scalar data.

Available properties:

- **BoundaryConditions** (short form: **bc**); `int`
  This integer property specifies the type of boundary conditions:
  - ◊ **None** (**BoundaryConditions**=0) implies that the bounding box has no effect and data can extend outside it.
  - ◊ **Periodic** (**BoundaryConditions**=1) implies that the opposite sides of the bounding box are identical, i.e. the value of `x=-1.1` is identical to `x=0.9` (the size of the bounding box is 2 in OpenGL units);

  Additional options for boundary conditions can be added in the future.
- **ConvertArraysToFloat** (short form: **vtkca**); `bool`
  IFrIT works with scalar data in float format only. Since VTK files may contain scalar data in different numerical formats, this boolean property, if set to `true`, causes all scalar data to be converted to float.
- **Erase** (short form: **e**); `bool( data )`
  A method (script function) property that erases some of the loaded data from memory. It accepts one argument of type **data id**that specifies the data type to be erased. For example, calling this method with the argument **"Tensors"** will erase the built-in Tensors data.
- **IncludeParticles** (short form: **hip**); `bool`
  A property that specifies how ART data files are loaded. If set to **1**, then loading a variables data file will cause loading a corresponding particle file (if it exists).
- **IncludeVariables** (short form: **hiv**); `bool`
  A property that specifies how ART data files are loaded. If set to **1**, then loading a particles data file will cause loading a corresponding variables file (if it exists).
- **Load** (short form: **l**); `bool( data , string )`
  A method (script function) property that loads a data file. The first argument of this function has the type **data id** and specifies the data type to be loaded. For example, an argument **"Scalars"** will load the built-in Scalars data. The second argument is the name of the file. If it starts with the plus sign, the plus sign is replaced with the name of the default data directory for this type of data file.
- **MaxZDimension** (short form: **mzd**); `int`
  A limit on the size of the Z dimension of the uniform mesh data (a scalar, vector, or tensor field data) or 0, if the limit is not set. It is useful for loading only a fraction of a large data file.
- **PropertyIncluded** (short form: **gpi**); `bool[4]`
  A 4-component bool array that specifies which properties of GADGET particles are included. The 4 components are particle id, particle mass, gas internal energy, and gas density.
- **ROICenter** (short form: **hrc**); `vector`
  ART-ROICenter, **ART-ROISize**, and **ART-ROIType** properties specify the region of the ART grid to load, where type is 0 for a cube and 1 for a sphere.
- **ROISize** (short form: **hrs**); `double`

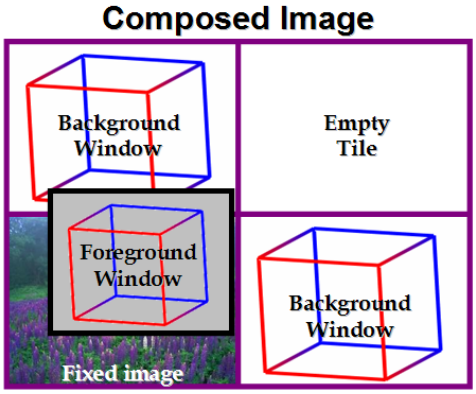
See ART-ROICenter.

- **ROIType** (short form: **hrt**); `int`
  See ART-ROICenter.
- **Reload** (short form: **r**); `bool( )`
  Some of **DataReader** properties only take effect when the data are loaded. When such properties are changed, they request **DataReader** to reload the appropriate data file(s). The **DataReader** accumulates such requests; this method (script function) causes all accumulated data files to be reloaded.
- **ScalePositions** (short form: **vtksp**); `bool`
  Internally, IFrIT positions are kept in OpenGL coordinates that go from -1 to 1 in each direction. If this boolean property is set to `true`, all positions in loaded VTK data will be rescaled to fit into the [-1,1] cube, with axis ratios preserved.
- **ScaledDimension** (short form: **sd**); `int`
  An integer value from -2 to 2, specifying which dimension (X, Y, or Z) of the uniform mesh data (a scalar, vector, or tensor field data) to fit into the bounding box. A value of -2 means the largest dimension, a valur of -1 means the shortest dimension, and a value between 0 and 2 means an X to Z dimension respectively. If the uniform mesh data is an exact cube, this property has no effect.
- **Shift** (short form: **s**); `double[3]`
  A three-dimensional floating point array that specifies the amounts (in units of the bounding box size = 2) by which the data should be shifted in each of 3 dimensions relative to their location in the data files. This is especially useful for periodic boundary conditions: it allows to bring to the center of the view a feature which, otherwise, may be cut into several peacies by the bounding box edges.
- **VoxelLocation** (short form: **vl**); `int`
  A property specifying the location of the data on the uniform mesh. Value 0 means that the data are point data, i.e. they are located at the vertices of the mesh; value 1 implies cell data, which are located at the centers of mesh cells (see Cell and Point Data).

3.2.10 ImageComposer object

Short form: **ic**

ImageComposer can combine several independent visualization windows into one output image. For example, it can tile several windows together in a regular mesh, or place a reduced representation of one visualization window as a insert inside another visualization window.



An example of an image composed from 3 different visualization windows

Images from different visualization windows are split into *background* and *foreground* images. The background images are tiles on a regular grid. Some of grid tiles may remain empty, or a fixed image from an external file can placed into an empty tile. The foreground images can be scaled down and placed anywhere in front of the background grid of tiles. Both, background and foreground images may have borders.

If no background or foreground visualization windows are specified, then the **Image Composer** is considered to be inactive (or bypassed): it will simply create an image of whatever is shown in the current visualization window.

Available properties:

- **BackgroundWindowIndex** (short form: **bgi**); `int`
  An array with number of components equal to the number of background tiles. Each component of the array gives an index of the visualization window whose image goes into that tile, or 0, if that tile is empty.
- **BackgroundWindowIndex2** (short form: **bgi2**); `int`
  See BackgroundWindowIndex3.
- **BackgroundWindowIndex3** (short form: **bgi3**); `int`
  Properties **BackgroundWindowIndex2** and **BackgroundWindowIndex3** specify additional visualization windows for a given tile for psedo-color composing. They are similar to **BackgroundWindowIndex** property, but if more than one visualization window is specified for a given background tile, images from multiple windows are combined in pseudo-color single image as follows: each image is turned into a grayscale image, and then the grayscale image from the primary window is used to specify the red channel of the combined image, and grayscale images from windows 2 and 3 (if present) are used to provide the green and the blue channels. If no windows 2 and 3 are specified, then normal composing is used - the image from the primary window is displayed in the respective tile as is.
- **BackgroundWindowWallpaperFile** (short form: **bgw**); `string`
  An array with number of components equal to the number of background tiles. Each component of the array gives a name of the image file whose image goes into that tile, or an empty string, if no image should be placed in that tile. If both, the visualization window and the image file are specified for a tile, the visualization window takes precedence.
- **BorderColor** (short form: **bc**); `color`
  The color of the border around background images. In properties colors are specified as 3-component RGB values `int,int,int` (like 255,0,0 for red).
- **BorderWidth** (short form: **bw**); `int`
  The width (in pixels) of the border around background images.
- **ForegroundWindowBorderColor** (short form: **fgc**); `color`
  An array with number of components equal to the number of foreground windows. Each component of the array gives the color of the border for the corresponding visualization window.
- **ForegroundWindowBorderWidth** (short form: **fgw**); `int`
  An array with number of components equal to the number of foreground windows. Each component of the array gives the width of the border (in pixels) for the corresponding visualization window.
- **ForegroundWindowIndex** (short form: **fgi**); `int`
  An array with number of components equal to the number of foreground windows. Each component of the array gives an index of the visualization window whose image goes into that tile. Each foreground image must have a visualization window associated with it.

- **ForegroundWindowIndex2** (short form: **fgi2**); `int`
  See ForegroundWindowIndex3.
- **ForegroundWindowIndex3** (short form: **fgi3**); `int`
  Properties **ForegroundWindowIndex2** and **ForegroundWindowIndex3** are used for psedo-color composing in a foreground window, completely analogous to **BackgroundWindowIndex2/3** properties.
- **ForegroundWindowPositionX** (short form: **fgx**); `int`
  See ForegroundWindowPositionY.
- **ForegroundWindowPositionY** (short form: **fgy**); `int`
  ForegroundWindowPositionX and **ForegroundWindowPositionY** are two arrays with number of components equal to the number of foreground windows. Each component of the arrays gives the horizontal and vertical location of the lower left corner of the corresponding foreground window in the composed image.
- **ForegroundWindowScale** (short form: **fgs**); `float`
  An array with number of components equal to the number of foreground windows. Each component of the array gives a floating point number less or eaual to 1, by which the corresponding visualization window is scaled.
- **ForegroundWindowZoom4Line** (short form: **fgz4**); `bool`
  See ForegroundWindowZoomSource.
- **ForegroundWindowZoomFactor** (short form: **fgzf**); `float`
  See ForegroundWindowZoomSource.
- **ForegroundWindowZoomSource** (short form: **fgz**); `int`
  Properties **ForegroundWindowZoom...** control the appearance of a foreground window when it is presented as a zoom onto another window. The zoom is displayed as a frame with 2 or 4 lines reaching from the corners of the frame into the corners of the foreground window. The property **ForegroundWindowZoomSource** sets the source window for the zoom: if this value is positive, it is taken as an index of another foreground window; if it is negative, it is taken as minus an index of a background window (background windows are numbered from left to right and from bottom to top). A zero value for the **ForegroundWindowZoomSource** property indicates that no zoom is displayed. The location and the size of the zoom frame are controled by the **ForegroundWindowZoomX**, **ForegroundWindowZoomY**, and **ForegroundWindowZoomFactor** properties. The first two set the center of the zoom frame in units of the source window size (i.e. both change from 0 to 1), while the latter sets the scale factor of the zoom frame relative to the original window (i.e., if the zoom is by a factor of 10, then **ForegroundWindowZoomFactor** should be set to 0.1). Finally, the **ForegroundWindowZoom4Line** boolean property controls whether the zoom frame is connected to the foreground window with 2 (**ForegroundWindowZoom4Line**=`false`) or 4 (**ForegroundWindowZoom4Line**=`true`) lines.

  If the zoom source and this foreground window are clones of each other, are both in parallel projections, and the foreground window displays a portion of the source window (i.e. it is a true zoom), then the zoom location and size will be set automatically to reflect the relationship between the two windows. In other words, whenever the foreground window is a true zoom onto another window, IFrIT will try to do its best to determine this relationship.
- **ForegroundWindowZoomX** (short form: **fgzx**); `float`
  See ForegroundWindowZoomSource.
- **ForegroundWindowZoomY** (short form: **fgzy**); `float`
  See ForegroundWindowZoomSource.
- **ImageHeight** (short form: **h**); `int( )`
  See ImageWidth.

- **ImageWidth** (short form: **w**); `int( )`
  ImageWidth and **ImageHeight** are read-only properties that give the total width and height of the composed image, including the border. The dimensions of the image are determined by the number of background tiles in both direction, by the size of each tile (which is taken to be the size of the largest visualization window set as a background tile), and the width of the border.
- **InnerBorder** (short form: **ib**); `bool`
  A boolean switch specifying whether, if the background images have a border, this border should be present between the inner sides of background tiles, or only on the outside.
- **NumForegroundWindows** (short form: **nfg**); `int`
  The number of foreground windows.
- **NumTiles** (short form: **nt**); `int[2]`
  The number of background tiles in two dimensions (stored as two components of this array).
- **ScaleBackground** (short form: **sb**); `bool`
  A boolean switch specifying whether the background images, if they are smaller than the size of the background tile, should be scaled or padded to the tile size.

3.2.11 ImageWriter object

Short form: **iw**

ImageWriter is a helper object that controls how actual writing of an image or a movie (animation) file.

Available properties:

- **ImageFormat** (short form: **f**); `int`
  The format of the image file produces. The accepted values are:
  - **0**: Portable Network Grapics(.png);
  - **1**: Joint Photographic Experts Group (.jpg);
  - **2**: Portable pixmap (.ppm);
  - **3**: Windows bitmap (.bmp);
  - **4**: Tag Image File Format (.tif);
  - **5**: Encapsulated Postrscript (.eps).

