DTP Computational Physiology Documentation

Release 2019.09-alpha

University of Auckland

Sep 12, 2019

Contents:

Welcome to the Computational Physiology module of the Doctoral Training Programme (DTP) from the MedTech CoRE.

Note: DTP Computational Physiology module documentation release 2019.09-alpha. Some previous releases are available at: http://readthedocs.org/projects/dtp-compphys/versions/

Anonymous feedback on any part of this module can be left at: http://goo.gl/forms/0OOhzkwnFo.

CHAPTER 1

Overview

The topics in this section contain information to help you get started to explore a selection of the methods used by scientists in the MedTech CoRE working in the field of computational physiology - the application of engineering and mathematical sciences to the study of physiology.

This series of tutorials will first guide you through the main steps in a “standard” clinical workflow that takes advantage of computational physiology: *image processing*, *model building*, *simulation*, and *visualisation*. You will then be led through the development and implementation of a complete clinically focused workflow which will take advantage of the skills you have developed in the earlier modules. In demonstrating each of these steps we make use of examples for various organ systems being investigated under the MedTech CoRE. Each session will begin with a introductory lecture from a scientist working on the example organ system for that session who will present current research relevant to the MedTech CoRE.

Following the clinical workflow tutorials we venture deeper into the world of standards and repositories to explore their importance in enabling the reproducibility and reusability of computational models as used in the clinical workflow tutorials. Latest research methods in the contruction and management of these computational models will be presented and time devoted to *playing* with these methods in order to emphasise the importance of reuse and reproducibility in the computational physiology domain.

Finally, several mini-projects are offered for students to work through on their own or in groups. These projects will collectively demonstrate the application of all the previous material in *real life* scientific research scenarios.

1.1 Software Tools

Here we introduce the tools used in this module and provide instructions for getting them set up to perform the tasks in this module.

Contents:

1.1.1 MAP Client

The Musculoskeletal Atlas Project Client (MAP Client) is an open-source cross-platform framework for managing workflows. A workflow, as far as MAP Client is concerned, consists of a number of connected workflow steps. The MAP Client framework is a plugin-based application where the plugins are workflow steps. MAP Client is a Python based application which makes use of the Qt widget library.

Documentation for the project can be found at MAP Client documentation.

MAP Client is available to download from MAP Client download

To get the best out of the MAP Client you will need to get some plugins, by default the MAP Client comes with five basic plugins. A collection of available plugins can be found at MAP Client plugins.

1.1.2 OpenCOR

OpenCOR is a very flexible model management, editing, and simulation tool with many features. The full *Tutorial on CellML, OpenCOR & the Physiome Model Repository* tutorial covers a lot of material that is relevant to this tutorial, but before continuing here it is useful to quickly run through the section *Create and run a simple CellML model: editing and simulation* to get a bit of familiarity with the tool.

Instructions for obtaining and using OpenCOR are covered in the *Install and Launch OpenCOR*. If you are using one of the DTP computers, then the correct version of OpenCOR will already be installed, but for the purposes of this module you need to install the *17 March 2018 snapshot*, or newer, verison of OpenCOR.

Connecting OpenCOR to your PMR account

Once you have OpenCOR installed, we want to connect it to PMR to enable you to archive and share your work. If you follow the instructions over at *Using PMR with OpenCOR* to create an account on the teaching instance of the Physiome Model Repository (PMR) and then set up OpenCOR to use that account. You can use this CellML model as the test model for that part of the tutorial.

Here is a video tutorial explaining how to connect the PMR with OpenCOR.

Note: The teaching instance of the repository is a mirror of the main repository site found at https://teaching.physiomeproject.org/, running the latest development version of *PMR2*. User accounts are periodicly synchronised from the main repository, but if you recently created an account on the main site you might need to also create a new account on the teaching instance.

Any changes you make to the contents of the teaching instance are not permanent, and will be overwritten with the contents of the main repository whenever the teaching instance is upgraded to a new release of PMR2. For this reason, you can feel free to experiment and make mistakes when pushing to the teaching instance. Please subscribe to the cellml-discussion mailing list to receive notifications of when the teaching instance will be refreshed.

