ABI Software Book

Release 0.2-PMR@HARMONY

Auckland Bioengineering Institute

Mar 27, 2017

Contents

This book is a collection of documentation covering software used within the ABI. This includes software developed internally and also other commonly used applications.

Contents:

CHAPTER 1

About the ABI

History, research groups, opportunities etc etc.

Contents:

ABI Resources

Documentation covering what resources are available for Msc and PhD students to carry out their research.

Computer hardware and software

- Details of computers available to students.
- Operating system options
- List of software packages available for installation
- License issues
- Information about help, IT staff etc.

workshop

Other

CHAPTER 2

ABI research case studies

Examples of research at the ABI - goals, tools and techniques used, outcomes.

To serve as examples of how to use the available tools, as well as promotional material.

If you would like your research to appear in this section, please let me know, or fork this documentation on GitHub!

CHAPTER 3

CellML API user documentation

CHAPTER 4

Cmgui user documentation

This documentation provides an overview of the functionality of the cmgui application, as well as some information about the architecture of the software, and some of the concepts behind the visualizations.

This documentation is current for cmgui version 2.9.0.

Migration documentation for users of previous versions

Some of the changes in recent versions of cmgui will cause some old commands to break - changes may need to be made to some old command files. Below are links to documents detailing these changes.

Changes in recent versions

Migrating to Cmgui 2.8

Scenes and graphics filters

The biggest change in cmgui 2.8 is that graphics are now associated with regions and not stored in a separate ‘scene graph’. The Scene Editor displays the region tree, and each region has a graphical ‘rendition’ consisting of a list of graphics (lines, surfaces, node_points etc.) which visualize the fields of that region. This change should not affect most users, because previous versions of cmgui made the scene graph mirror the flat list of regions and groups used in most models and the commands to set up these graphics should work as before.

However, if your command file created new scenes (or modified certain attributes of scene ‘default’), or your model consisted of more than just a simple list of regions and groups (i.e. regions within regions within the root region) you will need to make some changes.

Rendition or ‘graphical element’

Since graphics now belong to a region, the commands to create them now take the region path instead of the name of the ‘graphical element’ (now termed ‘rendition’) on a particular scene:

```
gfx modify g_element REGION_PATH ...
```

Argument scene NAME is now redundant and ignored. Since the automatic name for the 'g_element' was the region or group name, most previous commands work as before.

Scenes

Scenes remain as the objects determining which graphics are displayed in a window or exported by various commands. However they are considerably simplified in that their main attributes are (1) the top region they show graphics for and (2) A graphics 'filter' object which controls which graphics from the region tree are visible (see later).

The command for creating or modifying a scene is as follows:

```
Usage : gfx define scene ??
  SCENE_NAME
<add_light LIGHT_NAME|none[none]>
<region PATH_TO_REGION[/]>
<remove_light LIGHT_NAME|none[none]>
<filter GRAPHICS_FILTER_NAME|none[none]>
```

(gfx modify scene is the same as gfx define scene. Note future plans are to remove lights from scenes and list them as special graphics as part of a rendition.)

The gfx create scene command has been removed because invariably the command file must be manually updated to replace it, and it is now no longer necessary to create 'child' scenes to make hierarchical graphics. Cmgui developers are happy to help with migration.

Graphics filters

Graphics filters are new objects which control which graphics are shown in a scene. Each filter has a criterion for matching attributes of a graphic or region/rendition, and if the result is true the graphic is shown. The initial graphics types include:

- matching visibility flags
- matching graphic name
- matching (within) region path
- logical operator OR
- logical operator AND

Furthermore, all criteria can be inverted to create logical not:

```
Usage : gfx define graphics_filter ??
  GRAPHICS_FILTER_NAME
      *  Filter to set up what will be and what will not be
      *  included in a scene. The optional inverse_match flag
      *  will invert the filter's match criterion. The behaviour
      *  is to show matching graphic with the matching criteria.
      * <match_graphic_name> filters graphic with the matching
      *  name. <match_visibility_flags> filters graphic with
      *  the setting on the visibility flag. <match_region_path>
      *  filters graphic in the specified region or its subregion.
      * <operator_or> filters the scene using the logical operation
      *  'or' on a collective of filters. <operator_and> filters
      *  the scene using the logical operation 'and' on a collective
      *  of filters. Filters created earlier can be added or
      *  removed from the <operator_or> and <operator_and> filter.
<operator_or
  <add_filters[add_filters]|remove_filters>
      FILTER_NAMES >
```

```
<operator_and
  <add_filters[add_filters]|remove_filters>
      FILTER_NAMES >
<match_graphic_name MATCH_NAME>
<match_visibility_flags>
<match_region_path REGION_PATH>
<inverse_match|normal_match[normal_match]>
```

The default filter matches the familiar visibility flags of the region/rendition and each graphic:

```
gfx define graphics_filter default normal_match match_visibility_flags;
```

The following makes a filter that shows all graphics satisfying the above default OR if graphics is named "bob":

```
gfx define graphics_filter bob normal_match match_graphic_name "bob";
gfx define graphics_filter default_or_bob normal_match operator_or add_filters␣
↪default bob;
gfx define scene default filter default_or_bob;
```

Note it is easy for us to add new filter types as needed by users.

Drawing static graphics

Cmgui now associates all graphics with a region rather than a separate 'scene graph', so all previous commands to create and draw 'static' graphics on a scene are removed or changed. The equivalent functionality is achieved by adding a 'point' graphic to a region rendition in the scene editor, which can show any glyph with controllable scale, offset, material and optional label. The commands for reproducing the graphics can be obtained using `gfx list g_element REGION_PATH commands`.

- `gfx create axes` now just writes a migration note as it is redundant. Several axes objects are predefined in the glyph list to be used with point graphics. See comment on `gfx draw` below.
- `gfx create annotation` has been removed. You must now define a string field in the respective region with the required text (`gfx define field ![REGION_PATH/]NAME string constant "Your text here"`) and use it as a label, probably with glyph none, on a point graphic.
- `gfx create colour_bar` works as before but the colour bar is now put in the list of glyphs able to be shown with any point graphic.
- `gfx create lines/surfaces/node_points ...` have been removed altogether. You must now use `gfx modify g_element REGION_PATH lines/surfaces/node_points ...`.
- 'gfx draw' now creates a point graphic in the root region rendition with the glyph matching the graphics name specified. If you had previously created axes with the default name "axes" it finds the glyph however the scaling, offset and material are lost: edit these in the scene editor. The scene is ignored by this command so you will need to make changes if you were drawing to multiple scenes, particularly an overlay scene (see migration notes for overlay graphics below).
- 'gfx erase' has been removed.

Overlay graphics and coordinate systems

Previous versions of cmgui required an 'overlay scene' attribute to be set for each graphics window. Any graphics drawn in the selected overlay scene were drawn in a window-relative coordinate system on that window.

This has been replaced by a far simpler and more powerful mechanism. Each graphic in the scene editor has a 'graphics coordinate system' which can be:

- LOCAL = subject to the graphical transformations of the renditions in the region tree relative to world (the default).

- WORLD = in the world coordinate system of the top region of the scene
- NORMALISED_WINDOW_FILL = ranges from ![-1,+1] across all dimension of window so distorting if window is non-square. This was the mode used by the overlay scene.
- NORMALISED_WINDOW_FIT_LEFT/RIGHT/BOTTOM/TOP = ranges from ![-1,+1] in the largest square fitting in the window, aligned to the specified side. Non-distorting so perferable to NORMALISED_WINDOW_FILL.
- WINDOW_PIXEL_BOTTOM_LEFT/TOP_LEFT = In screen pixels from (0,0) at the specified origin of the window, with +x to the right, +y up. TOP_LEFT has negative y coordinates on screen.

Choosing any window-relative coordinate system causes the graphic to be drawn as an overlay, i.e. on top of all non-overlay graphics, on rendering windows. Window-relative graphics cannot be exported to VRML and other formats without a viewport.

Future plans are to allow layers to be specified independently from the graphics coordinate system, with layers 'background', 'default' and 'overlay' predefined and the ability to add more layers and control whether the depth buffer is cleared between layers.

Removal of 'general settings' for graphics

In previous versions of cmgui, the graphical rendition of a region (`g_element` commands) had a set of 'general settings' controlling the 'discretisation' (number of line segments used to approximate curved element boundaries), and also a default coordinate field to apply to graphics which do not have the coordinate field specified.

For cmgui 2.8 all these general settings have been removed and must be specified for each graphic in the rendition. This has the benefit of allowing different graphics to use different discretisations, which are now set via 'tessellation' objects (see later).

To minimize migration issues, all previous `gfx modify g_element REGION_PATH general ...` options are read and become defaults for subsequent g_element commands to add graphics. If the general 'element_discretization' attribute is set then on creating new graphics requiring a tessellation it finds or creates one giving the same effect as the discretization; it will have an automatically generated name such as 'temp2'. Changing these general options now has no effect on graphics that have already been created. The only non-deprecated `g_element general` command option is to clear all graphics from the rendition.

Note that region renditions still have transformation attributes which give the 4x4 transformation from local to parent coordinate systems.

Tessellation

Tessellation objects have been introduced to replace the general 'element_discretization' attribute and solve several problems:

- they allow any number of graphics – and not just from the same region – to share tessellation settings, allowing graphics quality across complicated model visualisations to be changed from a few controls.
- they allow tessellation quality to automatically switch from minimum for linear basis functions to fine for non-linear bases and coordinate systems. (Note: bilinear and trilinear Lagrange are considered linear even though they have a few quadratic or cubic product terms.)
- we anticipate adding more options in future e.g. adaptive curvature-dependent triangulation, but graphics will still only require one object to be chosen.

However, tessellation objects do not solve one problem, that choosing a very large number causes Cmgui to lock up while all affected graphics are regenerated: take care when setting large values!

Note that elements_points and streamlines do not have a tessellation object set by default. Instead they have a separate fixed 'discretization' setting which defaults to "1*1*1" (i.e. 1 point or streamline per element), but you may add a tessellation which then acts as a multiplier on the discretization. The optional native_discretization field also acts as a multiplier.

Commands for defining tessellations are:

```
Usage : gfx define tessellation ??
  TESSELLATION_NAME
      * Defines tessellation objects which control how finite
      *  elements are subdivided into graphics. The minimum_divisions
      *  option gives the minimum number of linear segments
      *  approximating geometry in each xi dimension of the
      *  element. If the coordinate field of a graphic uses
      *  non-linear basis functions the minimum_divisions is
      *  multiplied by the refinement_factors to give the refined
      *  number of segments. Both minimum_divisions and refinement_factors
      *  use the last supplied number for all higher dimensions,
      *  so "4" = "4*4" and so on.
<minimum_divisions "#*#*..."["1"]{>=0}>
<refinement_factors "#*#*..."["1"]{>=0}>
```

You can also list all available tessellations with:

```
gfx list tessellation
```

The default tessellation exists from start-up, but can be edited:

```
gfx define tessellation default minimum_divisions "1" refinement_factors "4";
```

A tessellation editor dialog can be opened by clicking on the “Edit...” button beside the tessellation chooser in the scene editor.

Migrating to Cmgui 2.9

In the 2.9.0 release, significant changes have been made to the way ‘groups’ are handled. We have also removed the automatic creation of default lines on reading a model.

While we’ve taken every effort to minimise the impact of the changes, usually by making commands do the nearest equivalent behaviour, some breaks are unavoidable. So if your command files are not working as before, please try to see why in the following explanation, but if that doesn’t help, see the Cmgui development team. As always, if you find what looks like a bug in the software, please report it on our tracker.

Removal of ‘group regions’

Background

A Cmgui model is made up of a tree of objects called regions, each of which owns a set of nodes, elements, data points (an extra set of nodes) and fields defined on them, plus 0 or more child regions. Regions are like the folders in a file system; fields (and other ‘intra region’ objects such as sets of nodes and meshes) are like files in the folder. The root region is denoted by “/”, and region names in a path are separated by “/”.

A ‘group’ is a subset of the nodes, elements and data points owned by a region. Prior to this release, each group was created as a pseudo region, what we called a ‘group region’. This meant they looked like a region except they contained a subset of the objects of the parent/master region, and listed all fields of the master region (but could only address those defined over its subset). This also meant every group could individually have graphics associated with it. For users, the downsides of group regions were that they hid away the definitions of their subgroups so it was not possible to intersect or union them in field expressions, it was not possible to make invisible subgroups (they all had a graphical rendition), and it was impossible to make hierarchical groups and groups containing whole regions. For developers they have been a maintenance hazard and a feature we did not wish to expose to our API.

The Change

From Cmgui 2.9, groups are implemented as fields. A group field returns 1 at any location in the domain (at nodes, in elements etc.) which is in the group, 0 elsewhere. From the API you can add the whole region to a group, but this is not available through gfx commands. Group fields can have subregion groups in child regions, so a group tree can contain an arbitrary subset of a region tree. The group field can have zero or more 'subobject group fields' representing subsets of specific object types that are present in the group, each of which give the value 1 on the respective object type in the group, 0 otherwise. For example, if you read cube example a2 and write "gfx list fields" you will now see 5 new fields:

```
cube : group = generic group object
cube.cmiss_mesh_1d : sub_group_object = subset of "cmiss_mesh_1d" in group cube
cube.cmiss_mesh_2d : sub_group_object = subset of "cmiss_mesh_2d" in group cube
cube.cmiss_mesh_3d : sub_group_object = subset of "cmiss_mesh_3d" in group cube
cube.cmiss_nodes : sub_group_object = subset of "cmiss_nodes" in group cube
```

Note: In several new commands, "cmiss_nodes" is the name you use to reference the set of nodes in the region, "cmiss_data" references the set of data points, and "cmiss_mesh_Nd" references the 1, 2 and 3-D meshes owned by that region. In future we anticipate allowing any number of meshes to be created with their own names.

Since Cmgui 2.8, when you select objects in the graphics window, you are making a group which is automatically given the name "cmiss_selection", and any selected objects are added to subobject groups coordinated under it. For example, selecting a face element of the cube creates:

```
cmiss_selection : group
cmiss_selection.cmiss_mesh_2d : sub_group_object
```

There is now only one type of group; the previous group regions and the selection have been merged into the same class.

Prefer Regions to Groups

Shortly we will explain several of the workarounds that make most existing command files work as before, most of the time. But first we wish to recommend using regions instead of groups whenever possible, and if you can do this your command files will likely work just as before. Needlessly using a group means all your objects are listed twice, once for the region and once for the group, which consumes more memory. To migrate to regions, just change lines in EX files from:

```
Group name: NAME

to (and the leading "/" is required):

Region: /NAME
```

(The Region can have a full path e.g. /bob/fred; the group is just a name which is within the namespace of the surrounding region. At the start of parsing EX files, the file's root region "/" is active.)

Another approach is to specify the region on the read command "gfx read node/element/data region PATH", but bear in mind this will continue to create groups at the named path. This effectively maps the file's root region to the named region.

Warning: Be aware that there are still cases where you can't use regions, so continued use of groups will be necessary:

1. Whenever several element groups share common nodes. Elements can only reference nodes from the same region.

2. Certain features such as the compose/find_mesh_location fields that do not work inter-region.

We certainly wish to remedy these in time. We need to hear user input about what functionality they need to work between regions.

Possible breaks with the removal of group regions

1. Groups no longer have a rendition

Each region has a 'rendition' which is the set of graphics for visualising its fields. When groups were presented as regions, they too had their own rendition. We now interpret existing commands in the form:

```
gfx modify g_element [REGION_PATH]/GROUP_NAME ...
```

as:

```
gfx modify g_element [REGION_PATH]/ ... subgroup GROUP_NAME
```

Hence the graphics now belong to the parent region. All graphics now have an optional subgroup field and when supplied only shows primitives for which the field is non-zero. Hence the new group fields are all usable, but so is any other scalar field expressions (e.g. less_than(mag(coordinates), 100.0). When you re-list graphics with "gfx list g_element" you will see the new form. Note: this also replaces the "visibility_field" option on node_points and data_points so you will need to rename this.

The "gfx modify g_element [REGION_PATH]/GROUP_NAME" command clears only graphics using the group of the GROUP_NAME as the subgroup field. Beware that clearing the graphics of the parent region will also clear those referencing any subgroup.

2. Groups no longer have a rendition, hence have no individual 'default' arguments

Cmgui 2.8 (the last public release) deprecated all arguments to the "general" command for modifying graphics except for the clear command.

gfx modify g_element PATH general clear default_coordinate NAME element_discretization "#*#*..." ...

To keep old command files working, the options specified in the g_element command are stored and used as defaults when new graphics are created. With the removal of group regions, these no longer have independent defaults from each other and from the parent region. Continuing to rely on these defaults can produce spurious results; Cmgui now warns whenever these defaults change.

Conclusion: always prefer to specify coordinate field, tessellation object etc. for each individual graphic. Never use the general defaults: remove them from your command files.

3. Groups no longer have a rendition, hence no transformation

Each region rendition has a transformation attribute which transforms all graphics relative to the parent region rendition, but groups no longer have either. There is no workaround here except to use a child region instead. Please talk to Cmgui developers if you have particular needs here.

4. Can't have a group and field of same name

Groups are now fields, and no two fields can have the same name. You will have to change the name of one of them.

Workaround for particular commands

A number of commands inconsistently use the keyword region or group followed by a region path. Many of these commands have been migrated to identity if the path is to a group and work with it as before, however, we haven't fixed the inconsistent keywords.

1. gfx create|modify dgroup|ngroup|egroup

These all create or modify the new group fields but should otherwise work as before.

For egroup these now have an option to manage_subobjects which means when you add an element to the group, all faces, lines and nodes are added to the related group. This is the default so as to reproduce previous behaviour, but can also be turned off by specifying <no_manage_subobjects> (as managing subobjects is expensive and may be unnecessary).

2. gfx define field integration|xi_texture_coordinates

Added option to specify mesh as cmiss_mesh_1d, cmiss_mesh_2d or cmiss_mesh_3d, or group_name.cmiss_mesh_Nd for a subset of the mesh. Removed region option since node must be from current region.

3. gfx define field nodal_lookup|quaternion_SLERP

Removed region option since node must be from current region. Added option to specify nodeset cmiss_nodes|cmiss_data, so a node or data point can be used.

4. gfx list data|elements|faces|lines|nodes

Added conditional FIELD_NAME option. Use a group field as the conditional to list only objects in the group (previously used region path).

5. gfx list dgroup|egroup|ngroup

Removed. Use gfx list data|elements|faces|lines|nodes conditional GROUP_NAME instead. See point 4.

6. gfx list g_element

A new feature! Omit the region to list commands for all region renditions with new default recursive option. Default is now to list commands rather than the wordy description.

7. gfx list group

A new command: lists the groups in a region.

No default line graphics

Prior to this release, when you read in a model into cmgui, line graphics for all 1-D elements in any region under the root region were automatically added. However, lines were not added to regions created empty which included the root region itself. Particularly with the removal of 'group regions', there is no way that this behaviour can be sensibly maintained. The automatic lines feature was about as annoying to some users as it was beneficial to others.

Hence, we have removed this 'feature'. Now, in order to see any model you read in, you must open the scene editor and manually add any graphics. After you have created graphics you may need to click 'View All' in the graphics window to ensure they are in view.

Some users' command files will have been written assuming lines are automatically present; these now need to be modified to explicitly add lines, e.g. using the 'gfx modify g_element PATH_TO_REGION lines coordinate NAME ...'. You can do this interactively by adding the lines in the scene editor and using this command to list all commands to reproduce the graphics for all regions:

```
gfx list g_element
```

Another partial workaround is to add lines to the root region after your model has been read:

```
gfx modify g_element "/" lines coordinate coordinates;
```

(or whatever your coordinate field is called.)

Note: Alternative approaches to creating a default view of a model have been investigated but they are complex, hence they haven't been tried. However, we'd appreciate feedback and ideas from users in this area.

Simplified creation of nodal derivatives

The specification of nodal derivatives and versions with commands:

gfx modify nodes|data ...

have been changed to match the derivative names and limitations of the external API, since this code has been changed to test the external API internally. Users needing the removed functionality (different derivatives and versions per component) should talk to the Cmgui developers.

Getting started with CMGUI

This section will introduce you to the capabilities of CMGUI, as well as how to install and begin using the software.

Where to go for help

There is a large collection of examples which can be used to learn about the capabilites and operation of CMGUI.

Issues with the software can be reported and discussed on the tracker, which is found at https://tracker.physiomeproject.org.

What is CMGUI and what can it do?

CMGUI is an advanced 3D visualization open source software package with modelling capabilities. It originated as part of CMISS, a collection of software for “Continuum Mechanics, Image processing, Signal processing and System identification”. CMISS includes cm, a closed-source application for finite element method computation and related computation. Historically CMGUI has been mostly used in conjunction with cm. OpenCMISS-Iron (www.opencmiss.org) is an open source project, currently under development, aiming to replace the closed source CMISS-cm.

A gallery demonstrating some of the uses of CMGUI is available.

Some of the main areas of functionality of CMGUI are as follows:

- 3D visualization of finite element and boundary element meshes
- Mesh creation
- Mathematical field visualization and manipulation

CMGUI is mainly controlled from its built in “command line” interpreter. Usually script files are read or commands are typed in the CMGUI command window to read in your mesh, create a 3D visualisation and manipulate it. Many tasks can also be accomplished by using the mouse to select options from various menus and dialogs. CMGUI has a large amount of functionality but it can be difficult for a new user to figure out what command to use to accomplish exactly what they want to do. Fortunately there are a large number of examples demonstrating the use of various different commands.

CMGUI is also used as part of the Zinc extension to Mozilla Firefox. The Zinc extension allows users to view Zinc applications and visualizations in Firefox. A Zinc application uses CMGUI for visualization while providing a nice customized user interface for a specific task. For example Zinc applications have been written to do the following:

- Interactively explore Colon endoscopy
- Explore the Physiome eye model
- Create data points to give a digital description of a geometry based on medical images.

Note that Zinc is compiled as a separate application from CMGUI. You do NOT have to install CMGUI to use Zinc. For more information on Zinc see http://www.cmiss.org/CMGUI/zinc.

Installation

CMGUI is currently available for download for a wide variety of platforms. You can download the appropriate executable for your operating system from the release centre.

CMGUI is quite a large application so the executable has been archived (tarred) and compressed (zipped) to make it faster to download. Once you have downloaded the executable all you need to do is untar and unzip it and it is ready to run. On Linux you can untar and unzip the tar.gz file with the command:

```
tar -xf filename.tar.gz.
```

On Windows you will need 7Zip or something similar to unzip the file. You should now be able to run CMGUI by either clicking on the executable or by changing into the directory it is located in and then typing `./` followed by the name of the executable at the command prompt, eg. `./CMGUI-wx`. This will bring up the CMGUI command window interface.

For convenience it is a good idea to place the directory containing CMGUI into your PATH environmental variable. That way you will be able to run CMGUI without having to be in the directory that contains the executable. To do this on a Linux system you can specify your PATH variable in your .profile file. On Windows you can edit your PATH variable by right clicking on my computer and then selecting: properties, advanced, environment variables. Ask your local system administrator if you do not know how to edit the PATH variable for your operating system.

Introduction to CMGUI

CMGUI: The Windows

In order to become acquainted with the features of CMGUI, it is useful to run through some of the simple example files provided. CMGUI is a command line driven program, and all of its features are available via a command line interface - however, it also provides a more accessible GUI interface allowing many (but not all!) of these commands to be executed via a more familiar graphical interface. The interface is broken up into a number of windows that allow access to the different functions of the program. The first example will familiarise you with the most commonly used windows.

The CMGUI examples with thumbnails can be obtained from the CMISS website at http://cmiss.bioeng.auckland.ac.nz/development/examples/a/index_thumbs.html - These examples normally consist of a comfile and some other data files for creating a mesh.

Example a1 - Graphical Element Groups - Viewing a Cube

Load CMGUI to begin the first example. The first window that will present itself when loading CMGUI is the command window. This window has a selection of menus and three functional areas; a history area which shows commands that have been executed, a command line where commands may be entered directly, and an output area that shows the text output of any commands, including error and help messages.

In the file menu, select *Open* then *Com file*, then locate the example-a1.com file. This is a text file containing all the commands used in this example, as well as useful comments on what the various commands are doing. You will see the comfile window appear, displaying the file you have just loaded. All the lines that begin with a # character are explanatory comments, the rest are commands. The comfile window has three buttons at the bottom - the *All* button executes the entire comfile, while the *Selected* button will execute only selected portions. *Close* closes the comfile. Individual commands from the comfile can be executed by double clicking on them - notice that clicking to select a command in the comfile window makes that command appear in the command line panel in the command window, and clicking again then executes it. It is important to note that the order the commands

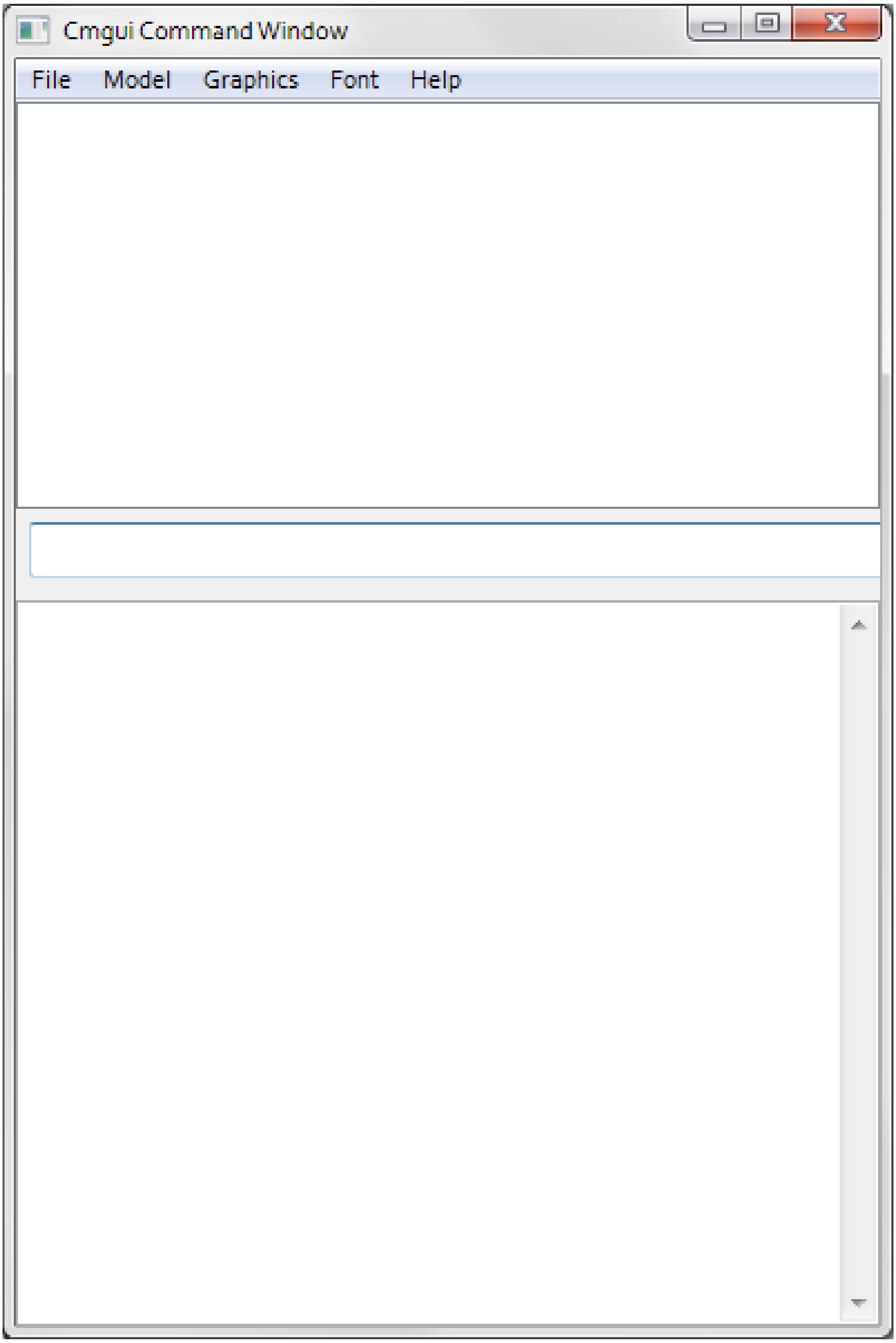
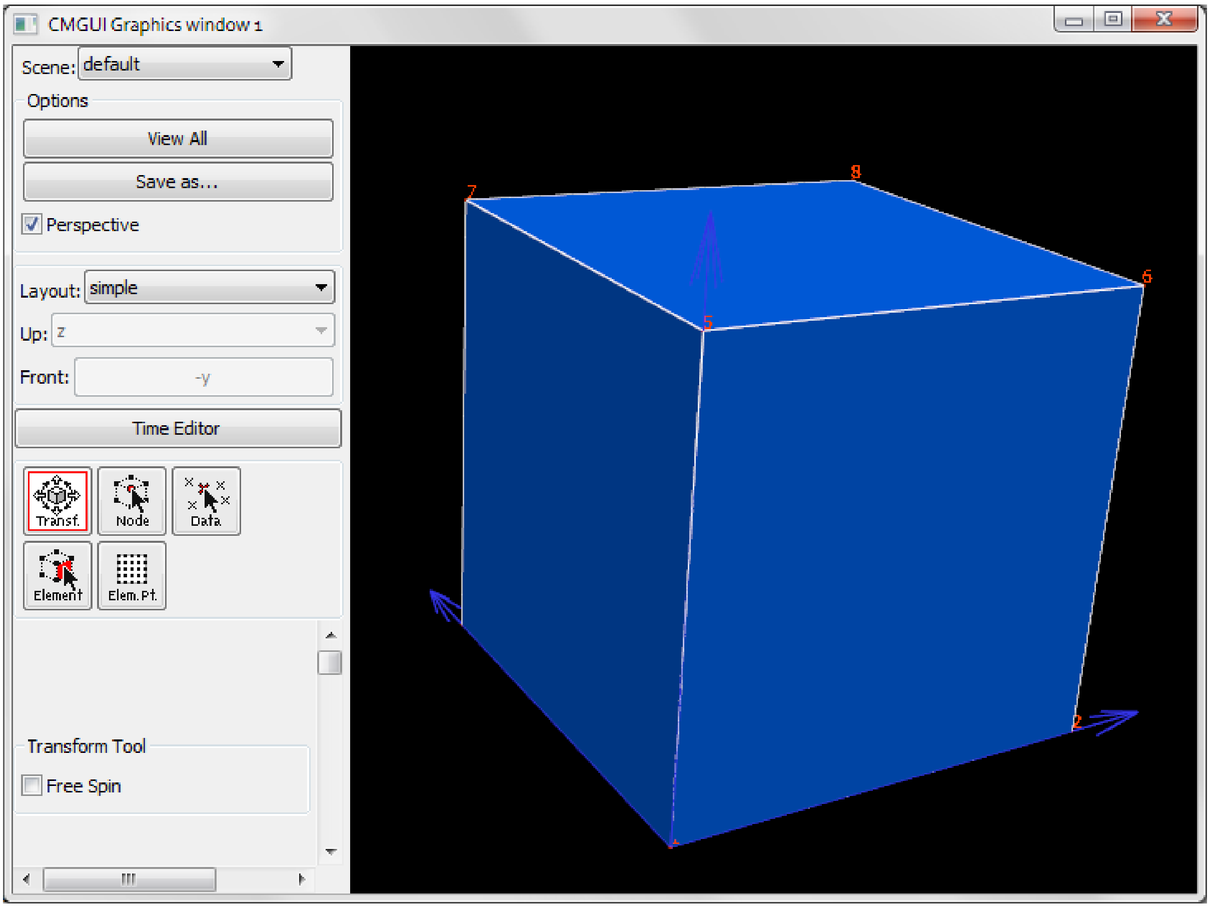


Fig. 4.1: **The CMGUI main window**

are executed in can be important, especially in later examples, so click on the commands in the order they appear in the example comfile.

Double-click the first two commands in the comfile - a new window will appear, displaying a wireframe representation of a cube. This is the graphics window. You can rotate, zoom and translate the view of the cube in this window by holding down the left, right and middle mouse buttons respectively while moving the mouse in the display area. Have a go at moving the cube around, and click the *perspective* button on and off to see the difference it makes to the display. To reset the view, click the *View All* button.

Once you have run through all of the commands in this example (either by double-clicking them or by using the *All* button in the comfile window) you should have a blue, somewhat shiny cube, a set of blue axes, and red numbers at each corner of the cube. These are the numbers of the nodes, the points that make up the 3D model. Another window will also have appeared, called the *Node Viewer*. This window allows you to manipulate the data that define the nodes - in this case the corners of the cube. In the cube example, each node has a number and coordinates. You can click on each of these buttons (at this stage a small GUI bug means you need to manually resize the window after clicking) to show editable boxes containing the actual data contained in the nodes. This window therefore allows direct access to the node data for editing. If you edit coordinate data and click the *Apply* button, you will see changes in the shape of the cube in the graphics window.



Scene: default

☑ Perspective

Layout: simple

Up: z

Front: -y

□ Free Spin

Fig. 4.2: **The CMGUI graphics window**

Other Interface Windows: Editors

That is the end of the example file commands, but there are other windows you can now use to manipulate the cube shown in the graphics window. Go into the Graphics menu in the Command Window, and select *Material editor*. This brings up a window that allows you to create and edit materials, which define the appearance of objects in the graphics window. You can manipulate the colour, specularity (shininess), emitted light (glow), and alpha (transparency) of a material using this window. Your edited material is used to create the sphere in the preview panel at the bottom of the window - you may need to resize the window in order to get a good sized preview panel. Click on some of the already defined materials in the list at the top to see how they affect the preview sphere, then play around with the sliders to alter the material.

Click on the material *bluey* - this is the material used to show the faces of the cube in this first example, as defined in the comfile. Use the colour and property sliders to make it quite different; perhaps red and highly transparent, then click *Apply* to change the cube in the graphics window to your new material.

Now close the material editor and go to the Graphics menu, selecting *Scene editor*. The scene editor allows you to manipulate which objects appear in the graphics window, and how they are rendered. It also allows you to create new objects, as well as change their order in the scene, which is particularly important when rendering transparent objects. As a quick example of what can be done with this editor, select the *lines* setting in the settings list (shown selected below), and un-tick the check box. The white lines along the edges of the cube will disappear in the graphics window. Similarly, you can choose whether to show the cubes surfaces or the node numbers using the appropriate check boxes. You will notice that as you select settings (such as surfaces) in the settings list, their properties will appear in the settings editor below. Using this editor you can change a large number of the display properties of objects in the scene. There are many more important functions to this editor window which will be explained in more depth later.

CMGUI Architecture

Regions and Fields

The basic structural unit in CMGUI is called a *region* - region is a container for the objects representing a model. Regions own a set of fields and may contain child regions, thereby allowing a hierarchical model structure. For example, you may have a "body" region that contains "heart", "lung", and "stomach" regions.

When you start CMGUI, a *root region* exists; this is an empty space into which other regions are loaded. When you load in exnode and exelem files, a new region will be created in the *root region* named for the group declared in the files. For example, if you were to load in heart.exnode and heart.exelem files which declare a group "heart" (via the line `Group name:  heart`), a new region called "heart" would be created containing all the nodes, elements and fields defined in these files.

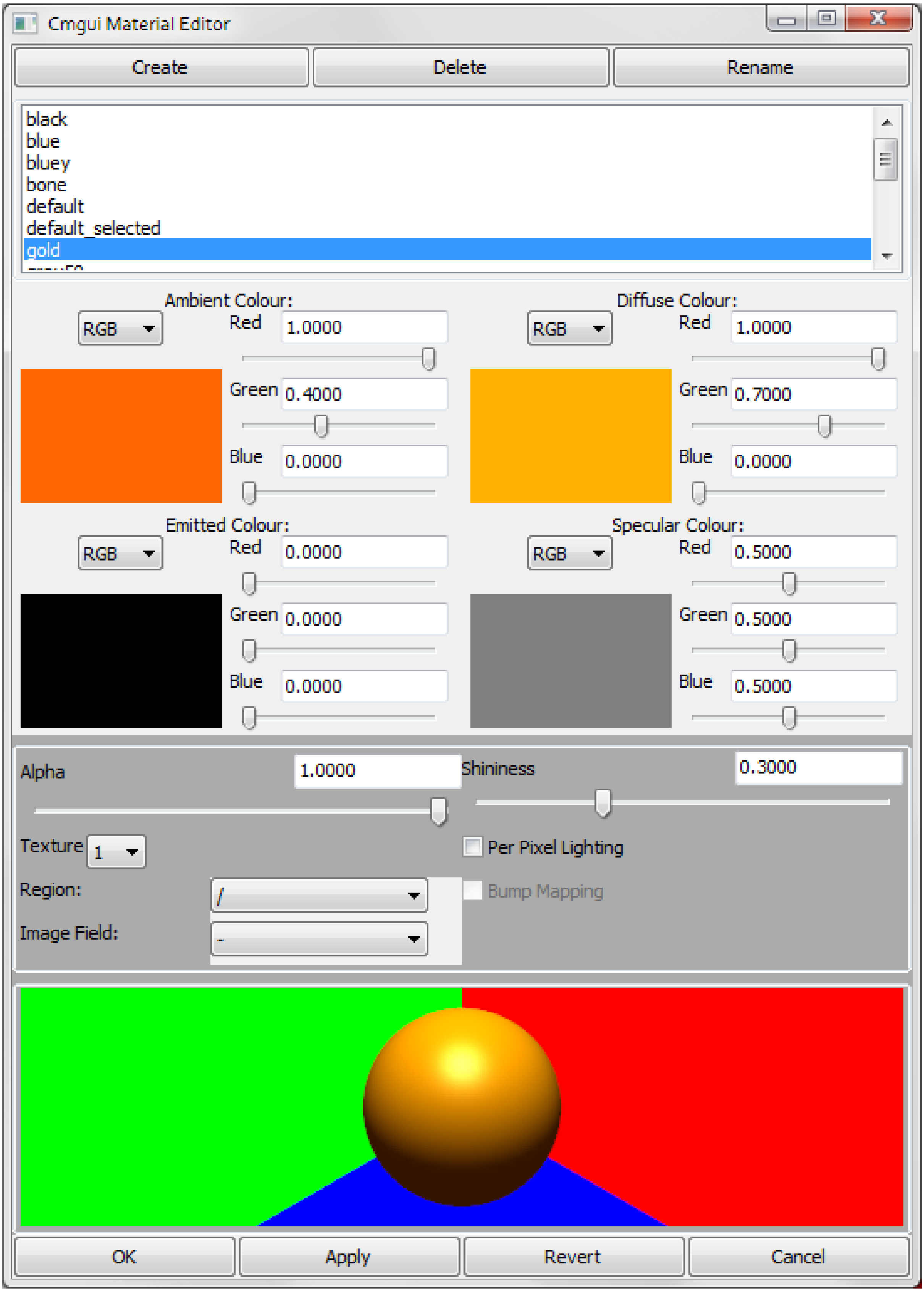
Each region contains a set of *fields* from which a model is built up. A field is a function that returns values over a domain; for example, the "potential" field could provide a value for the electrical potential of each point in the heart domain, which could be a finite element mesh. The geometry of the heart is itself defined by the coordinate field (this field is often called "coordinates" in our models). Finite element fields in CMGUI are generally defined by storing discrete parameters at nodes (eg the coordinates of the node, the electrical potential at each node) and interpolated across the ξ *space* spanned by elements to give the continuous field.

Other examples of fields include fibre orientation in the heart, temperature, pressure, material properties, strain and so on. In CMGUI you can also define fields that are computed from other fields. For example, you might define a field called "scaled_coordinates" by scaling coordinates x, y, and z to x, y, and 10z, perhaps to exaggerate the height of topological features. You might also compute strain from deformed and undeformed coordinates, fibre axis from fibre Euler angles, and so on.

Other Data in CMGUI

The structure of the CMGUI *command data* (that is, the complete set of data used by CMGUI) is divided into a number of *containers* and categories. Besides the regions which contain fields, there are also the following lists and managers, which contain other data used to create visualizations:

- Scene manager - scenes contain definitions of how to display visualizations. The region created when you load exnode and exelem data (such as the heart region described above) is used to automatically create a



black
blue
bluey
bone
default
default_selected
gold

Ambient Colour: RGB
Red 1.0000
Green 0.4000
Blue 0.0000

Diffuse Colour: RGB
Red 1.0000
Green 0.7000
Blue 0.0000

Emitted Colour: RGB
Red 0.0000
Green 0.0000
Blue 0.0000

Specular Colour: RGB
Red 0.5000
Green 0.5000
Blue 0.5000

Alpha 1.0000

Shininess 0.3000

Texture 1

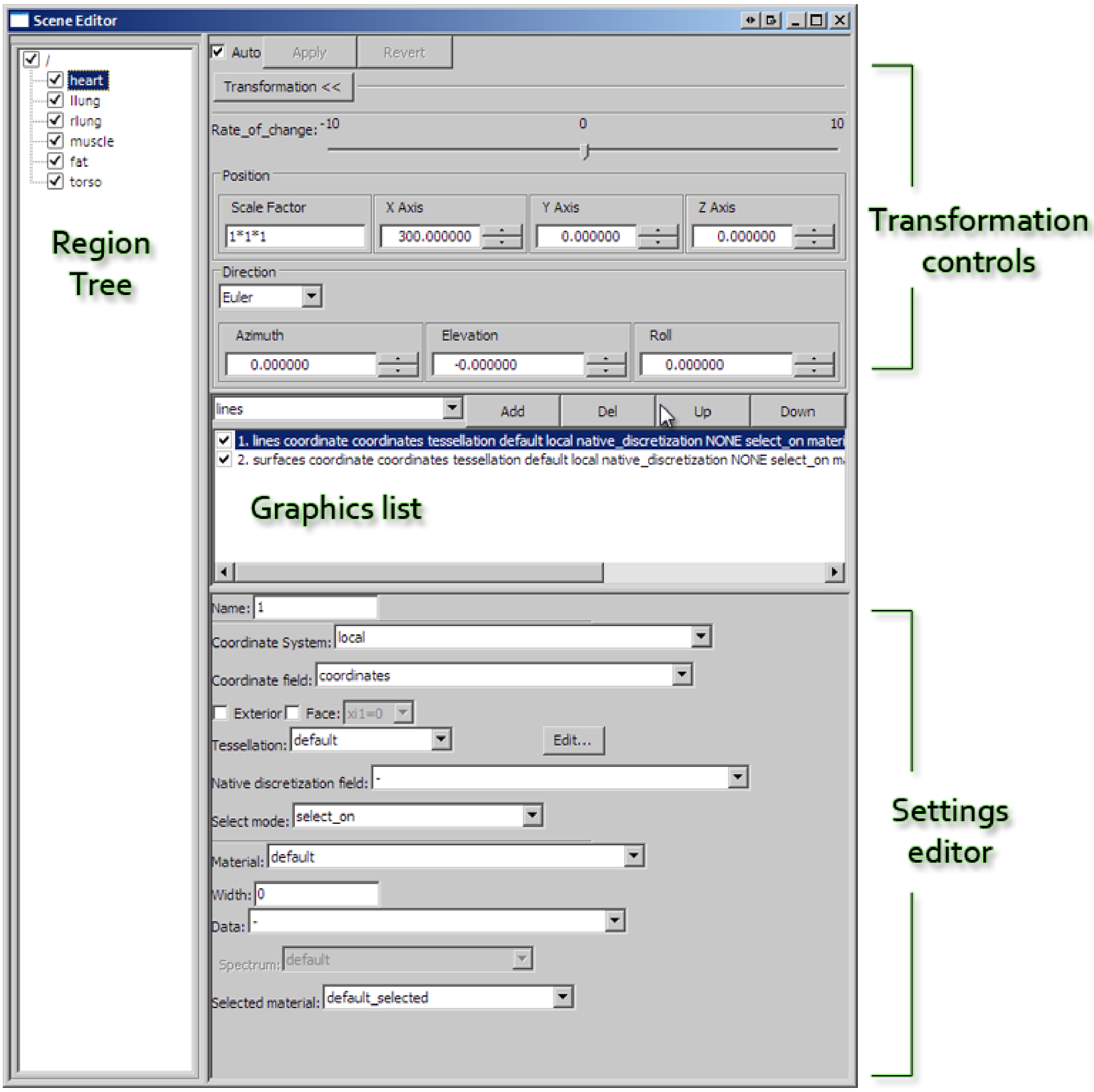
☐ Per Pixel Lighting

Region: /

☐ Bump Mapping

Image Field: -

Fig. 4.3: **The CMGUI graphical material editor window**



☑ /
☑ heart
☑ llung
☑ rlung
☑ muscle
☑ fat
☑ torso

☑ Auto

Rate_of_change: -10 0 10

Position

Scale Factor	X Axis	Y Axis	Z Axis
1*1*1	300.000000	0.000000	0.000000

Direction: Euler

Azimuth	Elevation	Roll
0.000000	-0.000000	0.000000

lines

☑ 1. lines coordinate coordinates tessellation default local native_discretization NONE select_on materi

☑ 2. surfaces coordinate coordinates tessellation default local native_discretization NONE select_on m

Name: 1

Coordinate System: local

Coordinate field: coordinates

□ Exterior □ Face: xi1=0

Tessellation: default

Native discretization field: -

Select mode: select_on

Material: default

Width: 0

Data: -

Spectrum: default

Selected material: default_selected

Fig. 4.4: **The CMGUI scene editor window**

graphical representation in scene "default" when loaded. Scenes contains either (1) graphical renditions of regions (2) graphics objects (3) child scenes.

- Graphical material manager - contains a list of graphical materials that can be applied to graphical representations or objects.
- Texture manager - contains a list of textures that can be used in material definitions or volume renderings, etc. Textures are 2D or 3D images.
- Spectrum manager - contains a list of spectra that can be used in material definitions. These control how the graphical material is changed when graphics are coloured by a data field.
- Graphics object list - contains simple graphics objects which are not tied to a model representation, but can be drawn in a scene. These can animate with time.
- Glyph list - contains glyphs that can be used to represent data points in the graphics window - such things as spheres, cubes, arrows, axes, points or lines. Objects from the graphics objects list may be added to the glyph list, in order to create customized glyphs.

CMGUI is not like most common software packages that can save a single file containing all of your work. In CMGUI, the data being worked on is often loaded in as a number of separate files, and the editing of the visual representation of this data often does not change it; it only alters how it is represented in the 3D window. Currently, it is not possible to save your work in a single file that can be loaded in order to recreate all your work on the representation. For example; if you load in a model, change the viewpoint, alter the materials used to render it, and add glyphs to important data points, there is no way to simply "save" all of these changes in one file.

It is possible to save most of your work in CMGUI, using the menu item *write all files*. Go to the *file* menu, select *write* and then *all files*. You will be prompted to give names for a com file, exdata file, exnode file, and an exelem file. CMGUI will then prompt you for the name of a zip file to save all these separate files in. In this way it is possible to take a fairly complete "snapshot" of your work in CMGUI. This feature is undergoing improvement to make it more comprehensive.

CMGUI Command Window

This is the first window that appears when CMGUI is loaded. It consists of standard menus at the top, and three panels below. From top to bottom these are the history, command line, and output panels. The history panel is where all commands that have been executed are displayed. The second, single-line panel is the command line where commands may be entered directly. The lowermost panel is the output panel; this displays the text output of commands, error messages, as well as help information.

The history and output areas have scrollbars that allow you to view all of the contents of these panels. In the case of the history panel, this enables you to scroll back and click on commands - clicking on a command in the history panel will enter that command into the command line. This command can then be edited and executed. In the case of lengthy commands, this can save a lot of typing. Double-clicking on a command in the history window will execute it immediately.