  Frankly speaking, I have no idea why anyone would want all of them...
- **ImageRootName** (short form: **in**); `string`
  The name of the file into which images are written; individual images are formed from that root by adding *-00001.png*, *-00002.png*, etc (actual file extension is determined by the **ImageWriter.ImageFormat** property). If the root name begins with a plus sign, files are created in the directory specified by the environment variable IFRIT_IMAGE_DIR; otherwise, the string is treated as a complete file name on the underlying file system. IFrIT does not create new directories for image files, so all directories encoded in the image root name must exist.
- **MovieOutput** (short form: **mo**); `int`
  The integer-valued property that controls the final output of a movie:
  - **0**: movie is stored in a sequence of files;
  - **1**: movie is stored as an MPEG2 movie (if VTK is linked against MPEG2 library);
  - **2**: movie is stored as an AVI movie (the latter option may not be available on all platforms).

- **MovieRootName** (short form: **mn**); `string`
  The name of the file(s) into which a movie is written. For a non-zero value of the **ImageWriter.MovieOutput** property, this is the root part of the actual movie file name (i.e. without the suffix); if **ImageWriter.MovieOutput=0**, then each movie frame is written into a separate file, and the file name is constructed in the following manner: if **MovieRootName** does not end with a minus sign, a directory with such name is created, and subsequent animation files are written into that directory with names *frame-00001.png*, *frame-00002.png*, etc (actual file extension is determined by the **ImageWriter.ImageFormat** property); otherwise, **MovieRootName** is treated as the actual root for the file name, and individual frames are formed from that root by adding *00001.png*, *00002.png*, etc. In order to avoid accidentally overwriting a previously created animation, if a second animation is run with the same root name, the root name will be appended with subsequent numbers 2, 3, etc. If the root name starts with a plus sign, files are created in the directory specified by the environment variable IFRIT_IMAGE_DIR; otherwise, the string is treated as a complete file name on the underlying file system. IFrIT does not create new directories for movie files, so all directories encoded in the movie root name must exist.
- **PostScriptOrientation** (short form: **po**); `int`
  The orientation of the PostScript images as portrait (0) or landscape (1).
- **PostScriptPaperFormat** (short form: **pf**); `int`
  The paper format for the PostScript image format. The valid values are from 0 to 10, which select the followinf formats respectively: A0, A1, A2, A3, A4, A5, A6, A7, A8, Letter, 11x17.
- **Write** (short form: **w**); `bool( )`
  An action that creates an image of the current visualization scene. This is simply an alias for **Window.WriteImage**.

3.2.12 Label object

Short form: **lb**

Label object represents a file record label that displays a properly modified current record of an animatable data file. If the curent data file is not animatable, the label is not displayed.

The specific contents of the label is controled by **Name**, **Scale**, **Offset**, **Unit**, and **NumDigits** properties. Namely, the label is shown as the following equation:

```
name=value unit
```

where *name* and *unit* are strings specified by string-values properties **Name** and **Unit**, and *value* is the mathematical product of the property **Scale** and the difference between value of the current file record and the **Offset** property (*value* = **Scale***[record-**Offset**]). The number of decimal places to show in the *value* is set by the **NumDigits** property.

For example, if **Name**="Power", **Unit**="Watts", **Offset**=0, and **Scale**=0.01, then a file with the record 1234 will have a label

```
Power=12.34Watts
```

A special value **Scale**=0 forms a label with name "z", no unit, and the value equal to 1e4/*record*-1, so that a file with the record 1234 will have a label


```
z=7.10
```

As you see, IFrIT remembers its origin!

Available properties:

- **Name** (short form: **n**); `string`
  A variable name to display in the file record label (see **Label**).
- **NumDigits** (short form: **d**); `int`
  The number of digits after the decimal place to display in the file record label.
- **Offset** (short form: **o**); `float`
  An offset for forming a numeric value of the label (see **Label**).
- **Scale** (short form: **s**); `float`
  A scale for forming a numeric value of the label (see **Label**).
- **Unit** (short form: **u**); `string`
  A string to diplay as a unit after the numeric value (see **Label**).
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

3.2.13 Lights object

Short form: **ls**

Lights object represents a collection of lights that shine on the scene. The collection consists of five lights, a **main** light, a **fill** light, a **headlight**, and two auxiliary **back** lights. The main light is usually positioned so that it appears like an overhead light (like the sun, or a ceiling light). It is generally positioned to shine down on the scene from about a 45 degree angle vertically and at least a little offset side to side. The main light usually at least about twice as bright as the total of all other lights in the scene to provide good modeling of object features.

The other lights in the kit (the fill light, headlight, and a pair of back lights) are weaker sources that provide extra illumination to fill in the spots that the main light misses. The fill light is usually positioned across from or opposite from the main light (though still on the same side of the object as the camera) in order to simulate diffuse reflections from other objects in the scene. The headlight, always located at the position of the camera, reduces the contrast between areas lit by the main and fill light. The two back lights, one on the left of the object as seen from the observer and one on the right, fill on the high-contrast areas behind the object. The **LightIntensity** property sets the intensities of the main, fill, and head lights. The back lights intensities are set automatically.

All lights are directional lights (infinitely far away with no falloff). Lights move with the camera. For simplicity, the position of lights in the **LightKit** can only be specified using two angles: the elevation (latitude) and azimuth (longitude) of the main light with respect to the camera, expressed in degrees. (Lights always shine on the camera's focal point.) For example, a light at (elevation=0, azimuth=0) is located at the camera. A light at (elevation=90, azimuth=0) is above the lookat point, shining down. Negative azimuth values move the lights clockwise as seen above, positive values counter-clockwise. So, a light at (elevation=45, azimuth=-20) is above and in front of the object and shining slightly from the left side.

Available properties:

- **Direction** (short form: **d**); `vector`
  A 3D vector that specifies the location of the main light in the camera reference frame. When this direction is along the Z axis (0,0,1) the light is on top of the main camera.
- **Intensities** (short form: **i**); `float[3]`
  Three intensities for the main, fill, and head lights.
- **TwoSided** (short form: **ts**); `bool`
  A boolean switch that toggles one- or two-sided lighting in OpenGL.

3.2.14 Marker object

Short form: **m**

Marker is a small resizable object that can be placed at a specified position inside the visualization scene. It can be used simply to mark a particular point, or other visualization objects can be "attached" to a marker. For example, markers can be used as staring points for streamlines of a vector field, or as centers of probing surfaces. A text caption - a body of text connected to the object by a line - can be attached to a marker to provide a short explanation. A legend listing all markers and their captions can also be placed at the bottom of the scene. The marker caption can be moved interactively using **Marker.MoveCaption()** method.

Available properties:

- **CaptionPosition** (short form: **cx**); `pair[*]`
  An array of positions of marker captions in the visualization window in viewport coordinates (in which the visualization window ranges from 0 to 1 in each direction). Each position is a pair of numbers `x,y` represented as appropriate in the used scripting language (pairs are represented differently in different scripting languages).
- **CaptionText** (short form: **ct**); `string[*]`
  An array of strings with the text of captions for all markers. The caption will be shown if the string is non-empty.
- **Color** (short form: **c**); `color[*]`
  Properties **Color** and **Opacity** set these two properties for visualization objects, which are represented as solid surfaces (OpenGL polygonal mesh) - which are all visualization objects except the **Volume** object. These two properties can be either scalar (1-component) or arrays (multi-component), depending on the type of the visualization objects they belong to (check descriptions for specific objects for more details). The **Color** property takes a 3-component RGB value `int,int,int` (like 255,0,0 for red). The **Opacity** properties takes a floating point number between 0 and 1.
- **Delete** (short form: **dlt**); `bool( int )`
  A method (script function) to delete a given marker.
- **Legend** (short form: **l**); `bool`
  A boolean property array that toggles whether the legend for each marker is shown or not.
- **LegendPosition** (short form: **lp**); `int`
  A position on the screen of marker legends. This property takes only 2 values: 0 (for lower-left corner) or 1 (for lower-right corner).
- **MoveCaptions** (short form: **mc**); `bool( )`
  Calling this method puts the current visualization window into a **Caption Move** mode. In that mode the mouse interaction with the visualization scene is disabled. The only mouse interaction that is allowed is to click on the marker caption and drag it around the screen. To

exit the **caption Move** mode, click anywhere outside a marker caption.

- **New** (short form: **new**); `bool( )`
  A method (script function) to create a new marker. The newly created marker will be initially identical to the last one.
- **Number** (short form: **num**); `int`
  The number of markers in the scene. All marker properties become **Number**-sized arrays, and individual properties of separate markers can be set independently as different components of each property array.
- **Opacity** (short form: **o**); `float[*]`
  See Color.
- **Position** (short form: **x**); `vector[*]`
  A position of a marker in the visualization scene.
- **Scaled** (short form: **sc**); `bool[*]`
  A boolean property that controls whether the marker is scaled during the scene interaction so that its aparent size remains the same, or behaves as all other objects - unscaled marker will become larger/smaller if the scene is zoomed in/out.
- **Size** (short form: **s**); `double[*]`
  The size of the marker.
- **TransparentCaptions** (short form: **tc**); `bool`
  A boolean property that toggles the transparency of marker captions.
- **Type** (short form: **t**); `int[*]`
  The type of the marker. The following values are available:
  - **0**: Point
  - **1**: Sphere
  - **2**: Tetrahydron
  - **3**: Cube
  - **4**: Octahedron
  - **5**: Icosahedron
  - **6**: Dodecahedron
  - **7**: Cone
  - **8**: Cylinder
  - **9**: Arrow
  - **10**: Cluster (a cloud of points)
  - **11**: Galaxy

Child objects:

- **Material**

3.2.15 Material object

Short form: **_mat**

A **Material** object is a helper object that provides access to rarely used parameters that control four OpenGL material properties: **Ambient**, **Diffuse**, **Specular**, and **SpecularPower** (also called *shininess* in some OpenGL books). They all take a floating-point value between 0 and 1. You need to know how OpenGL handles material properties to understand what these values mean. Most likely, you will never find a need to use them.

Available properties:

- **Ambient** (short form: **a**); `float`
  See Material.
- **Diffuse** (short form: **d**); `float`
  See Material.
- **ShadingMode** (short form: **sm**); `int`
  If the shading mode is -1 then shading is always off (so that an object looks the same no matter what its orientation relative to lights and a camera is), if it is +1 then shading is always on, and a value of 0 implies "automatic" shading, i.e. the object owning the material decides when it needs shading and when it does not. The automatic choice is normally good enough.
- **Specular** (short form: **s**); `float`
  See Material.
- **SpecularPower** (short form: **p**); `float`
  See Material.

3.2.16 MeasuringBox object

Short form: **mb**

A **Measuring Box** object is a semi-transparent cube that is always placed at the focal point of the camera. Its size can be adjusted, and is always displayed on the screen, so the size of features that fit into the box are always known.

Available properties:

- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

3.2.17 Palette object

Short form: **pal**

Palette objects represents palettes used by IFrIT.

Available properties:

- **Blue** (short form: **b**); `pair[*]`
  Array properties **Red**, **Green**, and **Blue** specify piecewise-linear functions to use for the three respective color ramps. Each piecewise-linear function is an array of pairs of numbers (represented as appropriate in the used scripting language - pairs are represented differently in different scripting languages) that give `x,y` values for each point along the piecewise-linear function.
- **Green** (short form: **g**); `pair[*]`
  See Blue.
- **Name** (short form: **n**); `string`
  A string-valued name for the palette.

- **New** (short form: **new**); `bool( )`
  This method adds an empty, unnamed palette, to the list of the available palettes.
- **Number** (short form: **num**); `int`
  The number of instances (existing independent realizations) of this object. This property is different from any other property in that it cannot be set, i.e. an assignment to it will fail, so it can only be used for checking the number of instances of a particular object.
- **Red** (short form: **r**); `pair[*]`
  See Blue.