See the section *Migrating content to the main repository* for instructions on how to migrate any content from the teaching instance to the main (permanent) Auckland Physiome Repository.

Install and Launch OpenCOR

Download OpenCOR from www.opencor.ws. Versions are available for Windows, OS X and Linux[1]. Note that some aspects of this tutorial require OpenCOR snapshot 2017-02-10 (or newer). Create a shortcut to the executable (found in the bin directory) on your desktop and click on this to launch OpenCOR. A window will appear that looks like Fig. 1.1(a).

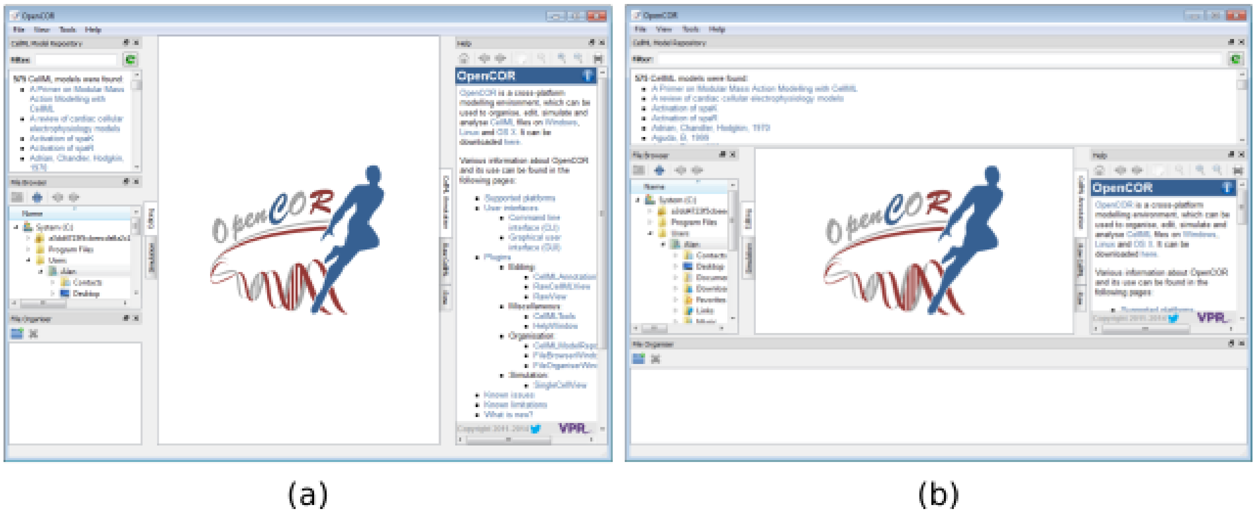


Fig. 1.1: OpenCOR application (a) Default positioning of dockable windows. (b) An alternative configuration achieved by dragging and dropping the dockable windows.

Dockable Windows

The central area is used to interact with files. By default, no files are open, hence the OpenCOR logo is shown instead. To the sides, there are dockable windows, which provide additional features. Those windows can be dragged and dropped to the top or bottom of the central area as shown in Figure 1(b) or they can be individually undocked or closed. All closed panels can be re-displayed by enabling them in the *View* menu, or by using the *Tools* menu *Reset All* option. The key combination `Control-spacebar` removes (for less clutter) or restores these two side panels[2].