The output panel is where any text output from commands or error messages appears. If something is not working in CMGUI, this panel is the first place to look for a reason. This panel also displays the help information when you execute a command with the ? or ?? argument.

Useful commands

To "save" a set of material settings:

```
gfx list material commands
```

Copy the resulting text and save it as a text file. You can use this to save any parts of the visualisation you have created - simply type:

```
gfx list ?
```

To get a list of the objects that can be listed with this command.

CMGUI Fields

Fields in CMGUI: finite element models

CMGUI models are organized into regions, which may contain child regions, so that models can be organized into hierarchies. The representation of the model within each region is given by *fields*. Much of the power of CMGUI is in its generalized representation and manipulation of fields. Mathematically, a field is a function returning values at locations within its domain. Most commonly the values are real numbers, though they may be scalars or multi-component (vectors, tensors etc.).

Consider an object such as a solid cube over which a value like the temperature varies. We would represent this in CMGUI as a field "temperature" which is a function of location in the cube. A cube is a very simple geometry and it is easy to come up with a simple method to specify any location within it; for example the global x, y, z. But real geometries are seldom simple so we follow the finite element method and divide them up into simple shapes such as cubes, wedges, tetrahedra, etc. for 3D geometries; squares, triangles etc. for 2D geometries; lines for 1D geometries, and other shapes in other dimensions.

These finite elements have a local coordinate system of limited span, which is called an element chart. In cmiss we commonly use the greek character xi (ξ) with subscript to denote each independent coordinate within an element, hence we will commonly refer to the chart of a given element as its ξ *space*. A 3-D cube element in CMISS/CMGUI has a range of 0 to 1 in each ξ direction, as shown in figure 1A.

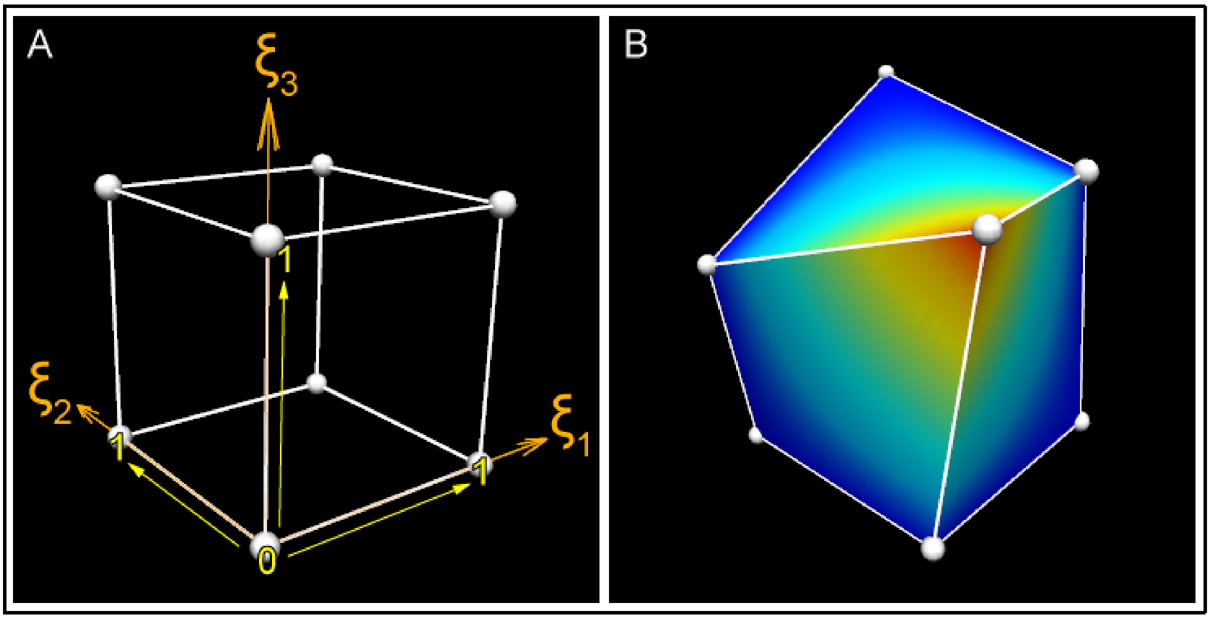


Fig. 4.5: **Figure 1: Coordinates and ξ space of a 3D element.** A) This shows the unit cube ξ space, where each dimension of the element ranges from 0-1. It is easy to imagine that the coordinates of this cube could also be 0-1 in the x, y, and z axes. B) The cube element in this picture has been distorted, such that its coordinates are no longer 0-1 in the x, y, and z axes. Despite this, the element's ξ values are still 0-1 in each ξ direction. This cube has a "temperature" field that is rendered as a rainbow spectrum.

We must define a coordinate field over the domain in order to give it its true position in 3-dimensional space. This applies even to the simple cube model: it can be treated as a unit cube in ξ space, but the coordinate field allows its real position to be transformed such that it is not aligned with the global coordinate system, and in fact can be generally distorted as shown in the figure 1B, above. About the only thing special about a coordinate field is that - provided it is of the same dimension as the element ξ space - it is usually bijective with it - this means if you have one you can find the other, eventually. (I say usually because the relation cannot generally be enforced:

it is possible for coordinates of parts of the mesh to penetrate other parts of the mesh, including within a single element where the jacobian is zero or negative.)

A set of finite elements which make up a domain is called a mesh.

In CMGUI data structures we usually also define the faces of elements as separate elements of lower dimension, and share faces between adjacent elements. The faces of 3D "top-level" elements are called face elements, the faces of 2D elements are called line elements. The top-level elements, whether 3D or 2D (or nD) have unique integer element identifiers in their region. Face and line elements also have unique identifiers within their own type; in other words you can have element 1, face 1 and line 1 and they are not the same element.

The zero dimensional counterpart to elements and faces are called nodes. There is one set of nodes per region, again with their own unique integer identifiers. Nodes can be considered as a set of points at which fields are defined. When we define field functions over elements (finite element fields) we most commonly store explicit values and derivatives of the field at nodes and interpolate them across the element. To define such fields each element maintains a list of the nodes which contribute to fields in that element, called the local node list. The number and arrangement of nodes depends very much on the basis function used to compute the field value. Linear Lagrange and simplex basis functions are a simple, common case where nodes contain values of the field at the corners of elements, and these are linearly interpolated in the ξ space between. Figure 2 shows the arrangements of faces, lines and nodes (for linear basis) for 3D and 2D elements.

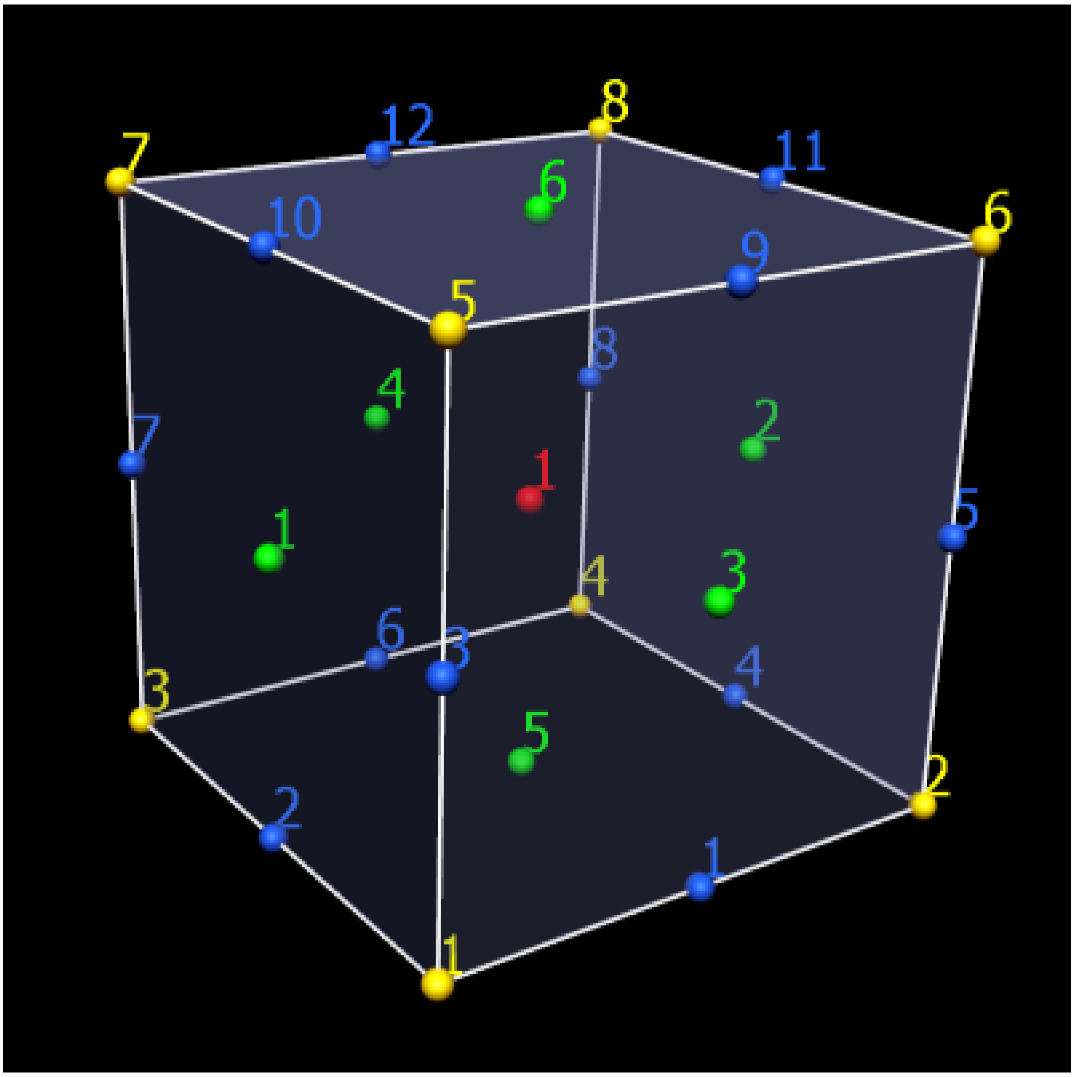


Fig. 4.6: **Figure 2: How nodes, lines and faces make up a mesh** The simple cube mesh again; nodes (yellow), elements (red), faces (green), and lines (blue) are numbered. A single cube element requires 8 nodes, 12 lines, and 6 faces.

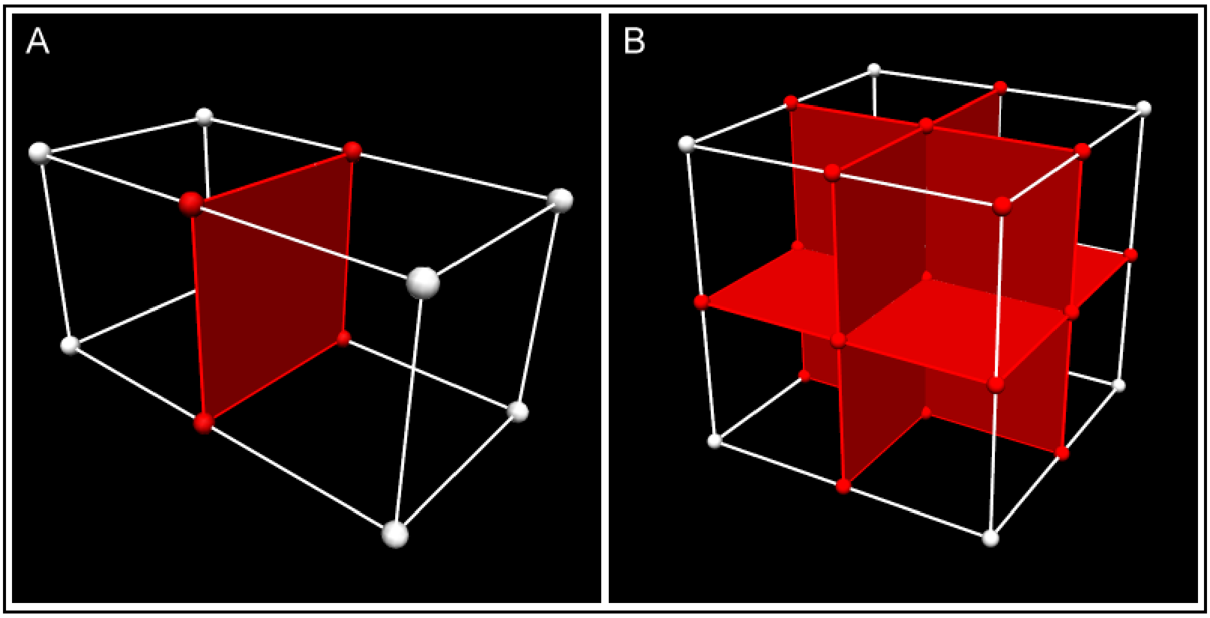


Fig. 4.7: **Figure 3: Node, face and line sharing between connected elements** In more complex meshes, connected elements share nodes, lines and faces (shared nodes lines and faces are shown in red). In panel A, this two-cube mesh has only 12 nodes; 4 of the nodes are shared by both elements. In panel B, an eight-cube mesh is made up of only 27 nodes - many of the nodes are shared by more than one of the elements. The central node in this example is shared by all 8 elements. Field values are continuous across these shared parts.

Getting back to the original statement of what a field is, we can generally state that a domain is a set of 0..N-dimensional manifolds, i.e. there can be any number of manifolds of any dimension. In CMGUI finite element meshes, nodes supply the point manifolds and elements supply all the higher dimensional manifolds. There is no requirement for the domain to be connected. CMGUI fields are not limited to returning real numbers; there are fields that can return integers, strings, element-ξ locations and other values.

Computed fields

We have now introduced the main building blocks of finite element fields which are just one type of field representation in CMGUI. In CMGUI we offer a large number of other field types which do not work directly with finite element meshes, but may be based on finite element fields. Most of these are mathematical operators which act on one or more source fields to produce a new field result.

A simple example is the *add* field which adds two other fields together to return a new field. The add field has two source fields as arguments. If you made field C which added finite element field A to finite element field B, the resulting field is defined over the intersection of the domains of field A and field B.

Other types of field

- **Arithmetic and transcendental functions:** Add, subtract, multiply, divide, sum_components, log, power, square root, exp.
- **Trigonometric functions:** sin, cos, tan, asin, acos, atan, atan2.
- **Derivative functions:** Derivative, divergence, gradient, curl.
- **Vector functions:** Dot_product.
- **Matrix functions:** Matrix_multiply, matrix_invert, transpose, projection, eigenvalues, eigenvectors, quaternion_to_matrix, matrix_to_quaternion.
- **Logical functions:** Less_than, greater_than.

- **Conditional functions:** If.
- **Constant fields:** Constant fields have a value which is independent of location within the domain. You may also have a constant field that is constant (non-varying) across chosen dimension/s but varies across other dimension/s.
- **Composite field:** Makes a new field built from other fields and field components in any order.
- **Image-based fields:** These can be used for texture-mapping and image processing. Image processing fields using ITK filters.
- **Many more.**

These types of fields can be created via `gfx define field` commands or through the API. Fields are a modular part of the CMGUI application. If a new function is required, it can be added as a field. To get a list of the computed fields available in CMGUI, enter `gfx define field ??` in the command line.

Fields and visualization

When creating visualizations, you need to choose which field controls which part of a graphics object. Coordinates in one, two or three dimensions can be used to create spatial representations. Texture coordinate fields can be used to position textures. Orientation or data fields can be used to position glyphs or colour objects such as surfaces. CMGUI allows an enormous amount of flexibility in how fields can be visualized. Further information on visualizations is available in other documents such as those detailing *graphics*, *glyphs*, or *surfaces*.

Visualizing fields at points using glyphs

Glyphs are graphical objects that are used to represent information at points within a model. These glyphs can be coloured, scaled, and oriented according to the values of chosen fields. Glyphs might be used for something as simple as showing node locations (Figure 1), or something more complex such as showing the strain at points within a deformed mesh. cmgui has a range of glyphs available, for representing different types of point data. This document will explain how to position, scale and orient glyphs in a variety of ways.

Adding glyphs to a mesh

Glyphs are added and edited from within the scene editor. When you select a region in the scene editor, you can create a new graphic for placing glyphs at your points of interest. Glyphs can be added at point, node points, data points, or element points. Element points are points through the interior of an element at a controllable layout and density.

Unscaled glyphs

The simplest use of a glyph is to visually mark a point, with no direction or scale information. Unscaled glyphs are simply glyphs for which you have not selected an *orientation/scale* field. This means that they simply appear at the points you specify, at the default orientation, and with the specified the base size. The default base size is 1*1*1. If you want to run through this example, load up example a1 in cmgui and run the entire com file.

Open the *scene editor* and make sure cube is selected in the *scene object list*. You will notice that there are three items in the *graphics list*: *lines*, *node_points*, and *surfaces*. Select the node_points graphics item. You will see in the settings below that these node_points already have glyphs associated with them - in the *Glyph* drop-down menu, the glyphs currently used to display the nodes are *point* glyphs. These are the default glyph, and are by default a single white pixel. Figure 2 shows the scene editor window, with the three areas labelled.

Use the drop-down menu to select another type of glyph for the node points: select *sphere*. In the graphics window, large orange spheres will appear at the corners (node points) of the cube. As the cube in this example is 1*1*1 and the default size of the glyphs is also 1*1*1, the glyphs will be far too large to be useful. For unscaled glyphs, you control the size by entering your desired size in the *Base glyph size* text-box, which will currently contain 1*1*1

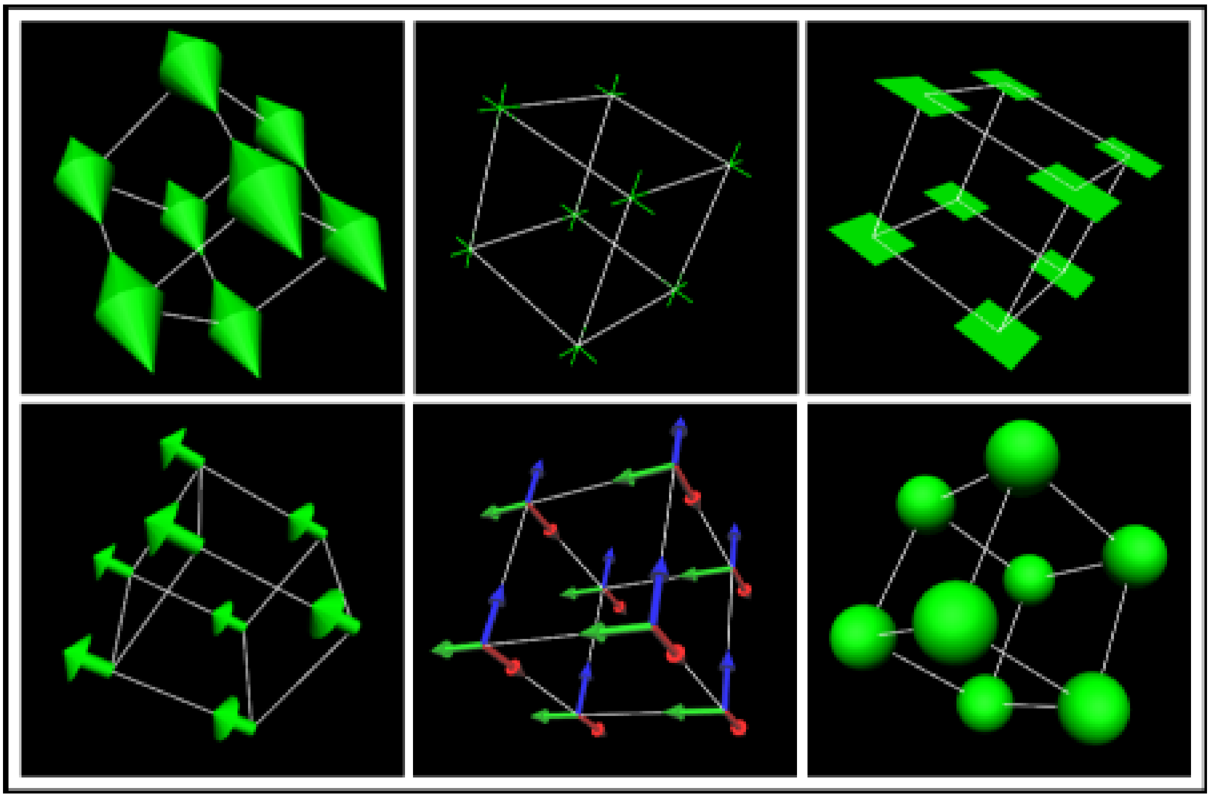


Fig. 4.8: **Figure 1: Different glyphs placed at the nodes of a cube mesh.**

(x*y*z dimensions). Change this to 0.1*0.1*0.1. If you enter a single value in the base glyph size box, it will use this value for all three dimensions. After changing the glyphs to these dimensions, you will notice that the spheres have shrunk to a more practical size (Figure 3). For unscaled glyphs, the final size is determined entirely by the *Base glyph size*.

Final glyph size = b1*b2*b3

This is the most basic use of glyphs - to indicate points such as nodes. Other uses of glyphs allow you to represent data in addition to simple location by using the size, shape, and colour of glyphs.

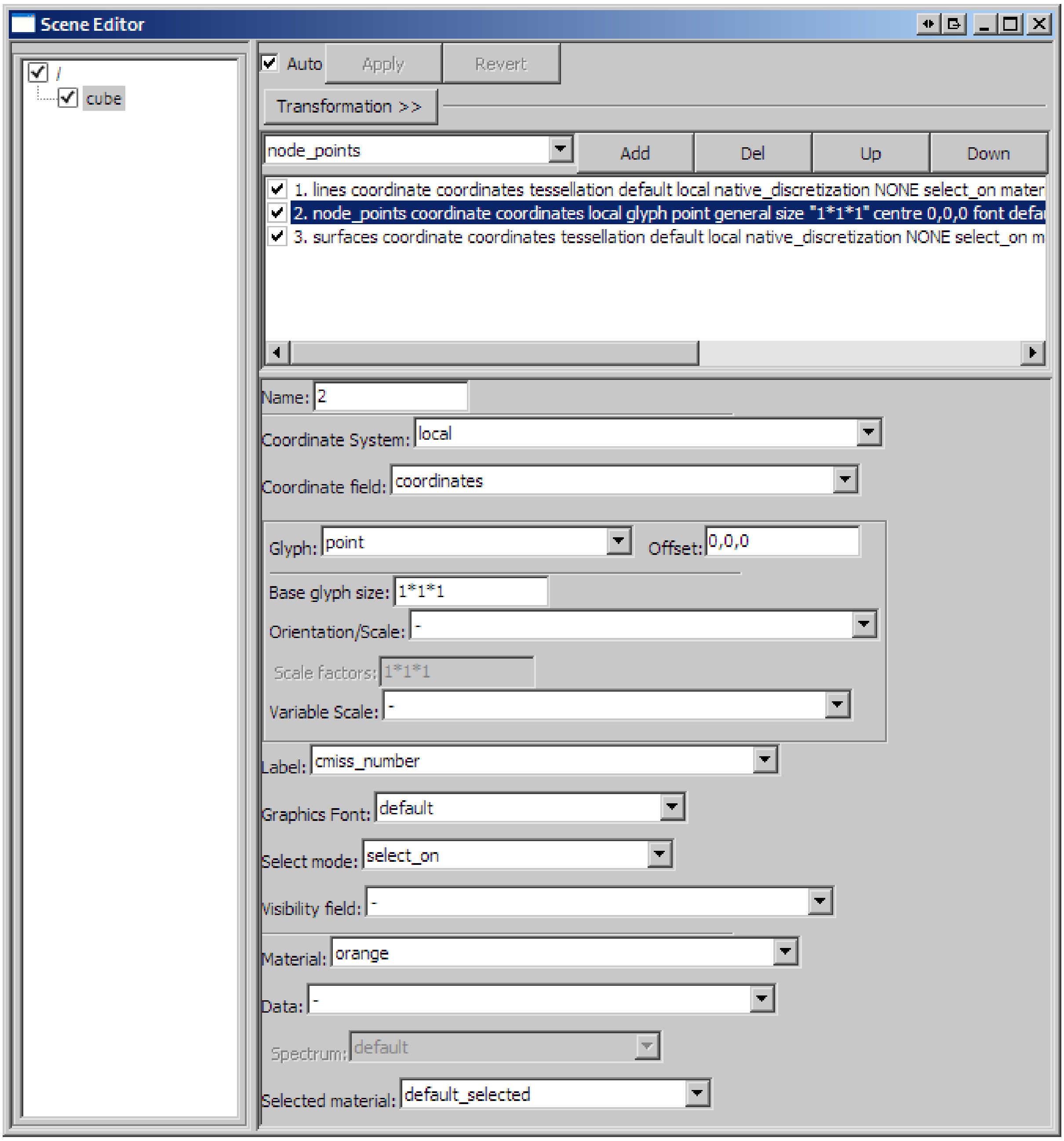
Scalar glyphs

Scalar glyphs can be used to represent the value of a scalar field at selected points. They have a single component *orientation/scale* field, which we will call *S*. For this case the value of S is used for all three directions of glyph scaling. The size of each glyph dimension (x, y, and z) is determined by the *base size* (which we will call b) plus the *scale factor* (which we will call c) multiplied by the *orientation/scale* field.

Any scale factors that are given a zero value do not add anything to the base size - in this way it is possible to scale glyphs in one, two, or three dimensions simply by using a zero value for the dimensions along which you do not wish them to scale. For example, for glyphs that only scale in the z direction, c1, c2, and c3 could be 0, 0, and 0.1 respectively. It can also be useful to set the *base size* to zero in order that the final glyph size is exactly proportional to the *orientation/scale* field.

Vector glyphs

Glyphs used to represent a single vector most often use a three component *orientation/scale* field, but it is possible to use a two component field for two-dimensional vectors. The alignment of the glyph is determined by the alignment of the vector defined by the field; the glyph's first direction aligns to the vector, and its second and third directions are arbitrarily aligned orthogonal to this.



node_points

☑ 1. lines coordinate coordinates tessellation default local native_discretization NONE select_on mater
☑ 2. node_points coordinate coordinates local glyph point general size "1*1*1" centre 0,0,0 font defa
☑ 3. surfaces coordinate coordinates tessellation default local native_discretization NONE select_on m

Name: 2

Coordinate System: local

Coordinate field: coordinates

Glyph: point

Offset: 0,0,0

Base glyph size: 1*1*1

Orientation/Scale: -

Scale factors: 1*1*1

Variable Scale: -

Label: cmiss_number

Graphics Font: default

Select mode: select_on

Visibility field: -

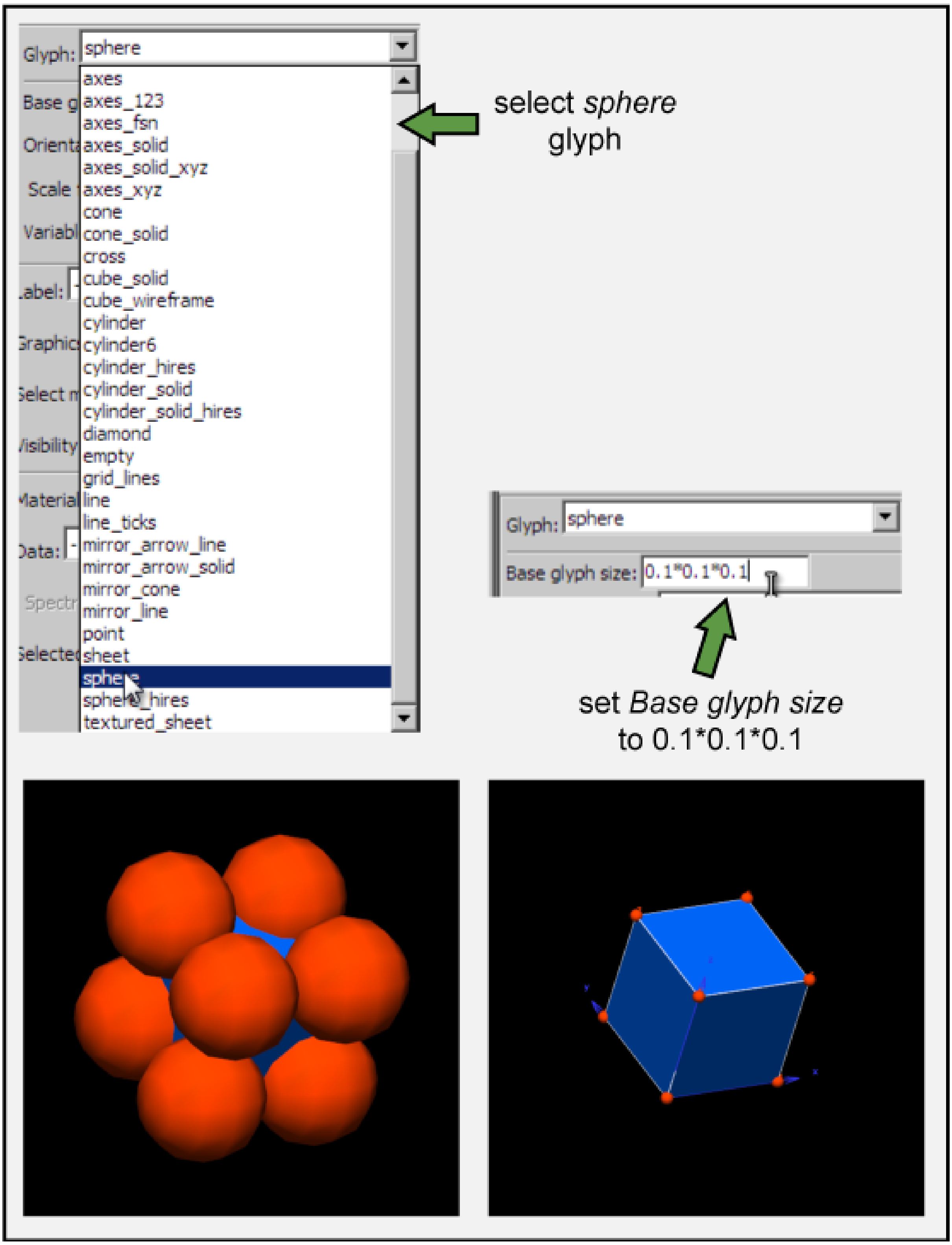
Material: orange

Data: -

Spectrum: default

Selected material: default_selected

Fig. 4.9: **Figure 2: The scene editor.**



select *sphere* glyph

Glyph: sphere

Base glyph size: 0.1*0.1*0.1

set *Base glyph size* to 0.1*0.1*0.1

Fig. 4.10: **Figure 3: Changing the glyph and its base size.**

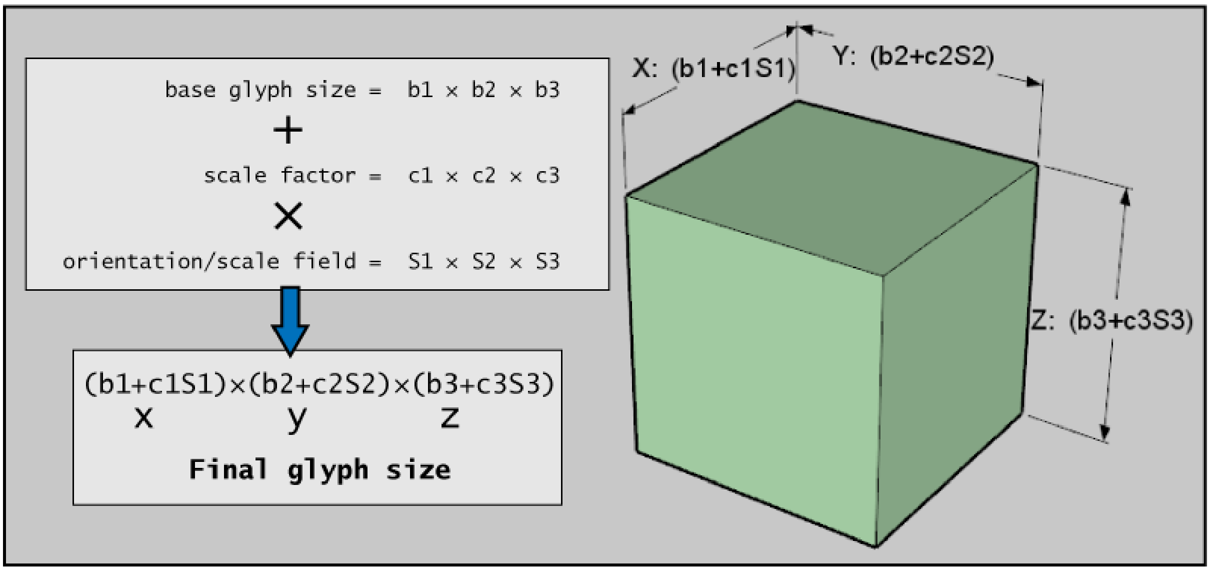

base glyph size = b1 × b2 × b3
+
scale factor = c1 × c2 × c3
×
orientation/scale field = S1 × S2 × S3
(b1+c1S1)×(b2+c2S2)×(b3+c3S3)
x
y
z
Final glyph size
X: (b1+c1S1)
Y: (b2+c2S2)
Z: (b3+c3S3)

Fig. 4.11: **Figure 4: How glyphs are scaled by a scalar orientation/scale field.**

The glyph scaling is proportional to the magnitude of the vector, which we will call M. All three directions of the glyph are scaled by this value.

Final glyph size = (b1+c1M)*(b2+c2M)*(b3+c3M)

In most cases, b1 (base size one) is set to zero so that the final length of the glyphs is directly proportional to M; that is, their size in direction one is entirely determined by the scaling factor c1 multiplied by the magnitude M. Likewise, c2 and c3 can be set to zero so that the width and height of the glyphs are constant; in this case b2 and b3 would be set to the desired constant sizes in these directions.

Two vector glyphs

These are rarely used. They have either four (2D vectors) or six (3D vectors) component *orientation/scale* fields. Vector 1 is defined by the first 2 or 3 components, and vector 2 by the second 2 or 3 components. The glyphs will orient their first direction along vector 1, and their second direction along vector 2. The glyph's direction 3 direction 3 is equal to the cross product of vectors 1 and 2.

The glyph scaling is proportional to the magnitude of vector 1 (M1) in direction 1, the magnitude of vector 2 (M2) in direction 2, and the cross product of vectors 1 and 2 in direction 3. For two 2D vectors, cmgui assumes a z value of 0 in order to obtain the cross product.

M1 = magnitude of vector 1 M2 = magnitude of vector 2 M3 = magnitude of cross product of vectors 1 and 2

Final glyph size = (b1+c1M1)*(b2+c2M2)*(b3+c3M3)

Three vector glyphs

Three vector glyphs use a nine component *orientation/scale* field: vector 1 is defined by components 1,2,3, vector 2 by components 4,5,6, and vector 3 by components 7,8,9. The glyph is oriented in directions 1, 2, and 3 by the directions of vectors 1, 2, and 3 respectively. The scaling along the three directions is determined by the magnitude of the three vectors.

Final glyph size = (b1+c1M1)*(b2+c2M2)*(b3+c3M3)

Using the fibre field

A special case of three vector glyphs is when you choose a *fibre field* for the *scale/orientation* field. This option automatically creates a three vector "fibre axes" field from it together with the coordinate field used by that graphic.

This is equivalent to defining a field using the command `gfx define field NAME fibre_axes`

Variable scale glyphs

Variable scale glyphs use an extra "variable scale" field to give a signed magnitude; this not only multiplies the magnitude of the orientation_scale field (so it is doubly-scaled) but its magnitude provides its "sense". A good example of this would be extension (positive) versus compression (negative) for strain. Negative values of the variable scale field reverse glyphs about their origin along their orientation. There are several special "mirror" glyphs designed specifically for this purpose.

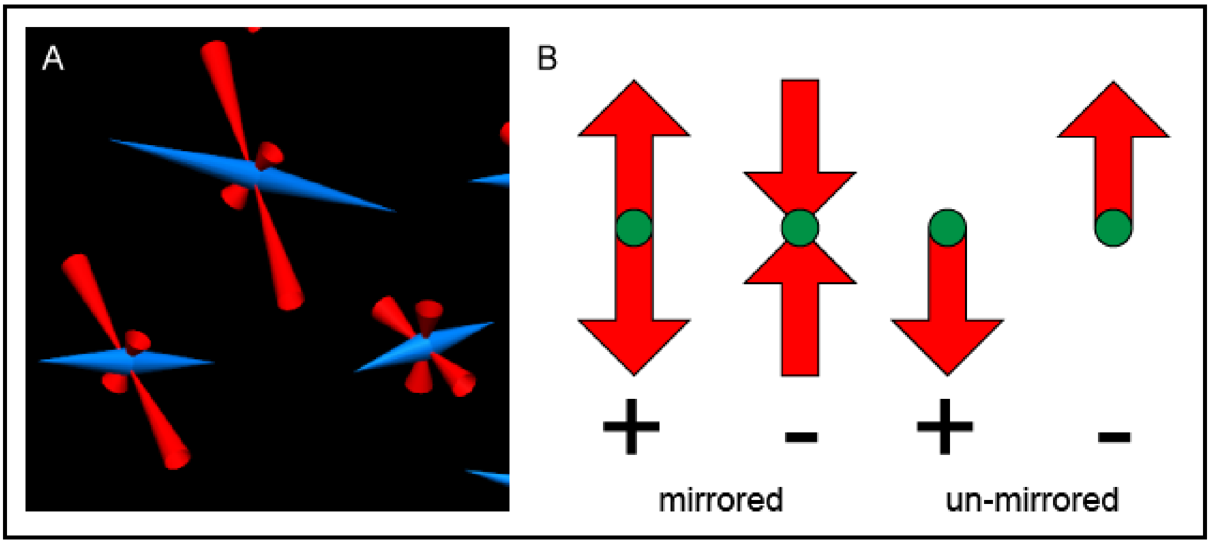


Fig. 4.12: **Figure 5: Mirror glyphs and glyph reversal using the variable scale field.** A) Mirror-cone glyphs being used in the large strain example, with the magnitude and sign of strain indicated by the length and direction of the glyphs respectively. B) How glyphs are represented with differently signed variable scale fields. Un-mirrored glyphs are not as useful for representing this information.

Variable scale glyphs need both:

- direction: *orientation_scale* field
- magnitude: *variable_scale* field

The variable scale field is an extra scaling factor in addition to the magnitude of the vector. For the final glyph size equation I will call the variable scale field "lambda" - this is because the variable scale field is often the eigenvalue of an eignevector calculated from deformations. Run through the large_strain example (*a/large_strain*) to see this in action.

Final glyph size = (b1+c1M*lambda1)*(b2+c2M*lambda2)*(b3+c3M*lambda3)

It is most common to use a variable scale field with single vector glyphs, such as in the large strain example.

Additional tricks with glyphs

Adjusting the glyph offset

All glyphs have a default origin; this is the point which is positioned at the chosen point within the graphical representation. This can be edited by entering values in the *offset* value box in the settings editor. This appears next to the glyph drop-down menu.

By default, glyphs have a 0,0,0 coordinate point (origin) that is logically positioned according to the purpose of the glyph. For directional glyphs, the "long axis" is always the x axis. Spheres, cubes and cylinders have their origin positioned in the spatial centre of a bounding unit cube. Directional glyphs such as arrows have their origin at the base of the arrow, and axis glyphs have their origin at the intersection of the axes.

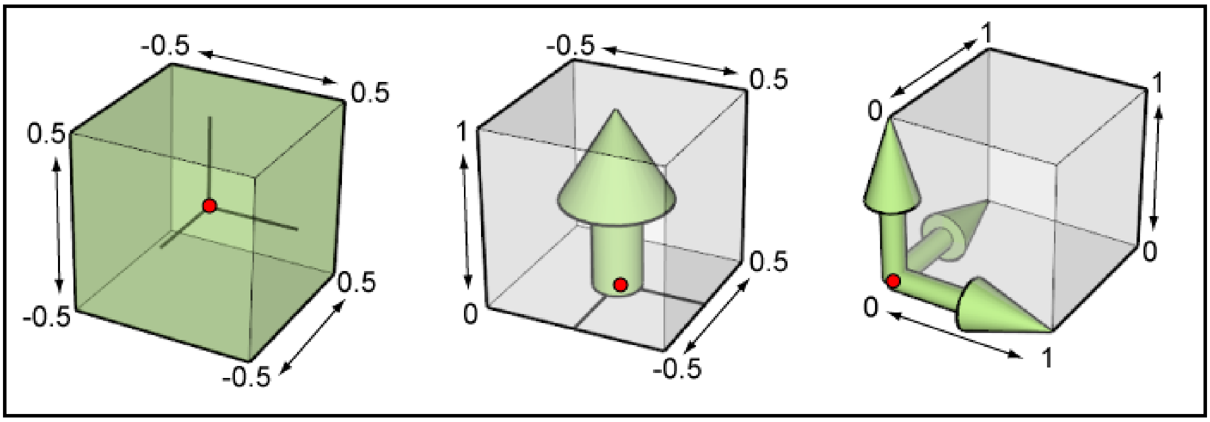


Fig. 4.13: **Figure 6: Origins of various glyph types within their bounding cubes.** Origin of each glyph family is indicated by a red dot.

Using the *offset* value box, you can adjust the position of your selected glyph.

Using custom glyphs

It is possible in cmgui to create your own glyphs from obj model files. An example of this in action is the biplane example, where a model of a biplane is used to create a custom glyph.

Graphics in CMGUI

General description of graphics

graphics are the building blocks used to create any visualization displayed in the CMGUI *graphics window*. They are created, edited, re-ordered and deleted from within the scene editor, or via the command line. Most graphics have the following settings in common:

- Name: This allows you to set the name of the graphic.
- Coordinate System: This setting can be used to render the graphic according to a range of different coordinate systems. This is useful for creating “static” overlays of data on visualizations.
- Coordinate field: This setting has a drop-down menu showing a list of fields that can be selected as the coordinate field for the selected graphic.
- Tessellation: Tessellation settings are used to set the level of detail of an object. This replaces the discretization settings from old versions of cmgui.

Note: The only graphics which do not have a tessellation setting are *node points*, *data points*, and *point*.

- Select mode: This drop-down menu allows you to select different selection behaviours for the graphic:
  - select_on - The default setting; graphics are able to be selected and selected items are highlighted by rendering in the *default_selected* material.
  - no_select - No selection or highlighting of the graphic.
  - draw_selected - only selected items are drawn.
  - draw_unselected - only unselected items are drawn.

- Material: This drop-down menu allows you to select which material should be used to render the graphic. Materials are defined and edited in the *material editor window*.
- Data: This setting has a drop-down menu, allowing you to select which field will be mapped on to the graphic. This enables you to colour the graphic according to the values of some field, for example. The check box also activates the *spectrum* drop-down menu.
- Spectrum: This drop-down menu is used to select which spectrum is to be used to colour the graphical element according to the field selected in the *data* setting. Spectra are edited in the *spectrum-editor-window*.
- Selected material: This drop-down menu allows you to set which material will be used to render parts of the graphic which are selected.

The eight types of graphics

- **node_points**

  Node points are used to visualize nodes. You can use *glyphs* to represent node points. There are a range of built-in glyphs in CMGUI, and it is possible to create custom glyphs as well. *Node points* graphics have the following settings in addition to the common ones listed above:

  - Glyph: this drop-down menu allows you to choose the glyph that will be rendered at the node points.
  - Offset: this box allows you to offset the origin of the glyph in order to alter its placement with respect to the node point.
  - Base glyph size: This box allows you to enter a size (x,y,z) for the glyph in the same scale as the coordinate system for the region.
  - Orientation/Scale: This check box enables a drop-down menu that allows selection of the field that the glyphs will be oriented and scaled according to.
  - Scale factors: This box allows you to enter how each dimension scales according to the orientation/scale field value.
  - Variable scale: This check box enables a drop-down menu that allows selection of the field that acts as the variable scale field. For more information on this and other of the *node points* options, see the document on working with *glyphs*.
  - Label: use this drop-down menu allows you to add field-value labels to the glyphs.

- **data_points**

  Data points are used to visualize data points. Like node points, they can be represented using *glyphs*. They have the same settings as *node points*.

- **lines**

  Lines are used to visualize 1D elements, or the edges of 2D or 3D elements. They are simple, unshaded lines that have a fixed, specified width. They have the following specific settings:

  - Exterior: This check box will automatically only render lines on exterior surfaces of a mesh.
  - Face: This check box enables a drop-down menu that allows you to choose which faces are drawn according to xi values. You can select xi=0 or 1 for each of the three xi-directions.
  - Width: this allows you to specify the width of the lines in pixels. This is a constant width that does not scale according to the zoom level.

- **cylinders**

  Cylinders are used to visualize the same things as lines. They are shaded cylinders of a specified radius. They have the following specific settings:

  - Exterior: This check box will automatically only render lines on exterior surfaces of a mesh.
  - Face: This check box enables a drop-down menu that allows you to choose which faces are drawn according to xi values. You can select xi=0 or 1 for each of the three xi-axes.

  - Constant radius: This allows you to set the radius of the cylinders, in the units of the coordinate system.
  - Scalar radius: This check box will activate a drop-down menu allowing you to select which field will be used to scale the radius of the cylinders. It will also activate a text box in which you can enter the scale factor, or how the scale field will scale the radius.
  - Circle discretization: This sets the number of sides used to render the cylinders in the 3D window.
  - Texture coordinates: This drop-down menu allows you to select which field will be used to position any textures applied by the material setting.

- **surfaces**

  Surfaces are used to visualize 2D elements or the faces of 3D elements. They are shaded surfaces of zero thickness that are automatically shaped according to the nodes defining the element they represent. Their level of detail is specified per surface by choosing a *tessellation* object. They have the following specific settings:

  - Exterior: This check box will automatically only render surfaces on exterior surfaces of a mesh.
  - Face: This check box enables a drop-down menu that allows you to choose which faces are drawn according to xi values. You can select xi=0 or 1 for each of the three xi-axes.
  - Render type: This drop down menu allows you to select shaded (default) or wireframe rendering of surfaces. Wireframe rendering renders the surfaces as grids of shaded lines, with the grid detail determined by the *tessellation* setting.
  - Texture coordinates: This drop-down menu allows you to select which field will be used to position any textures applied by the material setting.

- **iso_surfaces**

  Iso-surfaces are used to represent a surface that connects all points that share some common value. For example, in example a7 an iso-surface is used to represent a surface at which every point has a temperature of 100 degrees C. They have the following specific settings:

  - Use element type: This drop down menu allows you to select which type of element will have surfaces rendered on it. Type *use_elements* is the default. The types *use_faces* and *use_lines* will render element points only on those components of elements. If faces or lines are chosen, the following options are activated:
    - Exterior: This check box will automatically only render iso-surfaces on exterior surfaces of a mesh.
    - Face: This check box enables a drop-down menu that allows you to choose on which faces iso-surfaces are drawn, according to xi values. You can select xi=0 or 1 for each of the three xi-axes.

  It is worth noting that if you select *use_surfaces* then the equivalent of iso-surfaces becomes iso-lines. If you select *use_lines* then you will not get any visual representation.

  - Iso-scalar: This drop down menu allows you to select the field that the iso-surface will be rendered according to the values of.
  - Iso-values: This settings box contains the following settings:
    - List: This radio button activates a text box that allows you to enter a value at which to draw the iso-surface.
    - Sequence: This radio button activates three text boxes that allow you to enter a sequence of evenly spaced values to draw iso-surfaces at. The *Number* box allows you to enter the number of iso-surfaces you want. The *First* and *Last* boxes allow you to enter the starting and ending values of the iso-surfaces. The sequence will automatically space the number of surfaces between these two values.
  - Render type: This drop down menu allows you to select shaded (default) or wireframe rendering of surfaces. Wireframe rendering renders the surfaces as grids of shaded lines, with the grid detail determined by the chosen *tessellation* object.