3.2.18 Particles object

Short form: **p**

Particles object represents a collection of particles (points) and associated with them variables (or variables - for example, mass, size, etc). The set of particles consists of one or more independent subsets called groups. Each group is controlled indepedently, particles within each group can be connected with lines to, for example, represent trajectories. Particles in different groups can have different representation (dots, spheres, etc), color, size, they can be colored or sized according to values of one of the variables. Properties of a particular group are stored as corresponding components of arrays-valued **Particles** properties. For example, the representation type of group #2 is referenced as **Particles.Type[2]** property. Groups can be created when particle data have one or more variables. One of the variable (set with the **SplitVar** property) is used for splitting particles into group: all particles within one group have the value of the variable with the index **SplitVar** within a given range. Since ranges of different groups may overlap, a given particle may belong to more than one group. The ranges for the `data[SplitVar]` for each group are set with the array-values property **SplitRanges** property. Each range is set by a **pair** of values that give `min,max` values for each range (pairs are represented differently in different scripting languages). For example, the assignment in Python

```
Particles.SplitRanges = ((0,0.5),(0.3,0.9))
```

creates two groups, the first one containing particles for which `data[SplitVar]` is between 0 and 0.5, and the second one with `data[SplitVar]` between 0.3 and 0.9 (and so particles with values between 0.3 and 0.5 belong to both groups). After that statement, all array-valued particle properties (like **Particles.Type**) become 2-element array. If, now, **SplitRanges** is reassigned,

```
Particles.SplitRanges = ((-3,2),(2.5,4),(9,11))
```

all array-valued properties automatically become 3-element arrays. Setting **SplitVar** to -1 will place all particles into a single and only group.

Available properties:

- **AutoScaled** (short form: **au**); `bool[*]`
  If this boolean property is set to `true`, IFrIT will scale the particles automatically so that their size appear to be fixed when the scene is zoomed in or out. That requires a re-calculation of particle properties and may be slow.
- **Color** (short form: **c**); `color[*]`
  Properties **Color** and **Opacity** set these two properties for visualization objects, which are represented as solid surfaces (OpenGL polygonal mesh) - which are all visualization objects

except the **Volume** object. These two properties can be either scalar (1-component) or arrays (multi-component), depending on the type of the visualization objects they belong to (check descriptions for specific objects for more details). The **Color** property takes a 3-component RGB value `int,int,int` (like 255,0,0 for red). The **Opacity** properties takes a floating point number between 0 and 1.

- **ConnectVar** (short form: **cv**); **`int[*]`**
  Particles in a group can be connected by lines. This array-valued property specifies the variables that determines the order by which particles are connected for each group. If this property is set to -1, no particles are connected; otherwise, particles are connected by lines in the increasing order of the variable with index **ConnectVar**. For example, if there are 3 particles with `variable[`**`ConnectVar`**`]` values of 3, 1, and 2, then the particle #2 will be connected to the particle #3 and that one will be connected to the particle #1.
- **ConnectionBreakVar** (short form: **cbv**); **`int[*]`**
  If **ConnectVar** property is set to a valid index of a particle variable, all particles are connected by a single line in the increasing order of the variable with the index **ConnectVar**. To show more than 1 line, particles must have another variable, and the **ConnectionBreakVar** property should be set to that variable index. Then only particles having the same value of `data[`**`ConnectionBreakVar`**`]` are connected. For example, if there are 4 particles with 2 variables each, (1,1), (2,1), (4,2), and (6,2), and **ConnectVar** and `ConnectionBreakVar=1`, then the particle #1 will be connected to the particle #2 and the particle #3 will be connected to the particle #4 (increasing order of `data[`**`ConnectVar`**, the same value of `data[`**`ConnectionBreakVar`**`]`), but the particle #2 will *not* be connected to the particle #3, since they have different values of **ConnectionBreakVar**.
- **FixedSize** (short form: **s**); **`float[*]`**
  The basic size of all particles (see **ScaleVar** property for the description of particle sizing).
- **LineWidth** (short form: **lw**); **`int[*]`**
  The width of the line that connects particles, if they connected with lines
- **Opacity** (short form: **o**); **`float[*]`**
  See Color.
- **PaintVar** (short form: **pv**); **`int[*]`**
  The index of a variable to paint the object with. If the object does not visualize scalar data (for examle, a vector field), the scalar data should be compatible with the object data. The **Palette** property sets the respective palette to be used on the color bar.
- **Palette** (short form: **p**); **`int[*]`**
  Palette property takes an integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of -1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.
- **RenderSortVar** (short form: **rsv**); **`int`**
  This is a special variable index that, if set, changes the order by which particles are rendered. By default in OpenGL, particles are plotted from back to front in the order of their distance from the camera (we call this true rendering). This property causes the particles to be ordered by the increasing value of the variable. If the variable index is not -1, this option will disable true rendering (by setting the property **Window:TrueRendering** to false) and will also disable splitting particles by pieces.

- **ReplicationFactors** (short form: **rf**); `int[6]`
  Some of visualization objects can be replicated, i.e. their identical replicas placed outside the bounding box. This 6-dimensional integer property specifies how many replicas should be placed in -X, +X, -Y, +Y, -Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all visualizationw objects are replicatable).
- **ScaleFactor** (short form: **sf**); `float[*]`
  An additional factor used to scale particle (i.e. the particle size = **ScaleFactor** times the value of variable with the index **ScaleVar**.
- **ScaleVar** (short form: **sv**); `int[*]`
  The index of the particle variable used to scale particles in different groups. Sizes of individual particles are determined by the product of the value of the `data[`**`ScaleVar`**`]` and an additional factor **ScaleFactor**), i.e. for each particle `size =` **`ScaleFactor`**`*data[`**`ScaleVar`**`]`. If **ScaleVar** is -1, all particles have the same fixed size set by the property **FixedSize**. However, keep in mind that particles can be sized differently only if they are *not* represented by points. Points cannot be sized differentially in OpenGL.
- **SplitRanges** (short form: **r**); `pair[*]`
  An array property that sets the range of the values of `data[`**`SplitVar`**`]` that belong to a particular group (see description of the **Particles** object for the explanation on particle groups).
- **SplitRangesTiled** (short form: **rt**); `bool`
  If this boolean property is set, the ranges of separate particle groups are always adjoint ("tiled"), i.e. the max value for the group range of group N is always kept equal to the min value for the group range N+1. For example, the assignment (using Python as scripting language)

  ```
  Particles:SplitRanges[2] = (1,2)
  ```

  automatically changes the max range value of group #1 to 1 and the min range value of group #3 to 2. This is useful for making sure that every particle always belongs to exactly one group.
- **SplitVar** (short form: **v**); `int`
  The index of the particle variable used in splitting **Particles** into separate groups. All particles within one group have the value of `data[`**`SplitVar`**`]` within the given range.
- **Type** (short form: **t**); `int[*]`
  The representation of individual particles in a given group. The values of this property are the same as **Marker.Type**. Points are the fastest to show, but they cannot be sized differentially in OpenGL. If you need to size particles differentially, the fastest type to use is tetrahydron. Spheres are always the slowest type to visualize.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

Child objects:

- **Material**

3.2.19 Picker object

Short form: **pi**

Picker allows the user to "pick" an object in the visualization scene, and obtain information about the data at the picked location. Picking is activated by pointing the cursor to the desired location and pressing a key `P` on the keyboard. IFrIT then follows an imaginary line drawn through the cursor position, and finds the visualization objects that the line intersects. The information about the location of the intersection and properties of the visualization objects is then reported.

Picker operates in one of three modes:

- **Cell** mode is the slowest, but it is most intuitive. In this mode Picker picks a "cell" that first intersects the line of sight. For an OpenGL polygonal mesh the cell is a polygon of the mesh, for volume rendering, the cell is a cell in the volume, for particles a cell is a single particle - in the latter case the size of the cell is zero, so it is often ambigious to pick, the **Point** mode is best for picking particles.
- **Point** mode is also slow, and is best suited for picking **Particles**. It is important to keep in mind that IFrIT only picks an **OpenGL** point, which may or may not have any relation to the underlying data structure. For example, if IFrIT picks a point on the isosurface, the point most likely will not correspond to any vertex of the undelying grid. In addition, for a solid object (i.e. object consisting of the polygonal OpenGL mesh), only a vertex of a polygon can be picked. Thus, it is possible that you point the cursor on the solid surface, but nothing gets picked, because all polygon vertices are too far from the imaginary line. In that case, either use a **Cell** mode, or adjust the **Accuracy** property to reduce the maximum distance from the line-of-sight.
- **Object** mode is fast, because in this mode IFrIT uses the hardware to pick an object. However, the hardware picker may not pick translucent (non-opaque) objects correctly.

Available properties:

- **Accuracy** (short form: **a**); `float`
  The distance (tolerance) within which the **Picker** will search for the nearest point or cell. If this distance is too small, Picker may not find even one point or cell to pick. If this distance is too large, the picked point may be too far from the cursor position, and may not be the point you intended to pick. Adjust this parameter to get the best results for you data. This property has no effect in the **Object** mode.
- **PickMethod** (short form: **m**); `int`
  The picker mode: 0 is for **Cell** mode, 1 is for **Point** mode, and 2 is for **Object** mode.
- **PointSize** (short form: **ps**); `float`
  The size of the point the Picker places at the picked position, in OpenGL coordinates (the length of the bounding box is 2).
- **Position** (short form: **x**); `vector( )`
  The last picked position, returned as a 3D vector.

3.2.20 Ruler object

Short form: **rl**

A **Ruler** is an object displayed on top of the visualization window. The ruler is only visible in the parallel projection, because in the perspective projections the same distance projects on the screen differently depending on its location in the scene.

Available properties:

- **Scale** (short form: **s**); `float`
  The scale shown on the ruler. The scene will be zoomed in/out appropriatey to maintain consistency between the scale of the scene and the value shown on the ruler.
- **Title** (short form: **t**); `string`
  The title shown on top of the ruler. By default, there is no title.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

3.2.21 Surface object

Short form: **s**

Surface object represents a two-dimensional surface that samples the three-dimensional scalar data. The surface can be either an isosurface of a particular scalar variable, or a specified geometric shape like a sphere or a plane. A surface can be painted by varied colors that correspond to a value of one of scalar variables via a specified palette. Multiple instances of a **Surface** object can co-exist at the same time - for example, several isosurfaces corresponding to different values of a scalar variable can be used to represent the three-dimensional structure of the data.

Because a surface has two sides, IFrIT uses the following rule to determine which side is called "outside" and which is called "inside" for an isosurface:

1. For the **first** variable the outside side is the side where the value of the variable is **larger** than the level of the isosurface.
2. For the **second** and the **third** variables the outside side is the side where the value of the variable is **smaller** than the level of the isosurface.

It might help you to put more meaning in the words "inside" and "outside" for your isosurface by appropriately ordering variables in the data file.

Available properties:

- **AlternativeReductionMethod** (short form: **rda**); `bool[*]`
  If this property is set to 1, an alternative method for reducing the number of polygons in the isosurface will be used. Smetimes, the alternative method may be faster or may produce better results.
- **CellToPointMode** (short form: **cpm**); `int[*]`
  The method for converting ART cell data into point data. The two methods currently supported are the average of the 8 neighbors (0) or the median of the 8 neighbors (1).
- **Color** (short form: **c**); `color[*]`
  Properties **Color** and **Opacity** set these two properties for visualization objects, which are represented as solid surfaces (OpenGL polygonal mesh) - which are all visualization objects except the **Volume** object. These two properties can be either scalar (1-component) or arrays (multi-component), depending on the type of the visualization objects they belong to (check

descriptions for specific objects for more details). The **Color** property takes a 3-component RGB value `int,int,int` (like 255,0,0 for red). The **Opacity** properties takes a floating point number between 0 and 1.