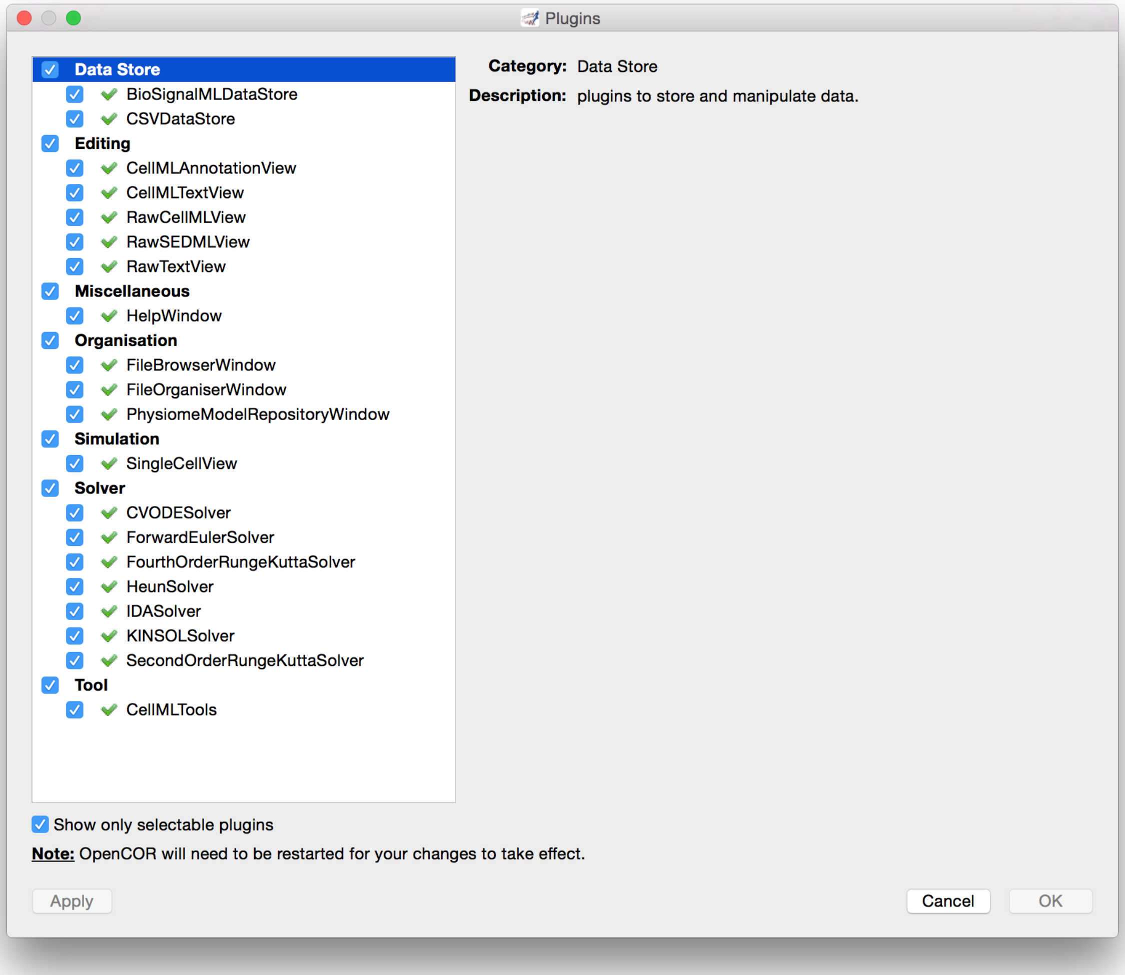
Any of the subpanels (*Physiome Model Repository*, *File Browser*, and *File Organiser*) can be closed with the top right delete button, and then restored from the *View* .. *Windows* .. menu. Files can be dragged and dropped into the File Organiser to create a local directory structure for your files.

Plugins

OpenCOR has a plugin architecture and can be used with or without a range of modules. These can be viewed under the *Tools* menu. By default they are all included, as shown in Fig. 1.2. Information about developing plugins for OpenCOR is also available.

[1] http://opencor.ws/user/supportedPlatforms.html

[2] `-spacebar` being the equivalent on OS X.