  - Texture coordinates: This drop-down menu allows you to select which field will be used to position any textures applied by the material setting.

- **element_points**

  Element points are used to visualize the discretized points within an element. Elements may be 1, 2 or 3 dimensional, in which case the element points are spaced along the line, across the surface, or throughout the volume according to the chosen *tessellation* object . They have the following specific settings:

  - Use element type: This drop down menu allows you to select which type of element will have element points rendered on/in it. Type *use_elements* is the default, and renders element points throughout 3D elements. The types *use_faces* and *use_lines* will render element points only on those components of elements. If faces or lines are chosen, the following options are activated:

    - Exterior: This check box will automatically only render element points on exterior surfaces of a mesh.

    - Face: This check box enables a drop-down menu that allows you to choose on which faces element points are drawn according to xi values. You can select xi=0 or 1 for each of the three xi-axes.

  - Xi discretization mode: this drop down menu allows you to select the method by which element points are distributed across the element.

  -

- **streamlines**

  Streamlines are a special graphic for visualizing *vector* fields - for example, a fluid flow solution. They can be used to visualize 3, 6 or 9 component vector fields within a 3 dimensional element. In example ao, streamlines are used to show the fibre and sheet directions in the heart. Streamlines will align along their length according to the first vector of a vector field, and across their “width” (eg the width of the *ribbon* or *rectangle* streamline types) to the second vector. For single vector (3 component) vector fields, the width of the streamlines will align to the curl of the vector.

  Note that streamlines can be quite expensive to compute; changes to streamline settings in the *scene editor* can take several seconds to appear in the 3D window, especially for complex scenes.

  Streamlines have the following specific settings:

  - Streamline type: This drop-down box allows you to select the shape of the streamlines; that is, the shape outline that is extruded along the length of the streamline. *Line* and *Cylinder* can be used to visualize streamlines without showing orientation (curl). *Ellipse*, *rectangle* and *ribbon* types will enable visualization of the direction of the vector orthogonal to the streamline direction.

  - Length: Enter a value into this box to set the length of the streamline/s.

  - Width: Enter a value into this box to set the width of the streamline/s.

  - Stream vector: This drop-down box allows you to select the vector that is being visualized by the streamlines.

  - Seed element: This setting has a check box which when ticked allows you to specify a single element to seed a streamline from.

  - Xi discretization mode: This drop-down box allows you to set the point in xi-space from which streamlines are seeded. The setting of *exact_xi* for example will always seed the streamline at the exact centre of the element’s xi-space.

  - Reverse: Checking this box reverses the streamline.

  - Seed element: Checking this box allows you to select the single element number from which the streamline will be seeded.

  - Xi: Entering three comma-separated values (between 0 and 1) allows you to set the xi location within elements from which streamlines will be seeded.

- **point**

Point graphics are used to add a single glyph to the scene. This is the graphical setting that is used to replace the old axis creation, for example.

CMGUI Graphics Window

The graphics window is where all visualizations set up in the *scene editor* window are displayed. It also has tools which allow some interactive manipulation of the data being visualized. The window consists of a control panel on the left hand side, and the display area on the left. At the top of the control panel area are a selection of general controls under the "Options" label: view all, save as, and perspective. The view all button will zoom out the viewing area so that all the graphics are visible. The save as option provides the ability to save the viewing area as a raster graphic, such as a png or jpg file. The perspective check box switches convergence on or off in the 3D display.

Immediately underneath the options section are controls for selecting how the 3D display is configured. The *Layout* drop down list contains a list of display layouts, such as orthogonal or pseudo-3D views. If an applicable layout is selected, the *up* and *front* controls will become available in order to change the viewpoint. Many of these layouts contain multiple *scene viewers*, which may be useful in specific situations. The default layout *simple* consists of a single 3D scene viewer.

Below the layout section is a button labelled "Time Editor". This opens the timeline controls (much like the controls in a media player) that allow you to play animations if they have been set up.

Below this are the 5 "Tools" buttons. These offer different ways of interacting with the 3D display.

The graphics window tools

- **Transformation mode**

  This is the default mode, and is used for simple viewing of the graphical representations you are working on. In this mode the objects in the 3D window can be rotated, translated and zoomed using the mouse. Holding the left mouse button within the 3D view and moving it around will rotate, or tumble the view. Holding down the right mouse button and moving the mouse up and down will zoom the view in and out, and holding the middle button will allow you to translate the view around along two axes. It is useful to spend some time getting used to the way these manipulations work.

  In CMGUI, the rotate function works slightly differently from how similar view manipulations work in most software. This may not be immediately obvious, as the function does not "feel" particularly different in use; nevertheless there are some useful features of CMGUI's particular technique for rotating the view.

  Essentially, where you initially click in the view is the "handle" that you then move around by moving the mouse. In CMGUI this handle is different to most applications, in that it is like "grabbing" a point on a sphere that bounds the object in the 3D window. This allows manipulations using the rotate function that are not possible in most 3D views, such as rotating the object around an arbitrary axis, or rotating it in a circular fashion around the centre of the view. These abilities can be useful when looking at data that has aligned features.

  The four other tools available are used for the selection and limited editing of the type of item they refer to. Selected items are able to be targeted by commands input to the command line, or edited from within the graphics window.

  When any of the following four tools is selected, holding down the `Ctrl` key will temporarily switch you back into transformation mode in order to manipulate the view.

- **Node Tool**

  The node tool allows the selection and editing of individual nodes from within the graphics window. Selected nodes will turn red by default - the selected colour of a node is editable via the *scene editor* window. There are a range of other options to allow the editing, (moving nodes within the scene viewer) deletion, or creation of nodes. It is also possible to create 1D, 2D or even 3D (lines, surfaces and volumes) elements using the node tool; this functionality is somewhat experimental and is of limited use in most cases.

NOTE: It is somewhat easier to edit nodes (or indeed any of the other editable items) when they are represented by an easily clickable glyph such as a sphere or cube, rather than a point.

- **Data Tool**

  The data tool allows you to select data points and edit them in the same ways that nodes are editable, with the exception of element creation/destruction which is only available in the node tool.

- **Element Tool**

  The element tool allows you to select and destroy elements. You can also set up filters that allow only the selection of line, face or volume elements. It is worth noting that volume elements have no indication of their selection unless they contain element points, which will turn red when the volume element they exist in is selected.

- **Element Point Tool**

  The element point tool allows the selection of element points within the scene viewer/s.

All of the tools that allow selection are useful for creating *groups* via the command line. You can add selected items to a group using the commands

Visualizing element fields using iso-surfaces

Iso-surfaces are graphical representations used to visualize 3D or 2D objects that connect every point where the value of a certain field is the same. This idea is analogous to the contour lines on a map, where a line connects every point of a certain height. In 3D this means that a surface connects every point of a certain value. An example of this is in example a7 where an iso-surface is used to show the surface where every point is at 100 degrees Celsius. For a simpler example, example a2 shows the creation of an iso-surface at x=0.5 within the simple cube mesh.

As with surfaces, the detail level of iso-surfaces is determined by the selected *tessellation* setting in *settings editor* area of the *scene editor* (Figure 2). Iso-surfaces are drawn to connect points on the edges of the sub-element divisions where the iso-scalar matches the chosen value. With an *tessellation* setting of 1, an iso-surface will only connect points on the line edges of the element.

Visualizing element fields using lines and cylinders

Lines and *cylinders* are graphical representations which can be used to visualize 1-D elements in CMGUI - line elements at the edges of 2-D faces or 3-D elements. In general, lines or cylinders are used to visualize the basic shape of a mesh. When you load a mesh (exnode and exelem files) into CMGUI, the mesh is by default represented by lines of the default colour and thickness: white lines 1 pixel thick. This means that a *lines graphic* is created in the default scene (see the *scene editor*), with the default settings (Figure 1).

If you wish to create these lines in the graphics window, use the go to the file menu in CMGUI and select *Read*, then *Node file*. Read in `cube.exnode` from the example a2 directory. Then using the *Read* and *Elements file* menu options, read in `cube.exelem` from the same directory. If you now create a graphics window (using the *Graphics* menu item *3-D window*) you will see this cube rendered in 1 pixel thick white lines.

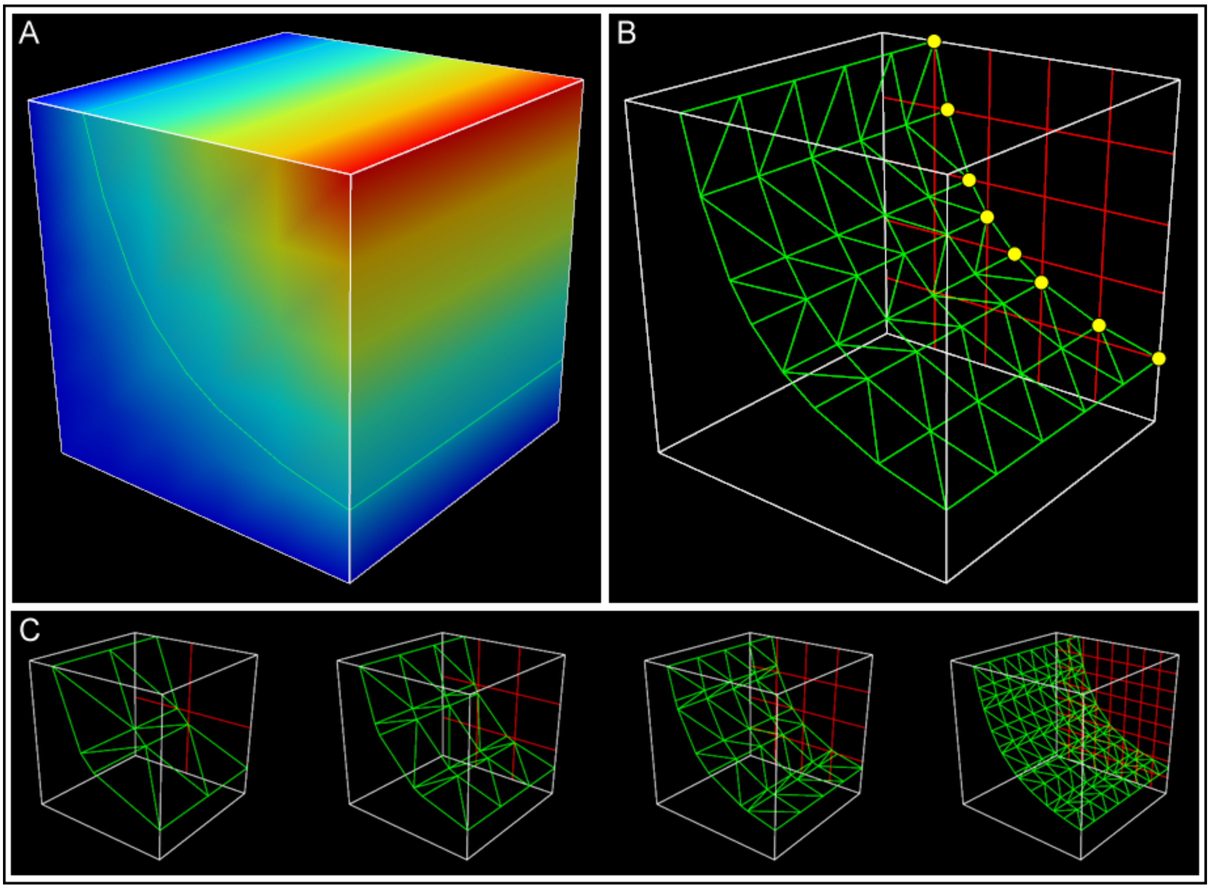


Fig. 4.14: **Figure 2: How tessellation affects iso-surface detail:** The *tessellation* setting of the iso-surface sets the number of divisions in each xi direction within each element. Here, an iso-surface is used to connect every point within the element cube where "temperature" is 20. The "temperature" field is shown on the surface of the element cube using a spectrum in panel A. In B, the iso-surface is shown in wireframe (green) and a single face of the cube is divided up according to the *tessellation* setting. You can see that the iso-surface links the discretization divisions at points where the iso-scalar is at the chosen value. These intersections are indicated by yellow circles. Panel C shows how different *tessellation* settings affect this iso-surface; settings of 2, 3, 4, and 8 minimum divisions are shown.

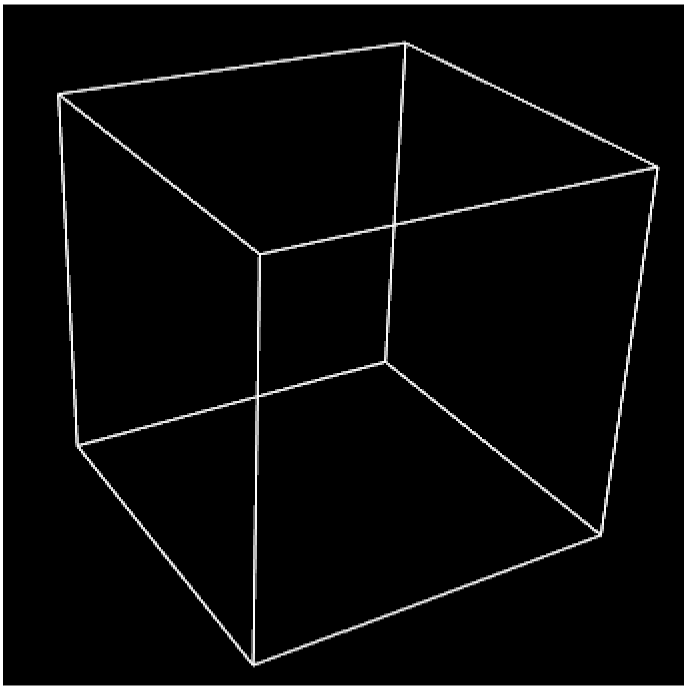


Fig. 4.15: **Figure 1: The default graphical setting lines created for a cube mesh.** This mesh was created by reading the `cube.exnode` and `cube.exelem` files from example a2. Note that the example a2 com file creates cylinders to visualize the cube mesh.

Settings for lines

Two default scene settings are important for lines and cylinders. In the scene editor, the selected *tessellation* controls the number of line segments used to draw each line. The *Circle discretization* value is used to control how many sides are used to draw each cylinder. Higher numbers will give "rounder" looking cylinders (Figure 2).

Lines have relatively few settings for altering their appearance (Figure 2). The following settings are available for lines:

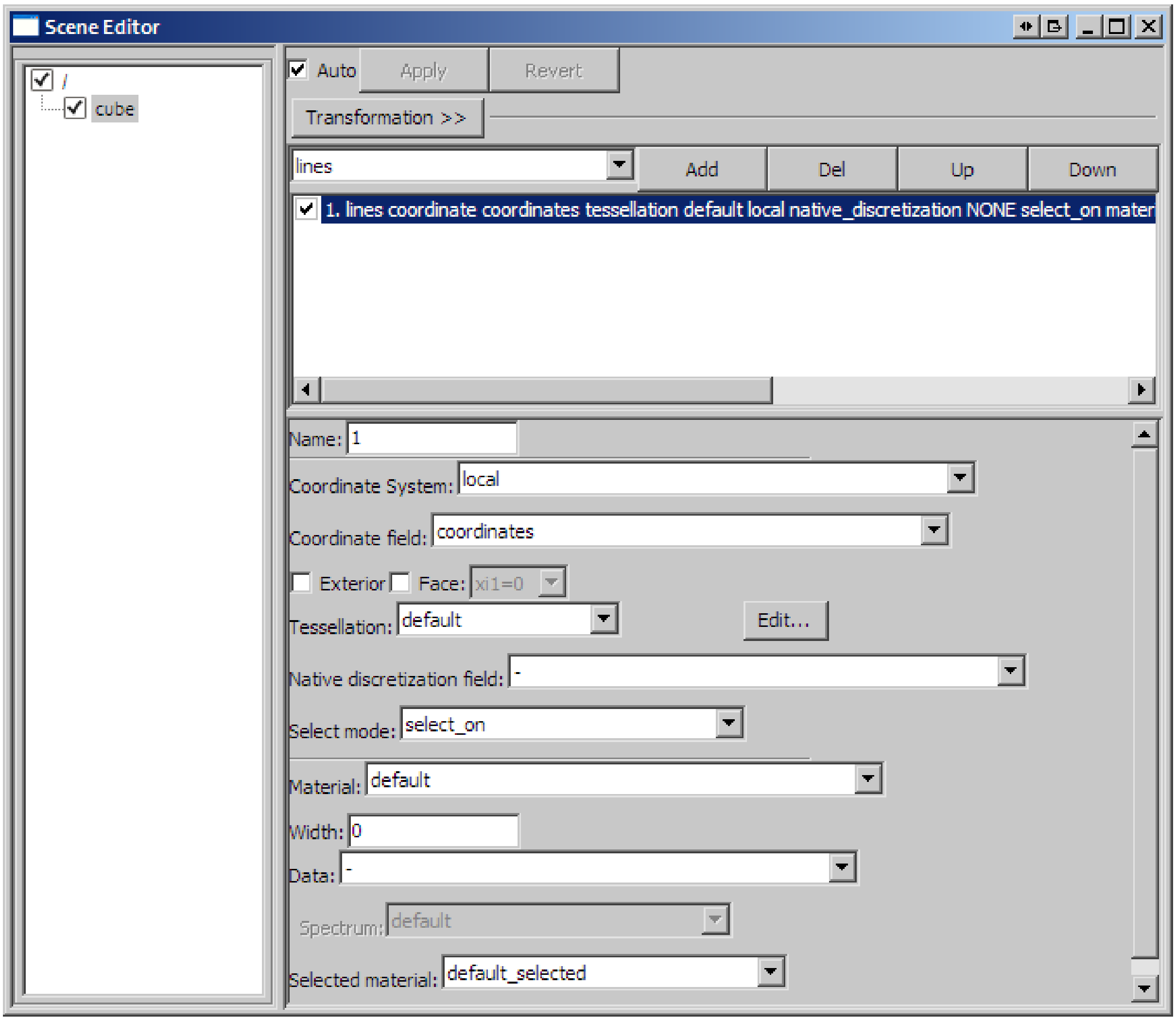
- **Coordinate field:** When you check this box, you are able to select the coordinate field that the lines are drawn according to. This is used any time the coordinate field used for the lines needs to differ from the default coordinate field used for the whole graphical element (in the general settings).
- **Exterior:** When this check box is selected, the lines will only be drawn on the exterior faces of a 3D mesh, or the outside edges of a 2D mesh. This can be useful with large, complex meshes.
- **Face:** Checking this box allows you to select which face of 3D elements is visualized by lines. Faces are selected according to one of the 3 xi directions of the element, and it's value (either 0 or 1).
- **Select mode:** This drop-down menu allows you to select different selection behaviours for the lines.
  - select_on - The default setting; line elements are able to be selected and selected elements are highlighted by rendering in the selected material.
  - no_select - No selection or highlighting of line elements.
  - draw_selected - only selected lines are drawn.
  - draw_unselected - only unselected lines are drawn.
- **Material:** This drop down menu allows you to select which material the lines will be rendered as. Materials are defined in the *material editor window*. Note: the material for lines is unshaded. This means that lines only use the *diffuse* colour for the selected material to draw the lines.
- **Width:** You can enter a value in this box to set the thickness of the lines in pixels. This width is independent of zoom, and remains constant through any transformation. Setting this value to 0 results in lines of 1 pixel

wide (the default).

- **Data:** This setting allows you to choose a field which is used to colour the lines according to a spectrum. Use the *Spectrum* drop-down menu to choose from one of the spectra defined in the spectrum editor window.
- **Selected material:** Use this drop-down menu to select which material will be used to render selected lines.

In addition to these settings there is a command line setting that can be very useful when using line based visualizations: `gfx modify window 1 set perturb_lines`. This command helps to prevent the "dotted lines" effect that occurs when lines and surfaces interfere.

Note: if no lines appear, you may not have added faces (and lines) to the mesh - try the `gfx define faces` command.



☑ 1. lines coordinate coordinates tessellation default local native_discretization NONE select_on mater

Name: 1

Coordinate System: local

Coordinate field: coordinates

□ Exterior □ Face: xi1=0

Tessellation: default Edit...

Native discretization field: -

Select mode: select_on

Material: default

Width: 0

Data: -

Spectrum: default

Selected material: default_selected

Fig. 4.16: **Figure 2: The scene editor settings available for a lines graphical setting.**

Settings for cylinders

Cylinders are very similar to lines, with a few additional parameters. A cylinders graphical setting will draw cylinders along all the same elements that a lines graphical setting would. Cylinders are different to lines in that they have a size relative to the mesh - therefore they scale with zooming just like other objects that have an actual "size" in the scene. The number of "faces" that are used to display cylinders is set under the *General settings* under *Circle dicretization*. The higher the number, the more circular the cylinders will appear. The default setting is for six sides. Cylinders have the following settings in addition to those for lines:

- **Constant radius:** This is the radius of the cylinders, in the same units as the coordinate system.
- **Scalar radius:** This drop-down menu allows you to select a field to scale the radius of the cylinders. A good example of this is shown in example a4.
- **Scale factors** This box allows you to enter three values as factors for the scaling in three dimensions. It is possible using this to exaggerate or reduce the scaling, or to restrict scaling to one or two dimensions.

CMGUI Material Editor Window

The material editor window is where you define materials to be applied to graphical elements or objects in the graphics window. Along the top of the material editor window are three buttons; create, delete and rename. You can create a new material, delete or rename an existing material using these buttons. Below these buttons is the list of currently defined materials. These will contain the default materials, as well as any defined in any comfile that has been run.

Below the material list is a panel containing four colour editors. These control the ambient, diffuse, emitted and specular colours.

- Ambient - The ambient colour is the “unlit” colour, or the colour of parts of the object that are in shadow.
- Diffuse - This is the overall colour of the material, the colour that the lit parts of the object will appear.
- Emitted - The emitted colour is the “glow” of a material; this colour will appear in both the lit and unlit parts of the material.
- Specular - This is the colour of the shine that appears on the material. This shine appears as a glossy highlight.

Each colour can be edited using three sliders or textboxes, and you can choose from three colourspaces for editing the colour. The default is RGB, but the HSV and CMY colourspaces are also available from a drop-down menu at the top of each colour editor. Each colour editor has a preview panel showing a flat sample of the colour that is being edited.

Below the colour editors is the surface editor. This panel allows you to set the alpha, shininess, and texture properties of the surface of the material being edited. The alpha value sets the transparency of the material. The shininess sets the “tightness” or size of the specular highlights of a material; generally the higher the shininess, the smaller and harder-edged the highlights. Higher shininess makes a material look glossier. The surface editor also allows you to assign a texture to the material surface - this option is unavailable unless you have created at least one texture by using the `gfx create texture` command to read in some bitmapped graphics. There are two other check box options available in the surface editor: “per pixel shading” which greatly increases the quality of the lighting of the material, and bump mapping which allows you to use a texture file to create surface details. These options are only available on hardware that supports advanced forms of shading.

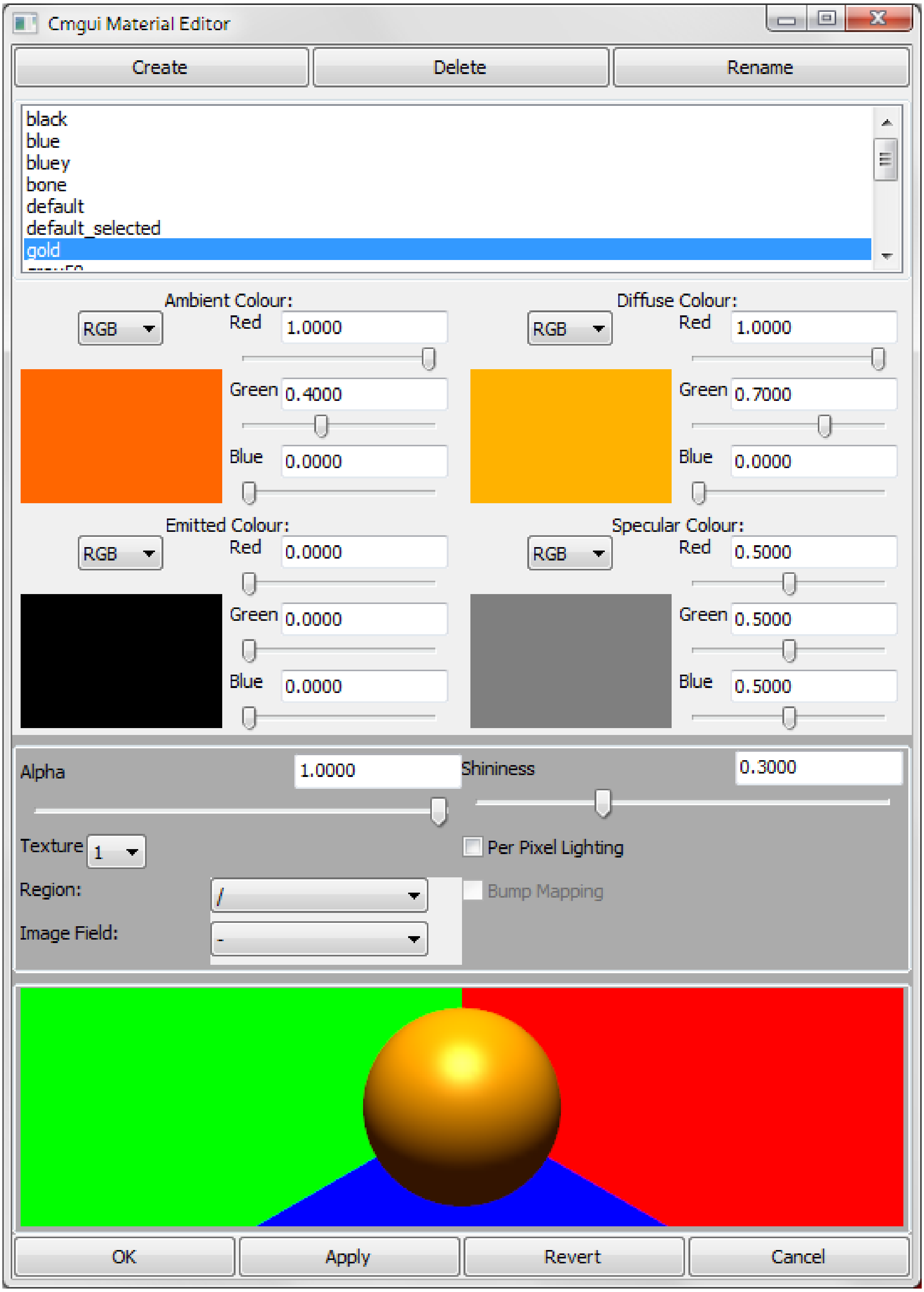
Below the surface editor is a panel that shows a preview of the currently edited material applied to a sphere. You may need to resize the material editor window in order to see a usefully large preview panel. Clicking in the preview panel will cycle the background from the RGB colours through black, white and back to RGB.

At the bottom of the window are four buttons: OK, apply, revert and cancel. The OK button leaves the editor and applies the changes made. The apply button immediately applies current changes to the materials, allowing you to see how they look if they are used in any objects shown in the graphics window. The revert button will undo any changes made to the currently edited material, and the cancel button exits the material editor window without applying changes.

CMGUI Scene Editor Window

The scene editor is used to control how visualizations appear in the graphics window. From this window you can:

- Toggle visibility for any part of the region tree.
- Toggle visibility of any individual graphics setting for a region.



black
blue
bluey
bone
default
default_selected
gold

Ambient Colour: RGB
Red 1.0000
Green 0.4000
Blue 0.0000

Diffuse Colour: RGB
Red 1.0000
Green 0.7000
Blue 0.0000

Emitted Colour: RGB
Red 0.0000
Green 0.0000
Blue 0.0000

Specular Colour: RGB
Red 0.5000
Green 0.5000
Blue 0.5000

Alpha 1.0000

Shininess 0.3000

Texture 1

□ Per Pixel Lighting

Region: /

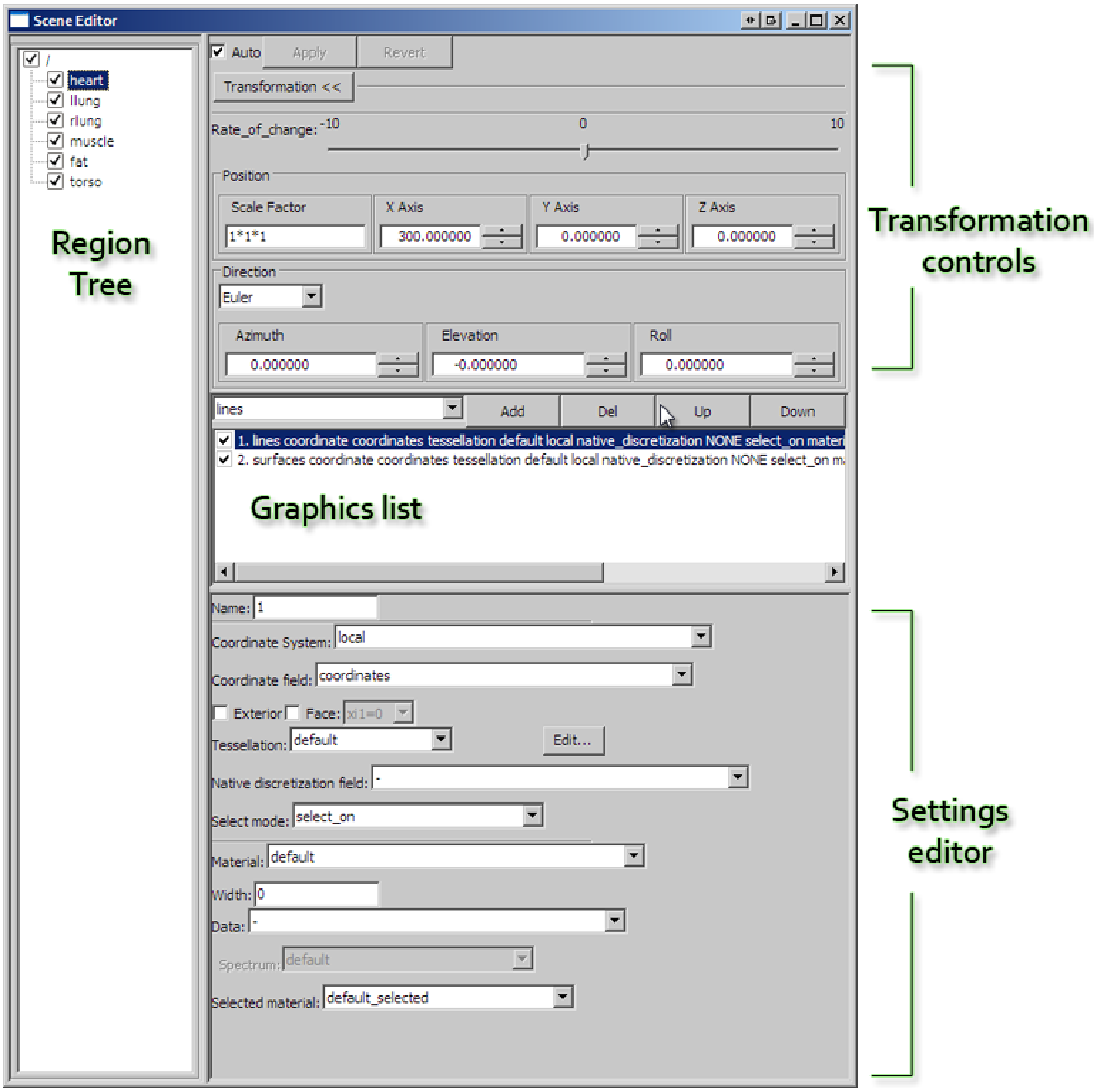
□ Bump Mapping

Image Field: -

Fig. 4.17: **Figure 1: The CMGUI Material Editor**

- Add, remove, and edit graphics.
- Alter the order in which graphics are drawn.

The window itself is broken into four main areas; region tree, transformation controls, graphics list, and settings editor (Figure 1).



☑ /
☑ heart
☑ llung
☑ rlung
☑ muscle
☑ fat
☑ torso

☑ Auto

Rate_of_change: -10 0 10

Position

Scale Factor	X Axis	Y Axis	Z Axis
1*1*1	300.000000	0.000000	0.000000

Direction: Euler

Azimuth	Elevation	Roll
0.000000	-0.000000	0.000000

lines

☑ 1. lines coordinate coordinates tessellation default local native_discretization NONE select_on materi

☑ 2. surfaces coordinate coordinates tessellation default local native_discretization NONE select_on m

Name: 1

Coordinate System: local

Coordinate field: coordinates

□ Exterior □ Face: xi1=0

Tessellation: default

Native discretization field: -

Select mode: select_on

Material: default

Width: 0

Data: -

Spectrum: default

Selected material: default_selected

Fig. 4.18: **Figure 1: The scene editor.**

Region tree

At the left hand side of the Scene Editor window is the region tree, where all regions are listed. Each region has a visibility toggle, controlled by clicking on the box alongside its name. All ticked regions will be visible in the graphics window, provided that they have some visible graphics. Regions may contain sub-regions, allowing powerful control over visibility of the parts of a complex model.

Alongside the region tree are the transformation settings, graphics list, and settings editor. These are used to control and edit the visualisations of the currently selected region in the region tree.

Note: Previous versions of cmgui also had a “General Settings” section in this window, which was used to set the level of detail (discretization) of objects. This system has been replaced by the new tessellation object. You are now able to use different levels of detail for different graphics, allowing you to choose the most appropriate for each. This also allows you to change the detail levels of a number of graphics by simply altering a single tessellation object that has been applied to them all. The tessellation is set for each graphical setting using the settings editor.

Graphics list

Below the transformation and general settings buttons is the *graphics list*. This is where the visual representations of the currently selected region are listed. This panel allows creation, deletion, visibility switching (on or off) and re-ordering of these visual representations.

Settings editor

Below the *graphics list* is the *settings editor* where each graphical setting can be edited. When a graphical setting is selected from the list, all of its editable properties appear in this area. The range of editable properties will vary depending on the type of graphics currently selected.

CMGUI Spectrum Editor Window

The spectrum editor window is where you define spectra to be applied to graphical elements or objects in the graphics window. Spectra are used to visualize continuous data ranges within models using colour ranges, and can be applied to the *graphics* that have been used to create your visualization. The window is divided into three basic areas; the spectrum list, the preview panel, and the settings editor (Figure 1).

It is useful to step through example a7 to get a feel for the use of spectra in cmgui. This example uses a mesh containing a number of fields that can be usefully visualized using spectra.

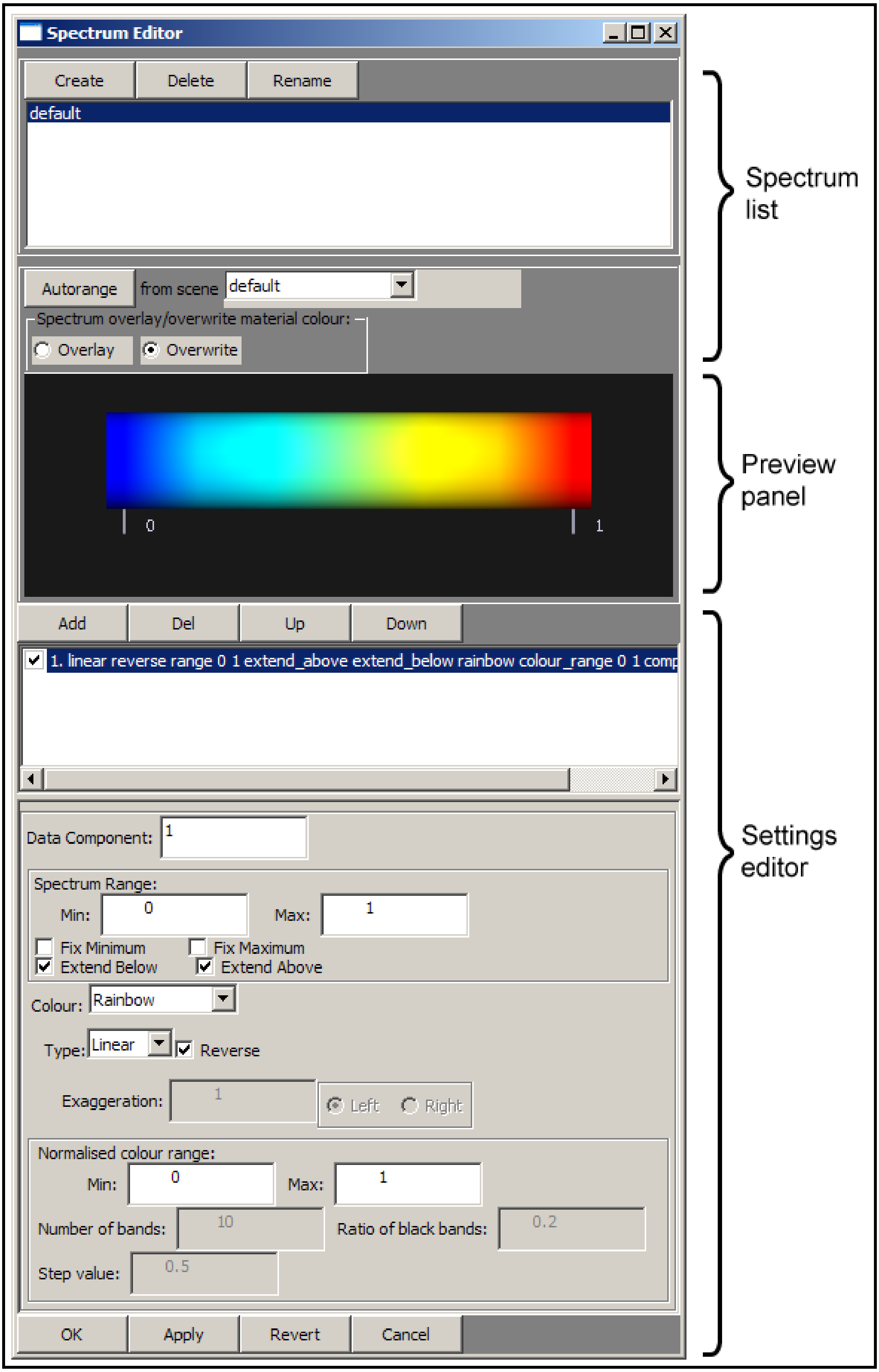
Spectrum List

The spectrum list shows all the currently defined spectra; it will always contain the *default* spectrum if no others have been defined. Three buttons at the top of the window allow you to *create*, *delete*, or *rename* spectra. Just below the list of spectra are some controls that allow you to set some of the general properties of the selected spectrum. The *autorange* button automatically sets the minimum and maximum values of the selected spectrum, according to the smallest and largest values it has been applied to in the *scene editor*. For example, if the default spectrum has been used to colour a temperature field in the default scene, and that field has values ranging from 0 to 100 degrees Celsius, pressing the *autorange* button will set the minimum and maximum values (in the *Spectrum range* settings - see below) of the selected spectrum to 0 and 100 respectively. If the selected spectrum has been used to colour objects according to more than one field, the *autorange* function will choose the smallest and largest values across all of these fields. The *from scene* drop-down menu allows you to select the scene from which the spectrum will be auto-ranged.

The *overlay* or *overwrite* options allow you to choose whether a spectrum will completely over-ride any other material settings (*overwrite*), therefore completely colouring the object as the spectrum appears in the preview panel - or whether the spectrum combines with the other material settings of the object so that the final colour is a combination of the spectrum and other material settings (*overlay*).

Preview Panel

This panel shows a horizontal bar, coloured using the selected spectrum. The bar also shows the range that the spectrum is currently set to, using a series of numbered labels. Clicking in the preview panel changes the number



default

Autorange from scene: default

Spectrum overlay/overwrite material colour:
○ Overlay ● Overwrite

☑ 1. linear reverse range 0 1 extend_above extend_below rainbow colour_range 0 1 comp

Data Component: 1

Spectrum Range:
Min: 0 Max: 1
□ Fix Minimum □ Fix Maximum
☑ Extend Below ☑ Extend Above

Colour: Rainbow

Type: Linear ☑ Reverse

Exaggeration: 1 ● Left ○ Right

Normalised colour range:
Min: 0 Max: 1
Number of bands: 10 Ratio of black bands: 0.2
Step value: 0.5

Fig. 4.9: Figure 1: The spectrum editor.

of these labels from 4 to 10 to 2, then cycles through these values. Currently the preview panel can not display multi-component spectra.

Settings Editor

The settings editor is where each spectrum is set up. It contains a number of controls.

- **Spectrum component list:** The top of this list has four buttons; *Add*, *Delete*, *Up*, and *Down*. Below these buttons is a list of the components that make up the selected spectrum. Spectra in cmgui can be made up of multiple components; these can be added, deleted or re-ordered using this list. Using these “sub-spectra” you are able to create spectra that have different colour ranges for different parts of the data range they cover, or spectra that have different colour ranges for different dimensions.
- **Data component:** This text box allows you to enter which data component of a multi-component field the spectrum will colour according to.
- **Spectrum range:** This set of controls is used to set up the range of values the spectrum covers. The *Min.* and *Max.* text boxes allow you to enter values for the minimum and maximum values of the spectrum. There are also four check boxes that allow you to change the behaviour of a spectrum at its start and end points.
- **Colour:** This drop-down menu provides a selection of pre-set colour settings that can be applied to the currently selected spectrum component. Although many of the colour ranges are fairly self-explanatory, a number of them have special features which are useful for creating specialized spectra. The *contour bands* and *step* colour settings have further settings associated with them.
  - Rainbow: This is a standard rainbow colour range.
  - Red: This is a red to black colour range. It is applied as a single “channel” of red to the spectrum; this can be used to add to other spectrum components to make multi-component spectra.
  - Green: This is a single channel green to black colour range.
  - Blue: This is a single channel blue to black colour range.
  - White to blue: This is a white to blue colour range.
  - White to red: This is a white to red colour range.
  - Monochrome: This is a black to white colour range.
  - Alpha: This is a single channel transparent to opaque spectrum.
  - Contour bands: This colour option creates a series of evenly spaced black bands that can be further edited to get the correct number, width and spacing desired.
  - Step: This is a colour range which has a sharp transition from saturated red to saturated green.
- **Type:** This drop down menu allows you to choose between linear and log scale spectra. The *Reverse* check box allows you to swap the direction of the spectrum. If a log scale is selected as the type, the *Exaggeration* settings become available.
  - Exaggeration: This text box allows you to specify how strongly the spectrum is exaggerated.
  - Left/Right: These radio buttons allow you to choose the direction of the exaggeration.
- **Normalised colour range:** These two text boxes allow you to specify the portion of the colour range that is displayed across the chosen range of values for the spectrum component. The default values of 0 and 1 display the entire colour range, and other values allow you to choose where in the colour range the spectrum component begins and ends. For example, 0.5 and 1 would display only the upper half of the colour range; 0 and 0.5 would display only the lower half; 0.25 and 0.75 would display the middle section of the colour range.
- **Number of bands:** This text box is enabled when the *contour bands* colour type is chosen. Enter a value to specify the number of contour bands.

- **Step value:** This text box is enabled when the *step* colour type is chosen. Enter a value somewhere between the values specified in *Spectrum range* to specify where the step occurs in the spectrum.

Visualizing element fields using surfaces

Surfaces are graphical representations that can be used to represent two or three dimensional elements in CMGUI. Surfaces can be used to represent the faces of a 3D element or mesh, for example.

Surfaces and are created as *graphics* in the *scene editor*. To create a very simple surface graphic, go to the *File* menu, select *Read*, then *Node file*. Read in `cube.exnode` from the example a2 directory. Then using the *Read* and *Elements file* menu options, read in `cube.exelem` from the same directory. If you now create a *graphics window*, you will see a simple cube rendered in the default lines graphic. Open the *scene editor* window by selecting it from the *Graphics* menu. Select the *cube* scene object, then in the graphics panel select *surfaces* in the drop-down menu. Click the *Add* button to the right of this menu. A new graphical setting will appear in the list, below the lines graphical setting. In the 3D window, your cube will now be rendered with white shaded surfaces.

The *tessellation* setting of the surface graphical setting determines the detail level of surfaces. Each face in a surface graphical setting is divided into a number of squares (each made up of two triangles) determined by the tessellation. The discretization can be independently set for each of the three coordinate dimensions, allowing you to set different detail levels for different parts of surface.

Creating an AVI from a series of images - Windows

This tutorial will describe a method of compiling a series of images saved from CMGUI into an AVI format movie file on the Windows operating system. There are other methods, but the one described here can be achieved easily using free software, and produces excellent, highly customizable results.

An AVI file is a Windows format - however, it can be played on most computers including those running Mac OS X or Linux. It can also be uploaded to online video sites such as YouTube.

Software used

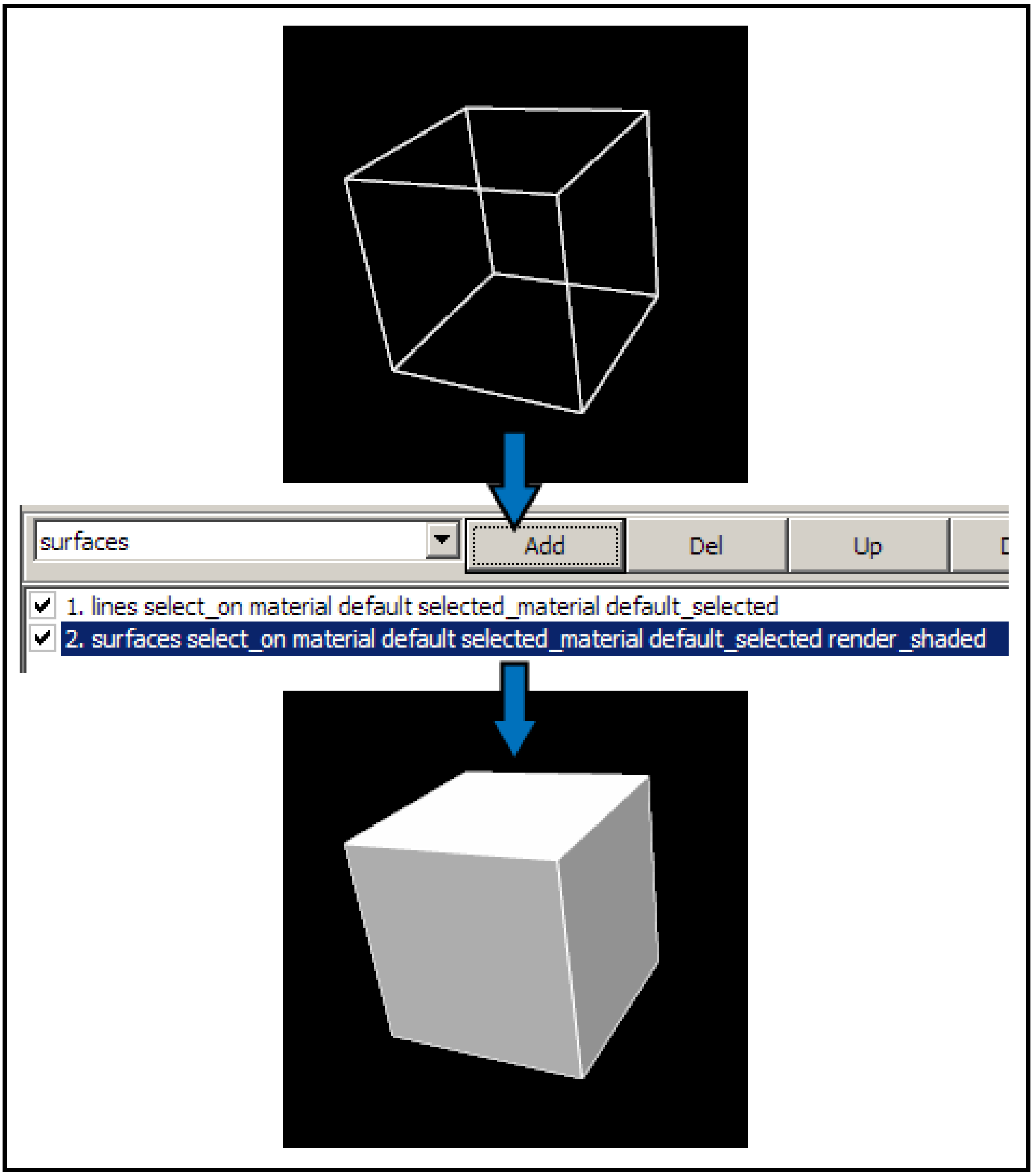
The following pieces of software are used in this tutorial - click on the name of the software to navigate to a download page for each item:

- VirtualDub - This is an open source AVI editing/capturing application. For the purposes of this tutorial, it allows you to load in a series of image files as movie frames, and save the resulting movie as an AVI.
- FFDShow - This is a Video for Windows and Directshow codec for decoding and encoding a large number of video and audio formats. With this installed you can save AVI files from VirtualDub using a large range of codecs.
- WinFF - This is a free windows front-end to **ffmpeg_** (. This tool allows easy conversion of videos from one format to another.
- CamStudio - This is a free open source tool for recording screen activity. This represents an alternative method for capturing quick movies of CMGUI.