- **Delete** (short form: **dlt**); `bool( int )`
  A method (script function) to delete a given instance of this object.
- **GradientNormals** (short form: **gn**); `bool[*]`
  A boolean switch that sets whether the normals to the isosurface are computed as the gradients of the isosurface field, rather than as geometric normals of the polygons making the surface.
- **IsoLevel** (short form: **l**); `float[*]`
  The value for the isosurface level in the isosurface mode.
- **IsoMethod** (short form: **im**); `int[*]`
  A method for creating the isosurface. The **IsoSurfaceMethod**=1 selects a commonly used Marching Cubes method, but it may not be available in all versions of VTK. A value of **IsoSurfaceMethod**=0 selects an alternative method, which may vary from one VTK version to another. Method **0** should *always* work, while method **1** may not always work due to VTK bugs. IFrIT attempts to choose the best method as a default setting.
- **IsoOptimization** (short form: **op**); `bool[*]`
  A boolean property that toggles whether the isosurface is optimized after construction. Optimizing the isosurface takes time, but the optimized isosurface will render faster. Setting this property to `true` only makes sense if you plan to send long time working with a given isosurface.
- **IsoReduction** (short form: **rd**); `int[*]`
  The level of reduction in the number of polygons that represent the isosurface. This property takes values from 0 to 3: value 0 means no reduction, 1 attempts to reduce by 75%, 2 by 90%, and 3 by 99%. Reduction does not change the topology of the isosurface (at least, should not), so the target level may not be achievable in practice.
- **IsoSmoothing** (short form: **sm**); `int[*]`
  The integer factor from 0 to 10 that controls the degree of additional smoothing for the isosurface. The value of zero switches smoothing off. Smoothing an isosurface takes time, but makes it look better.
- **IsoVar** (short form: **v**); `int[*]`
  The index of the scalar variable whose isosurface is created.
- **MaxLevel** (short form: **maxl**); `int`
  The maximum level of the mesh to use for rendering the surface.
- **Method** (short form: **m**); `int[*]`
  Specifies the method for creating the surface. Three methods are currently supported:
  - **0**: isosurface of the scalar variable with index **IsoSurfaceVar** at the value **IsoSurfaceLevel**,
  - **1**: sphere of radius **SphereSize** at the position **Position**,
  - **2**: plane with normal **PlaneDirection** at the position **Position**.
- **New** (short form: **new**); `bool( )`
  A method (script function) to create a new instances of this object. The newly created instance will be initially identical to the last one.
- **NormalsFlipped** (short form: **nf**); `bool[*]`
  This boolean property, if set to `true`, reverses the direction of normals (i.e. the sense of inside-outside for the surface) relative to the default settings.
- **Number** (short form: **num**); `int`
  The number of instances (independent screen representations) of this object. All properties of the object become **Number**-sized arrays, and individual properties of separate instances can be set independently as different components of each property array.

- **Opacity** (short form: **o**); `float[*]`
  See Color.
- **PaintVar** (short form: **pv**); `int[*]`
  The index of a variable to paint the object with. If the object does not visualize scalar data (for examle, a vector field), the scalar data should be compatible with the object data. The **Palette** property sets the respective palette to be used on the color bar.
- **Palette** (short form: **p**); `int[*]`
  Palette property takes an integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of -1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.
- **PlaneDirection** (short form: **pd**); `vector[*]`
  The direction (specified as a 3-component double array) of the surface in plane mode.
- **PolishLevel** (short form: **pl**); `int[*]`
  This an internal property for further development.
- **Position** (short form: **x**); `vector[*]`
  The position of the sphere or the plane in the sphere and plane modes.
- **ReplicationFactors** (short form: **rf**); `int[6]`
  Some of visualization objects can be replicated, i.e. their identical replicas placed outside the bounding box. This 6-dimensional integer property specifies how many replicas should be placed in -X, +X, -Y, +Y, -Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all visualizationw objects are replicatable).
- **SphereSize** (short form: **ss**); `double[*]`
  The radius of the sphere in the sphere mode.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

Child objects:

- **Material**

3.2.22 TensorField object

Short form: **t**

TensorField object represents a tensor field. Currently, the only supported method for tensor field visualization is a "tensor glyph": a collection of ellipsods, which, at every point is oriented along the eigenvectors of the tensor and (optionally) scaled in proportion to tensor eigenvalues.

Available properties:

- **Color** (short form: **c**); `color[*]`
  Properties **Color** and **Opacity** set these two properties for visualization objects, which are represented as solid surfaces (OpenGL polygonal mesh) - which are all visualization objects except the **Volume** object. These two properties can be either scalar (1-component) or arrays (multi-component), depending on the type of the visualization objects they belong to (check descriptions for specific objects for more details). The **Color** property takes a 3-component RGB value `int,int,int` (like 255,0,0 for red). The **Opacity** properties takes a floating point number between 0 and 1.
- **GlyphSampleRate** (short form: **gr**); `int[*]`
  This property accepts integer positive numbers that specify the sub-sampling rate for the object. If this property is set to 1, every point in the data will be shown. For values larger than 1, only every **GlyphSampleRate** value in every direction will be shown.
- **GlyphSize** (short form: **gs**); `float[*]`
  The uniform size to scale all glyphs with.
- **GlyphType** (short form: **gt**); `int[*]`
  A geometric representation for the glyph. Currently, only 2 types are supported: 0 is for cubes and 1 is for spheres, and spheres will take much longer to render than cubes.
- **IsConnectedToScalars** (short form: **cs**); `bool( )`
  A boolean query (script function with no argumnets) returning whether the object can be colored by a scalar variables. If this function returns `false`, setting **PaintVar** property will have no effect.
- **MaxLevel** (short form: **maxl**); `int`
  MaxLevel and **MinLevel** properties specify the range of mesh levels to show.
- **Method** (short form: **m**); `int[*]`
  This property is reserved for future use. Currently, only one methd - tensor glyph - is supported.
- **MinLevel** (short form: **minl**); `int`
  See MaxLevel.
- **Opacity** (short form: **o**); `float[*]`
  See Color.
- **PaintVar** (short form: **pv**); `int[*]`
  The index of a variable to paint the object with. If the object does not visualize scalar data (for examle, a vector field), the scalar data should be compatible with the object data. The **Palette** property sets the respective palette to be used on the color bar.
- **Palette** (short form: **p**); `int[*]`
  Palette property takes an integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of -1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.
- **ReplicationFactors** (short form: **rf**); `int[6]`
  Some of visualization objects can be replicated, i.e. their identical replicas placed outside the bounding box. This 6-dimensional integer property specifies how many replicas should be placed in -X, +X, -Y, +Y, -Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all visualizationw objects are replicatable).

- **Scaling** (short form: **s**); `bool[*]`
  A boolean property that specifies whether the tensor glyphs should be scaled by the tensor eigenvalues. If it is `false`, all glyphs will have the same volume, but different shapes and orientations.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

Child objects:

- **Material**

3.2.23 VectorField object

Short form: **u**

VectorField object represents a vector field, like the flow of the fluid. A vector field can be represented as a "vector glyph" - a set of straight lines pointing in the direction of the vector field at every point with lengths proportional to the magnitude of the vector, or as a collection of stream lines - lines which would correspond to the fluid flow lines if the vector field was a real fluid velocity field. Streamlines originate at a **source object**, which can be a plane, a sphere, a disk, or a all existing **Marker**s.

Available properties:

- **Color** (short form: **c**); `color[*]`
  Properties **Color** and **Opacity** set these two properties for visualization objects, which are represented as solid surfaces (OpenGL polygonal mesh) - which are all visualization objects except the **Volume** object. These two properties can be either scalar (1-component) or arrays (multi-component), depending on the type of the visualization objects they belong to (check descriptions for specific objects for more details). The **Color** property takes a 3-component RGB value `int,int,int` (like 255,0,0 for red). The **Opacity** properties takes a floating point number between 0 and 1.
- **GlyphBaseSize** (short form: **gbs**); `int[*]`
  Since vector glyphs have no directiona sense, the base of the can be labeled by a point. That property specifies the point size. Just keep in mind that if the point size is smaller than the width of the glyph line, it will not be clearly visible.
- **GlyphSampleRate** (short form: **gr**); `int[*]`
  This property accepts integer positive numbers that specify the sub-sampling rate for the object. If this property is set to 1, every point in the data will be shown. For values larger than 1, only every **GlyphSampleRate** value in every direction will be shown.
- **GlyphSize** (short form: **gs**); `float[*]`
  The uniform size to scale all glyphs with.
- **IsConnectedToScalars** (short form: **cs**); `bool( )`
  A boolean query (script function with no argumnets) returning whether the object can be colored by a scalar variables. If this function returns `false`, setting **PaintVar** property will have no effect.
- **LineDirection** (short form: **ld**); `int[*]`
  The direction of the streamlines relative to the source. The following values are accepted:

- ◊ **0**: the up-stream direction (the direction the **vector field** points to);
- ◊ **1**: the down-stream direction (the direction opposite to the one the **vector field** points to);
- ◊ **2**: the forward direction (the direction of increasing coordinate);
- ◊ **3**: the backward direction (the direction of decreasing coordinate);
- ◊ **4**: both directions (the streamline is drawn from both sides of the source object).

♦ **LineLength** (short form: **ll**); `float[*]`
The maximum length for the streamline to draw.

♦ **LineQuality** (short form: **lq**); `int[*]`
The quality of the streamline. If the value of this property is zero, the streamline may appear "jaggy". Positive values of this property make the streamline smoother, at the expense of longer rendering time.

♦ **LineWidth** (short form: **lw**); `int[*]`
The width of the line representations: the glyph and the stream lines.

♦ **MaxLevel** (short form: **maxl**); `int`
MaxLevel and **MinLevel** properties specify the range of mesh levels to show.

♦ **Method** (short form: **m**); `int[*]`
The following modes for visualizing the vector field are supported:

- ◊ **Glyph (0)**: a set of straight lines pointing in the direction of the **vector field** at every point with lengths proportional to the magnitude of the vector;
- ◊ **Stream line (1)**: a set of lines, which would correspond to the fluid flow lines if the **vector field** was a real fluid velocity field;
- ◊ **Stream tube (2)**: a stream line that has a finite thickness that varies according to the flow speed (as if the mass flow is conserved);
- ◊ **Stream band (3)**: a pair of nearby stream lines with the surface between them. This is useful to visualize diverging or converging flows.

In the last three modes, streamlines are

♦ **MinLevel** (short form: **minl**); `int`
See MaxLevel.

♦ **NumberOfStreamLines** (short form: **nl**); `int[*]`
The number of streamlines to generate. If the source object is **Markers**, then this property has no effect, as the number of streamlines is equal to the number of available markers.

♦ **Opacity** (short form: **o**); `float[*]`
See Color.

♦ **PaintVar** (short form: **pv**); `int[*]`
The index of a variable to paint the object with. If the object does not visualize scalar data (for examle, a vector field), the scalar data should be compatible with the object data. The **Palette** property sets the respective palette to be used on the color bar.

♦ **Palette** (short form: **p**); `int[*]`
Palette property takes an integer value, that is interpreted as follows: if the value is positive, it is taken to be the index of the specific palette; if the value is negative, it is taken as minus the index of the palette, and the palette is reversed. For example, a value of 1 will select a traditional rainbow palette, while a value of -1 will select the rainbow palette but in the reversed order (pink is on the left, dark blue is on the right). Finally, a value of zero will select the special "brightness" palette, with which the object is colored by shades of its current color, with brightness of the shade related to the value of scalar variable used to color the object. The brightness palette cannot be reversed, and **Cross Section** and **Volume** objects cannot be colored with a brightness palette (they do not have a regular color), so the value 0 is ignored for them.

♦ **ReplicationFactors** (short form: **rf**); `int[6]`
Some of visualization objects can be replicated, i.e. their identical replicas placed outside the

bounding box. This 6-dimensional integer property specifies how many replicas should be placed in -X, +X, -Y, +Y, -Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all visualizationw objects are replicatable).

- **ShowSourceObject** (short form: **sso**); `bool[*]`
  A boolean property that toggles whether the source object is shown in the visualization scene.
- **SourceDirection** (short form: **sd**); `vector[*]`
  The direction (specified as a 3-component double array) of the streamline source object when it is a plane or a disk.
- **SourceOpacity** (short form: **so**); `float[*]`
  The opacity of the streamline source object (between 0 and 1), if it is shown.
- **SourcePosition** (short form: **sx**); `vector[*]`
  The position of the streamline source object.
- **SourceSize** (short form: **ss**); `double[*]`
  The size of the streamline source if it is a disk or a sphere.
- **SourceType** (short form: **st**); `int[*]`
  The type of a geometric shape of the streamline source object:
  - ◊ **0**: disk;
  - ◊ **1**: plane;
  - ◊ **2**: size.