- ☑ **Data Store**
  - ☑ ✓ BioSignalMLDataStore
  - ☑ ✓ CSVDataStore
- ☑ **Editing**
  - ☑ ✓ CellMLAnnotationView
  - ☑ ✓ CellMLTextView
  - ☑ ✓ RawCellMLView
  - ☑ ✓ RawSEDMLView
  - ☑ ✓ RawTextView
- ☑ **Miscellaneous**
  - ☑ ✓ HelpWindow
- ☑ **Organisation**
  - ☑ ✓ FileBrowserWindow
  - ☑ ✓ FileOrganiserWindow
  - ☑ ✓ PhysiomeModelRepositoryWindow
- ☑ **Simulation**
  - ☑ ✓ SingleCellView
- ☑ **Solver**
  - ☑ ✓ CVODESolver
  - ☑ ✓ ForwardEulerSolver
  - ☑ ✓ FourthOrderRungeKuttaSolver
  - ☑ ✓ HeunSolver
  - ☑ ✓ IDASolver
  - ☑ ✓ KINSOLSolver
  - ☑ ✓ SecondOrderRungeKuttaSolver
- ☑ **Tool**
  - ☑ ✓ CellMLTools

Category: Data Store

Description: plugins to store and manipulate data.

☑ Show only selectable plugins

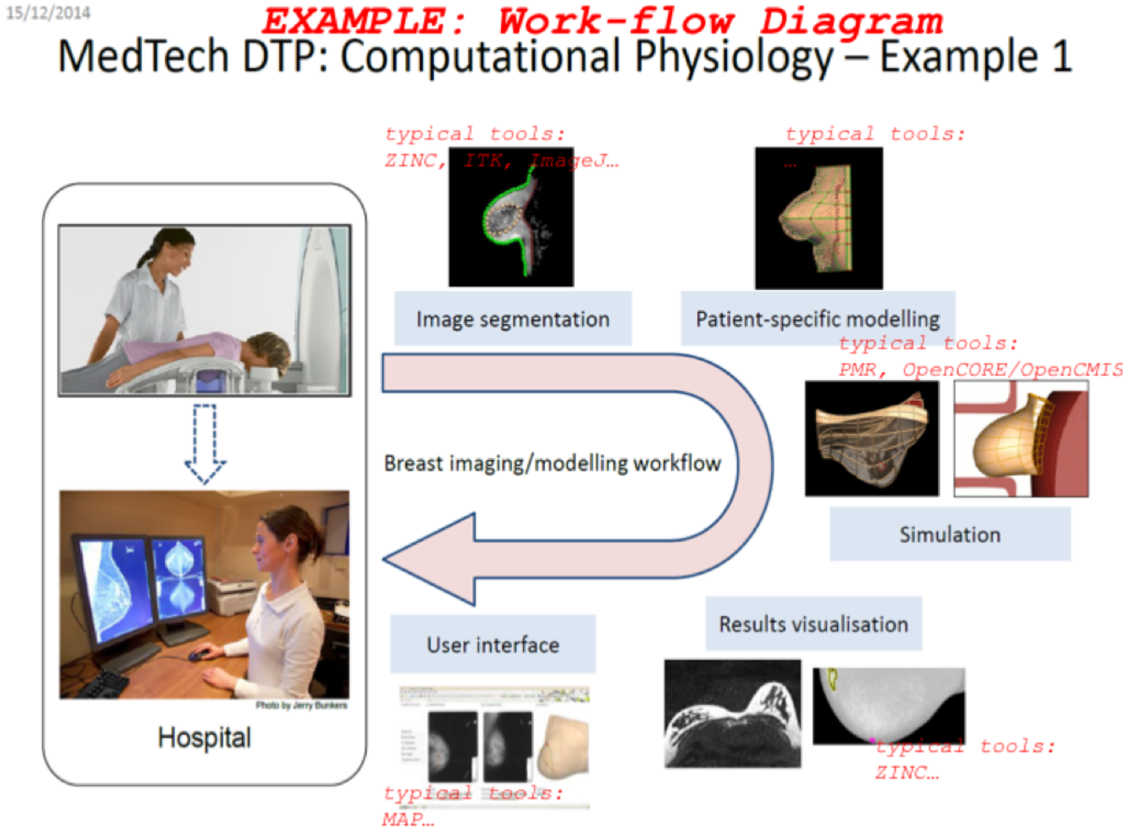
Note: OpenCOR will need to be restarted for your changes to take effect.