Installation of VirtualDub and FFDShow is a requirement for following this tutorial. You will need administrator rights on your computer to install FFDShow; VirtualDub is simply unzipped into a directory and run directly. Camstudio is only required if you wish to experiment with recording the CMGUI graphics window directly.

Compiling a series of images into a movie

When making a movie from a sequence of images, it is important to make sure that the images you created in CMGUI are saved at a usable resolution. It is always best to use familiar resolutions like 800x600, 1024x768,



☑ 1. lines select_on material default selected_material default_selected

☑ 2. surfaces select_on material default selected_material default_selected render_shaded

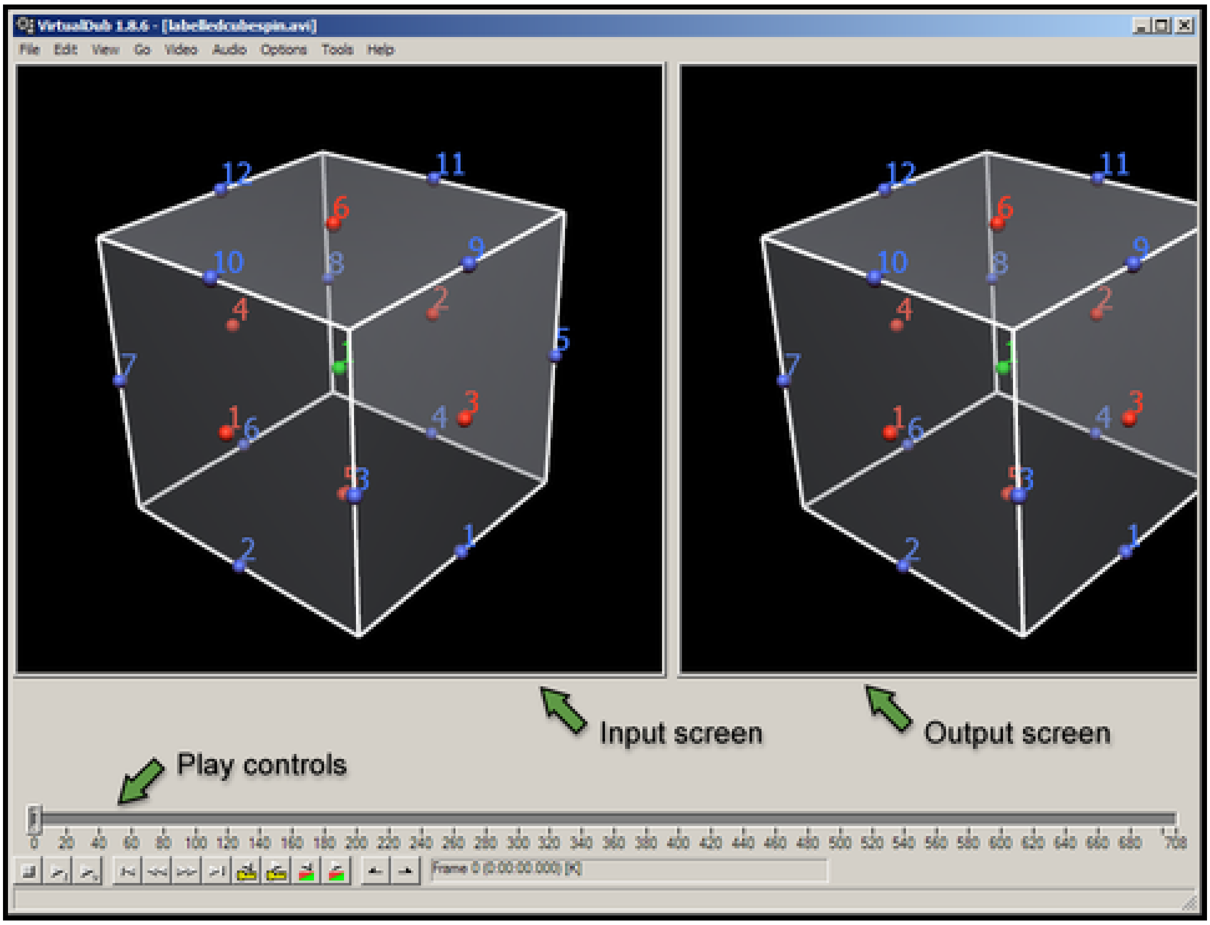
Fig. 4.20: **Figure 1: Adding surfaces to a mesh** Using the example a2 cube, adding a surface graphical setting creates a basic surface in the default material (white).

or 1920x1080. Many video formats require specific resolutions to work properly; codecs often are restricted to multiples of 2, 4, or 8 for both the height and width in pixels.

Creating a movie from a series of images using VirtualDub is very simple:

- Load VirtualDub. The first time you do this you may be presented with a window where certain options may be set. You can safely dismiss this window and load VirtualDub.
- Go to the *File* menu and select *Open video file...*. (Ctrl-O)
- Locate and load the first image in your sequence of images.

VirtualDub will automatically load the entire numbered sequence of images in as movie frames, in the correct order. This works whether you have a 00001-99999 or a 1-99999 style numbering scheme. You will now see two "screens" side by side, each displaying the first image in your sequence. This is the first frame of your movie, which can now be played within VirtualDub. The two screens display the *input* and the *output*. The *input* screen shows the movie you loaded, in this case the series of frames from the images. The *output* screen shows the original movie with any filters (such as resizing, contrast adjustments, sharpening etc) that you have applied to it.



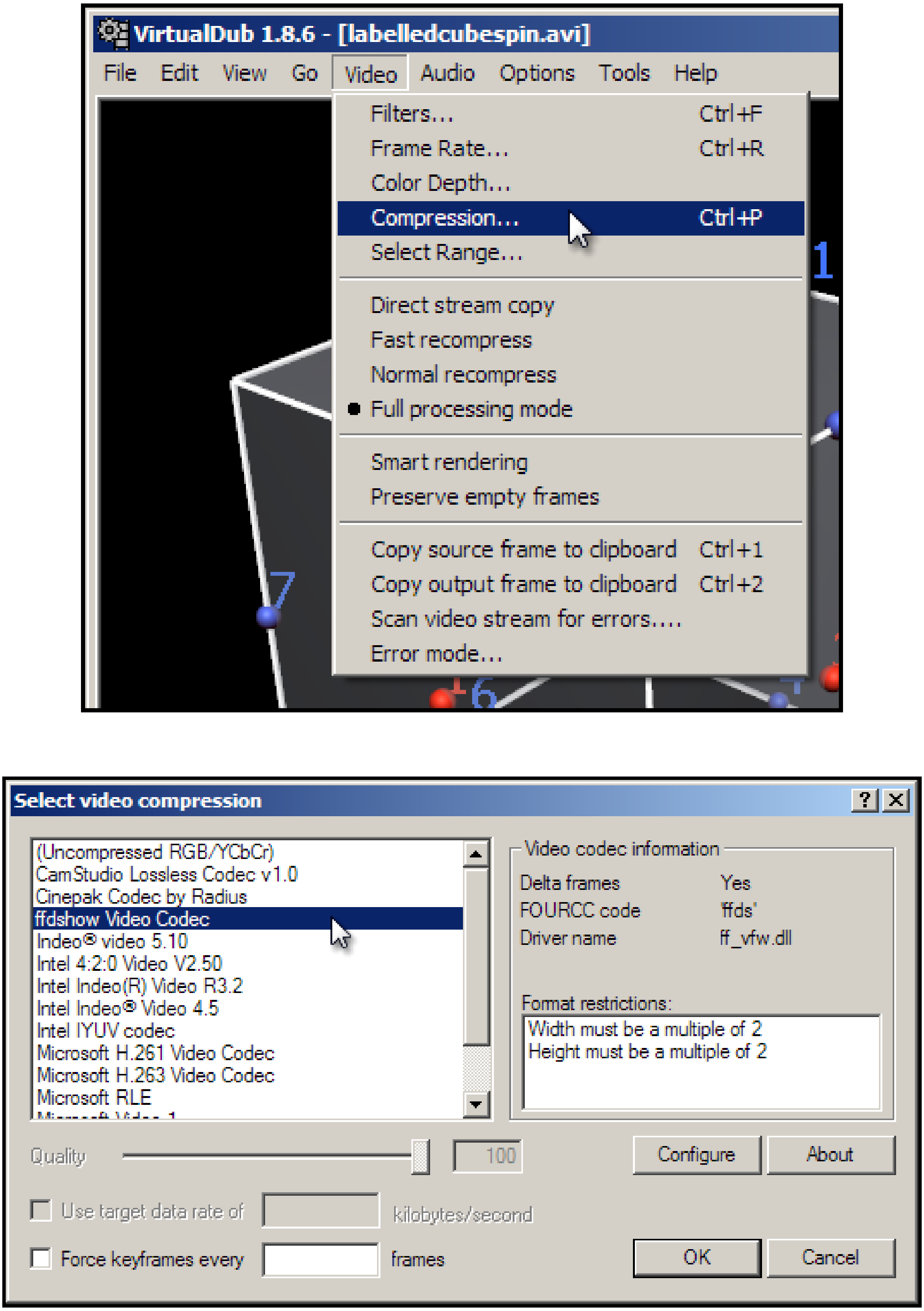
VirtualDub 1.8.6 - [labelledcubespin.avi]

Frame 0 (0:00:00.000) [K]

Fig. 4.21: **Figure 1: The main VirtualDub window** This window displays the play controls (which also include marking tools for editing the video), the input and output panels, and the menus.

The next step is to set up the compression and codec settings of the movie you are going to save.

- Go to the *Video* menu and select *Compression...*. (Ctrl-P)
- Select the *ffdshow Video Codec* from the list of codecs.



Video codec information

Delta frames	Yes
FOURCC code	'ffds'
Driver name	ff_vfw.dll

Format restrictions:
Width must be a multiple of 2
Height must be a multiple of 2

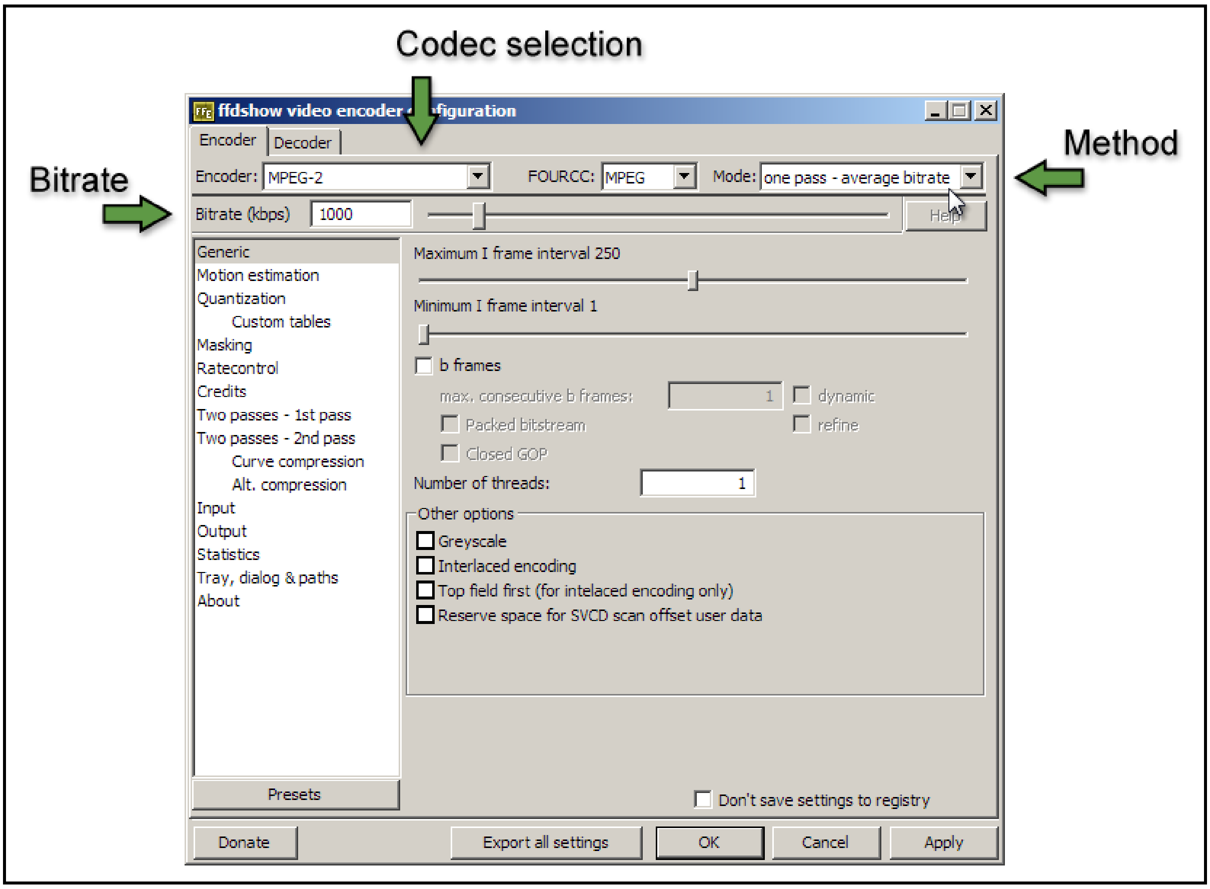
Quality 100

□ Use target data rate of kilobytes/second

□ Force keyframes every frames

Fig. 4.22: **Figure 3: Selecting a codec** The codecs listed may differ from those shown here. Select *ffdshow Video Codec*.

- Notice the *Format restrictions* box on the right hand side - this will inform you of any requirements your selected codec has for input. For the ffdshow codec, your video resolution must be a multiple of 2 in both width and height.
- Click on the *Configure* button. This will bring up a window with a range of settings for how the codec will be used to compress your movie.



Encoder: MPEG-2

FOURCC: MPEG

Mode: one pass - average bitrate

Bitrate (kbps) 1000

Maximum I frame interval 250

Minimum I frame interval 1

max. consecutive b frames: 1

Number of threads: 1

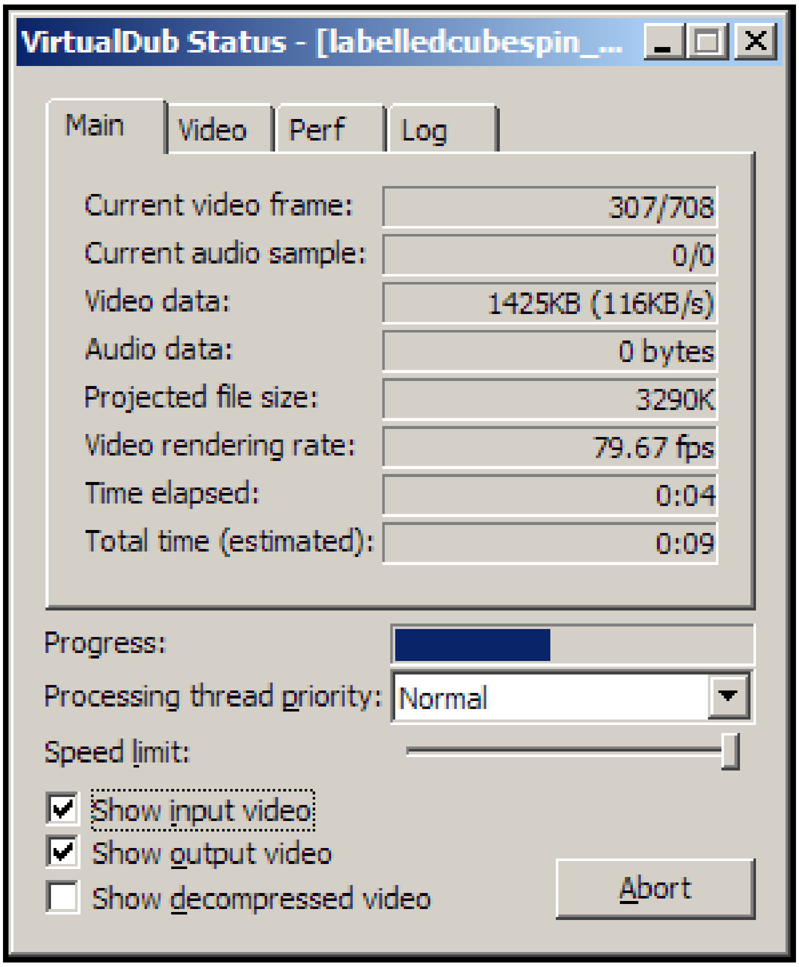
Fig. 4.23: **Figure 4: The codec configuration window** This window displays the controls for how the codec will compress your movie. The ffdshow codec actually enables you to encode your movie in any of a large number of different codecs; the codec is chosen from the *codec selection* drop-down menu.

From this settings window you can set up which codec you wish to use to compress your movie, and how much compression you wish to apply. Different codecs have different pros and cons; these days it is often best to stick to the more modern codecs, as a majority of computers on all platforms should be able to play them. Good codecs to choose are MPEG-2, MPEG4, and H.264. MPEG-2 is an older codec that offers good backwards compatibility, but encodes lower quality videos with larger file sizes. MPEG4 and its successor H.264 generally offer substantially better quality and/or smaller file sizes. The following steps will produce an AVI using the MPEG4 codec, which is widely supported. If you wish to use another codec, the steps are essentially the same.

- Select the *MPEG-4* codec from the codec selection drop-down menu.
- Make sure the *Mode* setting is *one pass - average bitrate*.
- Set the *Bitrate (kbps)* text box to `1000`.

- Click *Apply* and then *OK*. Click *OK* in the codec selection window.
- In the main VirtualDub window, go to the *File* menu and select *Save as AVI*. Enter a name for your movie and click *Save*.

A status window will appear, with some information about the progress of the AVI creation.



Current video frame: 307/708
Current audio sample: 0/0
Video data: 1425KB (116KB/s)
Audio data: 0 bytes
Projected file size: 3290K
Video rendering rate: 79.67 fps
Time elapsed: 0:04
Total time (estimated): 0:09

Progress:
Processing thread priority: Normal
Speed limit:

☑ Show input video
☑ Show output video
☐ Show decompressed video

Fig. 4.24: **Figure 5: Progress window**

Once the encoding is finished, find your file and double click on it to play it. Check the file size, and the quality of the movie. If the file is too big, change the codec settings to use a lower bitrate and save it again. If the movie is poor quality, set the bitrate higher and try again. 1000 kbps gives a good quality 640x480 movie in MPEG-4 encoding, with file sizes of about 7 megabytes per minute. With H264 encoding, bitrates as low as 300kbps can give remarkably good results at 640x480 depending on the content of the movie.

Creating a movie from images using FFMPEG

An alternative to using Virtualdub is to use the command line tool **'FFMPEG'**_. This is a powerful, open source tool that can perform a large number of video-related tasks. Documentation for FFMPEG can be found on the project website.

An example command line for creating a movie from image files using FFMPEG is:

```
ffmpeg -f image2 -i image%d.jpg output.mpg
```

For more information, please refer to the documentation linked to above.

Converting your movie to other formats

Sometimes you will want to have your movie in a format other than AVI, or you may wish to further tune the compression and quality of your movie. For example, using the H.264 codec with an AVI creates a non-standard file; it is usually better to encode an mp4 file using H.264.

Converting to mp4 using Handbrake

Handbrake is an open source video converter that is very fast and effective for converting videos to MP4 format. The application is quite easy to use, and has a number of built-in presets for converting video for certain purposes such as uploading to youtube, or playing on mobile devices.

Convert your movie to another format using WinFF

WinFF is a useful tool for optimizing the file size or quality of your movies, as well as for converting from AVI to other file formats such as mp4 or mov. When converting using WinFF, it is useful to have an uncompressed movie file to work with. To create an uncompressed AVI from VirtualDub, select the *Uncompressed RGB/YCbCr* codec when creating the AVI file as detailed above.

The following steps detail how to encode an H.264 codec mp4 file from an AVI:

- Load WinFF.
- Click on the *Options* button (far right) to show the advanced options.
- Click on the *Add* button, and select the movie file you wish to convert.
- Select *MP4* from the *Convert to...* drop-down menu.
- Select *H.264 in MP4(4:3)* from the preset drop-down menu, to the left of the *Convert to...* drop-down menu. The 4:3 refers to the aspect ratio of your video; standard sizes such as 320x240, 640x480 or 1024x768 have this aspect ratio. If your movie has a different aspect ratio, enter it into the *Aspect Ratio* text box, or enter the resolution of your movie directly into the *Video Size* text boxes.
- Enter a value such as `500` or `1000` into the *Video Bitrate* text box.
- Click on the *Convert* button.

By default, WinFF will create the converted movie file in your My Documents folder. You may choose a different desination folder for the converted movie by entering a different path in the *Output Folder* text box.

The cmgui EX format guide: exnode and exelem files

Contents

Overview

Spatially-varying "finite element" field definitions are commonly imported into cmgui in two parts:

1. ".exnode" files containing definitions of nodes which are points identified by a unique integer identifier, and for which field parameters are listed.
2. ".exelem" files containing definitions of elements which are n-dimensional coordinate spaces, and the functions giving field values over them, usually by interpolating node field parameters.

This division is arbitrary since they share a common format able to define both nodes and elements, and the fields defined over them. The extensions ".exnode" and "exelem" are merely conventions for the above uses. Any filename extension is acceptable, and full file names with extensions should always be specified in your scripts and command files.

Nodes with literal field values stored for them do not just support element interpolation, but are widely used to represent any point data, and there need not be any associated elements. You will occasionally see files with the ".exdata" extension which are identical to ".exnode" files but read in with a different command ("gfx read data . . .") so that the objects and associated fields are put in a different set of node objects referred to within cmgui as data rather than nodes, and which are unable to be interpolated in element fields. Note the current limitation of cmgui that each region consists of one list of node objects, one list of elements, and one list of data points.

General Syntax

Cmgui EX files are human-readable text files which are usually created by export from cm or OpenCMISS. They can also be created by scripts or by hand. Due to their complexity, particularly in defining element interpolation, editing an existing file is usually the easiest way to create new data files by hand.

The format has been around for many years and has several exacting rules which can catch the unwary:

1. Main blocks of the file begin with tokens such as "Group name", "Shape", "#Fields", "Node", "Element" which must be written in full with exact capitalization. Within these blocks are further tokens indicating sub-blocks of data to be read, each with their own rigid rules.
2. The characters used to separate tokens, identifiers, parameters and other data are quite inconsistent and could be white space, commas, colons, full stops (periods), equal signs and so on. In each case the precise character must be used. Whitespace separators generally includes any number of spaces, tabs, end of line and line feed characters. Non-whitespace separator characters can generally be surrounded by whitespace.
3. All array indices start at 1, but scale factor index 0 has the special meaning of using unit scale factor value, 1.0.
4. Node and element identifiers must be positive integers.

Fortunately, cmgui import commands ("gfx read node/element/data") report the line number in the file at which the first errant text is located (do not confuse this with line element numbers). Unfortunately the actual problem may be earlier: if the previous block was incomplete then the first text of the next block, however correct, will be complained about as not matching the tokens and data expected for the previous block.

Regions

The EX format defines fields, nodes, elements and groups (sets of nodes and elements) within regions. These objects belong to their respective region and are independent of similarly identified objects in other regions.

Cmgui version 2.6 (released: May 2009) introduces a "Region" keyword which specifies the path to the region which any following fields and objects are defined within until the next Region keyword or end of file. The format is as follows:

```
Region: /PATH_TO_REGION
```

Examples:

```
! The root region for this file:
Region: /

! Region "joe" within region "bob":
Region: /bob/joe
```

The region path must always be written as an absolute path from the root region of the file, i.e. as a series of zero or more region names starting with and separated by forward slash "/" characters. Region names must start with an alphanumeric character and contain only alphanumeric characters, underscores '_' and spaces; spaces are not permitted at the start or end of the name. Colons ':' and dots/periods '.' are permitted but discouraged.

Versions of cmgui prior to 2.6 implicitly read data into the root region, and will report an error for the unrecognised "Region" keyword. The first non-comment keyword in post Cmgui 2.6 EX files should be a Region keyword; to maintain compatibility with older files it may alternatively begin with a Group declaration.

With EX files you are free to put many regions (and groups) in the same file, or to keep each region (or group) in separate files. The choice is up to the user, although it is more flexible to use separate files, since commands for reading EX files into cmgui (e.g. gfx read nodes/elements/data) permit the file's root region to be mapped to any region. For maximum flexibility and to avoid potential merge conflicts, it is recommended that you keep your model data out of the Cmgui root region.

Cmgui example a/region_io gives several example EX files defining fields in regions and sub-groups.

Groups

The Group keyword indicates that all following nodes and elements until the next Group or Region keyword are "tagged" as belonging to a group (set) of the specified name:

```
Group name: GROUP_NAME
```

There can be any number of groups in a single region, each potentially sharing some or all of the same nodes and elements from the region they belong to. Groups are entirely contained within their particular region; groups with the same name in different regions are completely independent.

Groups have the same name restrictions as regions. They are not written as a path - just a single name within the enclosing region.

Older versions of Cmgui required nodes, elements and fields to always be defined within a group (within the implied root region), hence the Group keyword was always the first keyword in the file. Since Cmgui 2.6 this is no longer a limitation.

In older EX files it was common to have fields defined after/within a group declaration. However the grouping only ever applies to nodes and element - fields always belong to the region. It is more common to define the fields with nodes and elements under the region with no group, and list only node and element identifiers with no fields under the groups.

Field declaration headers

Whether the EX file is listing nodes or elements, there is broad similarity in the way in which they and their field data is listed. It consists of a header declaring the number and details of fields being defined for the respective object type, followed by the definitions of the objects themselves. An example header:

```
#Fields=1
1) coordinates, coordinate, rectangular cartesian, real, #Components=3
... field data specific to nodes or elements ...
```

The above header declares a field with the name "coordinates". The field is tagged as usage type "coordinate" - this is a hint which tells cmgui that this field is appropriate for use as a coordinate field. Other values for this hint are "anatomical" for special fibre fields, and "field" for all other fields. By default, cmgui will use the first coordinate field in alphabetical name order for the coordinates of graphics. The coordinates declared in this header are embedded in a rectangular Cartesian coordinate system with 3 real-valued components. The rectangular Cartesian coordinate system is assumed if none is specified. The value type real is frequently omitted as it is the default; other types such as integer, string and element_xi are only usable in node fields and integer for grid-based element fields. More complex declarations are given throughout this document. Note that if there is no header or the header has "#Fields=0", then nodes and elements can be defined or listed for adding to a group without defining fields.

Following each line declaring the main details of a field are the details on each field component including the name and the parameter values to be supplied with each node, or the basis functions and parameter mappings to be supplied with each element. These are described later.

There can be only one field of a given name in a region, but it can be defined on nodes, elements and data in that region provided the field is consistently declared in the header, including same value type, numbers of components and component names.

Note that the #Fields keyword in element field headers are additionally preceded by the following keywords and sub-blocks which are described in later examples:

```
#Scale factor sets=~
...
#Nodes=~
```

Comments

Since Cmgui version 2.6 EX files containing comments are now able to be read. Comment lines begin with ! (exclamation mark character) and may only be placed in parts of the file where a Region, Group, Shape, Node, Element, Values or #Field header keyword is permitted. Putting comments anywhere else will result in obscure errors or undefined behaviour!

Comments are useful for adding source details or copyright/license information to your files, or to document parts of the file. Cmgui ignores the comment lines: they will not be rewritten when exporting an EX file.

Some example comments preceding other keywords:

```
! Copyright (C) 2009 The Author
Region: /heart

! This following node is the apex of the heart:
! It has 10 versions of all nodal parameters
Node: 13
```

Note that errors are reported if you attempt to read EX files with comments into versions of Cmgui prior to 2.6.

Defining nodes and node fields

Example: specifying 3-D coordinates of nodes

Following is an example reading 8 nodes numbered from 1 to 8, with a field "coordinates" giving positions at the corners of a unit cube, adapted from cmgui example a/a2:

```
Region: /cube
Shape. Dimension=0
#Fields=1
1) coordinates, coordinate, rectangular cartesian, #Components=3
 x. Value index=1, #Derivatives=0
 y. Value index=2, #Derivatives=0
 z. Value index=3, #Derivatives=0
Node: 1
 0.0 0.0 0.0
Node: 2
 1.0 0.0 0.0
Node: 3
 0.0 1.0 0.0
Node: 4
 1.0 1.0 0.0
Node: 5
 0.0 0.0 1.0
Node: 6
 1.0 0.0 1.0
Node: 7
 0.0 1.0 1.0
Node: 8
 1.0 1.0 1.0
```

Notes:

- The first line indicates that the following objects will be put in a group called “cube”. Ex files must begin with a group declaration.
- The second line says that zero dimensional nodes are to be read until a different shape is specified. You will seldom see this line in any .exnode file because 0-D nodes are the default shape at the start of file read and when any new group is started.
- The next five lines up to the first Node is a node field header which declares finite element fields and indicates what field parameters will be read in with any nodes defined after the header. The first line of the header indicates only one field follows.
- The first line following the #Fields declares a 3-component coordinate-type field called “coordinates” whose values are to be interpreted in a rectangular Cartesian coordinate system. This field defaults to having real values.
- Following the declaration of the field are the details of the components including their names and the parameter values held for each, with the minimum being the value of the field at that node. The above node field component definitions indicate that there are no derivative parameters. The “Value index” is redundant since the index of where values for components “x”, “y” and “z” of the “coordinates” field are held in each node”s parameter vector is calculated assuming they are in order (the correct index is written for interest).
- Finally each of the nodes are listed followed by the number of parameters required from the node field header.

Example: multiple node fields and derivative parameters

A slightly more complex example adds a second field “temperature”:

```
Region: /heated_bar
#Fields=2
1) coordinates, coordinate, rectangular cartesian, #Components=2
 x. Value index=1, #Derivatives=0
 y. Value index=2, #Derivatives=0
2) temperature, field, rectangular cartesian, #Components=1
 1. Value index=3, #Derivatives=1 (d/ds1)
Node: 1
 0.0 0.0
 37.0 0.0
Node: 2
 1.0 0.0
 55.0 0.0
Node: 3
```

```
2.0 0.0
80.2 0.0
```

Notes:

- The coordinates field is now 2-dimensional. Beware it isn"t possible to have a two-component and three-component field of the same name in the same region.
- The temperature field is of CM type field which merely means it has no special meaning (as opposed to the coordinate CM field type hint for the "coordinates"). Although it is not entirely relevant, the coordinate system must still be specified since cmgui performs appropriate transformations whenever it is used as a coordinate field.
- The scalar (single component) temperature field has two parameters for each node, the first being the exact temperature at the node, the second being a nodal derivative. The optional text "(d/ds1)" labels the derivative parameter as the derivative of the temperature with respect to a physical distance in space. It isn"t until the definition of element interpolation that its contribution to the element field is known.

Example: node derivatives and versions in the prolate heart

The following snippet from example a/a3 shows the nodal parameters held at the apex of the prolate heart model:

```
#Fields=2
 1) coordinates, coordinate, prolate spheroidal, focus=0.3525E+02, #Components=3
  lambda. Value index=1,#Derivatives=3 (d/ds1,d/ds2,d2/ds1ds2)
  mu. Value index=5, #Derivatives=0
  theta. Value index=6,#Derivatives=0, #Versions=10
 2) fibres, anatomical, fibre, #Components=3
  fibre angle. Value index=16, #Derivatives=1 (d/ds1)
  imbrication angle. Value index=18, #Derivatives= 0
  sheet angle. Value index=19, #Derivatives=3 (d/ds1,d/ds2,d2/ds1ds2)
Node: 13
  0.984480E+00   0.000000E+00   0.000000E+00   0.000000E+00
  0.000000E+00
  0.253073E+00   0.593412E+00   0.933751E+00   0.127409E+01   0.188932E+01   0.
↪250455E+01   0.373500E+01   0.496546E+01   0.558069E+01   0.619592E+01
 -0.138131E+01  -0.117909E+01
  0.000000E+00
 -0.827443E+00  -0.108884E+00  -0.245620E+00  -0.153172E-01
```

Notes:

- This example uses a prolate spheroidal coordinate system for the coordinate field. This is inherently heart-like in shape allowing fewer parameters to describe the heart, and requires a focus parameter to set is scale and form.
- The "theta" component of the prolate coordinates has 10 versions meaning there are 10 versions of each value and derivative specified. In this case there are no derivatives so only 10 values are read in. 10 versions are used to supply the angles at which each line element heads away from the apex of the heart which is on the axis of the prolate spheroidal coordinate system.
- Node field parameters are always listed in component order, and for each component in the order:

value then derivatives for version 1
value then derivatives for version 2
etc.

- The second field "fibres" declares a field of CM type anatomical with a fibre coordinate system. In elements these fields are interpreted as Euler angles for rotating an ortho-normal coordinate frame initially oriented with element "xi" axes and used to define axes of material anisotropy, such as muscle fibres in tissue.

Example: embedded locations in a mesh

Following is the file “cube_element_xi.exdata” taken from cmgui example a/ar (exnode formats), which defines a node field containing embedded locations within elements:

```
Group name: xi_points
#Fields=1
 1) element_xi, field, element_xi, #Components=1
  1. Value index=1, #Derivatives=0
Node:      1
  E 1 3 0.25 0.25 0.75
Node:      2
  E 1 3 0.25 0.5 0.75
Node:      3
  E 1 3 1 0.25 0.75
Node:      4
  E 1 3 1 1 1
Node:      5
  E 1 3 0 0 0
Node:      6
  F 3 2 0.3 0.6
Node:      7
  L 1 1 0.4
```

Notes:

- The field named “element_xi” uses the special value type confusingly also called element_xi indicating it returns a reference to an element and a location within its coordinate “xi” space. Only 1 component and no derivatives or versions are permitted with this value type.
- Element xi values are written as:

[region path] {element/face/line} number dimension xi-coordinates
Where the region path is optional and element/face/line can be abbreviated to as little as one character, case insensitive. Hence node 1 lists the location in three-dimensional element number 1 where xi=(0.25,0.25,0.75); node 6 lists a location in two-dimensional face element number 3 with face xi=(0.3,0.6).

- Node fields storing embedded locations permit fields in the host element at those locations to be evaluated for the nodes using the special embedded computed field type, for example: “gfx define field embedded_coordinates embedded element_xi element_xi field element_coordinates”.

Special field types

The example a/ar (exnode formats) and a/aq (exelem formats) lists several other special field types including constant (one value for all the nodes it is defined on) and indexed (value indexed by the value of a second integer “index field”), but their use is discouraged and they may be deprecated in time. If such functionality is required, prefer computed fields created with “gfx define field” commands or request indexed functionality on the cmgui tracker.

Defining elements and element fields

Cmgui elements, shapes, faces and identifiers

Elements are objects comprising an n-dimensional (with n>0) coordinate space serving as a material coordinate system charting part or all of a body of interest. The set of elements covering the whole model is referred to as a mesh. We often use the Greek character “xi” to denote the coordinate within each element.

Each element has a shape which describes the form and limits of this coordinate space. Shapes are declared in the EX format are follows:

Shape. Dimension=n SHAPE-DESCRIPTION

Up to three dimensions, the most important shape descriptions are:

```
Shape. Dimension=0                            nodes (no shape)
Shape. Dimension=1 line                       line shape, xi covering [0,1]
Shape. Dimension=2 line*line          square on [0,1]
Shape. Dimension=2 simplex(2)*simplex triangle on [0,1]; xi1+xi2<1
Shape. Dimension=3 line*line*line             cube on [0,1]
Shape. Dimension=3 simplex(2;3)*simplex*simplex
           tetrahedron on [0,1]; xi1+xi2+xi3<1
Shape. Dimension=3 line*simplex(3)*simplex (and other permutations)
           triangle wedge; line on xi1 (etc.)
```

Pairs of dimensions can also be linked into polygon shapes; see example a/element_types.

The shape description works by describing the span of the space along each xi direction for its dimension. The simplest cases are the line shapes: “line*line*line” indicates the outer or tensor product of three line shapes, thus describing a cube. If the shape description is omitted then line shape is assumed for all dimensions. Simplex shapes, used for triangles and tetrahedra, cannot be simply described by an outer product and must be tied to another dimension; in the EX format the tied dimension is written in brackets after the first simplex coordinate, and for 3 or higher dimensional simplices all linked dimensions must be listed as shown for the tetrahedron shape.

The cmgui shape description is extensible to 4 dimensions or higher, for example the following denotes a 4-dimensional shape with a simplex across dimensions 1 and 2, and an unrelated simplex on dimensions 3 and 4:

```
Shape. Dimension=4 simplex(2)*simplex*simplex(4)*simplex
```

Beware that while cmgui can in principle read such high-dimensional elements from exelem files, this has not been tested and few graphics primitives and other features of cmgui will be able to work with such elements.

Elements of a given shape have a set number of faces of dimension one less than their own. A cube element has 6 square faces at xi1=0, xi1=1, xi2=0, xi2=1, xi3=0, xi3=1. Each square element itself has 4 faces, each of line shape. The faces of elements are themselves elements, but they are identified differently from the real “top-level” elements.

The cmgui EX format uses a peculiar naming scheme for elements, consisting of 3 integers: the element number, the face number and the line number, only one of which should ever be non-zero. All elements over which fields are defined or which are not themselves the faces of a higher dimensional element use the first identifier, the element number. All 2-D faces of 3-D elements use the face number and all 1-D faces of 2-D elements (including faces of faces of 3-D elements) use the line number. The naming scheme for faces of 4 or higher dimensional elements is not yet defined. The element identifier “0 0 0” may be used to indicate a NULL face.

Element nodes, scale factors and parameter mappings

Cmgui elements store an array of nodes from which field parameters are extracted for interpolation by basis functions. This local node array can be as long as needed. It may contain repeated references to the same nodes, however it is usually preferable not to do this.

Cmgui elements can also have an array of real-valued scale factors for each basis function (see below).

These two arrays are combined in mapping global node parameters to an array of element field parameters ready to be multiplied by the basis function values to give the value of a field at any “xi” location in the element.

Mappings generally work by taking the field component parameter at index i from the node at local index j and multiplying it by the scale factor at index k for the basis function in-use for that field component. It is also possible to use a unit scale factor by referring to the scale factor at index 0, and to not supply a scale factor set if only unit scale factors are in use.

Global-to-local parameter mappings are at the heart of the complexity in cmgui element field definitions and are described in detail with the examples.

Cmgui element basis functions

Basis functions are defined in cmgui ex format in a very similar manner to element shapes, by outer (tensor) product of the following functions along each xi axis:

```
constant                      constant
l.Lagrange                    linear Lagrange
q.Lagrange                    quadratic Lagrange
c.Lagrange                    cubic Lagrange
c.Hermite                     cubic Hermite
LagrangeHermite               Lagrange at xi=0, Hermite at xi=1
HermiteLagrange               Hermite at xi=0, Lagrange at xi=1
l.simplex                     linear simplex (see below)
q.simplex                     quadratic simplex (see below)
polygon                       piecewise linear around a polygon shape
```

It is not too difficult to extend these functions to higher order and to add other families of interpolation or other basis functions including the serendipity family, Bezier, Fourier etc. We are open to requests for new basis functions on the cmgui tracker.

Lagrange, Hermite, Bezier and many other 1-D basis functions are able to be combined in multiple dimensions by the outer product. This is not the case for simplex, polygon and serendipity families of basis functions which, like element shapes, require linked xi dimensions to be specified.

Some example element bases:

```
l.Lagrange*l.Lagrange*l.Lagrange              trilinear interpolation (8 nodes)
c.Hermite*c.Hermite                           bicubic Hermite (4 nodes x 4 params)
l.simplex(2)*l.simplex                        linear triangle (3 nodes)
q.simplex(2;3)*q.simplex*q.simplex    quadratic tetrahedron (10 nodes)
c.Hermite*l.simplex(3)*l.simplex              cubic Hermite * linear triangle (6␣
↪nodes, 2 parameters per node)
constant*constant*l.Lagrange          constant in xi1 and xi2, linearly varying in␣
↪xi3
```

Most element bases have one basis functions per node which multiplies a single parameter obtained from that node. For instance, a linear Lagrange basis expects 2 nodes each with 1 parameter per field component. A bilinear Lagrange basis interpolates a single parameter from 4 nodes at the corners of a unit square. A 3-D linear-quadratic-cubic Lagrange element basis expects 2*3*4 nodes along the respective xi directions, with 1 basis function and one parameter for each node. A linear triangle has 3 nodes with 1 parameter each; a quadratic triangle has 6 nodes with 1 parameter.

1-D Hermite bases provide 2 basis functions per node, expected to multiple two parameters: (1) the value of the field at the node, and (2) the derivative of that field value with respect to the xi coordinate. If this derivative is common across an element boundary then the field is C¬1 continuous there. Outer products of 1-D Hermite basis functions double the number of parameters per node for each Hermite term. A bicubic Hermite basis expect 4 nodes with 4 basis functions per node, multiplying 4 nodal parameters: (1) value of the field, (2) the derivative of the field with respect to the first xi direction, (3) the derivative with respect to the second xi direction and (4) the double derivative of the field with respect to both directions referred to as the cross derivative. Tri-cubic Hermite bases have 8 basis functions per node, one multiplying the value, 3 for first derivatives, 3 for second (cross) derivatives and a final function multiplying a triple cross derivative parameter.

The ex format requires nodes contributing parameters for multiplication by a basis to be in a very particular order: changing fastest in xi1, then xi2, then xi3, and so on. Note this is not necessarily the order nodes are stored in the element node array, just the order in which those nodes are referenced in the parameter map. In most example files the order of the nodes in the element node list will also follow this pattern.

Cmgui example a/element_types provides a large number of sample elements using complex combinations of basis functions on supported 3-D element shapes.

Example: tri-linear interpolation on a cube

Following is an example of a coordinate field defined over a unit cube, adapted from example a/a2, with faces and scale factors removed to cut the example down to minimum size, and assuming the cube node file from earlier has already been loaded:

```
Region: /cube
Shape.  Dimension=3  line*line*line
#Scale factor sets=0
#Nodes=8
#Fields=1
 1) coordinates, coordinate, rectangular cartesian, #Components=3
   x.  l.Lagrange*l.Lagrange*l.Lagrange, no modify, standard node based.
   #Nodes= 8
    1.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    2.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    3.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    4.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    5.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    6.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    7.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    8.  #Values=1
     Value indices:     1
     Scale factor indices:   0
   y.  l.Lagrange*l.Lagrange*l.Lagrange, no modify, standard node based.
   #Nodes= 8
    1.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    2.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    3.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    4.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    5.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    6.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    7.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    8.  #Values=1
     Value indices:     1
     Scale factor indices:   0
   z.  l.Lagrange*l.Lagrange*l.Lagrange, no modify, standard node based.
   #Nodes= 8
    1.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    2.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    3.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    4.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    5.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    6.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    7.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    8.  #Values=1
     Value indices:     1
     Scale factor indices:   0
 Element:     1 0 0
   Nodes:
     1     2     3     4     5     6     7     8
```

Notes:

- "Shape. Dimension=3 line*line*line" declares that following this point three dimensional cube-shaped elements are being defined. "line*line*line" could have been omitted as line shapes are the default.
- "#Scale factor sets=0" indicates no scale factors are to be read in with the elements that follow. Scale factors are usually only needed for Hermite basis functions when nodal derivative parameters are maintained with respect to a physical distance and the scale factors convert the derivative to be with respect to the element xi coordinate. Avoid scale factors when not needed.
- The 4th line "#Nodes=8" says that 8 nodes will be listed with all elements defined under this header.
- Under the declaration of the "coordinates" field (which is identical to its declaration for the nodes) are the details on how the field is evaluated for each field component. Each field component is always described separately: each may use different basis functions and parameter mappings. In this example the components "x", "y" and "z" all use the same tri-linear basis functions and use an identical parameter mapping except that parameters are automatically taken from the corresponding field component at the node.
- "no modify" in the element field component definition is an instruction to do no extra value manipulations as part of the interpolation. Other modify instructions resolve the ambiguity about which direction one interpolates angles in polar coordinates. The possible options are "increasing in xi1", "decreasing in xi1", "non-increasing in xi1", "non-decreasing in xi1" or "closest in xi1". For use, see the prolate heart example later.
- "standard node based" indicates that each element field parameter is obtained by multiplying one parameter extracted from a node by one scale factor. An alternative called "general node based" evaluates each element field parameter as the dot product of multiple node field parameters and scale factors; it is currently unimplemented for I/O but is designed to handle the problems like meshing the apex of the heart with Hermite basis functions without requiring multiple versions. A final option "grid based" is described in a later example.
- Following the above text are several lines describing in detail how all the element field parameters are

evaluated prior to multiplication by the basis functions. Being a tri-linear Lagrange basis, 8 nodes must be accounted for:

```
#Nodes= 8
```

Following are 8 sets of three lines each indicating the index of the node in the element"s node array from which parameters are extracted (in this case the uncomplicated sequence 1,2,3,4,5,6,7,8), the number of values to be extracted (1), the index of the each parameter value in the list of parameters for that field component at that node and the index of the scale factor multiplying it from the scale factor array for that basis, or zero to indicate a unit scale factor:

```
1.  #Values=1
 Value indices:     1
 Scale factor indices:   0
```

As mentioned earlier, the nodes listed in the mapping section must always follow a set order, increasing in xi1 fastest, then xi2, then xi3, etc. to match the order of the basis functions procedurally generated from the basis description.

- At the end of the example is the definition of element "0 0 1" which lists the nodes 1 to 8.