  The size of the disk or the sphere is set by the **SourceSize** property, the orientation of the disk or the plane is set by the **SourceDirection** property, and the center of the source object is set by the **SourcePosition** propery.
- **TubeRangeFactor** (short form: **tr**); `float[*]`
  TubeSize, **TubeRangeFactor**, and **TubeVariationFactor** control the shape of streamtubes. The **TubeSize** property sets the minimum diameter of the tube; all linear scales in the tube are proportional to the value of this property. The **TubeRangeFactor** property is the ratio of the maximum to the minimum sizes of the tube. The **TubeVariationFactor** determines how sensitive the size of the tube is to the value of the vector field visualized. It may take some experimentation with these three parameters to achieve a satisfactory-looking tubes for your data.
- **TubeSize** (short form: **ts**); `int[*]`
  See TubeRangeFactor.
- **TubeVariationFactor** (short form: **tv**); `float[*]`
  See TubeRangeFactor.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

Child objects:

- **Material**

3.2.24 Volume object

Short form: **v**

Volume object uses the volume rendering method to represent the full three-dimensional structure of the data. Several methods for volume rendering are available.

Available properties:

- **AllMethods** (short form: **am**); `string( )`
  This function returns all methods supported by this objects. Methods may differ for different extentsions.
- **BlendMode** (short form: **bm**); `int[*]`
  The **Blend Mode** is only meaningful for the Raycast and VolumePro methods. It accepts two values:
  - **0**: composite mode, when all values along a line-of-sight direction are composed together depending on the opacity function;
  - **1**: "maximum intensity" mode, when the maximum value along a line-of-sight direction is used to represent the opacity along the line-of-sight.
- **DepthDownsampleFactor** (short form: **dd**); `float[*]`
  See ImageDownsampleFactor.
- **ExtendOcts** (short form: **heo**); `bool[*]`
  When using the 2D texture method, the volume rendering by default uses only 4 textures for 8 cells in each oct. If this parameter is set to `true`, IFrIT will use 8 textures, achieving a better depth effect at the expense of using twice more texture memory.
- **Force2DTextures** (short form: **hf2d**); `bool[*]`
  This parameter forces the use of 2D textures for uniform rebinning mode.
- **ImageDownsampleFactor** (short form: **di**); `float[*]`
  Properties **ImageDownsampleFactor** and **DepthDownsampleFactor** specify downsampling factors in the plane of the screen and along the line-of-sight respectively. Both factors can be less than 1, which would imply *super-sampling* rather than down-sampling. These properties are not supported by all methods.
- **InterpolationType** (short form: **it**); `int[*]`
  The value of 0 sets the nearest neighbor interpolation; the value of 1 switches to linear interpolation.
- **MaxLevel** (short form: **maxl**); `int`
  The maximum level to use in volume rendering.
- **Method** (short form: **m**); `string[*]`
  Specifies the method for volume rendering. VTK provides several different methods, and different versions of VTK and different installations may or may not include all methods; IFrIT extensions may have their own sets of volume rendering methods. A specific method can be specified either by the **Method** property that accepts one of the following method names, or by the integer **MethodId** value:
  - **Ray casting** method computes volume properties along rays cast from every point on the screne into the volume; rendering is done in software, and is slow for large data sets.
  - **Fixed point ray casting** is an alternative software ray casting method. It is often faster than traditional ray casting.
  - **2D textures** method replaces the volume with a set of semi-transparent textures; this can be fast for medium-size volumes and high quality videocards, although the quality of rendering is usually worse then in the *Ray Casting* method; this method is not suitable for volumes with less than about 30x30x30 cells.
  - **3D textures** method uses 3-dimensional textures, similar to 2D textures, except the visual quality is usually much higher even for small volumes. It is also often faster that 2D textures. This method is only available if your hardware supports it - not all videocards can handle 3D textures, but most of the most recent ones can.
  - **GPU ray casting** uses the Graphic Processing Unit (GPU) board (if available). If your computer has a GPU, this is likely to be the fastest and the highest quality

method.
◊ **VolumePro board** method uses the VolumePro 1000 hardware volume rendering board; you need to have this expensive board to use this method.

- **MethodId** (short form: **mid**); `int[*]`
  See Method.
- **OpacityFunction** (short form: **of**); `pair[*]`
  The piece-wise function that specifies the opacity of a cell as a function of the variable value in the cell. The opacity function is specified in the same way as any of the **Palette** components.
- **OpacityScaleFactor** (short form: **os**); `float[*]`
  The extra factor by which the volume opacity is scaled - large values make the volume less transparent.
- **Palette** (short form: **p**); `int[*]`
  The palette to render the volume with.
- **QualityLevel** (short form: **hq**); `int[*]`
  The quality of uniform rebinning mapping: from low (0) to high (2).
- **ReplicationFactors** (short form: **rf**); `int[6]`
  Some of visualization objects can be replicated, i.e. their identical replicas placed outside the bounding box. This 6-dimensional integer property specifies how many replicas should be placed in -X, +X, -Y, +Y, -Z, and +Z directions, respectively. This property only has effect if the data are periodic in at least some of the directions, and if the object can be replicated (not all visualizationw objects are replicatable).
- **ScaleFactor** (short form: **hsf**); `float[*]`
  The scale factor for the uniform rebinning volume. By default, IFrIT chooses the volume for uniform rebinning that just fits into the camera frustrum.
- **SmoothTextures** (short form: **hst**); `bool[*]`
  This parameter toggles smoothing texture edges for the 2D texture method. May take up to 4 times more texture memory, though.
- **UniformOpacity** (short form: **huo**); `bool[*]`
  Because the dynamic range of ART data can easily be much more than the maximum dynamic range achieved with 1 byte OpenGL opacities (about 1410 = ln(1/255)/ln(254/255)), using standard uniform opacities for 2D texture mapping can produce an empty scene. By default IFrIT does not scale opacities by the cell size, to circumvent this problem - this is roughly equivalent to "density weighting", when small (and, presumably, dense) cells receive larger weight. If you set this boolean property to `true`, IFrIT will use standard unifrm opacities for all cells.
- **Var** (short form: **v**); `int[*]`
  The index of the scalar variable to do the volume rendering of.
- **Visible** (short form: **vis**); `bool`
  A boolean property that toggles whether the object is visible in the visualization scene.

Child objects:

- **Material**

3.2.25 Window object

Short form: **w**

The **Window** object is a single visualization window that displays the whole scene. Several **Window** objects can exist, and can either be independent, or be "clones" of another **Window**. In the latter case they share the data with the "parent" window but in all other respects behave as separate windows.The **Window** object is directly responsible for showing the **Bounding Box**, **Record Label**, **Clipping Plane**, and several other controls, but it delegates visualization of the data to its sub-objects: **CrossSection**, **Surface**, **Volume**, **Particles**, **VectorField**, and **TensorField**.

Available properties:

- **Antialiasing** (short form: **aa**); `bool[3]`
  A boolean property that toggles OpenGL antialiasing in the scene. The property is a vector of dimension 3, for antialiasing separate components of the rendered scene: points, lines, and polygons.
- **BackgroundColor** (short form: **bg**); `color`
  The background color of the scene, specified as a 3-component RGB value `int,int,int` (like 255,255,255 for white).
- **BackgroundImage** (short form: **bi**); `string`
  If this property is assigned a name of an existing image file, the image from the file will be used as a background for the visualization scene (instead of a fixed color background set by **BackgroundColor** property). To revert to a fixed color background, assign an empty string to this property. Notice, that rendering a scene with an image in the background is much slower than a scene with a fixed color background.
- **BackgroundImageFixedAspect** (short form: **bia**); `bool`
  A boolean property that determines whether the aspect ratio of a background image is preserved; if set to `0`, the image will be scaled to fill in the current window.
- **BoxSize** (short form: **bs**); `float`
  The size of the **Bounding Box** in physical units. Changing this property does not actually change the visualization scene and it does not affect the relation of objects with respect to each other, but it useful for assigning physical meaning to the distances in the scene. If **Box Size** is set to 0, then the default OpenGL coordinate system is used: coordinates within the bounding box go from -1 to 1 in each of the 3 directions; the center of the bounding box is at (0,0,0), and the linear size of the bounding box is 2. If **Box Size** is positive, coordinates go from 0 to the value of the **Box Size** in all 3 directions.
- **CameraAlignmentLabel** (short form: **ca**); `bool`
  A boolean property that toggles showing of coordinate axis when the came orientation is orthogonal, i.e. the direction of viewing and the view up vector are both parallel to coordinate axes. This is useful when visualizing **Cross Section** objects.
- **Clone** (short form: **clone**); `bool( int )`
  This method clones the current window (i.e., creates a new window that shares the data with the current one).
- **CloneOfWindow** (short form: **co**); `int`
  A method (script function) that returns the index of the parent window if this **Window** object is a clone of another one, or 0 if it is not a clone.
- **Copy** (short form: **copy**); `bool( int )`
  This method creates a new window and copies all the settings from the current window to the newly created one.
- **CurrentInteractorStyle** (short form: **is**); `int`
  The style of the visualization window *interactor*, i.e. the component that control how mouse clicks and keyboard keys are affect the visualization scene. IfrIT uses four different interactor styles:

- ◊ **0** or **1**: use **Display** mode described in Mouse and Keyboard Controls. The difference between values 0 and 1 is that the value 0 uses a *trackball* method of interaction, when you need to click on a mouse button and move the mouse to change the scene; the value of 1 uses a *joystick* mode, when just clicking on mouse button causes the scene to spin or zoom (depending on the button clicked) with a rate that depends on how far away the mouse cursor is from the center of the screen;
- ◊ **2**: use **Fly-by** mode described in Mouse and Keyboard Controls.
- ◊ **3**: use **Keyboard Interactor** mode described in Mouse and Keyboard Controls.

♦ **Delete** (short form: **dlt**); `bool( int )`
A method to delete the specific instances of this object - the index of the instance to be deleted is specified as an argument to that method. For example, `Window.Delete(2)` call will delete the window #2. If the index is equal 0, then the current instance will be deleted.

♦ **Export** (short form: **exp**); `bool( string )`
A method for exporting the current visualization scene in the OBJ, X3D, or VRML format into the file whose name supplied as a function agument. The format of the export is determined by the file extension (".obj" for OBJ, ".x3d" for X3D, ".vrml" for VRML).

♦ **FontScale** (short form: **fs**); `int`
FontScale and **FontType** properties set the font size and type of text displayed along **Color Bar**s, a **Ruler**, etc. The scale value of 0 means the default font size, negative values make the font smaller, positive values make it larger. The available values for the **FontType** property are:

- ◊ **-1**: use vector (drawn by lines) font. This font does not look too well, but is always available and looks the same on all systems.
- ◊ **0**: use Arial font family.
- ◊ **0**: use Courier font family. This font has equal space per letter ("fixed" or "monispace") font and is useful sometimes.
- ◊ **0**: use Times font family.

♦ **FontType** (short form: **ft**); `int`
See FontScale.

♦ **ImageMagnification** (short form: **ix**); `int`
The magnification factor of the image. If the produces image does not have to be the same as the image on the screen, but can be any factor larger. I have a 30,000 by 30,000 image on the wall of my office.

♦ **New** (short form: **new**); `bool( )`
This method creates a new window with default settings for all objects.

♦ **Number** (short form: **num**); `int`
The number of instances (existing independent realizations) of this object. This property is different from any other property in that it cannot be set, i.e. an assignment to it will fail, so it can only be used for checking the number of instances of a particular object.

♦ **Position** (short form: **sp**); `int[2]`
The position of the visualization on the screen (returned as two-component integer array). This property is read-only - use the mouse to move the windows on the screen.

♦ **Render** (short form: **ren**); `bool( )`
This method renders the visualization scene.

♦ **Size** (short form: **ss**); `int[2]`
This property controls the size of the visualization window on the screen. Opposite to the **Position** property, this property is writeable, i.e. changing this value also changes the size of the window on the screen.

♦ **StereoAlignmentMarks** (short form: **sam**); `bool`
A boolean property that toggles showing alighnment markers for dual window stereo mode. This property has no effect if the **Stereo Modee** mode is not *dual windows*.