Fig. 1.2: OpenCOR tools menu showing the plugins that are selectable. Untick the box on the bottom left to show all plugins.

1.2 Clinical Workflows

In Fig. 1.3 you can see an example of a clinical workflow that adopts a computational physiology approach. In summary:

1. Clinical observations are made (in this example, images from a mamographic scan).
2. A model is customised to those observations (a geometric model specific to the given patient).
3. Simulations are performed as part of the investigation (simulating mamographic compression on the geometrical model to determine the affect on internal structures).
4. Some kind of visualisation is performed to help interpret the simulation results in regard to the clinical observations (registering mamographic and MRI images to highlight potential calcifications in the breast tissue).
5. The preceeding steps are wrapped into a customised user interface which can be provided to the clinicians for them to execute these steps in the clinic.


15/12/2014

EXAMPLE: Work-flow Diagram

MedTech DTP: Computational Physiology – Example 1

Fig. 1.3: An example of a clinical workflow, showing the steps in the process which form the basis for the Computational Physiology module.

The goal of these tutorials is to make you aware of the key skills required at each step in the process described above and in Fig. 1.3. We achieve this here by leading you through some examples of the various skills and demonstrate their applicability across a range of organ systems and spatial scales.

1.2.1 Computational Physiology - Segmentation

Contents:

Image Segmentation

This tutorial is on image segmentation and image data post-processing. The objectives are to gain a basic understanding of the different types of images frequently acquired from medical devices. Understand some of the Digital Imaging and Communications in Medicine (DICOM) standard and the information that is useful to image segmentation. Gain some knowledge on manual and semi-automatic image segmentation and be aware of the reasons for post-processing segmented data. This tutorial includes the following sections:

Contents:

Overview

Segmentation begins with the acquisition of images. The images used in our domain originate from medical devices. The types of images that we come across can be roughly divided into two groups **macroscopic anatomical images** and **microscopic anatomical images**. The macroscopic image modalities that we typically come across are Magnetic Resonance Images (MRI), Computed Tomography (CT) images and Ultrasound (US). The microscopic images that we use are confocal images.

Most medical devices store their image output using the DICOM standard. The **DICOM standard** is a standard that covers the *handling, storing, printing* and *transmitting* of information in medical imaging. It includes a file format definition and a network protocol definition. We shall look at the information that can be stored in the DICOM file format and look at how we can make use of it.

With the information garnered from the DICOM file we will orient and display the images for visualisation, ready for segmentation. The task 1 is to investigate the DICOM headers from a set of tagged MRI images stored using the DICOM standard. In task 2 the segmentation of the images will use a technique called *point-and-click digitisation*, in task 3 we will perform semi-automatic segmentation using edge detection and edge erosion. In task 4 we will perform some post-processing on the segmented data to transform the data from the machine or magnet coordinates to heart coordinates. Finally, we will put it all together to create a segmentation workflow that will produce two point clouds suitable for left ventricle heart model fitting.

Task 1

As mentioned above, task 1 is to investigate the DICOM headers from a set of tagged MR images stored using the DICOM standard. It is quite common to see the DICOM standard used interchangeably with the DICOM image format. *It is important to remember that the DICOM standard is not only for the storing of images*. We will use the MAP Client application and load the 'DTP Image Segmentation Task 1' workflow, with the workflow loaded you should see something like Fig. 1.4.

In this workflow there are *three* steps:

1. The **Image Source** step from which we will load the DICOM images.
2. The **DICOM Header Query** step which will be used to query tags from the DICOM image.