Defining faces and lines

The above example is not ready to be fully visualised in cmgui because it contains no definitions of element faces and lines. Cmgui requires these to be formally defined because it visualises element surfaces by evaluating fields on face elements, and element edges via line elements. The command "gfx define faces ..." can create the faces and lines after loading such a file. Alternatively they can be defined and referenced within the ex file as in the following:

```
Region: /cube
 Shape.  Dimension=1
 Element: 0 0     1
 Element: 0 0     2
 Element: 0 0     3
 Element: 0 0     4
 Element: 0 0     5
 Element: 0 0     6
 Element: 0 0     7
 Element: 0 0     8
 Element: 0 0     9
 Element: 0 0    10
 Element: 0 0    11
 Element: 0 0    12
Shape.  Dimension=2
 Element: 0     1 0
   Faces:
   0 0     3
   0 0     7
   0 0     2
   0 0    10
 Element: 0     2 0
   Faces:
   0 0     5
   0 0     8
   0 0     4
   0 0    11
 Element: 0     3 0
   Faces:
   0 0     1
   0 0     9
   0 0     3
   0 0     5
 Element: 0     4 0
   Faces:
   0 0     6
   0 0    12
   0 0     7
   0 0     8
 Element: 0     5 0
   Faces:
   0 0     2
   0 0     4
   0 0     1
   0 0     6
 Element: 0     6 0
   Faces:
   0 0    10
   0 0    11
   0 0     9
   0 0    12
Shape.  Dimension=3
 Element:     1 0 0
   Faces:
   0     1 0
   0     2 0
   0     3 0
   0     4 0
   0     5 0
   0     6 0
   Nodes:
     1     2     3     4     5     6     7     8
```

Likewise scale factors can be read in as listed in the cube.exelem file from cmgui example a/a2, however with Lagrange basis functions the scale factors are all unit valued, so this is rather needless.

Example: collapsed square element

The following tricky example collapses a square element to a triangle by using the third local node twice:

```
Region: /collapse
Shape. Dimension=0
#Fields=1
1) coordinates, coordinate, rectangular cartesian, #Components=2
 x. Value index=1, #Derivatives=0
 y. Value index=2, #Derivatives=0
Node: 1
 0.0 0.0
Node: 2
 1.0 0.0
Node: 3
 0.5 1.0
Shape.  Dimension=1  line
 Element: 0 0 1
 Element: 0 0 2
 Element: 0 0 3
Shape.  Dimension=2  line*line
#Scale factor sets=0
#Nodes=3
#Fields=1
 1) coordinates, coordinate, rectangular cartesian, #Components=2
   x.  l.Lagrange*l.Lagrange, no modify, standard node based.
   #Nodes= 4
    1.  #Values=1
```

```
     Value indices:     1
     Scale factor indices:   0
    2.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    3.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    3.  #Values=1
     Value indices:     1
     Scale factor indices:   0
   y.  l.Lagrange*l.Lagrange, no modify, standard node based.
   #Nodes= 4
    1.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    2.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    3.  #Values=1
     Value indices:     1
     Scale factor indices:   0
    3.  #Values=1
     Value indices:     1
     Scale factor indices:   0
Element:      1 0 0
   Faces:
   0 0 1
   0 0 2
   0 0 3
   0 0 0
   Nodes:
     1     2     3
```

Notes:

- Element 1 0 0 has a node array with only 3 nodes in it, but the third and fourth parameter mappings both refer to the node at index 3 in the element node list.
- Note that face on the collapsed side of the element is undefined, as indicated by special face identifier 0 0 0.
- It is also possible to obtain an equivalent result by physically storing 4 nodes in the element but repeating node 3 in that array.

Simplex elements: triangles and tetrahedra

The cmgui example a/testing/simplex defines two fields on a triangle element, one using a 6-node quadratic simplex, the other a 3-node linear simplex basis. The element in that example stores 6 nodes in its node array, only 3 of which are used for the linear basis, but all 6 contribute parameters to the quadratic interpolation.

Example a/element_types has both linear and quadratic tetrahedra.

Example: multiple fields, bases and scale factor sets in the prolate heart

At the more complex end of the scale is this excerpt from the prolate heart model from cmgui example a/a3. It defines two element fields using different basis functions for each field component. It was exported from cm which always uses a full complement of scale factors i.e. one per basis function.

```
Shape.  Dimension=3
 #Scale factor sets= 4
   c.Hermite*c.Hermite*l.Lagrange, #Scale factors=32
```

```
  l.Lagrange*l.Lagrange*l.Lagrange, #Scale factors=8
  l.Lagrange*l.Lagrange*c.Hermite, #Scale factors=16
  l.Lagrange*c.Hermite*c.Hermite, #Scale factors=32
#Nodes=           8
#Fields=2
1) coordinates, coordinate, prolate spheroidal, focus=  0.3525E+02, #Components=3
  lambda.  c.Hermite*c.Hermite*l.Lagrange, no modify, standard node based.
  #Nodes= 8
   1.  #Values=4
    Value indices:     1   2   3   4
    Scale factor indices:   1   2   3   4
   2.  #Values=4
    Value indices:     1   2   3   4
    Scale factor indices:   5   6   7   8
   3.  #Values=4
    Value indices:     1   2   3   4
    Scale factor indices:   9  10  11  12
   4.  #Values=4
    Value indices:     1   2   3   4
    Scale factor indices:  13  14  15  16
   5.  #Values=4
    Value indices:     1   2   3   4
    Scale factor indices:  17  18  19  20
   6.  #Values=4
    Value indices:     1   2   3   4
    Scale factor indices:  21  22  23  24
   7.  #Values=4
    Value indices:     1   2   3   4
    Scale factor indices:  25  26  27  28
   8.  #Values=4
    Value indices:     1   2   3   4
    Scale factor indices:  29  30  31  32
  mu.  l.Lagrange*l.Lagrange*l.Lagrange, no modify, standard node based.
  #Nodes= 8
   1.  #Values=1
    Value indices:     1
    Scale factor indices:  33
   2.  #Values=1
    Value indices:     1
    Scale factor indices:  34
   3.  #Values=1
    Value indices:     1
    Scale factor indices:  35
   4.  #Values=1
    Value indices:     1
    Scale factor indices:  36
   5.  #Values=1
    Value indices:     1
    Scale factor indices:  37
   6.  #Values=1
    Value indices:     1
    Scale factor indices:  38
   7.  #Values=1
    Value indices:     1
    Scale factor indices:  39
   8.  #Values=1
    Value indices:     1
    Scale factor indices:  40
  theta.  l.Lagrange*l.Lagrange*l.Lagrange, decreasing in xi1, standard node based.
  #Nodes= 8
   1.  #Values=1
    Value indices:     1
```

```
     Scale factor indices:  33
    2.  #Values=1
     Value indices:     1
     Scale factor indices:  34
    3.  #Values=1
     Value indices:     1
     Scale factor indices:  35
    4.  #Values=1
     Value indices:     1
     Scale factor indices:  36
    5.  #Values=1
     Value indices:     1
     Scale factor indices:  37
    6.  #Values=1
     Value indices:     1
     Scale factor indices:  38
    7.  #Values=1
     Value indices:     1
     Scale factor indices:  39
    8.  #Values=1
     Value indices:     1
     Scale factor indices:  40
 2) fibres, anatomical, fibre, #Components=3
   fibre angle.  l.Lagrange*l.Lagrange*c.Hermite, no modify, standard node based.
   #Nodes= 8
    1.  #Values=2
     Value indices:     1   2
     Scale factor indices:  41  42
    2.  #Values=2
     Value indices:     1   2
     Scale factor indices:  43  44
    3.  #Values=2
     Value indices:     1   2
     Scale factor indices:  45  46
    4.  #Values=2
     Value indices:     1   2
     Scale factor indices:  47  48
    5.  #Values=2
     Value indices:     1   2
     Scale factor indices:  49  50
    6.  #Values=2
     Value indices:     1   2
     Scale factor indices:  51  52
    7.  #Values=2
     Value indices:     1   2
     Scale factor indices:  53  54
    8.  #Values=2
     Value indices:     1   2
     Scale factor indices:  55  56
   imbrication angle.  l.Lagrange*l.Lagrange*l.Lagrange, no modify, standard node␣
↪based.
   #Nodes= 8
    1.  #Values=1
     Value indices:     1
     Scale factor indices:  33
    2.  #Values=1
     Value indices:     1
     Scale factor indices:  34
    3.  #Values=1
     Value indices:     1
     Scale factor indices:  35
    4.  #Values=1
     Value indices:     1
     Scale factor indices:  36
    5.  #Values=1
     Value indices:     1
     Scale factor indices:  37
    6.  #Values=1
     Value indices:     1
     Scale factor indices:  38
    7.  #Values=1
     Value indices:     1
     Scale factor indices:  39
    8.  #Values=1
     Value indices:     1
     Scale factor indices:  40
   sheet angle.  l.Lagrange*c.Hermite*c.Hermite, no modify, standard node based.
   #Nodes= 8
    1.  #Values=4
     Value indices:     1   2   3   4
     Scale factor indices:  57  58  59  60
    2.  #Values=4
     Value indices:     1   2   3   4
     Scale factor indices:  61  62  63  64
    3.  #Values=4
     Value indices:     1   2   3   4
     Scale factor indices:  65  66  67  68
    4.  #Values=4
     Value indices:     1   2   3   4
     Scale factor indices:  69  70  71  72
    5.  #Values=4
     Value indices:     1   2   3   4
     Scale factor indices:  73  74  75  76
    6.  #Values=4
     Value indices:     1   2   3   4
     Scale factor indices:  77  78  79  80
    7.  #Values=4
     Value indices:     1   2   3   4
     Scale factor indices:  81  82  83  84
    8.  #Values=4
     Value indices:     1   2   3   4
     Scale factor indices:  85  86  87  88
 Element:            1 0 0
   Faces:
   0     1 0
   0     2 0
   0     3 0
   0     4 0
   0     5 0
   0     6 0
   Nodes:
         19           82           14           83            5           52       ␣
 ↪     1           53
   Scale factors:
     0.1000000000000000E+01   0.2531332864778986E+02   0.3202170161207646E+02   0.
↪8105758567679540E+03   0.1000000000000000E+01
   0.2540127674788437E+02   0.3851739595427941E+02   0.9783910342424932E+03   0.
↪1000000000000000E+01   0.2665607536220107E+02
   0.2913687357203342E+02   0.7766746977550476E+03   0.1000000000000000E+01   0.
↪2797776438705370E+02   0.3675988075068424E+02
   0.1028459282538834E+04   0.1000000000000000E+01   0.3107367883817446E+02   0.
↪3665266951220884E+02   0.1138933280984126E+04
   0.1000000000000000E+01   0.3053066581298630E+02   0.4220277992007600E+02   0.
↪1288478970118849E+04   0.1000000000000000E+01
   0.3612724280632425E+02   0.3339669014010959E+02   0.1206530333619314E+04   0.
↪1000000000000000E+01   0.3620256762563091E+02
```

```
   0.3810870609422361E+02   0.1379633009501423E+04
     0.1000000000000000E+01   0.1000000000000000E+01   0.1000000000000000E+01   0.
↪1000000000000000E+01   0.1000000000000000E+01
   0.1000000000000000E+01   0.1000000000000000E+01   0.1000000000000000E+01
     0.1000000000000000E+01   0.8802929891392392E+01   0.1000000000000000E+01   0.
↪7673250860396258E+01   0.1000000000000000E+01
   0.1368084332227282E+02   0.1000000000000000E+01   0.1181772996260416E+02   0.
↪1000000000000000E+01   0.8802929891392392E+01
   0.1000000000000000E+01   0.7673250860396258E+01   0.1000000000000000E+01   0.
↪1368084332227282E+02   0.1000000000000000E+01
   0.1181772996260416E+02
     0.1000000000000000E+01   0.3202170161207646E+02   0.8802929891392392E+01   0.
↪2818847942941958E+03   0.1000000000000000E+01
   0.3851739595427941E+02   0.7673250860396258E+01   0.2955536416463978E+03   0.
↪1000000000000000E+01   0.2913687357203342E+02
   0.1368084332227282E+02   0.3986170022398609E+03   0.1000000000000000E+01   0.
↪3675988075068424E+02   0.1181772996260416E+02
   0.4344183441691171E+03   0.1000000000000000E+01   0.3665266951220884E+02   0.
↪8802929891392392E+01   0.3226508800483498E+03
   0.1000000000000000E+01   0.4220277992007600E+02   0.7673250860396258E+01   0.
↪3238325173328371E+03   0.1000000000000000E+01
   0.3339669014010959E+02   0.1368084332227282E+02   0.4568948852893329E+03   0.
↪1000000000000000E+01   0.3810870609422361E+02
   0.1181772996260416E+02   0.4503583978457821E+03
```

Notes:

- It can be seen that for each Hermite term in the basis function there are twice as many parameter values for each node.
- The "Value indices" are indices into the array of parameters held for the field component at the node, starting at 1 for the value. It is not possible to refer to parameters by names such as "value", "d/ds1" or by version number.
- The "decreasing in xi1" option specified for the theta component of the coordinates field specifies that as xi increases, the angle of theta decreases. This needs to be stated since it is equally possible to interpolate this angle in the opposite direction around the circle.
- All scale factors are listed in a single block; in this case there are 32+8+16+32=88 scale factors, listed in the order of the scale factor set declaration at the top of the file excerpt. Scale factor indices are absolute locations in this array, but they are considered invalid if referring to parts of the array not containing scale factors for the basis used in the field component.
- Note how all the scale factors for the tri-linear Lagrange basis are equal to 1.0.

Example: per-element constant and grid-based element fields

Cmgui and its EX files also support storage of regular grids of real or integer values across elements. The grid is assumed regular across N divisions on lines, N*M divisions on squares and N*M*P divisions on cubes.

Per-element constants are a special case using constant bases together with 0 grid divisions. These have only been supported since Cmgui 2.7 (April 2010). See this extract from cmgui example a/element_constants:

```
1) temperature, field, rectangular cartesian, #Components=1
 value. constant*constant*constant, no modify, grid based.
 #xi1=0, #xi2=0, #xi3=0
Element: 1 0 0
  Values :
  48.0
```

Linear Lagrange interpolation is used when there are 1 or more element divisions. Following is an excerpt from cmgui example a/aq "element formats":

```
Group name: block
Shape.  Dimension=3
#Scale factor sets=0
#Nodes=0
#Fields=2
1) material_type, field, integer, #Components=1
 number. l.Lagrange*l.Lagrange*l.Lagrange, no modify, grid based.
 #xi1=2, #xi2=3, #xi3=2
2) potential, field, real, #Components=1
 value. l.Lagrange*l.Lagrange*l.Lagrange, no modify, grid based.
 #xi1=2, #xi2=3, #xi3=2
Element: 1 0 0
  Values:
  1 1 3
  1 1 3
  1 2 3
  1 2 2
  1 1 3
  1 1 3
  1 2 3
  2 2 2
  1 3 3
  1 3 3
  2 2 3
  2 2 2
  13.5 12.2 10.1
  14.5 12.2 10.1
  15.5 12.2 10.1
  16.5 12.2 10.1
  12.0 11.0 10.0
  13.0 11.0 10.0
  14.0 11.0 10.0
  15.0 11.0 10.0
  10.5 10.7 9.9
  11.5 10.7 9.9
  12.5 10.7 9.9
  13.5 10.7 9.9
```

Notes:

- "#xi1=2, #xi2=3, #xi3=2" actually refers to the number of divisions between grid points, so 3*4*3=36 values are read in, and represent values at the corners of the grid "cells". If there are 2 divisions along an xi direction, values are held for xi=0.0, xi=0.5 and xi=1.0. Under each element, values are listed in order of location changing fastest along xi1, then along xi2, then along xi3.
- Only constant (for number-in-xi = 0) or linear Lagrange bases are supported. The basis is irrelevant for integer-valued grids which choose the "nearest value", so halfway between integer value 1 and 3 the field value jumps directly from 1 to 3.
- Grid point values along boundaries of adjacent elements must be repeated in each element.

Further information

Further information about cmgui and its data formats can be found on the following web-site:

http://www.cmiss.org/cmgui

We also invite questions, bug reports and feature requests under the tracker at:

https://tracker.physiomeproject.org

Note: be sure to search and post under the "cmgui" project!

There are a number of examples which you can use to familiarise yourself with the functions and capabilities of cmgui.

Todo

Add some useful command examples to command window doc - eg. “saving” graphics

CHAPTER 5

OpenCMISS user documentation

OpenCMISS concepts

OpenCMISS has the following top level objects.

- *Basis Functions*
- *Coordinate Systems*
- *Regions*
- *Problems*
- Computational environments
- Base system(Diagnostics, I/O etc.)

Coordinate Systems

Coordinate system is a system for assigning an n-tuple of numbers or scalars to each point in an n-dimensional space. It anchors the regions within the real world. Coordinate system can have different types such as:

- Rectangular cartesian
- Cylindrical polar
- Spherical polar
- Prolate spheroidal
- Oblate spheroidal

There is a global (world) coordinate system aligned with 3D rectangular cartesian space.

Coordinate system has the following attributes:

- User number
- Finished tag
- Type
- Number of dimensions

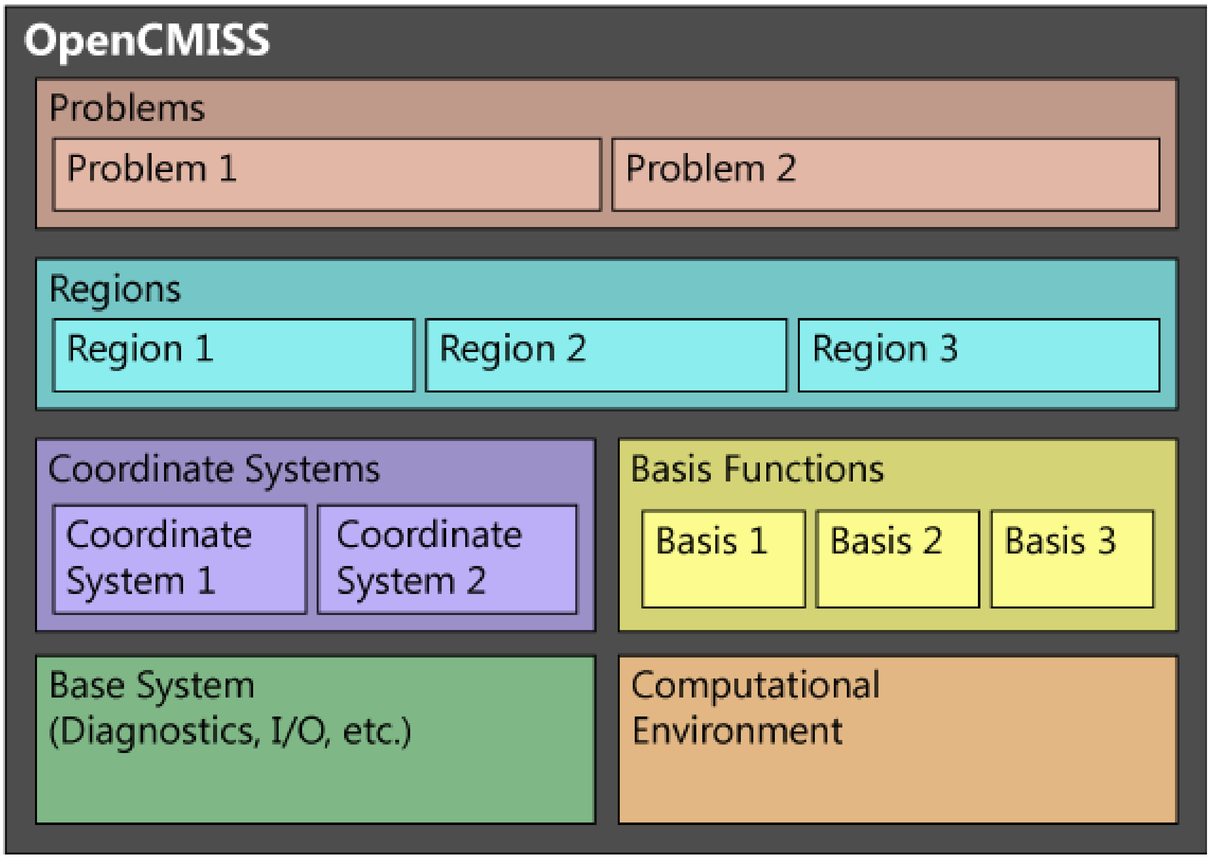


Fig. 5.1: **OpenCMISS top-level structure**

- Focus (for prolate-spheriodal system only)
- Origins
- Orientation

Basis Functions

Basis functions are the key item to specify the field approximation/interpolation and the linking of nodes and elements to form a mesh. Currently, it has two types: Lagrange-Hermite tensor product and Simplex. Lagrange-Hermite tensor product can be further divided into linear to cubic lagrange, cubic and quadratic hermite. It can be arbitrarily collapsed (two or more nodes in the same location) in any one direction or in any two directions to give a degenerate basis. Simplex basic functions could contain line, triangular and tetrahedral elements. It could be linear, quadratic or cubic. Arbitrary Gaussian quadrature can integrate from 1st to 5th order (3rd order for lines at the moment). Can only have the same order in each direction at the moment. Specifying a basis function automatically generates all necessary line and face basis functions as sub-bases of the basis function.

Basis function has the following attributes:

- User number
- Global number
- Family number
- Finished tag
- Type
- Is Hermite

- Number of XI
- Number of XI coordinates
- ...

Regions

Regions are one of the primary objects in openCMISS. Regions are hierarchical in nature in that a region can have one parent region and a number of daughter sub-regions. Daughter regions are related in space to parent regions by an origin and an orientation of the regions coordinate system. Daughter regions may only have the same or fewer dimensions as the parent region. There is a global (world) region (number 0) that has the global (world) coordinate system.

region_definition.JPG

Region has the following attributes:

- User number
- Finished tag
- Label
- Number of sub(daughter) regions
- Coordinate system pointer
- Nodes
- Meshes
- Fields
- Equations
- Parent region pointer
- Daughter regions pointers

region_structure.JPG

Nodes

There are three places storing nodal information. Nodes associated with region defines the nodes identification and the nodes geometric (initial) position.

Node has the following attributes:

- User number
- Global number
- Label
- Initial Position

Meshes

Meshes are topological constructs within a region which fields use to define themselves. Meshes are made up of a number of mesh components. All mesh components in a mesh must "conform", that is have the same number of elements, Xi directions etc.

Mesh has the following attributes:

- User number

- Global number
- Finished tag
- Region pointer
- Number of dimensions
- Number of components
- Embedded flag
- Embedding mesh pointer
- Embedded meshes pointers
- Number of elements
- Number of faces
- Number of lines
- Mesh topology pointers
- Decomposition pointers

mesh_structure.JPG

Mesh Topology

Mesh components (Topology) are made up from nodes, elements and basis functions. A new mesh component is required for each different form of interpolation e.g., one mesh component is bilinear Lagrange and another is biquadratic Lagrange. meshTopology_definition.JPG

Mesh topology has the following attributes:

- Mesh component number
- Mesh pointer
- Nodes pointers
- Element pointers
- DOFs pointers

meshTopology_structure.JPG

Decompositions

Mesh decomposition (partitioning) is used to split a computationally expensive mesh into smaller subdomains (parts) for parallel computing.

Decomposition has the following attributes:

- User number
- Global number
- Finished tag
- Mesh pointer
- Mesh component number
- Decomposition type
- Number of domains
- Number of edge cut

- Element domain numbers
- Decomposition topology pointer
- Domains pointers(list of domain which has the same size as the number of components in the mesh)

meshDecomposition_structure.JPG

Domain

Each domain stores domain information for relevant mesh component.

The domain object contains the following attributes:

- Decomposition pointer
- Mesh pointer
- Mesh component number
- Region pointer
- Number of dimensions
- Node domain(The domain number that the np'th global node is in for the domain decomposition. Note: the domain numbers start at 0 and go up to the NUMBER_OF_DOMAINS-1)
- Domain mappings(for each mapped object e.g. nodes, elements, etc)
- Domain topology pointer(elements, nodes, DOFs)

meshDecompositionDomain_structure.JPG

Domain Mappings

Stores information for each mapped object e.g. nodes, elements, etc.

The domain mapping contains the following attributes:

- Number of local
- Total number of local
- Numbers of domain local
- Number of global
- Number of domains
- Number of internal
- Internal list
- Number of boundary
- Boundary list
- Number of ghost
- Ghost list
- Local to global map
- Global to local map
- Number of adjacent domains
- Pointer to list of adjacent domains by domain number
- List of adjacent domains

meshDecompositionDomainMapping_structure.JPG

Fields

Fields are the central object for storing information and framing the problem. Fields have a number of field variables i.e., u, du/dn, du/dt, d2u/dt2. Each field variable has a number of components. A field is defined on a decomposed mesh. Each field variable component is defined on a decomposed mesh component.

Field can contains the following attributes:

- User number
- Global number
- Finished tag
- Region pointer
- Type(Geometric, Fibre, General, Material, Source)
- Dependent type(Independent, Dependent)
- Dimension
- Decomposition pointer
- Number of variables
- Variables
- Scalings sets
- Mappings(DOF->Field parameters)
- Parameter sets(distributed vectors)
- Geometric field pointer
- Geomatric field parameters
- Create values cache

field_structure.JPG

Field variable

Field variable stores variables for the field such as dependent variables. For example, in the Laplace's equation(FEM), it stores two variables: u and du/dn. Each field variable has a number of components.

Field variable has the following attributes:

- Variable number
- Variable type
- Field pointer
- Region pointer
- Max number of interpolation parameters
- Number of DOFs
- Total number of DOFs
- Global DOF List
- Domain mapping pointer
- Number of components
- Components

Field Variable Component

Field Variable Component has the following attributes:

- Component number
- Variable pointer
- Field pointer
- Interpolation type
- Mesh component number
- Scaling index
- Domain pointer
- Max number of interpolation parameters
- Mappings(Field paramters->DOF)

Parameter set

Parameter set stores values for each field variable component.

field_parameter_set_definition.JPG

Parameter set has the following Attributes:

- Set index
- Set type
- Parameters pointer

Equation Sets

Equations sets are aimed to have multiple classes, e.g. Elasticity, Fluid mechanics, Electromagnetics, General field problems, Fitting, Optimisation. Different equations are within each class, e.g. Bidomain, Navier-stokes etc. Each equation can use different solution techniques, e.g. FEM, BEM, FD, GFEM. The equation set is associated with a region and is built using the fields defined on the region.

The numerical methods are used which will result in a discretised matrix-vector form of the governing equations. openCMISS is designed to generate equations sets with a number of "equations" matrices.

e.g, damped mass spring system Mu + Cu + Ku = f will be represented as:

fieldEquationsets-matrix.JPG

Equations Set has the following attributes:

- User number
- Global number
- Finished tag
- Region pointer
- Class identifier
- Type identifier
- Sub type identifier
- Linearity type(?)
- Time dependence type(?)

- Solution method
- Geometry (fibre?) field pointer
- Materials field pointer
- Source field pointer
- Dependent field pointer
- Analytic info pointer(Analytic info stored in dependent field currently)
- Fixed conditions
- Equations pointer

fieldEquationssets-structure.JPG

Equations

Equation holds the matrices and mapping information.

The Field variable to matrix mappings maps each field variable onto the equations matrices or RHS vector.

e.g. Laplace(FEM): 2 variables, 1 component fieldEquationsetsEquations-mappingFEM.JPG

e.g. Laplace(BEM): 2 variables, 1 component fieldEquationsetsEquations-mappingBEM.JPG

e.g. Heat equation(explicit time/FEM space): 2 variables, 1 component fieldEquationsetsEquations-mappingHeat.JPG

TODO matrix distribution

Equations has the following attributes:

- Equation set pointer
- Finished tag
- Output type
- Sparsity type
- Interpolation pointer
- Linear equation data pointer
- Nonlinear equation data pointer
- Time(non-static) data pointer
- Equations mapping pointer
- Equations Matrices

Problems

A problem has a number of solutions (each with their solver) inside a problem control loop. Problem is associated with region via solution which maps to equations sets and hence links to region. Multiple problems can be in the same region, or multiple regions can work to solve one problem.

Problem has the following attributes:

- User number
- Global number
- Finished tag
- Class

- Type
- Subtype
- Control pointer
- Number of solutions
- Solutions pointer

problem_structure.JPG

Solutions

Solution has the following attributes:

- Solution number
- Finished tag
- Linear solution data pointer
- Nonlinear solution data pointer
- Time (non-static) solution data pointer
- Equations set to add (the next equations set to add)
- Index of added equations set(the last successfully added equations set)
- Solution mapping(which contains equations sets)
- Solver pointer

Solvers

Solver has the following attributes:

- Solution pointer
- Finished tag
- Solve type
- Output type
- Sparsity type
- Linear solver pointer
- Non-linear solver pointer
- Time integrationn solver pointer
- Eigenproblem solver pointer
- Solver matrices

problemSolutionSolver_structure.JPG

Control

Control has the following attributes:

- Problem pointer
- Finished tag
- Control type

Getting started with OpenCMISS

Tutorial one.

Using OpenCMISS from Python

Section author: Adam Reeve

In this tutorial we will walk through how to solve Laplace's equation on a 2D geometry using the Python bindings to OpenCMISS.

Getting Started

The Python code given here follows the Python Laplace example found in the OpenCMISS examples repository. You can run the code interactively by entering it directly into the Python interpreter, which can be started by running `python`. Alternatively you can enter the code into a text file with a .py extension and run it using the python command, eg:

```
python LaplaceExample.py
```

In order to use OpenCMISS we have to first import the `CMISS` module from the `opencmiss` package:

```
from opencmiss import CMISS
```

Assuming OpenCMISS has been correctly built with the Python bindings by following the instructions in the programmer documentation, we can now access all the OpenCMISS functions, classes and constants under the `CMISS` namespace.

Create a Coordinate System

First we construct a *coordinate system* that will be used to describe the geometry in our problem. The 2D geometry will exist in a 3D space, so we need a 3D coordinate system.

When creating an object in OpenCMISS there are at least three steps. First we initialise the object:

```
coordinateSystem = CMISS.CoordinateSystem()
```

This creates a thin wrapper that just points to the actual coordinate system used internally by OpenCMISS, and initially the pointer is null. Trying to use the coordinate system now would raise an exception. To actually construct a coordinate system we call the `CreateStart` method and pass it a user number. The user number must be unique between objects of the same type and can be used to identify the coordinate system later. Most OpenCMISS classes require a user number when creating them, and many also require additional parameters to the `CreateStart` method:

```
coordinateSystemUserNumber = 1
coordinateSystem.CreateStart(coordinateSystemUserNumber)
```

We can now optionally set any properties on the object. We will set the dimension so that the coordinate system is 3D:

```
coordinateSystem.dimension = 3
```

And finally, we finish creating the coordinate system:

```
coordinateSystem.CreateFinish()
```

Create a Region

Next we create a *region* that our fields will be defined on and tell it to use the 3D coordinate system we created previously. The `CreateStart` method for a region requires another region as a parameter. We use the world region that is created by default so that our region is a subregion of the world region. We can also give the region a label:

```
regionUserNumber = 1
region = CMISS.Region()
region.CreateStart(regionUserNumber, CMISS.WorldRegion)
region.label = "LaplaceRegion"
region.coordinateSystem = coordinateSystem
region.CreateFinish()
```

Create a Basis

The finite element description of our fields requires a *basis function* to interpolate field values over elements, so we create a 2D basis with linear Lagrange interpolation in both xi directions:

```
basisUserNumber = 1
basis = CMISS.Basis()
basis.CreateStart(basisUserNumber)
basis.type = CMISS.BasisTypes.LAGRANGE_HERMITE_TP
basis.numberOfXi = 2
basis.interpolationXi = [
    CMISS.BasisInterpolationSpecifications.LINEAR_LAGRANGE,
    CMISS.BasisInterpolationSpecifications.LINEAR_LAGRANGE,
]
basis.quadratureNumberOfGaussXi = [2, 2]
basis.CreateFinish()
```

When we set the basis type we select a value from the `BasisTypes` enum. To get a list of possible basis types and interpolation types you can use the `help` function in the python interpreter:

```
help(CMISS.BasisTypes)
help(CMISS.BasisInterpolationSpecifications)
```

Similarly, you can see what methods and properties are available for the various CMISS classes and get help information for these:

```
help(CMISS.Basis)
help(CMISS.Basis.CreateStart)
help(CMISS.Basis.interpolationXi)
```

Create a Decomposed Mesh

In order to define a simple 2D geometry for our problem we can use one of OpenCMISS's inbuilt generated meshes. We will create a 2D, rectangular mesh with 10 elements in both the x and y directions and tell it to use the basis we created previously:

```
generatedMeshUserNumber = 1
numberGlobalXElements = 10
numberGlobalYElements = 10
width = 1.0
length = 1.0

generatedMesh = CMISS.GeneratedMesh()
generatedMesh.CreateStart(generatedMeshUserNumber, region)
generatedMesh.type = CMISS.GeneratedMeshTypes.REGULAR
generatedMesh.basis = [basis]
generatedMesh.extent = [width, length, 0.0]
generatedMesh.numberOfElements = [
    numberGlobalXElements,
    numberGlobalYElements]
```

When setting the `basis` property, we assign a list of bases as we might want to construct a mesh with multiple components using different interpolation schemes.

The generated mesh is not itself a mesh, but is used to create a mesh. We construct the *mesh* when we call the `CreateFinish` method of the generated mesh and pass in the mesh to generate:

```
meshUserNumber = 1
mesh = CMISS.Mesh()
generatedMesh.CreateFinish(meshUserNumber, mesh)
```

Here we have initialised a mesh but not called `CreateStart` or `CreateFinish`, instead the mesh creation is done when finishing the creation of the generated mesh.

Because OpenCMISS can solve problems on multiple computational nodes, it must work with a *decomposed mesh*. We now decompose our mesh by getting the number of computational nodes and creating a decomposition with that number of domains:

```
decompositionUserNumber = 1
decomposition = CMISS.Decomposition()
decomposition.CreateStart(decompositionUserNumber, mesh)
decomposition.type = CMISS.DecompositionTypes.CALCULATED
decomposition.numberOfDomains = CMISS.ComputationalNumberOfNodesGet()
decomposition.CreateFinish()
```

Note that even when we have just one computational node, OpenCMISS still needs to work with a decomposed mesh, which will have one domain.

Defining Geometry

Now that we have a decomposed mesh, we can begin defining the *fields* we need on it. First we will create a geometric field to define our problem geometry:

```
geometricFieldUserNumber = 1
geometricField = CMISS.Field()
geometricField.CreateStart(geometricFieldUserNumber, region)
geometricField.meshDecomposition = decomposition
geometricField.ComponentMeshComponentSet(CMISS.FieldVariableTypes.U, 1, 1)
geometricField.ComponentMeshComponentSet(CMISS.FieldVariableTypes.U, 2, 1)
geometricField.ComponentMeshComponentSet(CMISS.FieldVariableTypes.U, 3, 1)
geometricField.CreateFinish()
```

The call to the `ComponentMeshComponentSet` method is not actually required here as all field components will default to use the first mesh component, but if we have defined a mesh that has multiple components (that use different interpolation schemes) then different field components can use different mesh components. For example, in a finite elasticity problem we could define our geometry using quadratic Lagrange interpolation, and the hydrostatic pressure using linear Lagrange interpolation.

We have created a field but all the field component values are currently set to zero. We can define the geometry using the generated mesh we created earlier:

```
generatedMesh.GeometricParametersCalculate(geometricField)
```

Setting up Equations

Now we have a geometric field we can construct an *equations set*. This defines the set of equations that we wish to solve in our problem on this region. The specific equation set we are solving is defined by the fourth, fifth and sixth parameters to the `CreateStart` method. These are the equations set class, type and subtype respectively. In this example we are solving the standard Laplace equation which is a member of the classical field equations set class and the Laplace equation type. When we create an equations set we also have to create an equations set field, however, this is only used to identify multiple equations sets of the same type on a region so we will not use it:

```
equationsSetUserNumber = 1
equationsSetFieldUserNumber = 2
equationsSetField = CMISS.Field()
equationsSet = CMISS.EquationsSet()
equationsSet.CreateStart(equationsSetUserNumber, region, geometricField,
        CMISS.EquationsSetClasses.CLASSICAL_FIELD,
        CMISS.EquationsSetTypes.LAPLACE_EQUATION,
        CMISS.EquationsSetSubtypes.STANDARD_LAPLACE,
        equationsSetFieldUserNumber, equationsSetField)
equationsSet.CreateFinish()
```

Now we use our equations set to create a dependent field. This stores the solution to our equations:

```
dependentFieldUserNumber = 3
dependentField = CMISS.Field()
equationsSet.DependentCreateStart(dependentFieldUserNumber, dependentField)
equationsSet.DependentCreateFinish()
```

We haven't used the `Field.CreateStart` method to construct the dependent field but have had it automatically constructed by the equations set.

We can initialise our solution with a value we think will be close to the final solution. A field in OpenCMISS can contain multiple *field variables*, and each field variable can have multiple *components*. For the standard Laplace equation, the dependent field only has a `U` variable which has one component. Field variables can also have different field *parameter sets*, for example we can store values at a previous time step in dynamic problems. In this example we are only interested in the `VALUES` parameter set:

```
componentNumber = 1
initialValue = 0.5
dependentField.ComponentValuesInitialiseDP(
    CMISS.FieldVariableTypes.U,
    CMISS.FieldParameterSetTypes.VALUES,
    componentNumber, initialValue)
```

Once the equations set is defined, we create the *equations* that use our fields to construct equations matrices and vectors. We will use sparse matrices to store the equations and enable matrix output when assembling the equations:

```
equations = CMISS.Equations()
equationsSet.EquationsCreateStart(equations)
equations.sparsityType = CMISS.EquationsSparsityTypes.SPARSE
equations.outputType = CMISS.EquationsOutputTypes.MATRIX
equationsSet.EquationsCreateFinish()
```

Defining the Problem

Now that we have defined all the equations we will need we can create our *problem* to solve. We create a standard Laplace problem, which is a member of the classical field problem class and Laplace equation problem type:

```
problemUserNumber = 1
problem = CMISS.Problem()
problem.CreateStart(problemUserNumber)
problem.SpecificationSet(CMISS.ProblemClasses.CLASSICAL_FIELD,
    CMISS.ProblemTypes.LAPLACE_EQUATION,
    CMISS.ProblemSubTypes.STANDARD_LAPLACE)
problem.CreateFinish()
```

The problem type defines a *control loop* structure that is used when solving the problem. We may have multiple control loops with nested sub loops, and control loops can have different types, for example load incremented loops or time loops for dynamic problems. In this example a simple, single iteration loop is created without any sub loops. If we wanted to access the control loop and modify it we would use the `problem.ControlLoopGet` method before finishing the creation of the control loops, but we will just leave it with the default configuration:

```
problem.ControlLoopCreateStart()
problem.ControlLoopCreateFinish()
```

Configuring Solvers

After defining the problem structure we can create the *solvers* that will be run to actually solve our problem. The problem type defines the solvers to be set up so we call `problem.SolversCreatStart` to create the solvers and then we can access the solvers to modify their properties:

```
solver = CMISS.Solver()
problem.SolversCreateStart()
problem.SolverGet([CMISS.ControlLoopIdentifiers.NODE], 1, solver)
solver.outputType = CMISS.SolverOutputTypes.SOLVER
solver.linearType = CMISS.LinearSolverTypes.ITERATIVE
solver.linearIterativeAbsoluteTolerance = 1.0e-10
solver.linearIterativeRelativeTolerance = 1.0e-10
problem.SolversCreateFinish()
```

Note that we initialised a solver but didn't create it directly by calling its `CreateStart` method, it was created with the call to `SolversCreateStart` and then we obtain it with the call to `SolverGet`. If we look at the help for the `SolverGet` method we see it takes three parameters:

controlLoopIdentifiers A list of integers used to identify the control loop to get a solver for. This always starts with the root control loop, given by `CMISS.ControlLoopIdentifiers.NODE`. In this example we only have the one control loop and no sub loops.

solverIndex The index of the solver to get, as a control loop may have multiple solvers. In this case there is only one solver in our root control loop.

solver An initialised solver object that hasn't been created yet, and on return it will be the solver that we asked for.

Once we've obtained the solver we then set various properties before finishing the creation of all the problem solvers.

After defining our solver we can create the equations for the solver to solve by adding our equations sets to the solver equations. In this example we have just one equations set to add but for coupled problems we may have multiple equations sets in the solver equations. We also tell OpenCMISS to use sparse matrices to store our solver equations:

```
solverEquations = CMISS.SolverEquations()
problem.SolverEquationsCreateStart()
solver.SolverEquationsGet(solverEquations)
solverEquations.sparsityType = CMISS.SolverEquationsSparsityTypes.SPARSE
equationsSetIndex = solverEquations.EquationsSetAdd(equationsSet)
problem.SolverEquationsCreateFinish()
```

Setting Boundary Conditions

The final step in configuring the problem is to define the boundary conditions to be satisfied. We will set the dependent field value at the first node to be zero, and at the last node to be 1.0. These nodes will correspond to opposite corners in our geometry. Because OpenCMISS can solve our problem on multiple computational nodes where each computational node does not necessarily know about all nodes in our mesh, we must first check that the node we are setting the boundary condition at is in our computational node domain:

```
nodes = CMISS.Nodes()
region.NodesGet(nodes)
firstNodeNumber = 1
lastNodeNumber = nodes.numberOfNodes
firstNodeDomain = decomposition.NodeDomainGet(firstNodeNumber, 1)
lastNodeDomain = decomposition.NodeDomainGet(lastNodeNumber, 1)
computationalNodeNumber = CMISS.ComputationalNodeNumberGet()

boundaryConditions = CMISS.BoundaryConditions()
solverEquations.BoundaryConditionsCreateStart(boundaryConditions)
if firstNodeDomain == computationalNodeNumber:
    boundaryConditions.SetNode(
        dependentField, CMISS.FieldVariableTypes.U,
        1, 1, firstNodeNumber, 1,
        CMISS.BoundaryConditionsTypes.FIXED, 0.0)
if lastNodeDomain == computationalNodeNumber:
    boundaryConditions.SetNode(
        dependentField, CMISS.FieldVariableTypes.U,
        1, 1, lastNodeNumber, 1,
        CMISS.BoundaryConditionsTypes.FIXED, 1.0)
solverEquations.BoundaryConditionsCreateFinish()
```

When setting a boundary condition at a node we can use either the `AddNode` method or the `SetNode` method. Using `AddNode` will add the value we provide to the current field value and set this as the boundary condition value, but here we want to directly specify the value so we use the `SetNode` method.

The arguments to the `SetNode` method are the field, field variable type, node version number, node user number, node derivative number, field component number, boundary condition type and boundary condition value. The version and derivative numbers are one as we aren't using versions and we are setting field values rather than derivative values. We can also only set derivative boundary conditions when using a Hermite basis type. There are a wide number of boundary condition types that can be set but many are only available for certain equation set types and in this example we simply want to fix the field value.

When `solverEquations.BoundaryConditionsCreateFinish()` is called OpenCMISS will construct the solver matrices and vectors.

Solving

After our problem solver equations have been fully defined we are now ready to solve our problem. When we call the `Solve` method of the problem it will loop over the control loops and control loop solvers to solve our problem:

```
problem.Solve()
```

Exporting the Solution

Once the problem has been solved, the dependent field contains the solution to our problem. We can then export the dependent and geometric fields to a FieldML file so that we can visualise the solution using cmgui. We will export the geometric and dependent field values to a `LaplaceExample.xml` file. Separate plain text data files will also be created:

```
baseName = "laplace"
dataFormat = "PLAIN_TEXT"
fml = CMISS.FieldMLIO()
fml.OutputCreate(mesh, "", baseName, dataFormat)
fml.OutputAddFieldNoType(
    baseName + ".geometric", dataFormat, geometricField,
    CMISS.FieldVariableTypes.U, CMISS.FieldParameterSetTypes.VALUES)
fml.OutputAddFieldNoType(
    baseName + ".phi", dataFormat, dependentField,
    CMISS.FieldVariableTypes.U, CMISS.FieldParameterSetTypes.VALUES)
fml.OutputWrite("LaplaceExample.xml")
fml.Finalise()
```

Using CellML in OpenCMISS

The aim of this tutorial is to give the user a bit of help in using CellML models in OpenCMISS. OpenCMISS(cellml) is the code that provides the mapping from OpenCMISS(cm) (Fortran) to the CellML API (C++). The API exposed by the OpenCMISS(cellml) library is then wrapped by the OpenCMISS(cm) API, with documentation available in the OpenCMISS programmer documentation. The programmer documentation provides the reference documentation for the OpenCMISS(cm) routines which are described in this tutorial. The monodomain example from the OpenCMISS examples repository provides a good demonstration of OpenCMISS+CellML in practice, while the CellML examples are used to perform functional testing of the OpenCMISS(cellml) library.

Overview

OpenCMISS encapsulates all interaction with CellML models within a *CellML environment*. A CellML environment is specific to a region, but each region may contain zero or more CellML environments. Having created a CellML environment you are then able to import CellML models into the environment. Each CellML environment may import an arbitrary number of CellML models. [TODO: what is a use-case for multiple CellML environments in a region?]

Variables in CellML models imported into a CellML environment are then able to flagged as *known* or *wanted* as desired by the user. Flagged variables are made available by the CellML environment for mapping to OpenCMISS fields, either for obtaining values from the CellML model (wanted) or setting values in the CellML model (known).

In order for the CellML environment to mange the computation of a CellML model, each environment requires the user to provide several fields suitable for such usage. These are the *models*, *state*, *intermediate*, and *parameters* fields. Typically the user will pass in empty fields and allow the CellML environment to create the fields appropriately.

With all desired fields set-up appropriately, the user then just needs to add their CellML environment(s) to the corresponding solver as part of their control loop set-up when defining the actual problem to solve.

The CellML environment

The OpenCMISS type for the CellML environment is `CMISSCellMLType`. As with most OpenCMISS types, a user-number is provided to uniquely identify this CellML environment to the user. The basic creation block is given below.

```
! declare the user number for the CellML environment we want to create
INTEGER(CMISSIntg), PARAMETER :: CellMLUserNumber=10
! and the actual handle for the CellML environment
TYPE(CMISSCellMLType) :: CellML

! We first initialise the CellML environment
CALL CMISSCellML_Initialise(CellML,Err)
```

```
! and then we are able to trigger the start of creation
CALL CMISSCellML_CreateStart(CellMLUserNumber,Region,CellML,Err)

! import models

! flag variables

! terminate the creation process
CALL CMISSCellML_CreateFinish(CellML,Err)
```

It is important to note that all required models must be imported and all desired variables flagged before terminating the CellML environment creation process. This is because the create finish method will proceed to make use of OpenCMISS(cellml) to instantiate each of the imported CellML models into executable code, and the generation of that executable code requires all knowledge of the flagged variables.

Models are imported into the CellML environment with the code shown below.

```
INTEGER(CMISSIntg) :: modelIndex

! Import the specified model into the CellML environment
CALL CMISSCellML_ModelImport(CellML,"model.xml",modelIndex,Err)
```

In this example, the CellML model `model.xml` is imported into the CellML environment and on successful return the variable `modelIndex` will be set to the index for this specific model in the CellML environment. The CellML model to import is specified by URL, and can be either absolute (e.g., `http://example.com/coolest/model/ever.xml`) or relative (e.g., `coolest/model/ever.xml`). If a relative URL is specified, it will be resolved relative to the current working directory (CWD) of the executed application. (It is anticipated that application developers would use their own logic to provide absolute URLs to define CellML models in OpenCMISS.)

Flagging variables

As mentioned above, all variables of interest in the imported CellML models must be flagged prior to terminating the CellML environment creation process. Variables in CellML models can be flagged as either *known* or *wanted*. Flagging variables in a model will inform OpenCMISS(cellml) that the given variable(s) need to be available for use by OpenCMISS(cm), and hence the executable code generated when the models are instantiated is required to provide this access. The process for flagging variables is shown below.

```
! Flag a variable as known
CALL CMISSCellML_VariableSetAsKnown(CellML,modelIndex,"fast_sodium_current/g_Na ",
↪Err)
! Flag a variable as wanted
CALL CMISSCellML_VariableSetAsWanted(CellML,modelIndex,"membrane/i_K1",Err)
```

Details on how to identify specific variables in a CelLML model are given below. The `modelIndex` should be the index of the desired model in the CellML environment, as returned by the model import described above.