- **StereoMode** (short form: **sm**); `int`
  The method to use to display stereo image. The following values are accepted:
  - ◊ **0**: Mono display (no stereo).
  - ◊ **1**: Dual windows (two eyes in two separate windows) for geowall-like set up.
  - ◊ **2**: Crystal Eyes (for shutter glasses).
  - ◊ **3**: Simple blue-red stereo.
  - ◊ **4**: The interlaced render stereo type is for output to a VRex stereo projector. All of the odd horizontal lines are from the left eye, and the even lines are from the right eye. The user has to make the render window aligned with the VRex projector, or the eye will be swapped.
  - ◊ **5**: Left eye only.
  - ◊ **6**: Right eye only.
  - ◊ **7**: Dresden Display special purpose hardware (vertical interleave).
  - ◊ **8**: Analgraph (red-cyan).
  - ◊ **9**: Checkerboard (odd-even interleave).
- **TrueRendering** (short form: **tr**); `int`
  In order to render translucent geometry correctly, OpenGL graphic primirives (points, lines, and polugons) need to be rendered from the back of the Z-buffer to front ("true rendering"). This property controls how precisely the true redering is implemented. If set to 0, VTK will render primitives in the order they are stored in memory, which may create visual artifacts; values from 1 to 3 will produce progressively higher fidelity images at the expense of longer rendering time. Opaque (i.e. non-translucent) objects will always be rendered correctly by OpenGL. Some special modes of visualizing particles may switch true rendering off.
- **UpdateRate** (short form: **ur**); `int`
  The interactive update rate. The main feature of VTK is that it can adjust rendering of a complex visualization scene to achieve a request update rate. If the rendering of one frames takes too long to keep up with the requested rate, some of the objects in the scene will be simplified temporarily to maintain interactive rate. The rate is measured in frames per second, rounded to the nearest integer.
- **WriteImage** (short form: **wi**); `bool( )`
  A function that creates an image of the current visualization scene (see **Image Composer** and **Image Writer** for more details on what can be output as an image).

Child objects:

- **ARTMesh**
- **Animator**
- **BoundingBox**
- **Camera**
- **ClipPlane**
- **ColorBar**
- **CrossSection**
- **Data**
- **DataReader**
- **ImageWriter**
- **Label**
- **Lights**
- **Marker**
- **MeasuringBox**
- **Particles**

- **Picker**
- **Ruler**
- **Surface**
- **TensorField**
- **VectorField**
- **Volume**

3.2.26 ifrit object

Short form: **i**

Object **ifrit** serves as a root of IFrIT object hierarchy. It establishes interactions between various visualization windows. For example, it can synchronize all windows, so that mouse interaction in one window affects all other windows in the same way.

Available properties:

- **AutoRender** (short form: **ar**); `bool`
  A switch controlling whether the visualization windows are rendered automatically after every request, or a render command needs to be issued explicitly. Normally, you do not need to use this switch.
- **NumberOfProcessors** (short form: **np**); `int`
  The number of processors to use in the parallel mode. Setting this property may not necessarily succeed, it will depend on whether the system is actually able to allocate that many processors for IFrIT to use, but for portability reasons the error message may not be generated.
- **OptimizationMode** (short form: **om**); `int`
  This integer property specifies the way IFrIT tries to optimize its performance. If it is set to 0, IFrIT will try to optimize for speed, using extra memory whenever it can lead to faster performance; the value 1 will set optimization for memory, and IFrIT will try to minimize its memory use at the expense of doing extra calculations. With the value 2, IFrIT will optimize for quality, i.e. it will try to make the highest quality rendering of the scene even if it takes more memory and more computing time.
- **RestoreState** (short form: **restore**); `bool( string )`
  This method reads the complete state of all IFrIT objects from a file with the previously saved state. The argument of this method is the string with the name of the file. If it starts with the plus sign, the plus sign will be replaced with the name of the default IFrIT directory. If the argument is an empty string, it will default to "+ifrit.ini".
- **SaveState** (short form: **save**); `bool( string )`
  This method saves the complete state of all IFrIT objects from a file, which can later be read with the RestoreState method. The argument of this method is the string with the name of the file. If it starts with the plus sign, the plus sign will be replaced with the name of the default IFrIT directory. If the argument is an empty string, it will default to "+ifrit.ini".
- **SynchronizeCameras** (short form: **sc**); `bool( int )`
  A method synchronizing (i.e. orienting them in the same way) cameras in different visualization windows.
- **SynchronizeInteractors** (short form: **si**); `bool`
  A boolean property switching the interactor synchronization between different visualization windows on and off. If interactors in different visualization windows are synchronized, mouse

interaction in one window will affect all other visualization windows. This can be useful for, for example, comparing two different data files, or for looking at two sides of the same scene. If the interactors are synchronized, the views in different windows do not have to be same - different cameras may be oriented differently, but they will all move in unison.

Child objects:

- **ImageComposer**
- **Palette**
- **Window**

3.3 Data Objects

3.3.1 ARTBlackHoleParticles data object

Short form: **abhp**

This object represents the part of ART particles that represent black holes.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`
  The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.
- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total

number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.

- ◊ **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
- ◊ **DownsampleMode**=3 selects particles to load at random.
- ◊ **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/`**`DownsampleFactor`** first particles will be selected.
- ◊ **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.

- ♦ **Erase** (short form: **e**); **`bool( )`**
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- ♦ **FileName** (short form: **fn**); **`string( )`**
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- ♦ **Load** (short form: **l**); **`bool( string )`**
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- ♦ **Name** (short form: **n**); **`string[*]`**
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- ♦ **Range** (short form: **r**); **`pair[*]`**
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- ♦ **ResetOnLoad** (short form: **rol**); **`bool`**
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- ♦ **Stretch** (short form: **s**); **`string[*]`**
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- ♦ **TypeIncluded** (short form: **ti**); **`bool`**
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- ♦ **UseExtras** (short form: **ue**); **`bool( )`**
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.2 ARTEddingtonTensorField data object

Short form: **aetf**

This object represents the ART Eddington tensor field treated a tensor field (rather than individual components).

Available properties:

- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.3 ARTParticles data object

Short form: **ap**

This object represents the ART particles data. Depending on the setting, this may include all particles, or only dark matter particles (if stellar particles and black holes are loaded as separate data). IFrIT recognizes all ART and ART particle data formats. In order to load particles, it is sufficient to load only the header file (PMcrd*.DAT or *.dph). Other data files will be loaded automatically.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`
  The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.
- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.
  - **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
  - **DownsampleMode**=3 selects particles to load at random.
  - **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/`**`DownsampleFactor`** first particles will be selected.
  - **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **GridSize** (short form: **gs**); `int`
  The size of the top level ART mesh in 1D as perceived by Particles Data (the factor `num_grid` in the code).

- **InputPositionsAreOffset** (short form: **ipo**); `bool`
  The position offset: 1 (as in HART) if this variable is **true** or 0 (as in CART) if it is set to **false**.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- **TypeIncluded** (short form: **ti**); `bool`
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- **UseExtras** (short form: **ue**); `bool( )`
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.4 ARTStellarParticles data object

Short form: **asp**

This object represents the part of ART particles that represent stars.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`
  The index of a particle variable that should be used for computing the spatial density. A

special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.

- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.
  - **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
  - **DownsampleMode**=3 selects particles to load at random.
  - **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/DownsampleFactor` first particles will be selected.
  - **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.

- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- **TypeIncluded** (short form: **ti**); `bool`
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- **UseExtras** (short form: **ue**); `bool( )`
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.5 ARTVariables data object

Short form: **av**

This object represents the ART gas-dynamic and other variables that live on the Adaptive Mesh, loaded from .d files.

Available properties:

- **ConvertToAbundance** (short form: **ca**); `bool`
  A boolean property that converts the densities of atomic species to their abundances during loading of the data file.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DeclaredVarIncluded** (short form: **dvi**); `bool[44]`
  A boolean array property that specifies whether a variables with a given index should be included. Setting some of the components of this property to `false` will exclude the respective variables from the loaded data - to reduce the memory use and speed up rendering.
- **DensityUnit** (short form: **du**); `int`
  The units for the density: a value of 0 will keep the density in code units (in units of the mean total density), a value of 1 switches density units to the mean baryonic density, and a value of 2 switched density units to cm^{-3}.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.

- **MaxLevel** (short form: **maxl**); `int`
  The maximum ART mesh level to load for the variables (.d) file.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **OptimizeCache** (short form: **oc**); `bool`
  A boolean property that specifies how the ART variables data are placed in memory. The default value (0) places the data as it is written in the file; a value of 1 will reorder the mesh to optimize future memory access at the expense of extra computational time. Setting this property to 1 only makes sense if you plan to work on agiven data file for some time.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.6 ARTVelocityField data object

Short form: **avf**

This object represents the ART velocity field treated as a vector field (rather than individual components).

Available properties:

- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on

the scene as **Color Bar** captions.

- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.7 GADGETBoundaryParticles data object G

Short form: **gzp**

This object represents the GADGET boundary particles.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`
  The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.
- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - ◊ **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - ◊ **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects

every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.

- ◊ **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
- ◊ **DownsampleMode**=3 selects particles to load at random.
- ◊ **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/`**`DownsampleFactor`** first particles will be selected.
- ◊ **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.

- ♦ **Erase** (short form: **e**); **`bool( )`**
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- ♦ **FileName** (short form: **fn**); **`string( )`**
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- ♦ **Load** (short form: **l**); **`bool( string )`**
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- ♦ **Name** (short form: **n**); **`string[*]`**
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- ♦ **Range** (short form: **r**); **`pair[*]`**
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- ♦ **ResetOnLoad** (short form: **rol**); **`bool`**
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- ♦ **Stretch** (short form: **s**); **`string[*]`**
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- ♦ **TypeIncluded** (short form: **ti**); **`bool`**
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- ♦ **UseExtras** (short form: **ue**); **`bool( )`**
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.8 GADGETBulgeParticles data object G

Short form: **gbp**

This object represents the GADGET bulge particles.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`
  The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.
- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.
  - **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
  - **DownsampleMode**=3 selects particles to load at random.
  - **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/DownsampleFactor` first particles will be selected.
  - **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example,

Data.Scalars.Erase() is equivalent to **DataReader.Erase("Scalars")**.

- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- **TypeIncluded** (short form: **ti**); `bool`
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- **UseExtras** (short form: **ue**); `bool( )`
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.9 GADGETDiskParticles data object G

Short form: **gdp**

This object represents the GADGET disk particles.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`

The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.

- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.
  - **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
  - **DownsampleMode**=3 selects particles to load at random.
  - **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/`**`DownsampleFactor`** first particles will be selected.
  - **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like

`((0.1,1),(2,3),(10,20))`.

- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- **TypeIncluded** (short form: **ti**); `bool`
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- **UseExtras** (short form: **ue**); `bool( )`
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.10 GADGETGasParticles data object G

Short form: **ggp**

This object represents the GADGET Gas particles.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`
  The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.
- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - ◊ **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.

- ◊ **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.
- ◊ **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
- ◊ **DownsampleMode**=3 selects particles to load at random.
- ◊ **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/`**`DownsampleFactor`** first particles will be selected.
- ◊ **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.

♦ **Erase** (short form: **e**); **`bool( )`**
A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.

♦ **FileName** (short form: **fn**); **`string( )`**
The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.

♦ **Load** (short form: **l**); **`bool( string )`**
A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.

♦ **Name** (short form: **n**); **`string[*]`**
A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.

♦ **Range** (short form: **r**); **`pair[*]`**
An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.

♦ **ResetOnLoad** (short form: **rol**); **`bool`**
A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.

♦ **Stretch** (short form: **s**); **`string[*]`**
A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

♦ **TypeIncluded** (short form: **ti**); **`bool`**
A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.

♦ **UseExtras** (short form: **ue**); **`bool( )`**
A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.11 GADGETHaloParticles data object G

Short form: **ghp**

This object represents the GADGET halo particles.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`
  The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.
- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.
  - **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
  - **DownsampleMode**=3 selects particles to load at random.
  - **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/DownsampleFactor` first particles will be selected.
  - **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example,

Data.Scalars.Erase() is equivalent to **DataReader.Erase("Scalars")**.

- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- **TypeIncluded** (short form: **ti**); `bool`
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- **UseExtras** (short form: **ue**); `bool( )`
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.12 GADGETStellarParticles data object G

Short form: **gsp**

This object represents the GADGET stellar particles (stars).

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`

The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.

- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - ◊ **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - ◊ **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.
  - ◊ **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
  - ◊ **DownsampleMode**=3 selects particles to load at random.
  - ◊ **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/`**`DownsampleFactor`** first particles will be selected.
  - ◊ **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like

`((0.1,1),(2,3),(10,20))`.

- **ResetOnLoad** (short form: **rol**); **`bool`**
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); **`string[*]`**
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- **TypeIncluded** (short form: **ti**); **`bool`**
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- **UseExtras** (short form: **ue**); **`bool( )`**
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.13 NativeVTKPolyData data object

Short form: **vtkp**

This object represents polygonal mesh data type in the VTK native format.