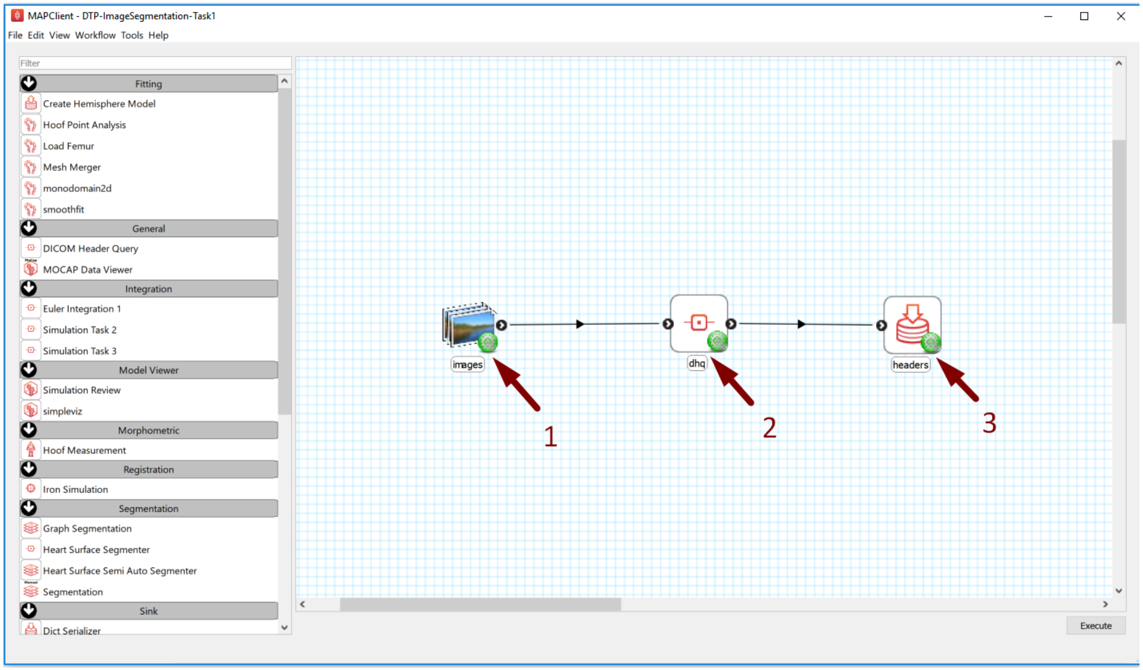


Fig. 1.4: Task 1 workflow

3. The **Dict Serializer** step so that we may store queries for future reference.

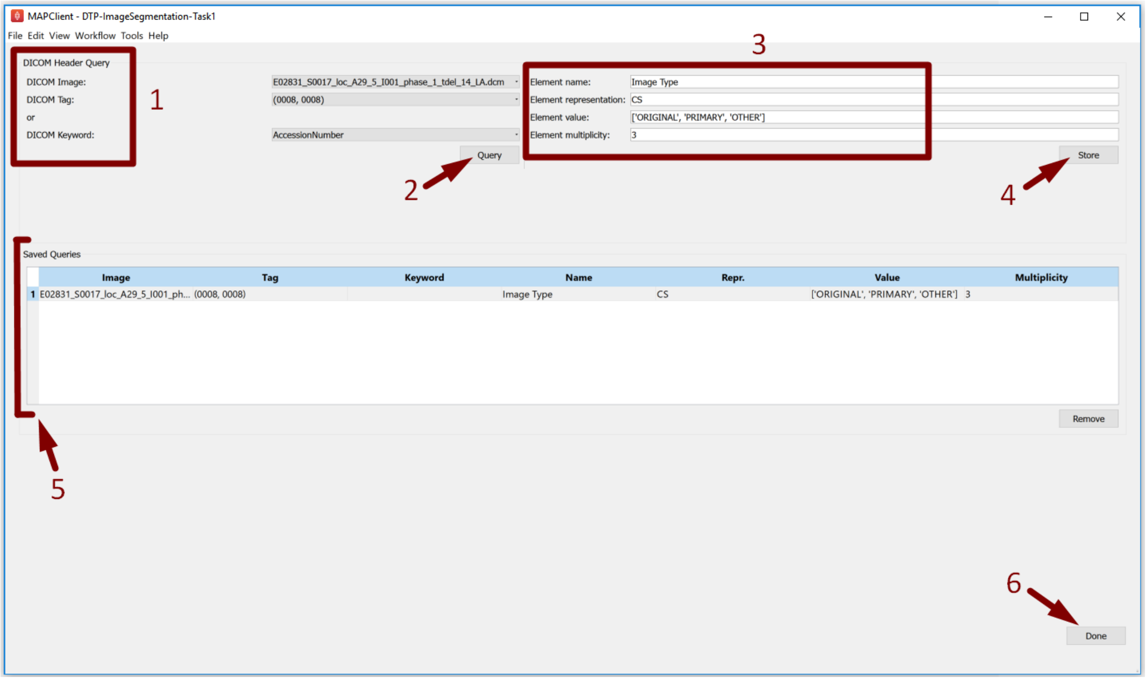
When we execute this workflow we are presented with Fig. 1.5 an interface for querying DICOM headers.