Flagging a variable as *known* indicates that the OpenCMISS user wants to control the value of the specified variable, thus taking ownership of the variable from the CellML model. Currently, only CONSTANT variables in CellML models can be flagged as known (see the Variable Evaluation Types in the CellML API documentation). This is typically used when parameters in the CellML model are to have their values defined by OpenCMISS fields.

Flagging a variable as *wanted* indicates that the OpenCMISS user wants to obtain the value of the specified variable from the CellML model. Currently, CONSTANT, PSEUDOSTATE_VARIABLE, and ALGEBRAIC variables in CellML models can be flagged as wanted (see the Variable Evaluation Types in the CellML API documentation). In order to be able to save the state of CellML models during integration steps, all state variables in models are automatically flagged as wanted. Depending on how the CellML model is being applied in the OpenCMISS simulation, variables that are considered CONSTANT by the CelLML API will actually be the variables of interest to the OpenCMISS user - commonly the case for mechanical constitutive relationships where the *wanted* strain energy components are algebraically related to the *known* strain components.

Identifying CellML variables

When identifying variables from CellML models in OpenCMISS, the convention is to address them with a string consisting of the variable's name and the name of the parent component. Given the following (invalid!) CellML model,

```
<model>
  <import href="http://models.cellml.org/bob/model.xml">
    <component name="imported_component" component_ref="source_component"/>
  </import>
  <component name="membrane">
    <variable name="i_K1"/>
    <variable name="i_stimulus"/>
    <variable name="T"/>
  </component>
  <component name="temperature">
    <variable name="temperature" units="K" initial_value="310.0" public_interface=
↪"out"/>
  </component>
  <connection>
    <map_components component_1="membrane" component_2="temperature"/>
    <map_variables variable_1="T" variable_2="temperature"/>
  </connection>
</model>
```

the `i_K1` variable in the `membrane` component is identified with the string `membrane/i_K1`. Similarly, `membrane/i_stimulus` identifies the stimulus current, `membrane/T` identifies the temperature variable in the membrane component, and `temperature/temperature` identifies the temperature variable in the temperature component. Due to the connection between the temperature variables in the membrane and temperature components, `membrane/T` and `temperature/temperature` can be treated interchangeably. (Internally, OpenCMISS(cellml) will always resolve variable references to the source variable and hence `membrane/T` will resolve to `temperature/temperature`.) Given the rules for naming CellML components and variables, these identifier strings are guaranteed to be unique for a specified model.

Using this method to identify variables in CellML models, it is not possible to address variables which are not described in the top-level model being imported into the OpenCMISS CellML environment. For example, the above CellML model imports the component `source_component` from the model `http://models.cellml.org/bob/model.xml` but the variables in that component are not available to the OpenCMISS user unless they are connected to variables in the model (i.e., there are connections that map the component `imported_component` to the component `membrane` or `temperature` in the above model).

Mapping between variables and fields

TODO: This section needs some checking and filling out!

CellML model variables which have been flagged (see above) are able to be mapped to components of OpenCMISS fields. CellML variables which are mapped from OpenCMISS field components will have their value updated from the field at each DOF prior to evaluation of the CellML model. OpenCMISS field components which are mapped from CellML variables will trigger evaluation of the CellML model(s) associated with the requested DOF.

The basic mapping process is as follows.

```
! Start the creation of CellML <--> OpenCMISS field maps
CALL CMISSCellML_FieldMapsCreateStart(CellML,Err)

! and set up the field variable component <--> CellML model variable mappings.
! Map the first component of the dependent field to the membrane potential in the␣
↪CellML model
CALL CMISSCellML_CreateFieldToCellMLMap(CellML,DependentField,CMISS_FIELD_U_
↪VARIABLE_TYPE,1,CMISS_FIELD_VALUES_SET_TYPE, &
  & modelIndex,"membrane/Vm",CMISS_FIELD_VALUES_SET_TYPE,Err)

! and the reverse mapping
CALL CMISSCellML_CreateCellMLToFieldMap(CellML,modelIndex,"membrane/Vm",CMISS_
↪FIELD_VALUES_SET_TYPE, &
  & DependentField,CMISS_FIELD_U_VARIABLE_TYPE,1,CMISS_FIELD_VALUES_SET_TYPE,Err)

! Finish the creation of CellML <--> OpenCMISS field maps
CALL CMISSCellML_FieldMapsCreateFinish(CellML,Err)
```

For a mechanical constitutive application, the mapping may look more like

```
!Start the creation of CellML <--> OpenCMISS field maps
CALL CMISSCellML_FieldMapsCreateStart(CellML,Err)
!Now we can set up the field variable component <--> CellML model variable␣
↪mappings.
!Map the strain components from OpenCMISS fields to the CellML model
CALL CMISSCellML_CreateFieldToCellMLMap(CellML,DependentField,CMISS_FIELD_U1_
↪VARIABLE_TYPE,1,CMISS_FIELD_VALUES_SET_TYPE, &
  & MooneyRivlinModelIndex,"equations/E11",CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateFieldToCellMLMap(CellML,DependentField,CMISS_FIELD_U1_
↪VARIABLE_TYPE,2,CMISS_FIELD_VALUES_SET_TYPE, &
  & MooneyRivlinModelIndex,"equations/E12",CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateFieldToCellMLMap(CellML,DependentField,CMISS_FIELD_U1_
↪VARIABLE_TYPE,3,CMISS_FIELD_VALUES_SET_TYPE, &
  & MooneyRivlinModelIndex,"equations/E13",CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateFieldToCellMLMap(CellML,DependentField,CMISS_FIELD_U1_
↪VARIABLE_TYPE,4,CMISS_FIELD_VALUES_SET_TYPE, &
  & MooneyRivlinModelIndex,"equations/E22",CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateFieldToCellMLMap(CellML,DependentField,CMISS_FIELD_U1_
↪VARIABLE_TYPE,5,CMISS_FIELD_VALUES_SET_TYPE, &
  & MooneyRivlinModelIndex,"equations/E23",CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateFieldToCellMLMap(CellML,DependentField,CMISS_FIELD_U1_
↪VARIABLE_TYPE,6,CMISS_FIELD_VALUES_SET_TYPE, &
  & MooneyRivlinModelIndex,"equations/E33",CMISS_FIELD_VALUES_SET_TYPE,Err)

! Map the material parameters - if any

! Map the stress components from the CellML model to the OpenCMISS dependent field
CALL CMISSCellML_CreateCellMLToFieldMap(CellML,MooneyRivlinModelIndex,"equations/
↪Tdev11",CMISS_FIELD_VALUES_SET_TYPE, &
  & DependentField,CMISS_FIELD_U2_VARIABLE_TYPE,1,CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateCellMLToFieldMap(CellML,MooneyRivlinModelIndex,"equations/
↪Tdev12",CMISS_FIELD_VALUES_SET_TYPE, &
  & DependentField,CMISS_FIELD_U2_VARIABLE_TYPE,2,CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateCellMLToFieldMap(CellML,MooneyRivlinModelIndex,"equations/
↪Tdev13",CMISS_FIELD_VALUES_SET_TYPE, &
  & DependentField,CMISS_FIELD_U2_VARIABLE_TYPE,3,CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateCellMLToFieldMap(CellML,MooneyRivlinModelIndex,"equations/
↪Tdev22",CMISS_FIELD_VALUES_SET_TYPE, &
  & DependentField,CMISS_FIELD_U2_VARIABLE_TYPE,4,CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateCellMLToFieldMap(CellML,MooneyRivlinModelIndex,"equations/
↪Tdev23",CMISS_FIELD_VALUES_SET_TYPE, &
  & DependentField,CMISS_FIELD_U2_VARIABLE_TYPE,5,CMISS_FIELD_VALUES_SET_TYPE,Err)
CALL CMISSCellML_CreateCellMLToFieldMap(CellML,MooneyRivlinModelIndex,"equations/
↪Tdev33",CMISS_FIELD_VALUES_SET_TYPE, &
  & DependentField,CMISS_FIELD_U2_VARIABLE_TYPE,6,CMISS_FIELD_VALUES_SET_TYPE,Err)
!Finish the creation of CellML <--> OpenCMISS field maps
```

```
CALL CMISSCellML_FieldMapsCreateFinish(CellML,Err)
```

CellML fields

The required fields and what you might want to do with them...

Evaluating CellML fields

Do we want to describe how simple algebraic-type evaluation can be done independently of the whole solver/problem/equation set thingy? is that even possible?

Adding CellML models to your Problem set-up

Really need a better title! Explain how CellML environments get added into control loops, etc. so that they get computed along with the rest of the model when performing a simulation.

Miscellaneous utilities

Collect all the other bits and pieces here? these are typically the OpenCMISS(cellml) functions that are used internally by OpenCMISS(cm) but maybe should be (and are?) exposed via the OpenCMISS(cm) API?

OpenCMISS tutorial videos

Videos of OpenCMISS tutorials.

MPI tutorial 31 January 2012 Chris Bradley

CUDA tutorial 1 17 February 2012 Chris Bradley

CUDA tutorial 2 24 February 2012 Chris Bradley

OpenMP tutorial 16 March 2012 Chris Bradley

Contact Mechanics tutorial 17 August 2012 Nancy Yan

Bidomain tutorial

**

Tutorial: Version Control With Git

Section author: Adam Reeve

Getting Started

If Git isn't installed already, it can be installed from the git-core package if you're using Linux or by downloading msysGit for Windows.

The first thing that must be done before using Git is to tell it your name and email address. These will be associated with any commits you make:

```
git config --global user.name "Adam Reeve"
git config --global user.email "aree035@aucklanduni.ac.nz"
```

If you are using a graphical Git client such as Tortoise Git on Windows or gitg on Linux there will be an option to edit these settings.

Creating a Repository

You can initialise a new Git repository in an existing directory with:

```
git init
```

Or to create a new directory:

```
git init my_project
```

Cloning an Existing Repository

You can clone an existing repository from a local path or from a URL. To clone the repository for this tutorial, run:

```
git clone https://github.com/adamreeve/git_tutorial.git
```

This will create a git_tutorial directory in your current directory, so change into that directory now:

```
cd git_tutorial
```

Have a look at the project history on the master branch with:

```
git log
```

Also try out some of the options to the log command:

```
git log --patch
git log --stat
git log --oneline --graph --all
```

The last example uses the "–all" option to show history in all branches.

Making and Committing Changes

Run **`git status`** and you should get a message that there are no changes to be staged and no staged changes to be committed.

Now make a change to the `hello.py` file and run **`git status`** again. You should see that you still have no changes staged to commit but you have an unstaged change.

Run **`git diff`** go see what changes you have made. Now add your change to the staging area:

```
git add hello.py
```

And run **`git status`** again. You will see that there are no unstaged changes but one file has changes that are staged.

Now run **`git diff`** and there should be no changes. This is because **`git diff`** compares your working directory with the staging area by default. To see what changes are in the staging area and would be committed, run:

```
git diff --cached
```

If you've decided that you no longer want to commit a change that you've added to the staging area, you'll need to unstage that file by running:

```
git reset HEAD hello.py
```

Note: git tells you how to stage and unstage changes in the output of the **`git status`** command.

If you then decide that your change is rubbish and you want to remove it completely, you can checkout a clean version of the changed file:

```
git checkout -- hello.py
```

The "–" before the file path isn't required but is recommended as it means that any further command options are file paths, which prevents confusion when you have a file named the same as a branch (as **`git checkout`** is also used for checking out a branch, as you'll see later).

Now make another change and add it to the staging area, then commit it:

```
git commit -m "My awesome change"
```

You can either specify a commit message on the command line with the "-m" option or if you leave that option off, git will open a text editor to allow you to enter a message. By default this is vim, but you might want to change it to something else. For example, if you're on Linux:

```
git config --global core.editor "gedit"
```

Or if you're on Windows and have TortoiseGit installed, you can use:

```
git config --global core.editor "notepad2"
```

Now make another change to the file and then add this change to the staging area, then run:

```
git commit --amend
```

The amend option lets you update the previous commit. It will also open the editor to let you update the commit message if required. This is useful if you realise you've made a small mistake in the previous commit and haven't yet pushed it to a public repository.

Note: The commit hash changes after you amend it. This is now a different commit to the one before, so the hash has changed.

Now we'll try using git's graphical interface for making a commit. Make a change to `hello.py` then run:

```
git gui
```

Stage the change you made then make another commit.

Branches

It's always a good idea to create a new branch for any new feature you're working on in a project:

```
git branch new_feature
```

This will create a new branch that points to the commit you have currently checked out. You can also specify which commit the new branch should point to. For example, to create a new branch that points to the same commit as the master branch:

```
git branch another_feature master
```

To delete a branch:

```
git branch -d another_feature
```

This will give an error if the branch hasn't been merged into another branch.

Now checkout the branch you created. If there are any differences between your previous head commit and the branch you are checking out, your working directory will be updated:

```
git checkout new_feature
```

You can create a new branch and check it out in one go by using the "-b" option to the checkout command:

```
git checkout -b my_feature
```

Now make some changes and commit them on your new branch. You can see a list of branches and the branch you're on at any time by running:

```
git branch
```

Have a look at the history of your branch and the position of other branches by running **gitk**.

Merging and Resolving Conflicts

Now we will practice merging one branch into another branch. We will create a new local branch that matches the "merge_into" branch from the origin repository, and merge in the "merge_from" branch. First create the local branch you will work on:

```
git checkout -b merge_into origin/merge_into
```

Now run:

```
gitk --all
```

This will show a tree with commits from all branches. Note where the heads of the merge_from, origin/merge_from and origin/merge_into branches are.

Now merge the origin/merge_from branch:

```
git merge origin/merge_from
```

And look at the result of your merge:

```
gitk --all
```

Now we will try another merge, but this time there will be a conflict:

```
git checkout -b merge_conflict origin/merge_conflict
gitk --all
git merge origin/conflicting
```

Read the output of the merge command to note that there is a conflict in the `hello.py` file. Also run **git status**. When you have conflicts in multiple files you can keep track of which conflicts have been resolved with the status command. Open that file in your editor and resolve the conflict. Then mark the conflict as resolved by adding the file:

```
git add hello.py
```

And now you have to manually make a merge commit:

```
git commit -m "Merge conflicting branch"
```

Remotes and Remote Branches

As you originally cloned this repository, you have one remote repository set up already called "origin". To list the remote repositories you've added with their urls:

```
git remote -v
```

Branches on a remote repository can be checked out or referred to in other commands by prefixing them with the remote name. For example, to show the head commit of the master branch on the origin repository:

```
git show origin/master
```

To see all branches including those on remote repositories, you can use:

```
git branch -a
```

Staging Parts of Files

Most Git graphical interfaces allow you to stage only some changes in a file. From the Git command line you can do this with the "–patch" or "-p" option to the add command. Change a line at the top of `hello.py` and then make another change at the bottom. Now run:

```
git add -p hello.py
```

Say yes to adding the first change but no to the second change, then run **`git status`**, **`git diff`**, and then **`git diff --cached`**.

Stashing Changes

Git's stash command is useful for storing changes you have made that you want to save but don't want to commit yet. For example, you can stash changes and then switch to another branch and do some work, making commits, then go to another branch and recover your stashed changes.

Run **`git status`**. You should have a change added to the staging area and another unstaged change from the last section. Otherwise make a change to `hello.py`. Now stash those changes:

```
git stash
```

And look at what this has done:

```
git status
git stash list
git stash show -p stash@{0}
```

Your working directory and index should now be clean, but your change is safely stored away in the list of stashes.

Now pop your stashed change off the top of the stash list:

```
git stash pop
git status
git diff
git stash list
```

Your changes are now back in your working directory, and have been removed from the stash list.

Stash your change again to get a clean working directory:

```
git stash
```

Note that you can use **`git stash apply`** to apply a stashed change without removing it from the stash list, and that you can also apply a stashed change on a different branch to the one it was made on.

Rewriting History

Git has powerful tools for allowing you to rearrange history so that you can clean up work to make the history more clear. The most useful one to know is **git rebase --interactive**. Rebasing means to move a series of commits onto a new base commit. You can also use the rebase command and keep the base the same, but still edit and rearrange commits.

Checkout a new branch that points to the same commit as origin/rebase_me:

```
git checkout -b rebasing origin/rebase_me
```

We will rebase these commits onto the latest master branch. First have a look at what we will rebasing by running **gitk --all** and looking at the origin/rebase_me branch, or **git log -p origin/master..rebasing**.

Now start the interactive rebase:

```
git rebase --interactive origin/master
```

Reorder the commits so that the "Update docstring" commit is first and the "Fix typo" commit is squashed into the "More excitement" commit.

Because you have squashed a commit, you will get the opportunity to change the message of the commit that was squashed into. Think about whether you should update the original commit message to account for the change that was squashed.

Run **gitk** to see what you've done.

Note that you can still access any rebased commits by their hash, and you can find the commits that you have recently checked out with the **git reflog** command. This means that if you have committed something it's very hard to permanently lose that work.

More on Remotes

Create an account on GitHub if you don't have one already, then fork this repository. Now make some changes to this repository (if you can, make the tutorial better!) and push them to a branch on your remote repository. To push you'll first have to create an alias for your remote repository:

```
git remote add github https://<your_username>@github.com/<your_username>/git_
↪tutorial.git
```

Where "<your_username>" has been replaced with your GitHub username. Now to push your changes to your remote repository:

```
git push github <name_of_branch>
```

Now on the web page for your fork of the git_tutorial repository, click on the pull request button and follow the steps to create a new pull request.

- genindex
- search

Laplace examples

42 master

For testing

Analytic Laplace

For testing

Laplace ellipsoid

For testing

Laplace Python

For testing

Laplace

For testing

Laplace C

For testing

Number Laplace

For testing

Parallel Laplace

For testing

CHAPTER 6

PMR user documentation

The documentation found here is mainly aimed towards providing information to users of the Physiome Model Repository. This includes users interested in obtaining and running models from the respository, and those who wish to add models to the repository.

If you wish to deploy an instance of the repository software, PMR2, please see the buildout repository on GitHub.

PMR - an introduction

The Physiome Model Repository, PMR (Physiome Model Repository), includes the CellML and FieldML repositories, and is powered by software called PMR2. PMR relies on the distributed version control system Mercurial (Hg), which allows the repository to maintain a complete history of all changes made to every file contained within repository *workspaces*. In order to use the Physiome Model Repository, you will need to obtain a Mercurial client for your operating system, and become familiar with the basic functions of Mercurial. There are many excellent resources available on the internet, such as Mercurial, the definitive guide. Mercurial clients may be downloaded from the Mercurial website, which also provides documentation on Mercurial usage. A graphical alternative to a command-line client is available for Windows, called TortoiseHg. This provides a Windows explorer integrated system for working with Mercurial repositories.

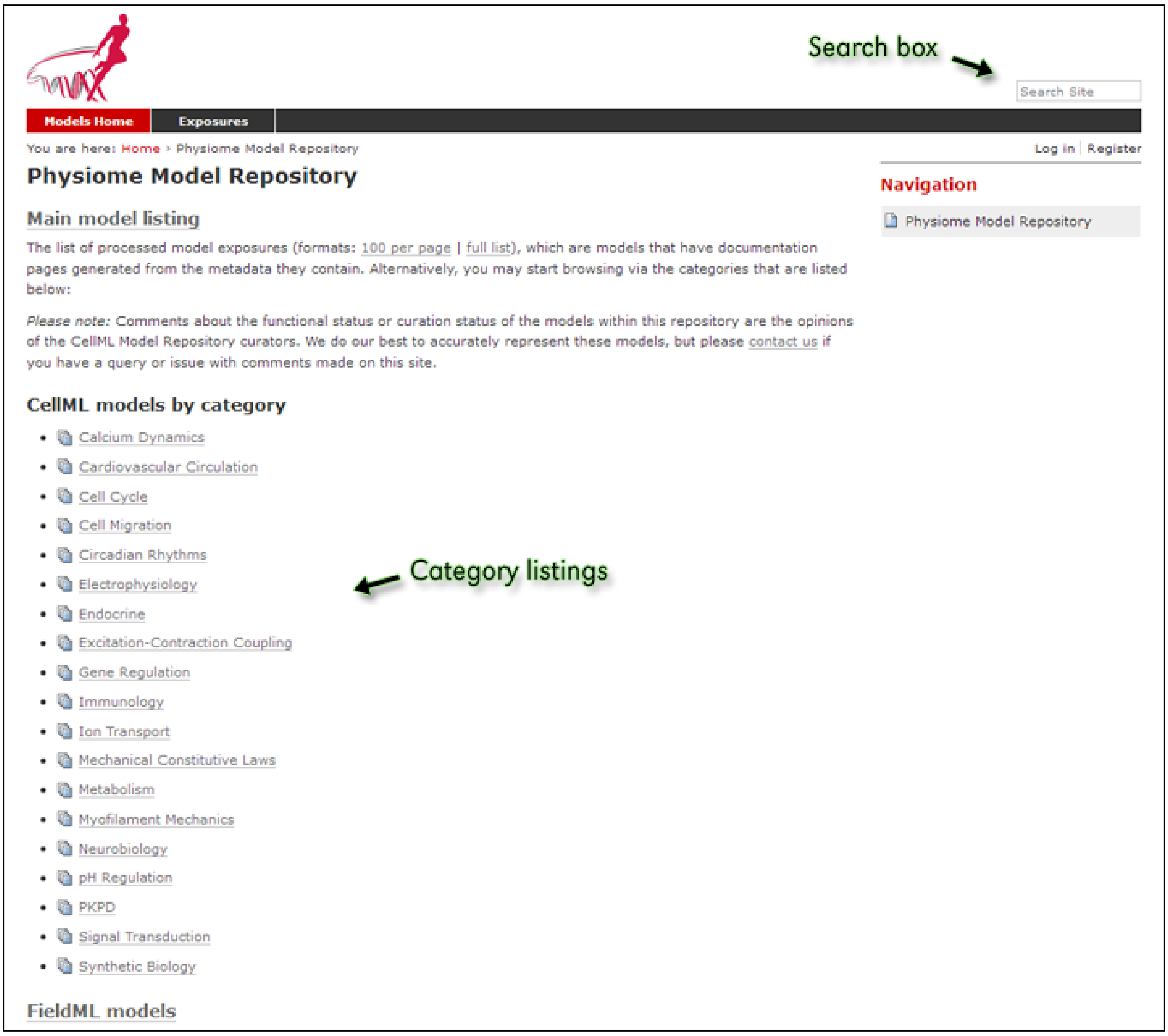
Downloading and viewing models from the Physiome Model Repository

There are several ways of obtaining and using models from the Physiome Model Repository, and which you choose will depend on the way you intend to use the models. If you are simply interested in running a particular model and viewing the output, you can use links found on model *exposure* pages to get hold of the model files. There links available for a large number of models that will load the model directly into the OpenCell application, allowing you to explore simulation results with the help of a model diagram.

If you intend to use the model for further work, for example saving changes to the model or creating a new model based on an existing model or parts of an existing model, you should use *Mercurial* to obtain the files. In this way you also obtain the complete revision history of the files, and can add to this history as you make your own changes.

Searching the repository

The Physiome Model Repository has a basic search function that can be accessed by typing search terms into the box at the top right hand side of the page. You can use keywords such as `cardiac` or `insulin`, author names, or any other terms relevant to the models you want to find.



You are here: Home › Physiome Model Repository

Physiome Model Repository

Main model listing

The list of processed model exposures (formats: 100 per page | full list), which are models that have documentation pages generated from the metadata they contain. Alternatively, you may start browsing via the categories that are listed below:

Please note: Comments about the functional status or curation status of the models within this repository are the opinions of the CellML Model Repository curators. We do our best to accurately represent these models, but please contact us if you have a query or issue with comments made on this site.

Physiome Model Repository

CellML models by category

- Calcium Dynamics
- Cardiovascular Circulation
- Cell Cycle
- Cell Migration
- Circadian Rhythms
- Electrophysiology
- Endocrine
- Excitation-Contraction Coupling
- Gene Regulation
- Immunology
- Ion Transport
- Mechanical Constitutive Laws
- Metabolism
- Myofilament Mechanics
- Neurobiology
- pH Regulation
- PKPD
- Signal Transduction
- Synthetic Biology

FieldML models

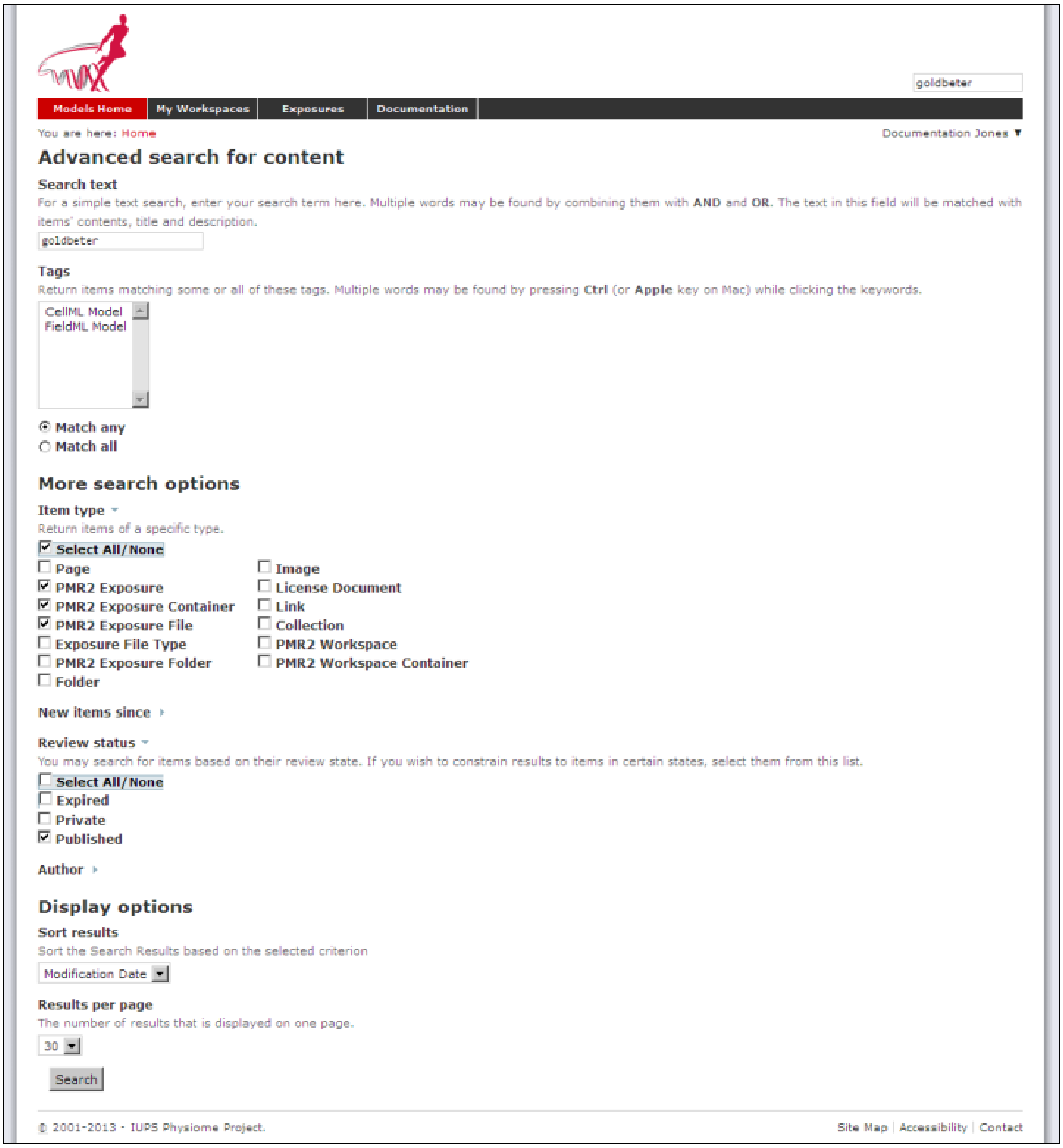
Fig. 6.1: The index page of the model repository provides two methods for finding models. There is a box for entering search terms, or you can click on categories based on model keywords to see all models in those categories.

If your search is yielding too many results, you may either try to narrow it down by choosing more or different keywords (*eg.* `goldbeter 1991` instead of just `goldbeter`), or you can click the *Advanced Search* link just under the search box on the results page. This will take you to a search page where you can select specific item types (*eg.* exposures or workspaces), statuses, and other specifics.

Once you have found the model you are interested in, there are several ways you can view or download it.

Viewing models via the respository web interface

The most common use of the Physiome Model Repository web interface is probably to view information about models found on exposure pages, and to then download the models from these pages for simulation in a CellML supporting application.



Advanced search for content

Search text

For a simple text search, enter your search term here. Multiple words may be found by combining them with **AND** and **OR**. The text in this field will be matched with items' contents, title and description.

goldbeter

Tags

Return items matching some or all of these tags. Multiple words may be found by pressing **Ctrl** (or **Apple** key on Mac) while clicking the keywords.

CellML Model
FieldML Model

● Match any
○ Match all

More search options

Item type

Return items of a specific type.

☑ Select All/None

☐ Page
☑ PMR2 Exposure
☑ PMR2 Exposure Container
☑ PMR2 Exposure File
☐ Exposure File Type
☐ PMR2 Exposure Folder
☐ Folder
☐ Image
☐ License Document
☐ Link
☐ Collection
☐ PMR2 Workspace
☐ PMR2 Workspace Container

New items since

Review status

You may search for items based on their review state. If you wish to constrain results to items in certain states, select them from this list.

☐ Select All/None
☐ Expired
☐ Private
☑ Published

Author

Display options

Sort results

Sort the Search Results based on the selected criterion

Modification Date

Results per page

The number of results that is displayed on one page.

30

Search

© 2001-2013 - IUPS Physiome Project.

Site Map | Accessibility | Contact

Fig. 6.2: In this search I have chosen to only have published exposures in my results.

Below is an example of a CellML exposure page. It contains documentation about the model(s), a diagram of the what the model(s) represent, and a navigation pane that allows the user to select between available versions of the model. Many models only have one version, but in this case there are two variants.

If you click on one of the model variant navigation links, you will be taken to a sub-page of the exposure which will allow you to view the actual CellML model in a number of ways.

On this page there are a number of options under a *Views available* panel at the right hand side.

- *Documentation* - displays the model documentation, already visible in the main area of the exposure page.
- *Model Metadata* - displays information such as the citation information, model authorship details, and PMR keywords.
- *Model Curation* - displays the curation stars for the model, also visible at the top right of the page. Future additions to the curation system mean that there will be additional information to be displayed on this page.
- *Mathematics* - displays all the equations in the model in graphical form.
- *Generated code* - shows a page where you can view the model in a number of different languages; C, C_IDA, Fortran 77, MATLAB, and Python. You can copy the generated code directly from this page to paste into your code editor.
- *Cite this model* - this page provides generic information about how to cite models in the repository.
- *Source View* - provides a raw view of the CellML (XML) model code.
- *Simulate using OpenCell* - this link will download the model and open it with OpenCell if you have the software installed. If the model has a session file, this will include an interactive diagram which can be clicked on to display traces of the simulation results.

The OpenCell session that is loaded when clicking on the Simulate using OpenCell link looks something like this:

Obtaining models via Mercurial

CellML Model Repository tutorial

Section author: David Nickerson, Randall Britten, Dougal Cowan

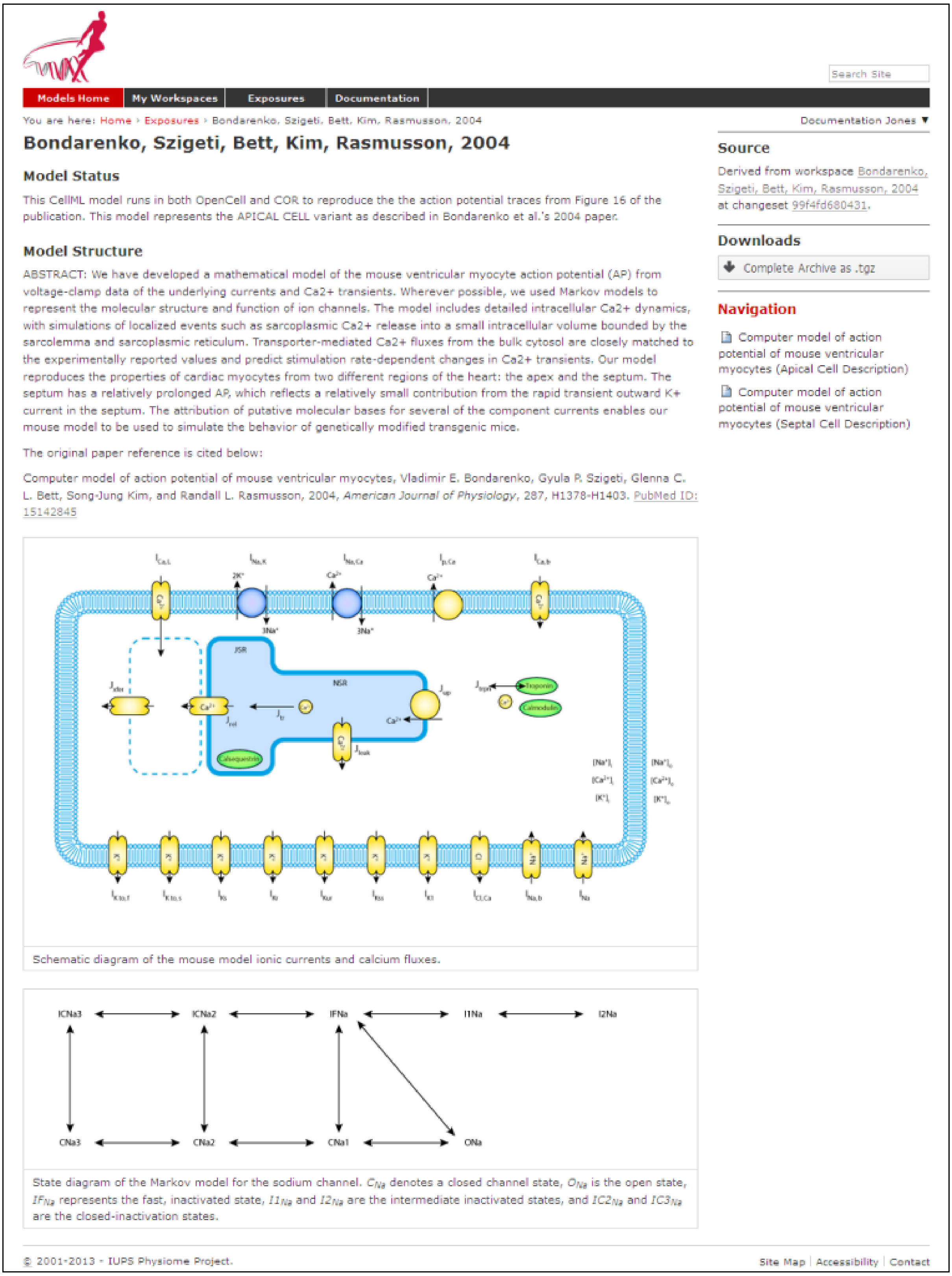
About this tutorial

The CellML model repository is an instance of the Physiome Model Repository (PMR) customised for CellML models. PMR currently relies on the distributed version control system Mercurial (Hg), which allows the repository to maintain a complete history of all changes made to every file it contains. This tutorial demonstrates how to work with the repository using TortoiseHg, which provides a Windows explorer integrated system for working with Mercurial repositories.

```
Brief mention of the equivalent command line versions of the TortoiseHg
actions will also be mentioned, so that these ideas can also be used without
a graphical client, and on Linux and similar systems. These will be denoted
by boxes like this.
```

This tutorial requires you to have:

- A Mercurial client such as TortoiseHg or Mercurial installed
- The OpenCell CellML modelling environment
- A text editor such as Notepad++ or gedit



You are here: Home › Exposures › Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004

Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004

Model Status

This CellML model runs in both OpenCell and COR to reproduce the the action potential traces from Figure 16 of the publication. This model represents the APICAL CELL variant as described in Bondarenko et al.'s 2004 paper.

Model Structure

ABSTRACT: We have developed a mathematical model of the mouse ventricular myocyte action potential (AP) from voltage-clamp data of the underlying currents and Ca2+ transients. Wherever possible, we used Markov models to represent the molecular structure and function of ion channels. The model includes detailed intracellular Ca2+ dynamics, with simulations of localized events such as sarcoplasmic Ca2+ release into a small intracellular volume bounded by the sarcolemma and sarcoplasmic reticulum. Transporter-mediated Ca2+ fluxes from the bulk cytosol are closely matched to the experimentally reported values and predict stimulation rate-dependent changes in Ca2+ transients. Our model reproduces the properties of cardiac myocytes from two different regions of the heart: the apex and the septum. The septum has a relatively prolonged AP, which reflects a relatively small contribution from the rapid transient outward K+ current in the septum. The attribution of putative molecular bases for several of the component currents enables our mouse model to be used to simulate the behavior of genetically modified transgenic mice.

The original paper reference is cited below:

Computer model of action potential of mouse ventricular myocytes, Vladimir E. Bondarenko, Gyula P. Szigeti, Glenna C. L. Bett, Song-Jung Kim, and Randall L. Rasmusson, 2004, *American Journal of Physiology*, 287, H1378-H1403. PubMed ID: 15142845

Schematic diagram of the mouse model ionic currents and calcium fluxes.

State diagram of the Markov model for the sodium channel. C_{Na} denotes a closed channel state, O_{Na} is the open state, IF_{Na} represents the fast, inactivated state, $I1_{Na}$ and $I2_{Na}$ are the intermediate inactivated states, and $IC2_{Na}$ and $IC3_{Na}$ are the closed-inactivation states.

Source

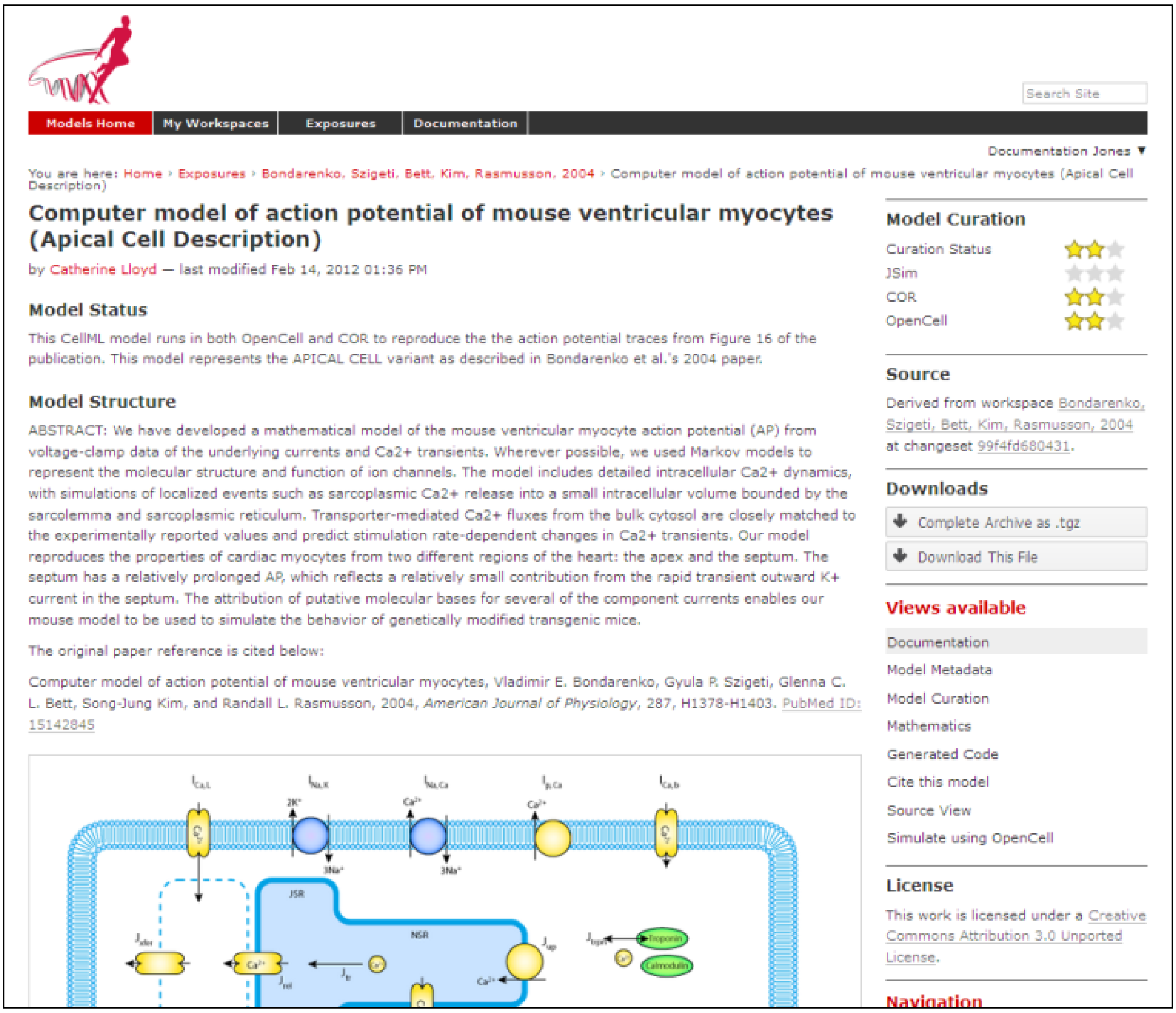
Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset 99f4fd680431.

Downloads

Navigation

- Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)
- Computer model of action potential of mouse ventricular myocytes (Septal Cell Description)

Fig. 6.3: An example of a CellML exposure page.



You are here: Home › Exposures › Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 › Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

by Catherine Lloyd — last modified Feb 14, 2012 01:36 PM

Model Status

This CellML model runs in both OpenCell and COR to reproduce the the action potential traces from Figure 16 of the publication. This model represents the APICAL CELL variant as described in Bondarenko et al.'s 2004 paper.

Model Structure

ABSTRACT: We have developed a mathematical model of the mouse ventricular myocyte action potential (AP) from voltage-clamp data of the underlying currents and Ca2+ transients. Wherever possible, we used Markov models to represent the molecular structure and function of ion channels. The model includes detailed intracellular Ca2+ dynamics, with simulations of localized events such as sarcoplasmic Ca2+ release into a small intracellular volume bounded by the sarcolemma and sarcoplasmic reticulum. Transporter-mediated Ca2+ fluxes from the bulk cytosol are closely matched to the experimentally reported values and predict stimulation rate-dependent changes in Ca2+ transients. Our model reproduces the properties of cardiac myocytes from two different regions of the heart: the apex and the septum. The septum has a relatively prolonged AP, which reflects a relatively small contribution from the rapid transient outward K+ current in the septum. The attribution of putative molecular bases for several of the component currents enables our mouse model to be used to simulate the behavior of genetically modified transgenic mice.

The original paper reference is cited below:

Computer model of action potential of mouse ventricular myocytes, Vladimir E. Bondarenko, Gyula P. Szigeti, Glenna C. L. Bett, Song-Jung Kim, and Randall L. Rasmusson, 2004, *American Journal of Physiology*, 287, H1378-H1403. PubMed ID: 15142845

Model Curation

Curation Status	★★☆
JSim	☆☆☆
COR	★★☆
OpenCell	★★☆

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset 99f4fd680431.

Downloads

- Complete Archive as .tgz
- Download This File

Views available

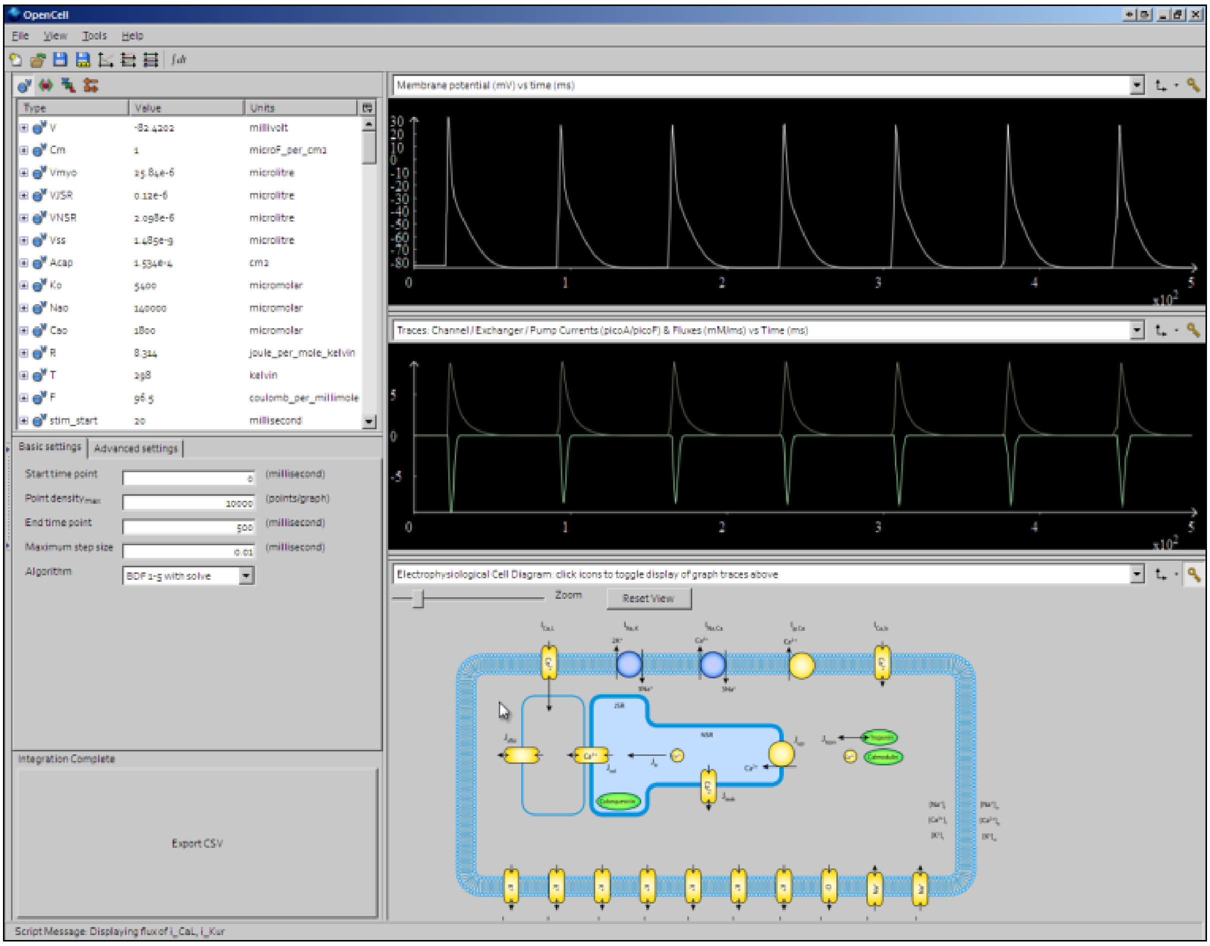
- Documentation
- Model Metadata
- Model Curation
- Mathematics
- Generated Code
- Cite this model
- Source View
- Simulate using OpenCell

License

This work is licensed under a Creative Commons Attribution 3.0 Unported License.

Navigation

Fig. 6.4: An example of a CellML exposure sub-page.