Available properties:

- **DataPresent** (short form: **dp**); **`bool( )`**
  A method that checks whether the data has been loaded.
- **Erase** (short form: **e**); **`bool( )`**
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); **`string( )`**
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); **`bool( string )`**
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); **`string[*]`**
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); **`pair[*]`**
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); **`bool`**
  A boolean switch toggling whether the range for available variables (property **Range**) is reset

to the range of the data in the current file after each file load.

- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.14 NativeVTKScalars data object

Short form: **vtks**

This object represents scalar data type in the VTK native format.

Available properties:

- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.15 NativeVTKTensors data object

Short form: **vtkt**

This object represents tensor data type in the VTK native format.

Available properties:

- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.16 NativeVTKVectors data object

Short form: **vtkv**

This object represents vector data type in the VTK native format.

Available properties:

- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.

- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.17 Particles data object

Short form: **p**

This object represents the built-in Particles data type.

Available properties:

- **AddOrderAsVar** (short form: **ov**); `bool`
  A boolean switch that specifies whether the order of particles in the data file should be stored as an additional particle variable. This can be useful for plotting, for example, trajectories, when individual particles in the data file are actually locations of the same object at different times.
- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **DensityVar** (short form: **dv**); `int`
  The index of a particle variable that should be used for computing the spatial density. A special value of 0 will compute the density of particles themselves, without weighting by the variable value. Setting this property to -1 would disable the density computation. IFrIT uses VTK classes to compute the particle density. These classes are not very efficient, so

computing density for a large number of particles is slow. Density computation is done in parallel, so using more than 1 processor would speed up the calculation.

- **DownsampleFactor** (short form: **df**); `int`
  See DownsampleMode.
- **DownsampleMode** (short form: **dm**); `int`
  DownsampleMode and **DownsampleFactor** properties specify which subset of all particles is to be loaded from a Particle Set file. The **DownsampleFactor** is the inverse of the fraction of all particles which must be loaded (i.e. **DownsampleFactor**=10 will load every tenth particle; but notice exceptions for **DownsampleMode**=1 and **DownsampleMode**=2 modes), while the **DownsampleMode** property specifies the way the particles must be subsampled, and takes values from 0 to 5:
  - **DownsampleMode**=0 selects every **DownsampleFactor** particle in the order particles are present in the data file.
  - **DownsampleMode**=1 assumes that particles are set on a square 2D mesh and selects every **DownsampleFactor** particle along each dimension. For example, if the total number of particles is 16 and **DownsampleFactor**=2, then particles are assumed to be located on a 4 by 4 mesh and particles 1 (i=1,j=1), 3 (i=3,j=1), 9 (i=1,j=3), and 11 (i=3,j=3) are selected.
  - **DownsampleMode**=2 is similar to **DownsampleMode**=1, but assumes that particles are set on a cubic 3D mesh and every **DownsampleFactor** particle along each of the 3 dimensions is selected.
  - **DownsampleMode**=3 selects particles to load at random.
  - **DownsampleMode**=4 selects the particles from the data file in order, until a needed number of particles is loaded. I.e., if the total number of particles in the file is `n`, then `n/`**`DownsampleFactor`** first particles will be selected.
  - **DownsampleMode**=5 selects the particles from the data file in order, but counting from the end of the file.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.

- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.
- **TypeIncluded** (short form: **ti**); `bool`
  A boolean array property which specifies whether the particular type of particles is loaded or not. This property only makes sense for extensions that support several different types of particles.
- **UseExtras** (short form: **ue**); `bool( )`
  A boolean read-only property that reports whether this particle data type supports extra properties (like adding particle order and/or particle density as extra variables).

3.3.18 Scalars data object

Short form: **s**

This object represents the built-in Scalars data type.

Available properties:

- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Smooth** (short form: **sm**); `bool( int , float )`
  This method (script function) will smooth the given variable with a Gaussian kernel of a given size (measured in cell sized of the grid). For example, a call to **Scalars.Smooth(2,1.5)**

will smooth the field of the scalar variable #2 with a Gaussian of width 1.5 cell sizes. Such smoothing will make eliminate a lot of small-scale noise (i.e., it is a low-pass filter).

- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.19 Tensors data object

Short form: **t**

This object represents the built-in Tensors data type.

Available properties:

- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

3.3.20 Vectors data object

Short form: **v**

This object represents the built-in Vectors data type.

Available properties:

- **DataPresent** (short form: **dp**); `bool( )`
  A method that checks whether the data has been loaded.
- **Erase** (short form: **e**); `bool( )`
  A method (script function) that erases this type of data. It is equivalent to calling **DataReader.Erase(...)** with this data type as an argument. For example, **Data.Scalars.Erase()** is equivalent to **DataReader.Erase("Scalars")**.
- **FileName** (short form: **fn**); `string( )`
  The name of last data file read. This property is read-only - assigning it a name of a file will not cause this file to be read. Use the **DataReader.Load(...)** method for loading data.
- **Load** (short form: **l**); `bool( string )`
  A method (script function) that loads this type of data from a given file. It is equivalent to calling **DataReader.Load(...)** with this data type and a file name as arguments. For example, **Data.Scalars.Load("mydata.bin")** is equivalent to **DataReader.Load("Scalars","mydata.bin")**.
- **Name** (short form: **n**); `string[*]`
  A string-valued **Name** array sets the names of all available variables. Names are displayed on the scene as **Color Bar** captions.
- **Range** (short form: **r**); `pair[*]`
  An array of ranges for available variables. Each component of an array is a pair `(min,max)` of values that are used to map colors to the numeric values of a particular variable: the value is clamped to lie within the range and then mapped to a given **palette** with the given stretch. For example, for the data with 3 variables the value of this property may look in Python like `((0.1,1),(2,3),(10,20))`.
- **ResetOnLoad** (short form: **rol**); `bool`
  A boolean switch toggling whether the range for available variables (property **Range**) is reset to the range of the data in the current file after each file load.
- **Stretch** (short form: **s**); `string[*]`
  A **Stretch** property sets the stretch (either "linear" or "logarithmic") for each available variable. String names can be shortened to the shortest recognizable sub-string, i.e. "lin" stands for the linear stretch.

A Appendices

A.1 Codes For Writing IFrIT Data Files

A.1.1 Code Examples

Examples of computer codes for writing IFrIT data files are available in `docs` directory of IFrIT source distribution.

A.1.2 Fortran

```
C
C  Write to text uniform scalars data file
C
      subroutine WriteIFrITUniformScalarsTxtFile(n1,n2,n3,var1,var2,var3,
     .     filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 var1(n1,n2,n3)
      real*4 var2(n1,n2,n3) ! Three scalar variables
      real*4 var3(n1,n2,n3)
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename)
      write(1,*) n1, n2, n3
      do k=1,n3
         do j=1,n2
            do i=1,n1
               write(1,*) var1(i,j,k), var2(i,j,k), var3(i,j,k)
            enddo
         enddo
      enddo
      close(1)
      return
      end
C
C  Write to binary uniform scalars data file
C
      subroutine WriteIFrITUniformScalarsBinFile(n1,n2,n3,var1,var2,var3,
     .     filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 var1(n1,n2,n3)
      real*4 var2(n1,n2,n3) ! Three scalar variables
      real*4 var3(n1,n2,n3)
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename, form='unformatted')
      write(1) n1, n2, n3
      write(1) (((var1(i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((var2(i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((var3(i,j,k),i=1,n1),j=1,n2),k=1,n3)
      close(1)
      return
      end
C
C  Write to text uniform vectors data file
C
      subroutine WriteIFrITUniformVectorsTxtFile(n1,n2,n3,vect,filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 vect(3,n1,n2,n3)  ! Vector field
```

```
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename)
      write(1,*) n1, n2, n3
      do k=1,n3
         do j=1,n2
            do i=1,n1
               write(1,*) vect(1,i,j,k), vect(2,i,j,k), vect(3,i,j,k)
            enddo
         enddo
      enddo
      close(1)
      return
      end
C
C  Write to binary uniform vectors data file
C
      subroutine WriteIFrITUniformVectorsBinFile(n1,n2,n3,vect,filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 vect(3,n1,n2,n3)  ! Vector field
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename, form='unformatted')
      write(1) n1, n2, n3
      write(1) (((vect(1,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((vect(2,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((vect(3,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      close(1)
      return
      end
C
C  Write to text uniform tensors data file
C
      subroutine WriteIFrITUniformTensorsTxtFile(n1,n2,n3,tens,filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 tens(6,n1,n2,n3)  ! Tensor field
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename)
      write(1,*) n1, n2, n3
      do k=1,n3
         do j=1,n2
            do i=1,n1
               write(1,*) tens(1,i,j,k), tens(2,i,j,k), tens(3,i,j,k),
     .              tens(4,i,j,k), tens(5,i,j,k), tens(6,i,j,k)
            enddo
         enddo
      enddo
      close(1)
      return
      end
C
C  Write to binary uniform tensors data file
C
      subroutine WriteIFrITUniformTensorsBinFile(n1,n2,n3,tens,filename)
      integer n1, n2, n3  ! Size of the computational mesh in 3 directions
      real*4 tens(6,n1,n2,n3)  ! Tensor field
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename, form='unformatted')
      write(1) n1, n2, n3
      write(1) (((tens(1,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(2,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(3,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(4,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(5,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      write(1) (((tens(6,i,j,k),i=1,n1),j=1,n2),k=1,n3)
      close(1)
      return
      end
C
C  Write to text basic particles data file
C
```



```
      subroutine WriteIFrITBasicParticlesTxtFile(n,xl,yl,zl,xh,yh,zh,
     .     x,y,z,attr1,attr2,attr3,filename)
      integer n  ! Number of particles
      real*4 xl, yl, zl, xh, yh, zh  ! Bounding box
      real*4 x(n), y(n), z(n)  ! Particle positions (can be real*8)
      real*4 attr1(n), attr2(n), attr3(n)  ! Particle attributes
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename)
      write(1,*) n
      write(1,*) xl, yl, zl, xh, yh, zh
      do i=1,n
         write(1,*) x(i), y(i), z(i), attr1(i), attr2(i), attr3(i)
      enddo
      close(1)
      return
      end
C
C  Write to binary basic particles data file
C
      subroutine WriteIFrITBasicParticlesBinFile(n,xl,yl,zl,xh,yh,zh,
     .     x,y,z,attr1,attr2,attr3,filename)
      integer n  ! Number of particles
      real*4 xl, yl, zl, xh, yh, zh  ! Bounding box
      real*4 x(n), y(n), z(n)  ! Particle positions (can be real*8)
      real*4 attr1(n), attr2(n), attr3(n)  ! Particle attributes
      character*(*) filename  ! Name of the file
      open(unit=1, file=filename, form='unformatted')
      write(1) n
      write(1) xl, yl, zl, xh, yh, zh
      write(1) (x(i),i=1,n)
      write(1) (y(i),i=1,n)
      write(1) (z(i),i=1,n)
      write(1) (attr1(i),i=1,n)
      write(1) (attr2(i),i=1,n)
      write(1) (attr3(i),i=1,n)
      close(1)
      return
      end
```