OpenCell
File
View
Tools
Help
Membrane potential (mV) vs time (ms)
30
20
10
0
-10
-20
-30
-40
-50
-60
-70
-80
0
1
2
3
4
5
x10^2
Traces: Channel / Exchanger / Pump Currents (picoA/picoF) & Fluxes (mM/ms) vs Time (ms)
5
0
-5
0
1
2
3
4
5
x10^2
Electrophysiological Cell Diagram: click icons to toggle display of graph traces above
Zoom
Reset View
Basic settings
Advanced settings
Export CSV

Type	Value	Units
V	-82.4202	millivolt
Cm	1	microF_per_cm2
Vmyo	25.84e-6	microlitre
VJSR	0.12e-6	microlitre
VNSR	2.098e-6	microlitre
Vss	1.485e-9	microlitre
Acap	1.534e-4	cm2
Ko	5400	micromolar
Nao	140000	micromolar
Cao	1800	micromolar
R	8.314	joule_per_mole_kelvin
T	298	kelvin
F	96.5	coulomb_per_millimole
stim_start	20	millisecond

Start time point	0	(millisecond)
Point density$_{max}$	10000	(points/graph)
End time point	500	(millisecond)
Maximum step size	0.01	(millisecond)
Algorithm	BDF 1-5 with solve	

Integration Complete

Script Message: Displaying flux of i_CaL, i_Kur

Fig. 6.5: An OpenCell session. Objects such as membrane channels in the diagram can be clicked - this will toggle the graph traces displaying the values for those objects.

PMR concepts

PMR and the CellML model repository use a certain amount of jargon - some is specific to the repository software, and some is related to distributed version control systems (DVCSs). Below are basic explanations of some of these terms as they apply to the repository.

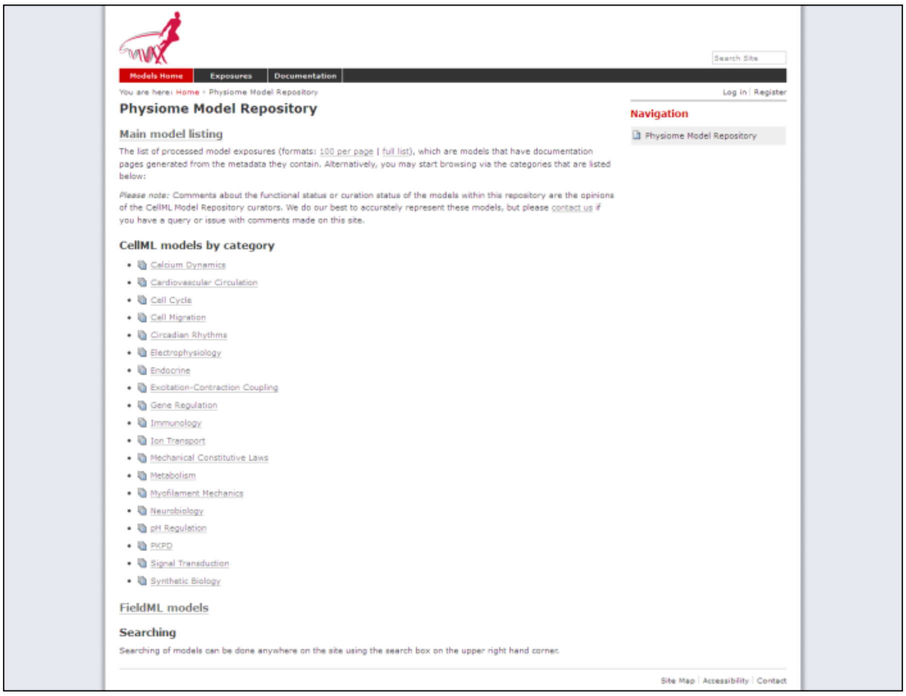
Workspace A container (much like a folder or directory on your computer) to hold the files that make up a model, as well as any other files such as documentation or metadata, etc. In practical terms, each workspace is a Mercurial repository.

Exposure An exposure is a publicly viewable presentation of a particular revision of a model. An exposure can present one or many files from your workspace, along with documentation and other information about your model.

The Mercurial DVCS has a range of terms that are useful to know, and definitions of these terms can be found in the Mercurial glossary: http://mercurial.selenic.com/wiki/Glossary.

Working with the repository web interface

This part of the tutorial will teach you how to find models in the Physiome model repository (http://models.physiomeproject.org), how to view a range of information about those models, and how to download models. The first page in the repository consists of basic navigation, a link to the main model listing, a search box at the top right, and a list of model category links as shown below.



You are here: Home › Physiome Model Repository

Physiome Model Repository

Main model listing

The list of processed model exposures (formats: 100 per page | full list), which are models that have documentation pages generated from the metadata they contain. Alternatively, you may start browsing via the categories that are listed below:

Please note: Comments about the functional status or curation status of the models within this repository are the opinions of the CellML Model Repository curators. We do our best to accurately represent these models, but please contact us if you have a query or issue with comments made on this site.

CellML models by category

- Calcium Dynamics
- Cardiovascular Circulation
- Cell Cycle
- Cell Migration
- Circadian Rhythms
- Electrophysiology
- Endocrine
- Excitation-Contraction Coupling
- Gene Regulation
- Immunology
- Ion Transport
- Mechanical Constitutive Laws
- Metabolism
- Myofilament Mechanics
- Neurobiology
- pH Regulation
- PKPD
- Signal Transduction
- Synthetic Biology

FieldML models

Searching

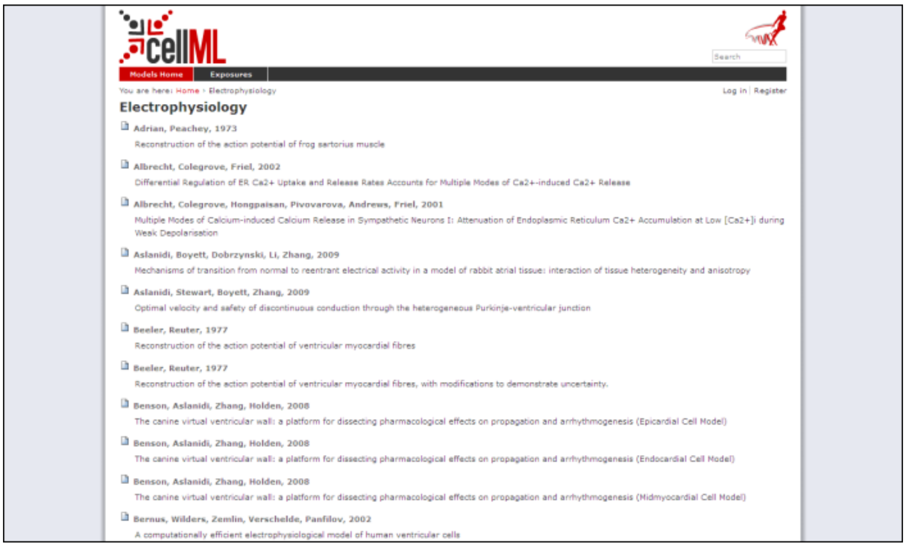
Searching of models can be done anywhere on the site using the search box on the upper right hand corner.

Fig. 6.6: The front page of the Physiome model repository.

Model listings

Clicking on the main model listing or any of the category listings will take you to a page displaying a list of exposed models in that category. Click on electrophysiology for example, and a list of over 100 exposed models in that category will be displayed, as shown here.

Clicking on an item in the list will take you to the exposure page for that model.



You are here: Home › Electrophysiology

Electrophysiology

Adrian, Peachey, 1973
Reconstruction of the action potential of frog sartorius muscle

Albrecht, Colegrove, Friel, 2002
Differential Regulation of ER Ca2+ Uptake and Release Rates Accounts for Multiple Modes of Ca2+-induced Ca2+ Release

Albrecht, Colegrove, Hongpaisan, Pivovarova, Andrews, Friel, 2001
Multiple Modes of Calcium-induced Calcium Release in Sympathetic Neurons I: Attenuation of Endoplasmic Reticulum Ca2+ Accumulation at Low [Ca2+]i during Weak Depolarisation

Aslanidi, Boyett, Dobrzynski, Li, Zhang, 2009
Mechanisms of transition from normal to reentrant electrical activity in a model of rabbit atrial tissue: interaction of tissue heterogeneity and anisotropy

Aslanidi, Stewart, Boyett, Zhang, 2009
Optimal velocity and safety of discontinuous conduction through the heterogeneous Purkinje-ventricular junction

Beeler, Reuter, 1977
Reconstruction of the action potential of ventricular myocardial fibres

Beeler, Reuter, 1977
Reconstruction of the action potential of ventricular myocardial fibres, with modifications to demonstrate uncertainty.

Benson, Aslanidi, Zhang, Holden, 2008
The canine virtual ventricular wall: a platform for dissecting pharmacological effects on propagation and arrhythmogenesis (Epicardial Cell Model)

Benson, Aslanidi, Zhang, Holden, 2008
The canine virtual ventricular wall: a platform for dissecting pharmacological effects on propagation and arrhythmogenesis (Endocardial Cell Model)

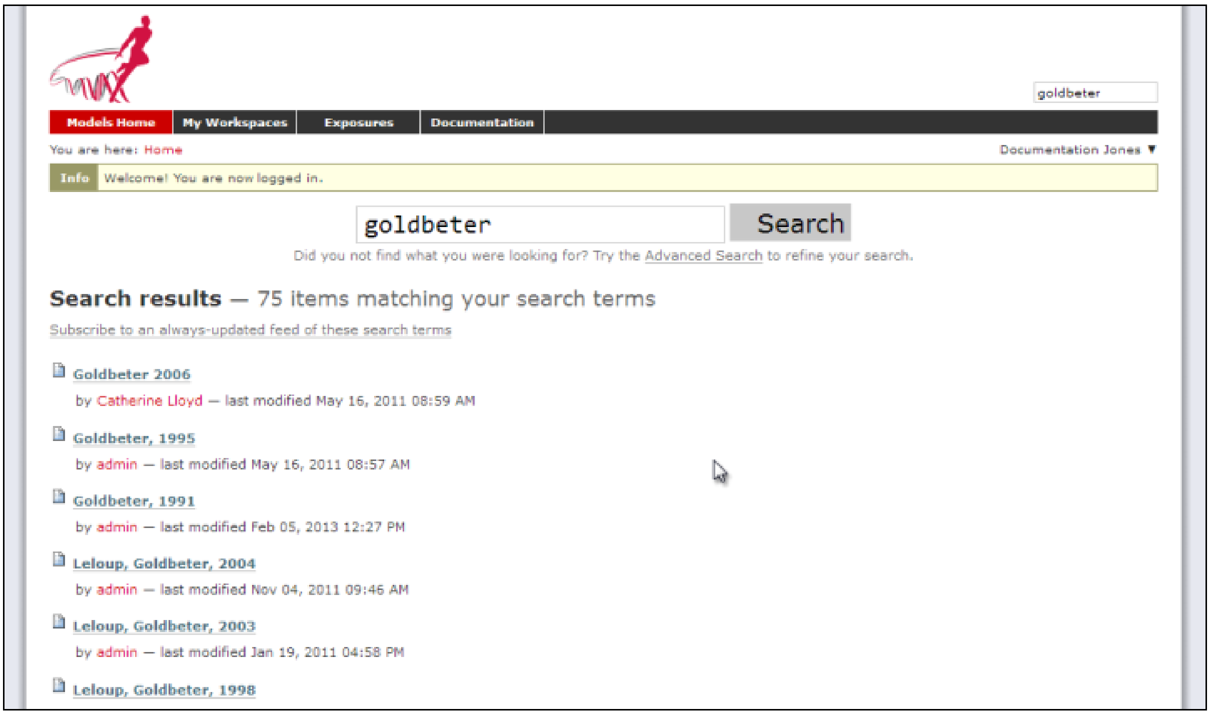
Benson, Aslanidi, Zhang, Holden, 2008
The canine virtual ventricular wall: a platform for dissecting pharmacological effects on propagation and arrhythmogenesis (Midmyocardial Cell Model)

Bernus, Wilders, Zemlin, Verschelde, Panfilov, 2002
A computationally efficient electrophysiological model of human ventricular cells

Fig. 6.7: A list of models in the electrophysiology category.

Searching the repository

You can search for the model that you wish to work on by entering a search term in the box at the top right of the page. Many of the models in the repository are named by the first author and publication date of the paper, so a good search query might be something like *goldbeter 1991*. A list of the results of your search will probably contain both workspaces and exposures - you will need to click on the workspace of the model you wish to work on. Workspaces can be identified because their links are pale blue and have no details line following the clickable link. In the following screenshot, the first two results are workspaces, and the remainder are exposures.



You are here: Home

Info Welcome! You are now logged in.

goldbeter

Did you not find what you were looking for? Try the Advanced Search to refine your search.

Search results — 75 items matching your search terms

Subscribe to an always-updated feed of these search terms

Goldbeter 2006
by Catherine Lloyd — last modified May 16, 2011 08:59 AM

Goldbeter, 1995
by admin — last modified May 16, 2011 08:57 AM

Goldbeter, 1991
by admin — last modified Feb 05, 2013 12:27 PM

Leloup, Goldbeter, 2004
by admin — last modified Nov 04, 2011 09:46 AM

Leloup, Goldbeter, 2003
by admin — last modified Jan 19, 2011 04:58 PM

Leloup, Goldbeter, 1998

Fig. 6.8: A search results listing on the Physiome Model Repository site.

Click on an exposure result to view information about the model and to get links for downloading or simulating the model. Click on workspaces to see the contents of the model workspace and the revision history of the model.

Working with the repository using Mercurial

This part of the tutorial will teach you how to clone a workspace from the model repository using a Mercurial client, create your own workspace, and then push the cloned workspace into your new workspace in the repository. We will be using a *fork* of an existing workspace, which provides you with a personal copy of a workspace that you can edit and push changes to.

Registering an account and logging in

First, navigate to the staging instance of the Physiome model repository at http://184.169.251.126/welcome.

Note: The staging instance of the repository is a mirror of the main repository site found at http://models.physiomeproject.org, running the latest development version of the PMR2 software.

Any changes you make to the contents of the staging instance are not permanent, and will be overwritten with the contents of the main repository whenever the staging instance is upgraded to the latest PMR2 release. For this reason, you can feel free to experiment and make mistakes when pushing to the staging instance.

In order to make changes to models in the CellML repository, you must first register for an account. The *Log in* and *Register* links can be found near the top right corner of the page. Your account will have the appropriate access privileges so that you can push any changes you have made to a model back into the repository.

Click on the Register link near the top right, and fill in the registration form. Enter your username and desired password. After completing the email validation step, you can now log in to the repository.

Note: This username and password are also the credentials you use to interact with the repository via Mercurial.

Once logged in to the repository, you will notice that there is a new link in the navigation bar, My Workspaces. This is where all the workspaces you create later on will be listed. The Log in and Register links are also replaced by your username and a Log out link.

Mercurial username configuration

Important: Username setup for Mercurial

Since you are about to make changes, your name needs to be recorded as part of the workspace revision history. When commit your changes using Mercurial, it is initially “offline” and independent of the central PMR2 instance. This means that you have to set-up your username for the Mercurial client software, even though you have registered a username on the PMR2 site.

You only need to do this once.

Steps for TortoiseHg:

- Right click on any file or folder in Windows Explorer, and select *TortoiseHg* → *Global Settings*.
- Select *Commit* and then enter your name followed by your e-mail address in “angle brackets” (i.e. less-than “<” and greater-than “>”). Actually, you can enter anything you want here, but this is the accepted best practice. Note that this information becomes visible publicly if the PMR2 instance that you push you changes to is public.

Steps for command line:

- **Edit the config text file:**
  - For per repository settings, the file in the repository: `<repo>\.hg\hgrc`
  - System-wide settings for Linux: `%USERPROFILE%\.hgrc`

  - System-wide settings for Windows: `%USERPROFILE%\mercurial.ini`

- Add the following entry:

```
[ui]
username = Firstname Lastname <firstname.lastname@example.net>
```

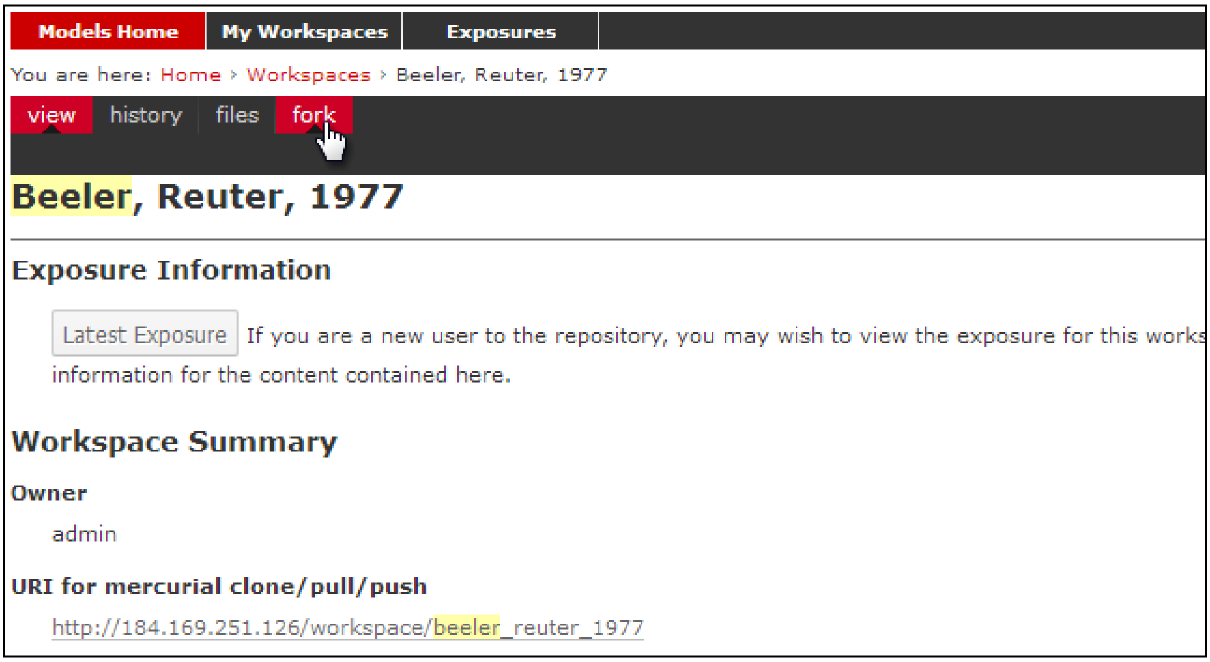
Forking an existing workspace

Important: It is essential to use a Mercurial client to obtain models from the repository for editing. The Mercurial client is not only able to keep track of all the changes you make (allowing you to back-track if you make any errors), but using a Mercurial client is the only way to add any changes you have made back into the repository.

For this tutorial we will *fork* an existing workspace. This creates new workspace owned by you, containing a copy of all the files in the workspace you forked including their complete history. This is equivalent to cloning the workspace, creating a new workspace for yourself, and then pushing the contents of the cloned workspace into your new workspace.

Forking a workspace can be done using the Physiome model repository web interface. The first step is to find the workspace you wish to fork. We will use the Beeler, Reuter 1977 *workspace* which can be found at: http://184.169.251.126/workspace/beeler_reuter_1977.

Now click on the *fork* option in the toolbar, as shown below.



Beeler, Reuter, 1977

Exposure Information

If you are a new user to the repository, you may wish to view the exposure for this works information for the content contained here.

Workspace Summary

Owner

admin

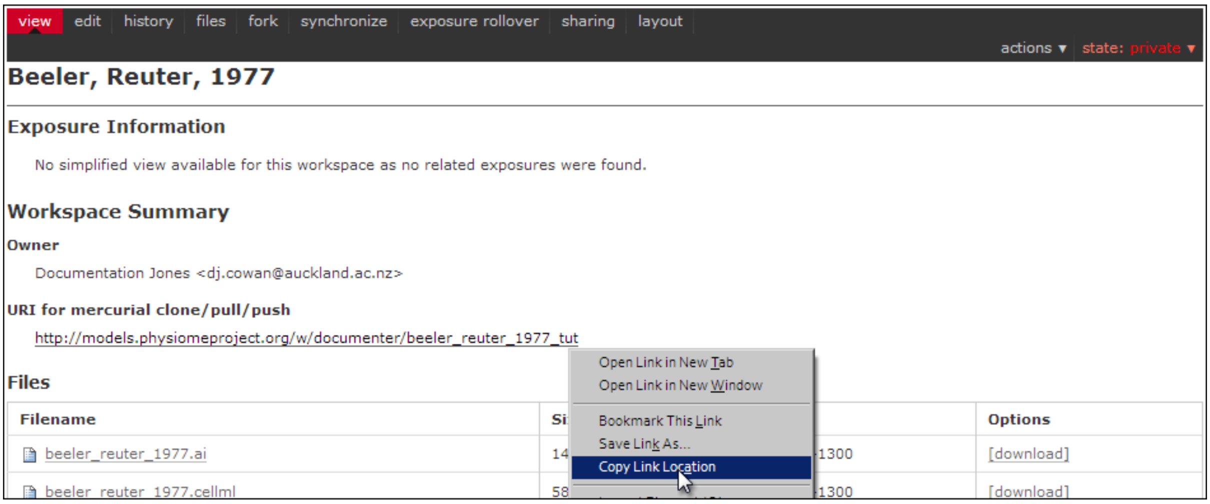
URI for mercurial clone/pull/push

http://184.169.251.126/workspace/beeler_reuter_1977

You will be asked to create a new ID for the workspace. Typically this is something like the existing workspace name plus initials, a text tag that indicates the purpose of the fork, or some other short addition to the original name. Create a fork called `beeler_reuter_1977_username`, for example. You will then be shown the page for your forked workspace.

Cloning your forked workspace

In order to make changes to your workspace, you have to *clone* it to your own computer. In order to do this, copy the URI for mercurial clone/pull/push as shown below:



Beeler, Reuter, 1977

Exposure Information

No simplified view available for this workspace as no related exposures were found.

Workspace Summary

Owner

Documentation Jones <dj.cowan@auckland.ac.nz>

URI for mercurial clone/pull/push

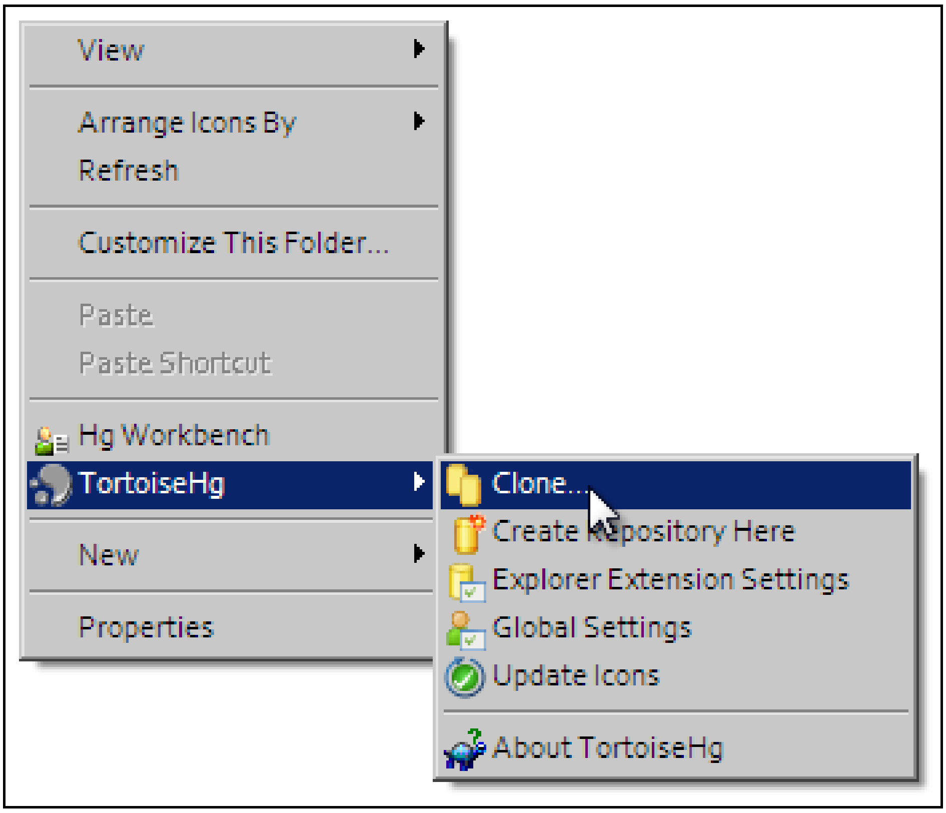
http://models.physiomeproject.org/w/documenter/beeler_reuter_1977_tut

Files

Filename	Si		Options
beeler_reuter_1977.ai	14	1300	[download]
beeler_reuter_1977.cellml	58	1300	[download]

Fig. 6.9: Copying the URI for cloning your workspace.

In Windows explorer, find the folder where you want to create the clone of the workspace. Then right click to bring up the context menu, and select *TortoiseHG* → *Clone* as shown below:



Paste the copied URL into the *Source:* area and then click the *Clone* button. This will create a folder called `beeler_reuter_1977_tut` that contains all the files and history of your forked workspace. The folder will be created inside the folder in which you instigated the clone command.

Command line equivalent

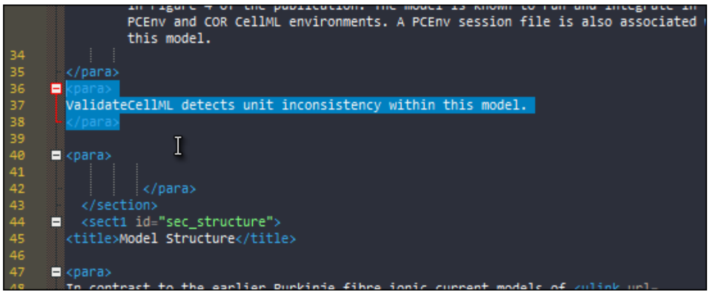
```
hg clone [URI]
```

You will need to enter your username and password to clone the workspace, as the fork will be set to *private* when it is created.

The repository will be cloned within the current directory of your command line window.

Making changes to workspace contents

Your cloned workspace is now ready for you to edit the model file and make a commit each time you want to save the changes you have made. As an example, open the model file in your text editor and remove the paragraph which describes validation errors from the documentation section, as shown below:



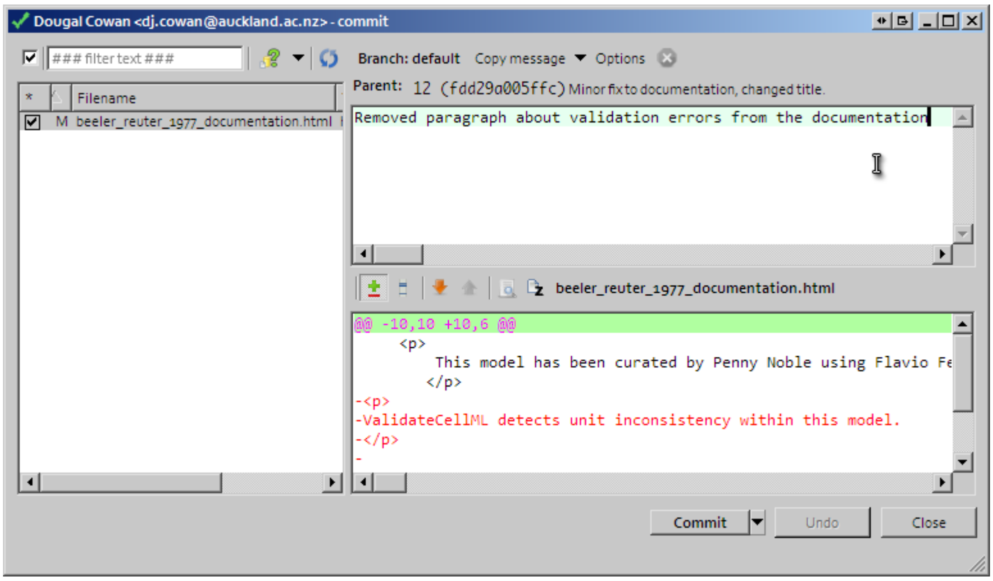
```
        PCEnv and COR CellML environments. A PCEnv session file is also associated
        this model.

</para>
<para>
ValidateCellML detects unit inconsistency within this model.
</para>

<para>

          </para>
  </section>
  <sect1 id="sec_structure">
<title>Model Structure</title>

<para>
```

Save the file. If you are using TortoiseHg, you will notice that the icon overlay has changed to a red exclamation mark. This indicates that the file now has uncommitted changes.

Committing changes

If you are using TortoiseHg, bring up the shell menu for the altered file and select *TortoiseHg* → *Hg Commit*. A window will appear showing details of the changes you are about to commit, and prompting for a commit message. Every time you commit changes, you should enter a useful commit message with information about what changes have been made. In this instance, something like "Removed the paragraph about validation errors from the documentation" is appropriate.

Click on the Commit button at the far left of the toolbar. The icon overlay for the file will now change to a green tick, indicating that changes to the file have been committed.



Dougal Cowan <dj.cowan@auckland.ac.nz> - commit

☑ ### filter text ###

Branch: default

Parent: 12 (fdd29a005ffc) Minor fix to documentation, changed title.

*	Filename
☑	M beeler_reuter_1977_documentation.html

Removed paragraph about validation errors from the documentation

beeler_reuter_1977_documentation.html

```
@@ -10,10 +10,6 @@
     <p>
         This model has been curated by Penny Noble using Flavio Fe
       </p>
-<p>
-ValidateCellML detects unit inconsistency within this model.
-</p>
-
```

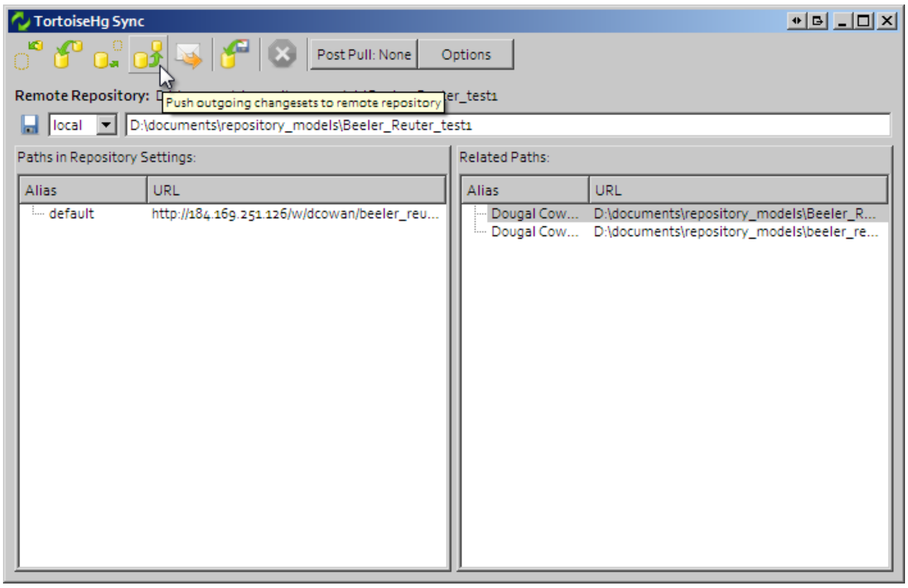
Command line equivalent

```
hg commit -m "Removed the paragraph about validation errors from the documentation"
```

Pushing changes to the repository

Your cloned workspace on your local machine now has a small history of changes which you wish to *push* into the repository.

Right click on your workspace folder in Windows explorer, and select *TortoiseHg* → *Hg Synchronize* from the shell menu. This will bring up a window from which you can manage changes to the workspace in the repository. Click on the Push button in the toolbar, and enter your username and password when prompted.



Remote Repository:

local D:\documents\repository_models\Beeler_Reuter_test1

Paths in Repository Settings:

Alias	URL
default	http://184.169.251.126/w/dcowan/beeler_reu...

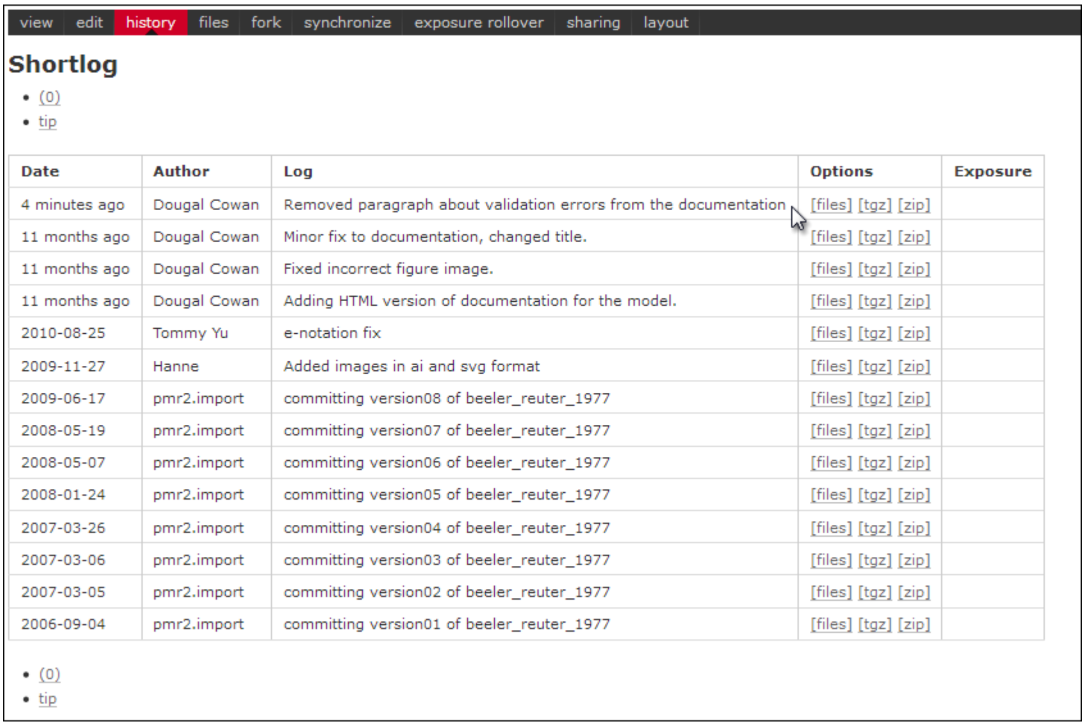
Related Paths:

Alias	URL
Dougal Cow...	D:\documents\repository_models\Beeler_R...
Dougal Cow...	D:\documents\repository_models\beeler_re...

Command line equivalent

```
hg push
```

Now navigate to your workspace and click on the history toolbar button. This will show entries under the Most recent changes, complete with the commit messages you entered for each commit, as shown below:



Shortlog

- (0)
- tip

Date	Author	Log	Options	Exposure
4 minutes ago	Dougal Cowan	Removed paragraph about validation errors from the documentation	[files] [tgz] [zip]	
11 months ago	Dougal Cowan	Minor fix to documentation, changed title.	[files] [tgz] [zip]	
11 months ago	Dougal Cowan	Fixed incorrect figure image.	[files] [tgz] [zip]	
11 months ago	Dougal Cowan	Adding HTML version of documentation for the model.	[files] [tgz] [zip]	
2010-08-25	Tommy Yu	e-notation fix	[files] [tgz] [zip]	
2009-11-27	Hanne	Added images in ai and svg format	[files] [tgz] [zip]	
2009-06-17	pmr2.import	committing version08 of beeler_reuter_1977	[files] [tgz] [zip]	
2008-05-19	pmr2.import	committing version07 of beeler_reuter_1977	[files] [tgz] [zip]	
2008-05-07	pmr2.import	committing version06 of beeler_reuter_1977	[files] [tgz] [zip]	
2008-01-24	pmr2.import	committing version05 of beeler_reuter_1977	[files] [tgz] [zip]	
2007-03-26	pmr2.import	committing version04 of beeler_reuter_1977	[files] [tgz] [zip]	
2007-03-06	pmr2.import	committing version03 of beeler_reuter_1977	[files] [tgz] [zip]	
2007-03-05	pmr2.import	committing version02 of beeler_reuter_1977	[files] [tgz] [zip]	
2006-09-04	pmr2.import	committing version01 of beeler_reuter_1977	[files] [tgz] [zip]	

- (0)
- tip

Create an exposure

As explained earlier, an *exposure* aims to bring a particular revision to the attention of users who are browsing and searching the repository.

There are two ways of making an exposure - creating a new exposure from scratch, or "Rolling over" an exposure. Rolling over is used when a workspace already has an existing exposure, and the updates to the workspace have not fundamentally changed the structure of the workspace. This means that all the information used in making the previous exposure is still valid for making a new exposure of a more recent revision of the workspace. Strictly speaking, an exposure can be rolled over to an older revision as well, but this is not the usual usage.

As you are working in a forked repository, you will need to create a new exposure from scratch. To learn how to create exposures, please refer to *Creating CellML exposures*.

Creating CellML exposures

Section author: Dougal Cowan

CellML models in the Physiome Model Repository are presented through *exposures*. An *exposure* is a view of a particular revision of a workspace, and is quite flexible in terms of what it can present. A workspace may contain one or more models, and any number of models may be presented in a single exposure. Exposures generally take the form of some documentation about the model(s), a range of ways of looking at the model(s) or their metadata, and links to download the model(s).

The example below shows the main exposure page for the Bondarenko *et al.* 2004 workspace. This workspace contains two models, which can be viewed via the *Navigation* pane on the right hand side of the page.

If you click on one of the model navigation links, it will take you to the page for that particular model. Exposures most often present a single model, although they can present any number of models, each with its own documentation and views.

Most of the CellML exposures in the repository are currently of this type, with a main documentation page containing navigation links to the model or models themselves.

The model pages have links that enable the user to do things like view the model equations, look at the citation information, or run the model as an interactive session using the OpenCell application. These links are found in the pane titled *Views available* on the right hand side of the page.

This tutorial contains instructions on how to create one of these standard CellML exposures, as well as information about how to create other alternative types of exposure.

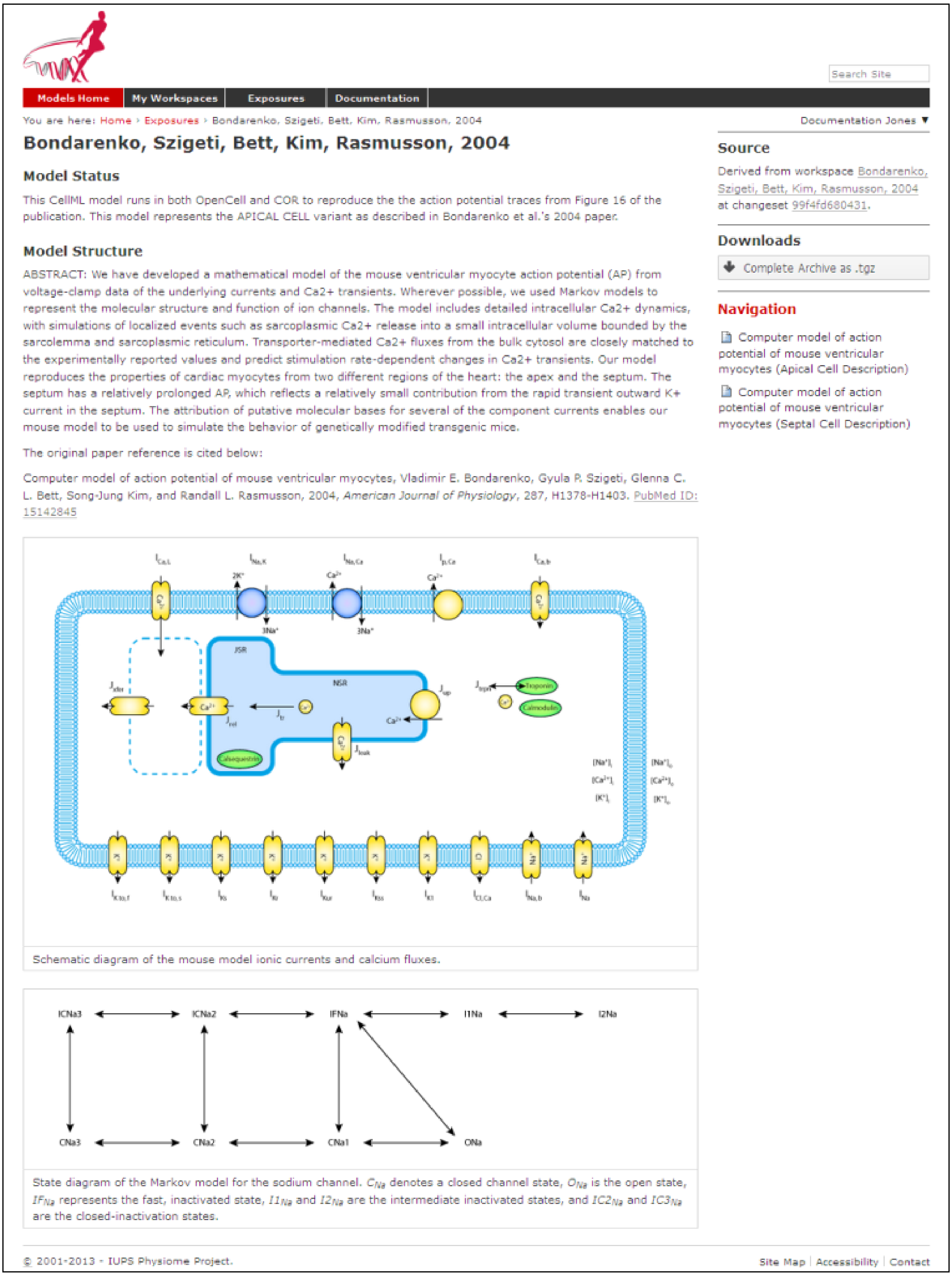
Creating standard CellML exposures

Note: In order to create an exposure of a workspace, the workspace must be *published*. You will either need to submit your workspace for publication and await review. It is not possible to create exposures in private workspaces.

In this example I will use a *fork* of the the Beeler Reuter 1977 workspace. Creating a *fork* of a workspace creates a *clone* of that workspace that you own, and can push changes to. You can *fork* any publicly available workspace in the Physiome model repository. For more information on this feature of PMR, refer to the information on features or collaboration, or see the *relevant section of the tutorial*.

At this point you will need to submit the workspace for publication, using the *state:* menu at the top right of the workspace view page.

You will need to wait for your workspace to be made public before you can carry on and create an exposure of your workspace.



You are here: Home › Exposures › Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004

Documentation Jones ▼

Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004

Model Status

This CellML model runs in both OpenCell and COR to reproduce the the action potential traces from Figure 16 of the publication. This model represents the APICAL CELL variant as described in Bondarenko et al.'s 2004 paper.

Model Structure

ABSTRACT: We have developed a mathematical model of the mouse ventricular myocyte action potential (AP) from voltage-clamp data of the underlying currents and Ca2+ transients. Wherever possible, we used Markov models to represent the molecular structure and function of ion channels. The model includes detailed intracellular Ca2+ dynamics, with simulations of localized events such as sarcoplasmic Ca2+ release into a small intracellular volume bounded by the sarcolemma and sarcoplasmic reticulum. Transporter-mediated Ca2+ fluxes from the bulk cytosol are closely matched to the experimentally reported values and predict stimulation rate-dependent changes in Ca2+ transients. Our model reproduces the properties of cardiac myocytes from two different regions of the heart: the apex and the septum. The septum has a relatively prolonged AP, which reflects a relatively small contribution from the rapid transient outward K+ current in the septum. The attribution of putative molecular bases for several of the component currents enables our mouse model to be used to simulate the behavior of genetically modified transgenic mice.

The original paper reference is cited below:

Computer model of action potential of mouse ventricular myocytes, Vladimir E. Bondarenko, Gyula P. Szigeti, Glenna C. L. Bett, Song-Jung Kim, and Randall L. Rasmusson, 2004, *American Journal of Physiology*, 287, H1378-H1403. PubMed ID: 15142845

Schematic diagram of the mouse model ionic currents and calcium fluxes.

State diagram of the Markov model for the sodium channel. C_{Na} denotes a closed channel state, O_{Na} is the open state, IF_{Na} represents the fast, inactivated state, $I1_{Na}$ and $I2_{Na}$ are the intermediate inactivated states, and $IC2_{Na}$ and $IC3_{Na}$ are the closed-inactivation states.

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset 99f4fd680431.

Downloads

Complete Archive as .tgz

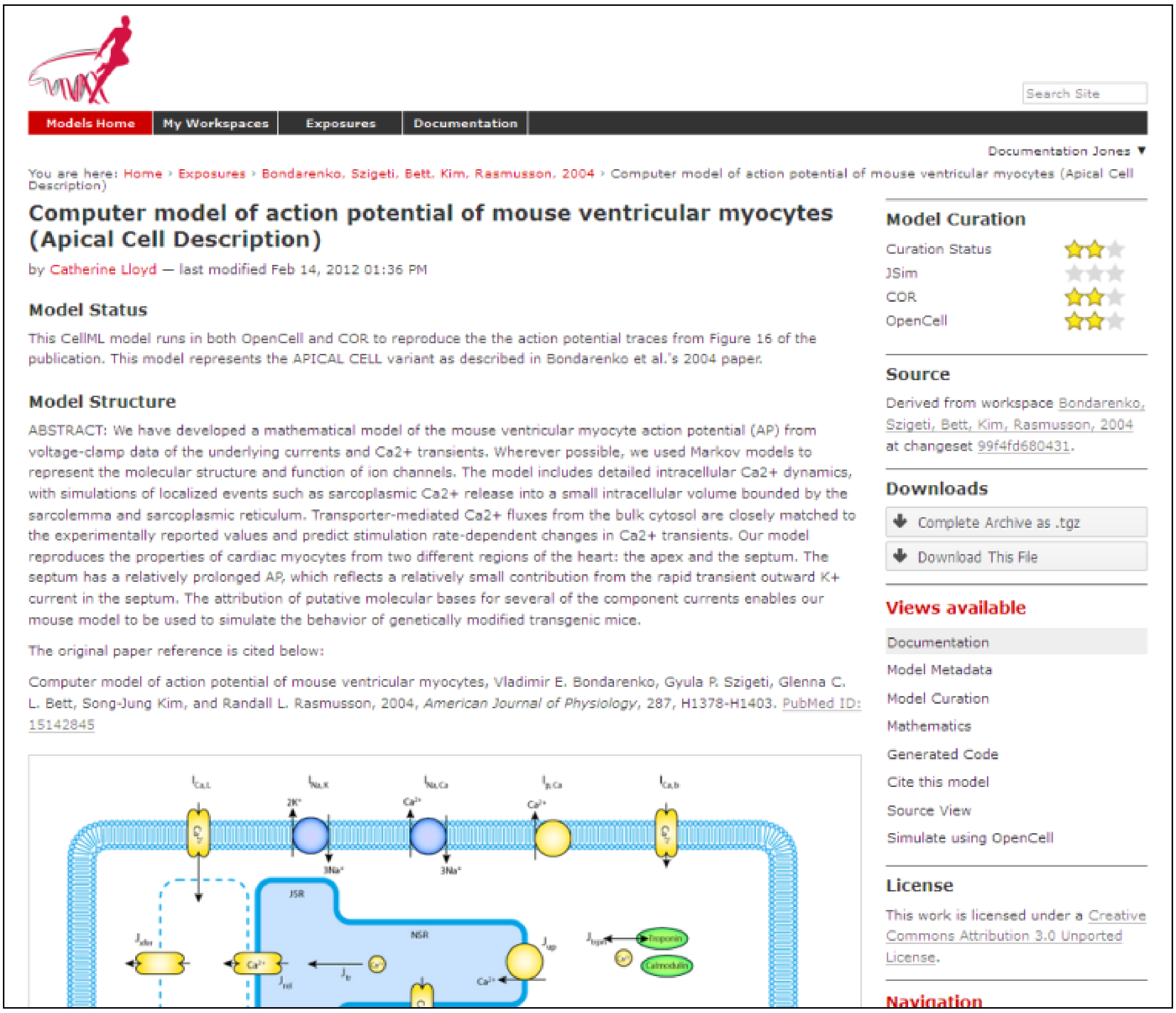
Navigation

- Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)
- Computer model of action potential of mouse ventricular myocytes (Septal Cell Description)

© 2001-2013 - IUPS Physiome Project.