A.1.3 C

```
#include <stdio.h>

/*  Write to text uniform scalars data file  */

int WriteIFrITUniformScalarsTxtFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *var1, float *var2, float *var3, /* Three scalar variables */
    char *filename) /* Name of the file */
{
   int i, j, k; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   fprintf(F,"%d %d %d\n",n1,n2,n3);
   for(k=0; k<n3; k++)
   {
      for(j=0; j<n2; j++)
      {
         for(i=0; i<n1; i++)
         {
            fprintf(F,"%g %g %g\n",var1[i+n1*(j+n2*k)],
                                   var2[i+n1*(j+n2*k)],
                                   var3[i+n1*(j+n2*k)]);
         }
      }
   }
```

```
   fclose(F);
   return 0;
}

/*  Write to binary uniform scalars data file  */

int WriteIFrITUniformScalarsBinFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *var1, float *var2, float *var3, /* Three scalar variables */
    char *filename) /* Name of the file */
{
   int ntemp; FILE *F; /* ntemp should be declared long on a 16-bit machine */
   F = fopen(filename,"wb"); if(F == NULL) return 1;
   ntemp = 12;
   fwrite(&ntemp,4,1,F);
   fwrite(&n1,4,1,F);
   fwrite(&n2,4,1,F);
   fwrite(&n3,4,1,F);
   fwrite(&ntemp,4,1,F);
   ntemp = 4*n1*n2*n3;
   fwrite(&ntemp,4,1,F); fwrite(var1,4,n1*n2*n3,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(var2,4,n1*n2*n3,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(var3,4,n1*n2*n3,F); fwrite(&ntemp,4,1,F);
   fclose(F);
   return 0;
}


/*  Write to text uniform vectors data file  */

int WriteIFrITUniformVectorsTxtFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *vect, /* Vector field */
    char *filename) /* Name of the file */
{
   int i, j, k; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   fprintf(F,"%d %d %d\n",n1,n2,n3);
   for(k=0; k<n3; k++)
   {
      for(j=0; j<n2; j++)
      {
         for(i=0; i<n1; i++)
         {
            fprintf(F,"%g %g %g\n",vect[0+3*(i+n1*(j+n2*k))],
                                   vect[1+3*(i+n1*(j+n2*k))],
                                   vect[2+3*(i+n1*(j+n2*k))]);
         }
      }
   }
   fclose(F);
   return 0;
}

/*  Write to binary uniform vectors data file  */

int WriteIFrITUniformVectorsBinFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *vect, /* Vector field */
    char *filename) /* Name of the file */
{
   int i, ntemp; FILE *F; /* ntemp should be declared long on a 16-bit machine */
   F = fopen(filename,"wb"); if(F == NULL) return 1;
   ntemp = 12;
   fwrite(&ntemp,4,1,F);
   fwrite(&n1,4,1,F);
   fwrite(&n2,4,1,F);
   fwrite(&n3,4,1,F);
   fwrite(&ntemp,4,1,F);
```



```
   ntemp = 4*n1*n2*n3;
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(vect+3*i+0,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(vect+3*i+1,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(vect+3*i+2,4,1,F);
   fwrite(&ntemp,4,1,F);
   fclose(F);
   return 0;
}

/*  Write to text uniform tensors data file  */

int WriteIFrITUniformTensorsTxtFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *tens, /* Tensor field */
    char *filename) /* Name of the file */
{
   int i, j, k; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   fprintf(F,"%d %d %d\n",n1,n2,n3);
   for(k=0; k<n3; k++)
   {
      for(j=0; j<n2; j++)
      {
         for(i=0; i<n1; i++)
         {
            fprintf(F,"%g %g %g %g %g %g\n",tens[0+6*(i+n1*(j+n2*k))],
                                            tens[1+6*(i+n1*(j+n2*k))],
                                            tens[2+6*(i+n1*(j+n2*k))],
                                            tens[3+6*(i+n1*(j+n2*k))],
                                            tens[4+6*(i+n1*(j+n2*k))],
                                            tens[5+6*(i+n1*(j+n2*k))]);
         }
      }
   }
   fclose(F);
   return 0;
}

/*  Write to binary uniform tensors data file  */

int WriteIFrITUniformTensorsBinFile(
    int n1, int n2, int n3, /* Size of the computational mesh in 3 directions */
    float *tens, /* Tensor field */
    char *filename) /* Name of the file */
{
   int i, ntemp; FILE *F; /* ntemp should be declared long on a 16-bit machine */
   F = fopen(filename,"wb"); if(F == NULL) return 1;
   ntemp = 12;
   fwrite(&ntemp,4,1,F);
   fwrite(&n1,4,1,F);
   fwrite(&n2,4,1,F);
   fwrite(&n3,4,1,F);
   fwrite(&ntemp,4,1,F);
   ntemp = 4*n1*n2*n3;
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+0,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+1,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+2,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
```

```
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+3,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+4,4,1,F);
   fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F);
   for(i=0; i<n1*n2*n3; i++) fwrite(tens+6*i+5,4,1,F);
   fwrite(&ntemp,4,1,F);
   fclose(F);
   return 0;
}

/*  Write to text basic particles data file  */

int WriteIFrITBasicParticlesTxtFile(
    int n, /* Number of particles */
    float xl, float yl, float zl, float xh, float yh, float zh, /* Bounding box */
    float *x, float *y, float *z, /* Particle positions (can be double) */
    float *attr1, float *attr2, float *attr3, /* Particle attributes */
    char *filename) /* Name of the file */
{
   int i; FILE *F;
   F = fopen(filename,"w"); if(F == NULL) return 1;
   fprintf(F,"%d\n",n);
   fprintf(F,"%g %g %g %g %g %g\n",xl,yl,zl,xh,yh,zh);
   for(i=0; n>i; i++)
   {
      fprintf(F,"%g %g %g %g %g %g\n",x[i],y[i],z[i],
                          attr1[i],attr2[i],attr3[i]);
   }
   fclose(F);
   return 0;
}

/*  Write to binary basic particles data file  */

int WriteIFrITBasicParticlesBinFile(
    int n, /* Number of particles */
    float xl, float yl, float zl, float xh, float yh, float zh, /* Bounding box */
    float *x, float *y, float *z, /* Particle positions (can be double) */
    float *attr1, float *attr2, float *attr3, /* Particle attributes */
    char *filename) /* Name of the file */
{
   int i, ntemp; FILE *F;
   F = fopen(filename,"wb"); if(F == NULL) return 1;
   ntemp = 4;
   fwrite(&ntemp,4,1,F); fwrite(&n,4,1,F); fwrite(&ntemp,4,1,F);
   ntemp = 24;fwrite(&ntemp,4,1,F);
   fwrite(&xl,4,1,F);
   fwrite(&yl,4,1,F);
   fwrite(&zl,4,1,F);
   fwrite(&xh,4,1,F);
   fwrite(&yh,4,1,F);
   fwrite(&zh,4,1,F);
   fwrite(&ntemp,4,1,F);
   ntemp = sizeof(x[0])*n;
   fwrite(&ntemp,4,1,F); fwrite(x,sizeof(x[0]),n,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(y,sizeof(y[0]),n,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(z,sizeof(z[0]),n,F); fwrite(&ntemp,4,1,F);
   ntemp = 4*n;
   fwrite(&ntemp,4,1,F); fwrite(attr1,4,n,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(attr2,4,n,F); fwrite(&ntemp,4,1,F);
   fwrite(&ntemp,4,1,F); fwrite(attr3,4,n,F); fwrite(&ntemp,4,1,F);
   fclose(F);
   return 0;
}
```


A.1.4 IDL

```
;
;  Write to text scalar field data file
;
pro WriteIFrITUniformScalarsTxtFile, n1, n2, n3, var1, var2, var3, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; var1, var2, var3: three scalar variables
; filename: name of the file
openw, 1, filename
printf, 1, n1, n2, n3
for k=0,n3-1 do $
for j=0,n2-1 do $
for i=0,n1-1 do $
printf, 1, var1[i,j,k], var2[i,j,k], var3[i,j,k]
close, 1
end
;
;  Write to binary scalar field data file
;
pro WriteIFrITUniformScalarsBinFile, n1, n2, n3, var1, var2, var3, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; var1, var2, var3: three scalar variables
; filename: name of the file
openw, 1, filename, /F77_UNFORMATTED
writeu, 1, long([n1,n2,n3])
writeu, 1, var1
writeu, 1, var2
writeu, 1, var3
close, 1
end
;
;  Write to text uniform vectors data file
;
pro WriteIFrITUniformVectorsTxtFile, n1, n2, n3, vect, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; vect[3,n1,n2,n3]: uniform vectors
; filename: name of the file
openw, 1, filename
printf, 1, n1, n2, n3
for k=0,n3-1 do $
for j=0,n2-1 do $
for i=0,n1-1 do $
printf, 1, vect[0,i,j,k], vect[1,i,j,k], vect[2,i,j,k]
close, 1
end
;
;  Write to binary uniform vectors data file
;
pro WriteIFrITUniformVectorsBinFile, n1, n2, n3, vect, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; vect[3,n1,n2,n3]: uniform vectors
; filename: name of the file
openw, 1, filename, /F77_UNFORMATTED
writeu, 1, long([n1,n2,n3])
writeu, 1, vect[0,*,*,*]
writeu, 1, vect[1,*,*,*]
writeu, 1, vect[2,*,*,*]
close, 1
end
;
;  Write to text uniform tensors data file
;
pro WriteIFrITUniformTensorsTxtFile, n1, n2, n3, tens, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; tens[6,n1,n2,n3]: uniform tensors
; filename: name of the file
openw, 1, filename
```

```
printf, 1, n1, n2, n3
for k=0,n3-1 do $
for j=0,n2-1 do $
for i=0,n1-1 do $
printf, 1, tens[0,i,j,k], tens[1,i,j,k], tens[2,i,j,k], $
           tens[3,i,j,k], tens[4,i,j,k], tens[5,i,j,k]
close, 1
end
;
;  Write to binary uniform tensors data file
;
pro WriteIFrITUniformTensorsBinFile, n1, n2, n3, tens, filename
; n1, n2, n3: size of the computational mesh in 3 directions
; tens[6,n1,n2,n3]: uniform vectors
; filename: name of the file
openw, 1, filename, /F77_UNFORMATTED
writeu, 1, long([n1,n2,n3])
writeu, 1, tens[0,*,*,*]
writeu, 1, tens[1,*,*,*]
writeu, 1, tens[2,*,*,*]
writeu, 1, tens[3,*,*,*]
writeu, 1, tens[4,*,*,*]
writeu, 1, tens[5,*,*,*]
close, 1
end
;
;  Write to text basic particles data file
;
pro WriteIFrITBasicParticlesTxtFile, n, xl, yl, zl, xh, yh, zh, $
x, y, z, attr1, attr2, attr3, filename
; n: number of particles
; xl, yl, zl, xh, yh, zh: bounding box (can be double)
; x, y, z: particle positions (can be double)
; attr1, attr2, attr3: particle attributes/
; filename: name of the file
openw, 1, filename
printf, 1, n
printf, 1, xl, yl, zl, xh, yh, zh
for i=0,n-1 do $
printf, 1, x[i], y[i], z[i], attr1[i], attr2[i], attr3[i]
close, 1
end
;
;  Write to binary basic particles data file
;
pro WriteIFrITBasicParticlesBinFile, n, xl, yl, zl, xh, yh, zh, $
x, y, z, attr1, attr2, attr3, filename
; n: number of particles
; xl, yl, zl, xh, yh, zh: bounding box (can be double)
; x, y, z: particle positions (can be double)
; attr1, attr2, attr3: particle attributes/
; filename: name of the file
openw, 1, filename, /F77_UNFORMATTED
writeu, 1, long(n)
writeu, 1, float([xl,yl,zl,xh,yh,zh])
writeu, 1, x
writeu, 1, y
writeu, 1, z
writeu, 1, attr1
writeu, 1, attr2
writeu, 1, attr3
close, 1
end
```


A.2 License Agreement

A.2.1 Overview

The standard edition of IFrIT is distributed under the GNU GPL License. Extensions of IFrIT may impose their own licenses.

For those not familiar with the GNU GPL, the license basically allows you to:

- Use the IFrIT software and source code at no charge.
- Distribute verbatim copies of the software in source form or as binaries you create.
- Sell verbatim copies of the software for a media fee, or sell support for the software.
- Distribute or sell your own modified version of IFrIT so long as the source code is made available under the GPL.

What this license **does not** allow you to do is make changes or add features to IFrIT and then sell a binary distribution without source code. You must provide source for any changes or additions to the software, and all code must be provided under the GPL.

A.2.2 GNU General Public License

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it. For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSETERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

  These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF

MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

A.2.3 ART Extension License

ART Extension of IFrIT adds capabilities to work with Oct-based Adaptive Mesh Refinement data and read directly the data files created by the ART (Hydrodynamic Adaptive Refinement Tree) code. Since ART is not open source, the extension license does not allow you to distribute IFrIT with ART extension, or to use it for any commercial purpose.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

This is proprietary software. Redistribution and modification in any form are expressly prohibited. You are given a non-exclusive right to use this software in original source and binary forms for your personal, non-commercial needs free of charge.

NO WARRANTY

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

END OF TERMS AND CONDITIONS