Site Map | Accessibility | Contact

Fig. 6.10: **Example of an exposure page**



You are here: Home › Exposures › Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 › Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

by Catherine Lloyd — last modified Feb 14, 2012 01:36 PM

Model Status

This CellML model runs in both OpenCell and COR to reproduce the the action potential traces from Figure 16 of the publication. This model represents the APICAL CELL variant as described in Bondarenko et al.'s 2004 paper.

Model Structure

ABSTRACT: We have developed a mathematical model of the mouse ventricular myocyte action potential (AP) from voltage-clamp data of the underlying currents and Ca2+ transients. Wherever possible, we used Markov models to represent the molecular structure and function of ion channels. The model includes detailed intracellular Ca2+ dynamics, with simulations of localized events such as sarcoplasmic Ca2+ release into a small intracellular volume bounded by the sarcolemma and sarcoplasmic reticulum. Transporter-mediated Ca2+ fluxes from the bulk cytosol are closely matched to the experimentally reported values and predict stimulation rate-dependent changes in Ca2+ transients. Our model reproduces the properties of cardiac myocytes from two different regions of the heart: the apex and the septum. The septum has a relatively prolonged AP, which reflects a relatively small contribution from the rapid transient outward K+ current in the septum. The attribution of putative molecular bases for several of the component currents enables our mouse model to be used to simulate the behavior of genetically modified transgenic mice.

The original paper reference is cited below:

Computer model of action potential of mouse ventricular myocytes, Vladimir E. Bondarenko, Gyula P. Szigeti, Glenna C. L. Bett, Song-Jung Kim, and Randall L. Rasmusson, 2004, *American Journal of Physiology*, 287, H1378-H1403. PubMed ID: 15142845

Model Curation

Curation Status	★★☆
JSim	☆☆☆
COR	★★☆
OpenCell	★★☆

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset 99f4fd680431.

Downloads

Views available

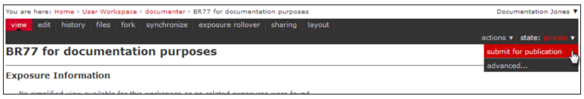
- Documentation
- Model Metadata
- Model Curation
- Mathematics
- Generated Code
- Cite this model
- Source View
- Simulate using OpenCell

License

This work is licensed under a Creative Commons Attribution 3.0 Unported License.

Navigation

Fig. 6.11: **Example of a model exposure page**



You are here: Home › User Workspace › documenter › BR77 for documentation purposes

BR77 for documentation purposes

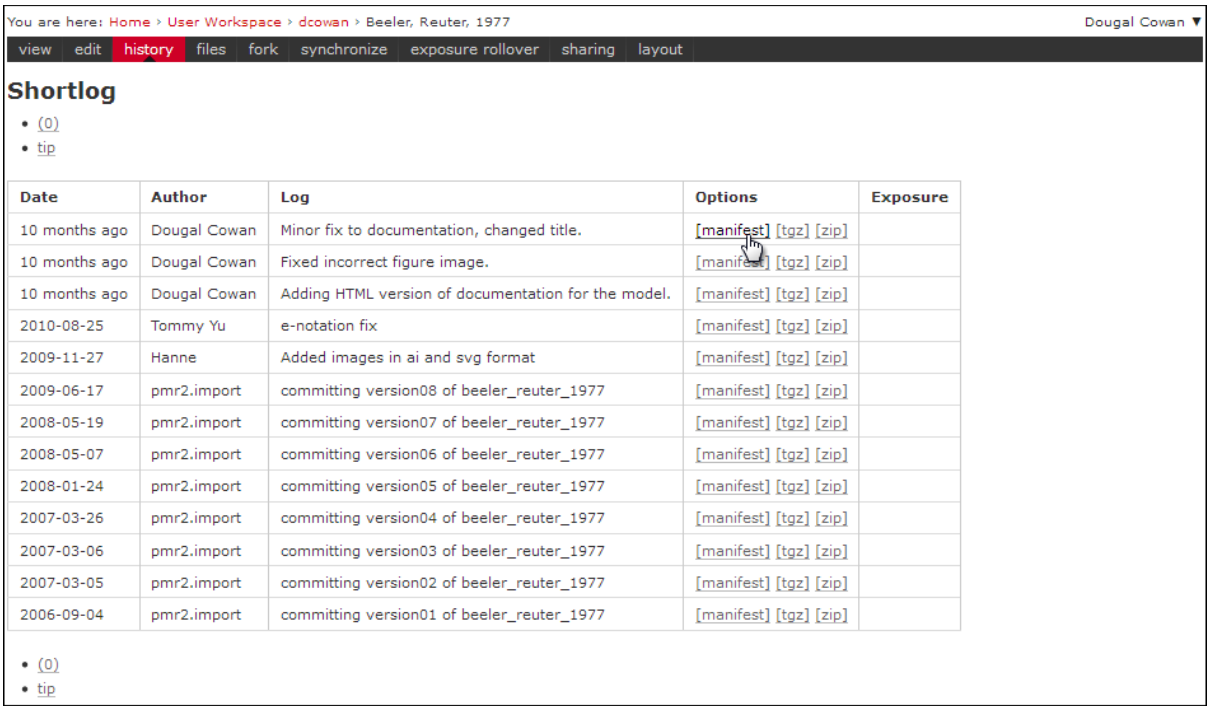
Exposure Information

No simplified view available for this workspace as no related exposures were found.

Fig. 6.12: **The state menu is used to submit objects such as workspaces for publication. Submitted items will be reviewed by site administrators and then published.**

Choose the revision to expose

As an exposure is created to present a particular revision of a workspace, the first thing to do is to navigate to that revision. To do this, first find the workspace - if this is your own workspace, you can click on the *My Workspaces* button in the navigation bar of the repository and find the workspace of interest in the listing displayed. After navigating to your workspace, click on the *history* button in the menu bar.



Shortlog

- (0)
- tip

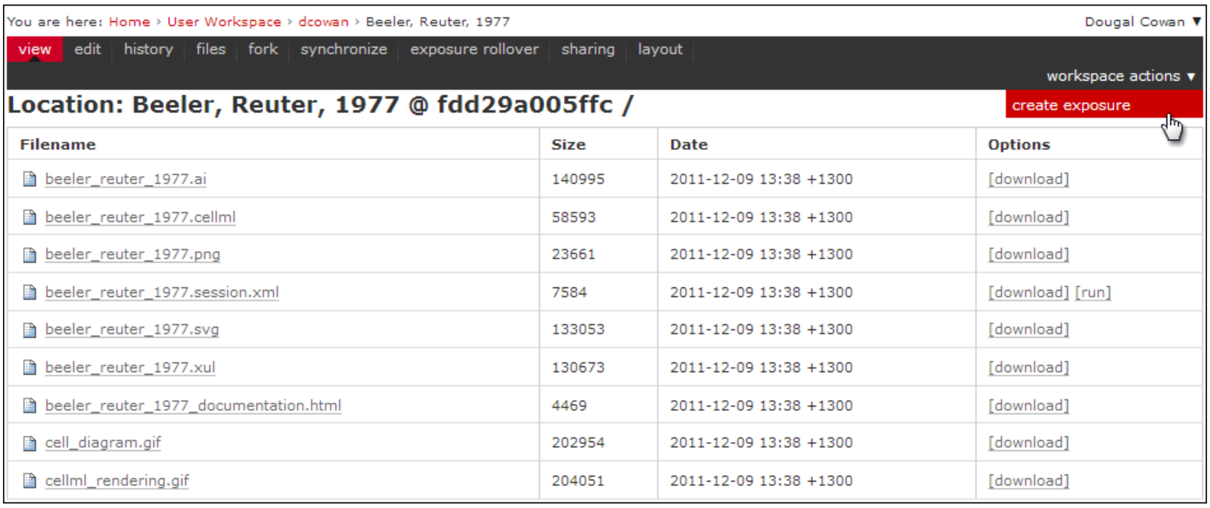
Date	Author	Log	Options	Exposure
10 months ago	Dougal Cowan	Minor fix to documentation, changed title.	[manifest] [tgz] [zip]	
10 months ago	Dougal Cowan	Fixed incorrect figure image.	[manifest] [tgz] [zip]	
10 months ago	Dougal Cowan	Adding HTML version of documentation for the model.	[manifest] [tgz] [zip]	
2010-08-25	Tommy Yu	e-notation fix	[manifest] [tgz] [zip]	
2009-11-27	Hanne	Added images in ai and svg format	[manifest] [tgz] [zip]	
2009-06-17	pmr2.import	committing version08 of beeler_reuter_1977	[manifest] [tgz] [zip]	
2008-05-19	pmr2.import	committing version07 of beeler_reuter_1977	[manifest] [tgz] [zip]	
2008-05-07	pmr2.import	committing version06 of beeler_reuter_1977	[manifest] [tgz] [zip]	
2008-01-24	pmr2.import	committing version05 of beeler_reuter_1977	[manifest] [tgz] [zip]	
2007-03-26	pmr2.import	committing version04 of beeler_reuter_1977	[manifest] [tgz] [zip]	
2007-03-06	pmr2.import	committing version03 of beeler_reuter_1977	[manifest] [tgz] [zip]	
2007-03-05	pmr2.import	committing version02 of beeler_reuter_1977	[manifest] [tgz] [zip]	
2006-09-04	pmr2.import	committing version01 of beeler_reuter_1977	[manifest] [tgz] [zip]	

- (0)
- tip

Fig. 6.13: **The revision history of a fork of the Beeler Reuter 1977 workspace**

Now you can select the revision of the workspace you wish to expose by clicking on the *manifest* of that revision. Usually you will want to expose the latest revision, which appears at the top of the list.

After selecting the revision you wish to expose, click on the *workspace actions* menu at the far right end of the menu bar and select *create exposure*.



Location: Beeler, Reuter, 1977 @ fdd29a005ffc /

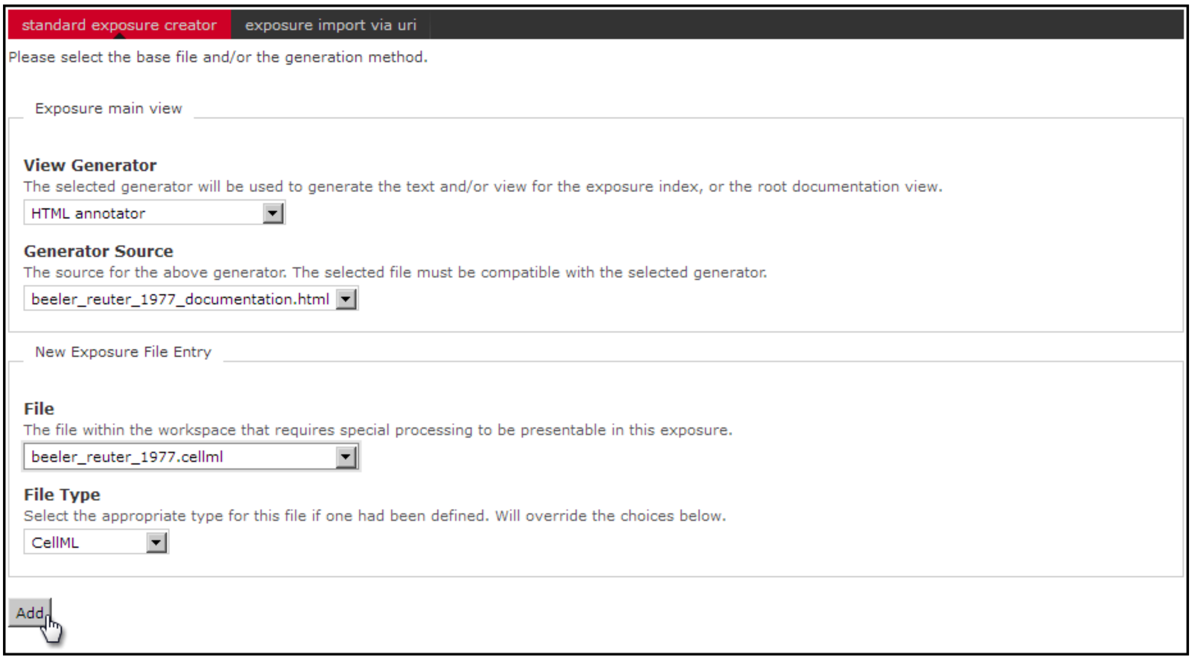
Filename	Size	Date	Options
beeler_reuter_1977.ai	140995	2011-12-09 13:38 +1300	[download]
beeler_reuter_1977.cellml	58593	2011-12-09 13:38 +1300	[download]
beeler_reuter_1977.png	23661	2011-12-09 13:38 +1300	[download]
beeler_reuter_1977.session.xml	7584	2011-12-09 13:38 +1300	[download] [run]
beeler_reuter_1977.svg	133053	2011-12-09 13:38 +1300	[download]
beeler_reuter_1977.xul	130673	2011-12-09 13:38 +1300	[download]
beeler_reuter_1977_documentation.html	4469	2011-12-09 13:38 +1300	[download]
cell_diagram.gif	202954	2011-12-09 13:38 +1300	[download]
cellml_rendering.gif	204051	2011-12-09 13:38 +1300	[download]

Fig. 6.14: **Selecting the manifest of the revision to expose**

Building the exposure

Selecting the *create exposure* option in the menu bar will bring you to the first page of the exposure *wizard*. This web interface allows you to select the model files, documentation files, and settings that will be used to create the exposure.

The initial page of the exposure creation wizard allows you to select the main documentation file and the first model file. Select the HTML annotator option and the HTML documentation file for the workspace in the *Exposure main view* section. For the *New Exposure File Entry* section, choose the CellML file you wish to expose, and select CellML as the file type.



Please select the base file and/or the generation method.

Exposure main view

View Generator
The selected generator will be used to generate the text and/or view for the exposure index, or the root documentation view.
HTML annotator

Generator Source
The source for the above generator. The selected file must be compatible with the selected generator.
beeler_reuter_1977_documentation.html

New Exposure File Entry

File
The file within the workspace that requires special processing to be presentable in this exposure.
beeler_reuter_1977.cellml

File Type
Select the appropriate type for this file if one had been defined. Will override the choices below.
CellML

Fig. 6.15: **Selecting the main documentation and the first CellML model file**

Note: Documentation should be written in HTML format. Some previous users of the CellML repository may be familiar with the tmpdoc style documentation, which has be deprecated. For an example of what a fairly standard HTML documentation file might look like, take a look at the documentation for the Beeler Reuter 1977 model.

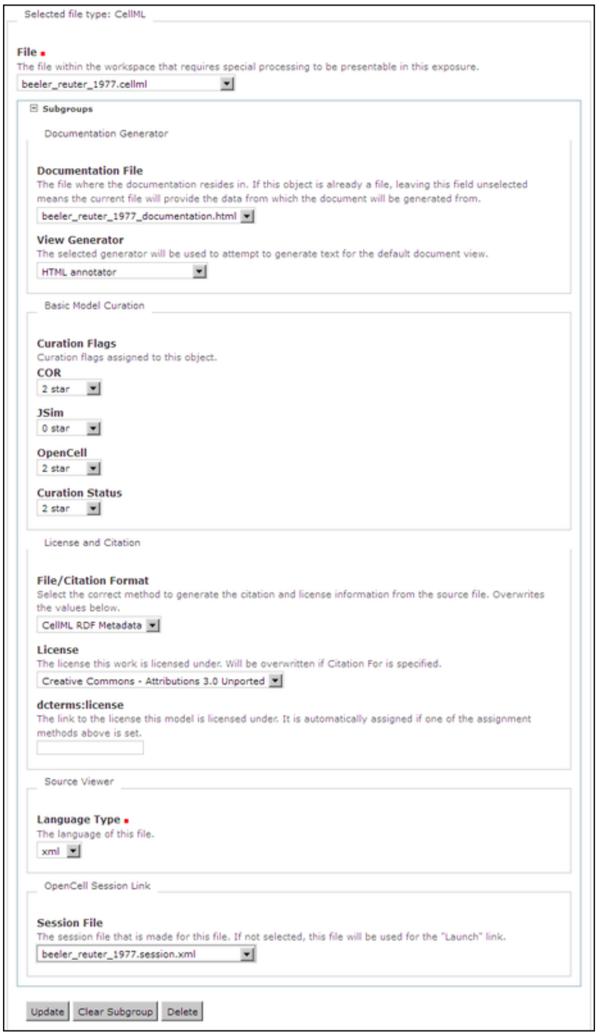
Once you have selected the documentation and model files and their types, click on the *Add* button. This will take you to the next step of the wizard, where you can select various options for the model you have chosen to expose, and will allow you to add further model files to the exposure if desired.

The wizard shows a *subgroup* for each CellML file to be included in the exposure. For each CellML file, select the following options:

- **Documentation**
  - Documentation file - select the HTML file created to document the model
  - View generator - select HTML annotator option
- **Basic Model Curation**
  - Curation flags - CellML model repository curators may select flags according to the status of the model
- **License and Citation**
  - File/Citation format - select CellML RDF metadata to automatically generate a citation page using the model RDF

  - License - select Creative Commons Attributions 3.0 Unported

- **Source Viewer**
  - Language Type - select xml
- **OpenCell Session Link**
  - Session File - select the session.xml if it has been created



Selected file type: CellML

File

The file within the workspace that requires special processing to be presentable in this exposure.

beeler_reuter_1977.cellml

⊟ **Subgroups**

Documentation Generator

Documentation File

The file where the documentation resides in. If this object is already a file, leaving this field unselected means the current file will provide the data from which the document will be generated from.

beeler_reuter_1977_documentation.html

View Generator

The selected generator will be used to attempt to generate text for the default document view.

HTML annotator

Basic Model Curation

Curation Flags

Curation flags assigned to this object.

COR

2 star

JSim

0 star

OpenCell

2 star

Curation Status

2 star

License and Citation

File/Citation Format

Select the correct method to generate the citation and license information from the source file. Overwrites the values below.

CellML RDF Metadata

License

The license this work is licensed under. Will be overwritten if Citation For is specified.

Creative Commons - Attributions 3.0 Unported

dcterms:license

The link to the license this model is licensed under. It is automatically assigned if one of the assignment methods above is set.

Source Viewer

Language Type

The language of this file.

xml

OpenCell Session Link

Session File

The session file that is made for this file. If not selected, this file will be used for the "Launch" link.

beeler_reuter_1977.session.xml

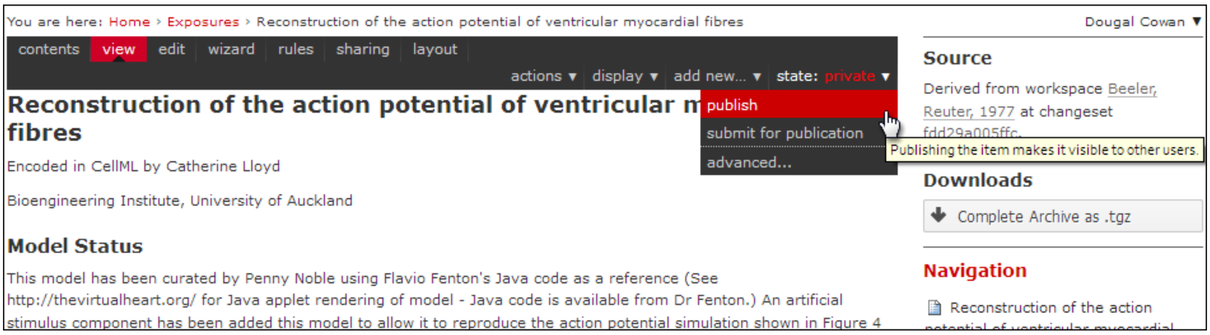
Fig. 6.16: **Selecting options for the model file subgroup**

After selecting the subgroup options, you need to click the *Update* button to set the chosen options for the exposure builder. If you do not update the subgroup, the options you selected will be replaced by the default options when you click *Build*.

For exposures where you wish to expose multiple models, click on the *Add file* button at this stage to create another subgroup. You can then use this to set up all the same options listed above for the additional model file. Remember to click *Update* when you have completed selecting the options for each subgroup before adding another subgroup.

After setting all the options for the models you wish to expose, click on the *Build* button. The repository software will then create the exposure pages and display the main page of the exposure.

In order to make the exposure visible and searchable, you will need to publish it. You can choose to submit your exposure for review, or if you have sufficient privileges you can publish it directly.



Reconstruction of the action potential of ventricular m fibres

Encoded in CellML by Catherine Lloyd

Bioengineering Institute, University of Auckland

Model Status

This model has been curated by Penny Noble using Flavio Fenton's Java code as a reference (See http://thevirtualheart.org/ for Java applet rendering of model - Java code is available from Dr Fenton.) An artificial stimulus component has been added this model to allow it to reproduce the action potential simulation shown in Figure 4

Source

Derived from workspace Beeler, Reuter, 1977 at changeset fdd29a005ffc.

Downloads

Navigation

Reconstruction of the action potential of ventricular myocardial

Fig. 6.17: **Publish your exposure to make it visible to others.**

Other types of exposure

Because the exposure builder uses HTML documentation, it is possible to create customized types of exposure that differ from the standard type shown above. For example, you might want to create an exposure that simply documents and provides links to models in a PMR workspace that are encoded in languages other than CellML. You can also use the HTML documentation to provide tutorials or other documents, with resources stored in the workspace and linked to from the HTML.

Examples of other exposure types:

- Andre's Hodgkin & Huxley CellML tutorial
- Testing nested SED-ML proposals with CellML
- Aslanidi et al. cardiac models encoded in C

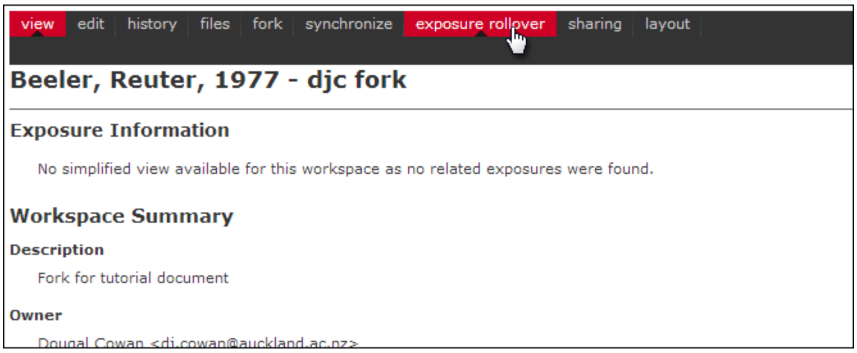
Making an exposure using "roll-over"

As explained earlier, an *exposure* aims to bring a particular revision to the attention of users who are browsing and searching the repository.

"Rolling over" an exposure is the method used when a workspace already has an existing exposure, and the updates to the workspace have not fundamentally changed the structure of the workspace. This means that all the information used in making the previous exposure is still valid for making a new exposure of a more recent revision of the workspace. Strictly speaking, an exposure can be rolled over to an older revision as well, but this is not the usual usage.

Note: A forked workspace contains all of the revision history of the workspace it was created from, but does not contain any of the exposures that existed for the original workspace. You will always need to create an exposure from scratch in newly forked repositories.

From the view page of your workspace, select "exposure rollover".



Beeler, Reuter, 1977 - djc fork

Exposure Information

No simplified view available for this workspace as no related exposures were found.

Workspace Summary

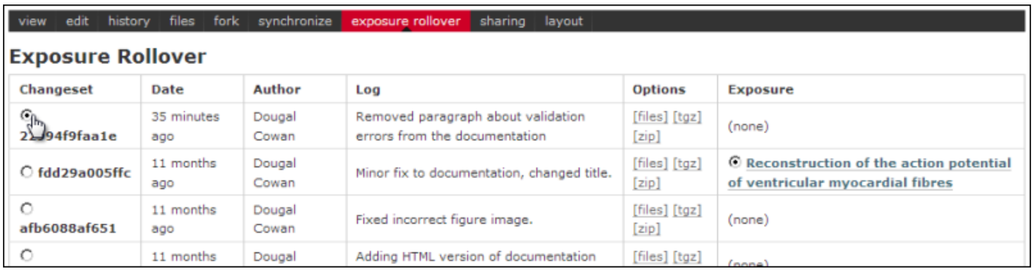
Description

Fork for tutorial document

Owner

Dougal Cowan <dj.cowan@auckland.ac.nz>

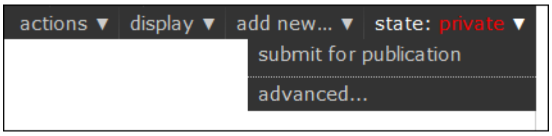
The exposure rollover button takes you to a list of revisions of the workspace, with existing exposures on the right hand side, and revision ids on the left. Each revision id has a radio button, used to select the revision you wish to create a new rolled over exposure for. Each existing exposure also has a radio button, used to select the exposure you wish to base your new one on. The most common use case is to select the latest exposure and the latest revision, and then click the *Migrate* button at the bottom of the list.



Exposure Rollover

Changeset	Date	Author	Log	Options	Exposure
● 2[illegible]94f9faa1e	35 minutes ago	Dougal Cowan	Removed paragraph about validation errors from the documentation	[files] [tgz] [zip]	(none)
○ fdd29a005ffc	11 months ago	Dougal Cowan	Minor fix to documentation, changed title.	[files] [tgz] [zip]	● Reconstruction of the action potential of ventricular myocardial fibres
○ afb6088af651	11 months ago	Dougal Cowan	Fixed incorrect figure image.	[files] [tgz] [zip]	(none)
○	11 months	Dougal	Adding HTML version of documentation	[files] [tgz]	(none)

The new exposure will be created and displayed. When a new exposure is created, it is initially put in the *private* state. This means that only the user who created it or other users with appropriate permissions can see it, and it will not appear in search results or model listings. In order to publish the exposure, you will need to select *submit for publication* from the *state* menu.



The state will change to "pending review". The administrator or curators of the repository will then review and publish the exposure, as well as expiring the old exposure.

Creating FieldML exposures

Section author: Dougal Cowan

FieldML models in the Physiome Model Repository are presented through *exposures*. A FieldML exposure has some similarities to a CellML exposure - usually consisting of a main documentation page with some information about the model, accompanied by a range of different views of the model data and or metadata. FieldML exposures also allow the real-time three-dimensional display of model meshes within the browser through the use of the Zinc plugin.

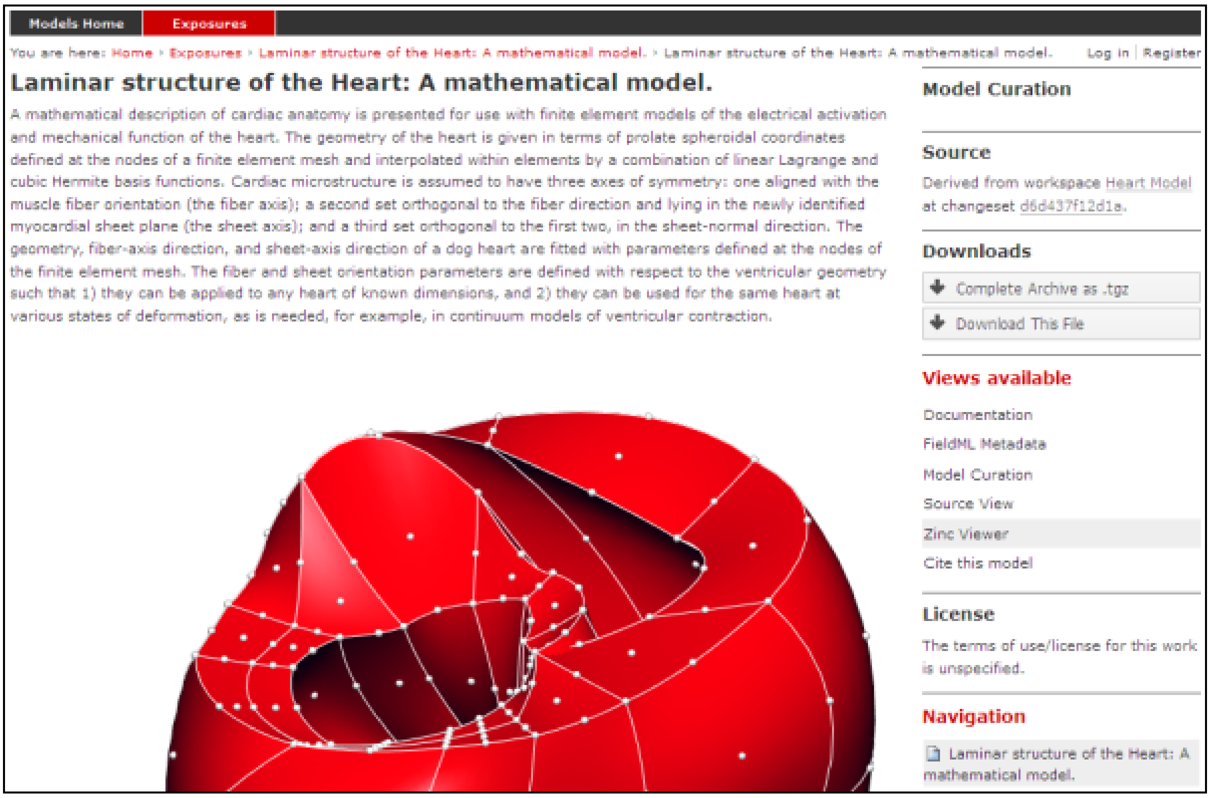
The example screenshots below show the main documentation page view and the 3D visualization provided by the Zinc viewer.

Creating the exposure files

To create a FieldML exposure, the following files will need to be stored in a workspace in PMR:

- The FieldML model file(s)
- An RDF file containing metadata about the model, and specifying the JSON file to be used to specify the visualization.
- The JSON file that specifies the Zinc viewer visualization.
- Optionally, documentation (HTML) and images (PNG, JPG etc).

The following example RDF file from comes from the Laminar Structure of the Heart workspace in the FieldML repository:


Models Home
Exposures
You are here: Home › Exposures › Laminar structure of the Heart: A mathematical model. › Laminar structure of the Heart: A mathematical model.
Log in | Register

Laminar structure of the Heart: A mathematical model.

A mathematical description of cardiac anatomy is presented for use with finite element models of the electrical activation and mechanical function of the heart. The geometry of the heart is given in terms of prolate spheroidal coordinates defined at the nodes of a finite element mesh and interpolated within elements by a combination of linear Lagrange and cubic Hermite basis functions. Cardiac microstructure is assumed to have three axes of symmetry: one aligned with the muscle fiber orientation (the fiber axis); a second set orthogonal to the fiber direction and lying in the newly identified myocardial sheet plane (the sheet axis); and a third set orthogonal to the first two, in the sheet-normal direction. The geometry, fiber-axis direction, and sheet-axis direction of a dog heart are fitted with parameters defined at the nodes of the finite element mesh. The fiber and sheet orientation parameters are defined with respect to the ventricular geometry such that 1) they can be applied to any heart of known dimensions, and 2) they can be used for the same heart at various states of deformation, as is needed, for example, in continuum models of ventricular contraction.

Model Curation

Source

Derived from workspace Heart Model at changeset d6d437f12d1a.

Downloads

Complete Archive as .tgz

Download This File

Views available

Documentation
FieldML Metadata
Model Curation
Source View
Zinc Viewer
Cite this model

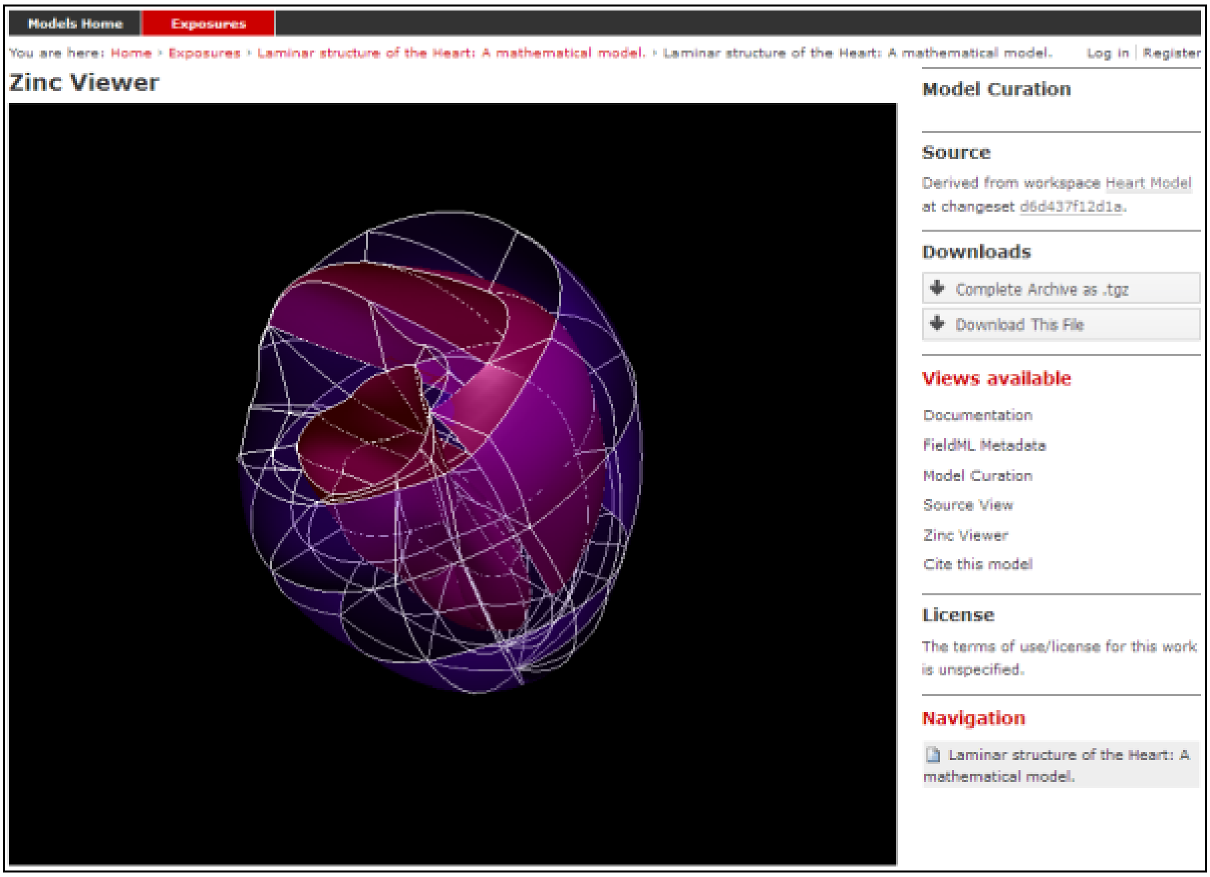
License

The terms of use/license for this work is unspecified.

Navigation

Laminar structure of the Heart: A mathematical model.

Fig. 6.18: The main documentation view of a FieldML exposure


Models Home
Exposures
You are here: Home › Exposures › Laminar structure of the Heart: A mathematical model. › Laminar structure of the Heart: A mathematical model.
Log in | Register

Zinc Viewer

Model Curation

Source

Derived from workspace Heart Model at changeset d6d437f12d1a.

Downloads

Complete Archive as .tgz

Download This File

Views available

Documentation
FieldML Metadata
Model Curation
Source View
Zinc Viewer
Cite this model

License

The terms of use/license for this work is unspecified.

Navigation

Laminar structure of the Heart: A mathematical model.

Fig. 6.19: The main Zinc viewer view of the same FieldML exposure

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <rdf:RDF
3        xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5        xmlns:dc="http://purl.org/dc/elements/1.1/"
6        xmlns:dcterms="http://purl.org/dc/terms/"
7        xmlns:vCard="http://www.w3.org/2001/vcard-rdf/3.0#"
8        xmlns:pmr2="http://namespace.physiomeproject.org/pmr2#">
9     <rdf:Description rdf:about="">
10       <dc:title>
11             Laminar structure of the Heart: A mathematical model.
12       </dc:title>
13       <dc:creator>
14          <rdf:Seq>
15             <rdf:li>LeGrice, I.J.</rdf:li>
16             <rdf:li>Hunter, P.J.</rdf:li>
17             <rdf:li>Smaill, B.H.</rdf:li>
18          </rdf:Seq>
19       </dc:creator>
20       <dcterms:bibliographicCitation>
21             American Journal of Physiology 272: H2466-H2476, 1997.
22       </dcterms:bibliographicCitation>
23       <dcterms:isPartOf rdf:resource="info:pmid/9176318"/>
24       <pmr2:annotation rdf:parseType="Resource">
25          <pmr2:type
26                rdf:resource="http://namespace.physiomeproject.org/pmr2/note#json_
   ↪zinc_viewer"/>
27          <pmr2:fields>
28             <rdf:Bag>
29                <rdf:li rdf:parseType="Resource">
30                   <pmr2:field rdf:parseType="Resource">
31                      <pmr2:key>json</pmr2:key>
32                      <pmr2:value>heart.json</pmr2:value>
33                   </pmr2:field>
34                </rdf:li>
35             </rdf:Bag>
36          </pmr2:fields>
37       </pmr2:annotation>
38    </rdf:Description>
39 </rdf:RDF>
```

This file provides citation metadata and a reference to the resource that specifies the Zinc viewer JSON file which will be used to describe the 3D visualisation of the FieldML model. The file breaks down into three main sections:

- Lines 3-8, namespaces used.
- Lines 10-23, citation metadata.
- Lines 24-37, resource description. Used to specify the JSON file that specifies the visualisation.

Example of the JSON file from the same (Laminar Structure of the Heart) workspace:

```
{
    "View" : [
      {
      "camera" : [9.70448, -288.334, -4.43035],
      "target" : [9.70448, 6.40667, -4.43035],
      "up"     : [-1, 0, 0],
      "angle" : 40
      }
    ],
    "Models": [
        {
            "files": [
13                  "heart.xml"
14              ],
15              "externalresources": [
16                  "heart_mesh.connectivity",
17                  "heart_mesh.node.coordinates"
18              ],
19              "graphics": [
20                  {
21                      "type": "surfaces",
22                      "ambient" : [0.4, 0, 0.9],
23                      "diffuse" : [0.4, 0,0.9],
24                      "alpha" : 0.3,
25                      "xiFace" : "xi3_1",
26                      "coordinatesField": "heart.coordinates"
27                  },
28                  {
29                      "type": "surfaces",
30                      "ambient" : [0.3, 0, 0.3],
31                      "diffuse" : [1, 0, 0],
32                        "specular" : [0.5, 0.5, 0.5],
33                      "shininess" : 0.5,
34                      "xiFace" : "xi3_0",
35                      "coordinatesField" : "heart.coordinates"
36                  },
37                  {
38                      "type": "lines",
39                      "coordinatesField" : "heart.coordinates"
40                  }
41              ],
42              "elementDiscretization" : 8,
43              "region_name" : "heart",
44              "group": "Structures",
45              "label": "heart",
46              "load": true
47          }
48      ]
49  }
```

- Lines 2-8, sets up the camera or viewpoint for the initial Zinc viewer display.
- Lines 12-18, specifies the FieldML model files
- Lines 19-41, set up the actual visualisations of the mesh - in this case, two different surfaces and a set of lines.
- Lines 42-46, specify global visualisation settings.

For more information on these settings, please see the cmgui documentation.

Note: The specifics of these RDF and JSON files are a work in progress, and may change with each new version of the Zinc viewer plugin or the PMR2 software.

Creating the exposure in the Physiome Model Repository

First you will need to create a workspace to put your model in, following the process outlined in the document on working with workspaces.

- Upload your FieldML model files and Zinc viewer specification files.
- Find revision of workspace you wish to expose and create exposure

Exposure wizard procedure

View generator as per CellML; select HTML annotator and HTML doc file

New exposure file entry: select .rdf file and select FieldML (JSON) type. Click *Add*.

Documentation file - same as above Curation flags - none (should be removed?) No other settings

Click *Update*.

Click *Build*.

To see the 3D visualisation, you will need to have the latest Zinc plugin installed.

Using SED-ML to specify simulations

Section author: Dougal Cowan

Hopefully PMR will support SED-ML simulations as part of the CellML views.

Embedded workspaces and their uses

Introduction

Uses

Best practice

CellML Curation in PMR1

As PMR2 contains much of the data ported over from the PMR1 based CellML Model Repository, the curation system from that system was ported to PMR2 verbatim. This document describing the curation aspect of the repository is derived from documentation on the CellML site.

CellML Model Curation: the Theory

The basic measure of curation in a CellML model is described by the curation level of the model document. We have defined four levels of curation:

- Level 0: not curated.
- Level 1: the CellML model is consistent with the mathematics in the original published paper.
- Level 2: the CellML models has been checked for (i) typographical errors, (ii) consistency of units, (iii) that all parameters and initial conditions are defined, (iv) that the model is not over-constrained, in the sense that it contains equations or initial values which are either redundant or inconsistent, and (v) that running the model in an appropriate simulation environment reproduces the results published in the original paper.
- Level 3: the model is checked for the extent to which it satisfies physical constraints such as conservation of mass, momentum, charge, etc. This level of curation needs to be conducted by specialised domain experts.

CellML Model Curation: the Practice

Our ultimate aim is to complete the curation of all the models in the repository, ideally to the level that they replicate the results in the published paper (level 2 curation status). However, we acknowledge that for some models this will not be possible. Missing parameters and equations are just one limitation; at this point it should also be emphasised that the process of curation is not just about "fixing the CellML model" so that it runs in currently available tools. Occasionally it is possible for a model to be expressed in valid CellML, but not yet able to be solved by CellML tools. An example is the seminal Saucerman et al. 2003 model, which contains ODEs as well as a set of non-linear algebraic equations which need to be solved simultaneously. The developers of the CellML editing and simulation environment OpenCell are currently working on addressing these requirements.

The following steps describe the process of curating a CellML model:

- **Step 1:** the model is run through OpenCell and COR. COR in particular is a useful validation tool. It renders the MathML in a human readable format making it much easier to identify any typographical errors in the model equations. COR also provides a comprehensive error messaging system which identifies typographical errors, missing equations and parameters, and any redundancy in the model such as duplicated variables or connections. Once these errors are fixed, and assuming the model is now complete, we compare the CellML model equations with those in the published paper, and if they match, the CellML model is awarded a single star - or level 1 curation status.
- **Step 2:** Assuming the model is able to run in OpenCell and COR, we then go onto compare the CellML model simulation output from COR and OpenCell with the published results. This is often a case of comparing the graphical outputs of the model with the figures in the published paper, and is currently a qualitative process. If the simulation results from the CellML model and the original model match, the CellML model is awarded a second star - or level 2 curation status.
- **Step 3:** if, at the end of this process, the CellML model is still missing parameters or equations, or we are unable to match the simulation results with the published paper, we seek help from the original model author. Where possible, we try to obtain the original model code, and this often plays an invaluable role in fixing the CellML model.
- **Step 4:** Sometimes we have been able to engage the original model author further, such that they take over the responsibility of curating the CellML model themselves. Such models include those published by Mike Cooling and Franc Sachse. In these instances the CellML model is awarded a third star - or level 3 curation status. While this is laudable, ideally we would like to take the curation process one step further, such that level 3 curation should be performed by a domain expert who is not the author of the original publication (i.e., peer review). This expert would then check the CellML model meets the appropriate constraints and expectations for a particular type of model.

A point to note is that levels 1 and 2 of the CellML model curation status may be mutually exclusive - in our experience, it is rare for a paper describing a model to contain no typographical errors or omissions. In this situation, Version 1 of a CellML model usually satisfies curation level 1 in that it reflects the model as it is described in the publication - errors included, while subsequent versions of the CellML model break the requirements for meeting level 1 curation in order to meet the standards of level 2. Taking this idea further, this means that a model with 2 yellow stars doesn't necessarily meet the requirements of level 1 curation but it does meet the requirements of level 2. Hopefully this conflict will be resolved when we replace the current star system with a more meaningful set of curation annotations.

Ultimately, we would like to encourage the scientific modeling community - including model authors, journals and publishing houses - to publish their models in CellML code in the CellML model repository concurrent with the publication of the printed article. This will eliminate the need for code-to-text-to-code translations and thus avoid many of the errors which are introduced during the translation process.

CellML Model Simulation: the Theory and Practice

As part of the process of model curation, it is important to know what tools were used to simulate (run) the model and how well the model runs in a specific simulation environment. In this case, the theory and the practice are essentially the same thing, and carry out a series of simulation steps which then translate into a confidence level as part of a simulator's metadata for each model. The four confidence levels are defined as:

- Level 0: not curated (no stars);
- Level 1: the model loads and runs in the specified simulation environment (1 star);
- Level 2: the model produces results that are qualitatively similar to those previously published for the model (2 stars);
- Level 3: the model has been quantitatively and rigorously verified as producing identical results to the original published model (3 stars).

PMR glossary

Clone Clone is a Mercurial term that means to make a complete copy of a Mercurial repository. This is done in order to have a local copy of a repository to work in.

Embedded workspace A Mercurial concept that allows workspaces to be virtually embedded within other workspaces.

Exposure

Exposures A publicly available page that provides access to and information about a specific revision of a workspace. Exposures are used to publish the contents of workspaces at points in time where the model(s) contained are considered to be useful.

Exposures are created by the PMR software, and offer views appropriate to the type of model being exposed. CellML files for example are presented with options such as code generation and mathematics display, whereas FieldML models might offer a 3D view of the mesh.

Fork A copy of the workspace which includes all the original version history, but is owned by the user who created the fork.

Mercurial Mercurial is a distributed version control system, used by the Physiome Model Repository software to maintain a history of changes to files in *workspaces*.

Synchronize Used to pull the contents or changes from other *Mercurial* repositories into a workspace via a URI.

User folder A folder on the Physiome Model Repository, created automatically when you register your username, which is used to store all of your workspaces.

The purpose of the user folder is to avoid workspace name clashes - users Jane and Bob can both have a workspace called *great_model_1*, for example.

Workspace

Workspaces A *Mercurial* repository hosted on the Physiome Model Repository. This is essentially a folder or directory in which files are stored, with the added feature of being version controlled by the distributed version control system called Mercurial.

Todo

- Update all PMR documentation to reflect workspace ID changes and user workspace changes, if they go ahead.
- Get embedded workspaces doc written.
- Get some best practice docs written.

CHAPTER 7

Documentation to do list

General

Todo

- Add back the MAP client section when the application is ready for testing.
- Update all screenshots to Windows 7 (Or OS-X, or latest Ubuntu, as appropriate).
- Change names of OpenCMISS and cmgui to OpenCMISS Iron and OpenCMISS Zinc.
- Finalise content for About ABI section; avoid overlap with ABI website but provide links.
- Populate CellML API section
- Add many more references (`.. _like-this:`) to docs for cross-referencing.

Within sections

Todo

- Add back the MAP client section when the application is ready for testing.
- Update all screenshots to Windows 7 (Or OS-X, or latest Ubuntu, as appropriate).
- Change names of OpenCMISS and cmgui to OpenCMISS Iron and OpenCMISS Zinc.
- Finalise content for About ABI section; avoid overlap with ABI website but provide links.
- Populate CellML API section
- Add many more references (`.. _like-this:`) to docs for cross-referencing.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/abibook/checkouts/latest/ABIBook-TODO.rst, line 10.)

Todo

- This entire section.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/abibook/checkouts/latest/MAP/index.rst, line 13.)

Todo

- Update all PMR documentation to reflect workspace ID changes and user workspace changes, if they go ahead.
- Get embedded workspaces doc written.
- Get some best practice docs written.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/abibook/checkouts/latest/PMR/index.rst, line 26.)

Todo

Add some useful command examples to command window doc - eg. “saving” graphics

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/abibook/checkouts/latest/cmgui/index.rst, line 34.)

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index