In Fig. 1.5 we can see a number of the GUI elements. On the left of the screen, [1] we can see the `DICOM Image` that we have chosen to query and two combo boxes(*DICOM Tag* and *DICOM Keyword*) below from which we can select one to perform a query with the identified tag. The *Query* button [2] when clicked will query the selected DICOM image with the header tag from the last activated combo box. The results of the query will appear on the right-hand side [3]. On the right-hand side we have four text boxes (*Element name*, *Element representation*, *Element value* and *Element multiplicity*) that will be populated with the result of our query. The `Element name` is related to the DICOM keyword but it is not an exact match, the `Element representation` is defined by the standard and is used to determine the format of the element value. The `Element value` is the actual value of tag queried and the `Element multiplicity` is the number of values in the value element,usually the element multiplicity is one. When the *Store* button [4] is clicked,the result of the query is saved to the saved queries table [5], rows in this table maybe deleted by selecting the row to be deleted with the mouse and clicking the *Remove* button. When the *Done* button [6] is clicked the workflow will finish and return to the workflow edit screen.

The DICOM standard is a rather large and ungainly document freely available on the web, of interest to us here is part three of the standard dealing with Information Object Definitions and part six of the standard dealing with the Data Dictionary in particular table 6-1 which relates Tags, Names, Keywords, Element Representation and Element Multiplicity. If you take a look at table 6–1 you will see that it defines a great number of terms and in any given DICOM file most of these terms will not be defined. What is of interest here are the tags relating to the image position in relation to the patient, position of the patient, the pixel spacing, the size of the image and the image data itself. It is the values taken from these tags that will enable us to correctly orient the images of the patient when we come to segment the left ventricle in task two. Also available are some data regarding the actual patient and study.

To finish this task, see if you can locate the following information:

1. What is the age of the patient?
2. What is the patient position?
3. Which manufacturer built the equipment?


MAPClient - DTP-ImageSegmentation-Task1
File Edit View Workflow Tools Help
1
2
3
4
5
6
Query
Store
Remove
Done

DICOM Header Query

DICOM Image: E02831_S0017_loc_A29_5_I001_phase_1_tdel_14_LA.dcm

DICOM Tag: (0008, 0008)

or

DICOM Keyword: AccessionNumber

Element name: Image Type

Element representation: CS

Element value: ['ORIGINAL', 'PRIMARY', 'OTHER']

Element multiplicity: 3

Saved Queries

	Image	Tag	Keyword	Name	Repr.	Value	Multiplicity
1	E02831_S0017_loc_A29_5_I001_ph...	(0008, 0008)		Image Type	CS	['ORIGINAL', 'PRIMARY', 'OTHER']	3

Fig. 1.5: DICOM Header Query Interface

4. In pixel spacing, what does *DS* in `Element representation` signify?

Task 2

Task two is to segment the left ventricle of the heart. Using the MAP Client application again, load the 'DTP Image Segmentation Task 2' workflow. In this workflow we see five steps - two image source steps,a heart segmentation step and two point cloud serialisation steps. In this workflow we have two image source steps one for the long axis images and another for the short axis images. We will also differentiate between the endocardial surface and the epicardial surface of the heart which will result in two separate point clouds.

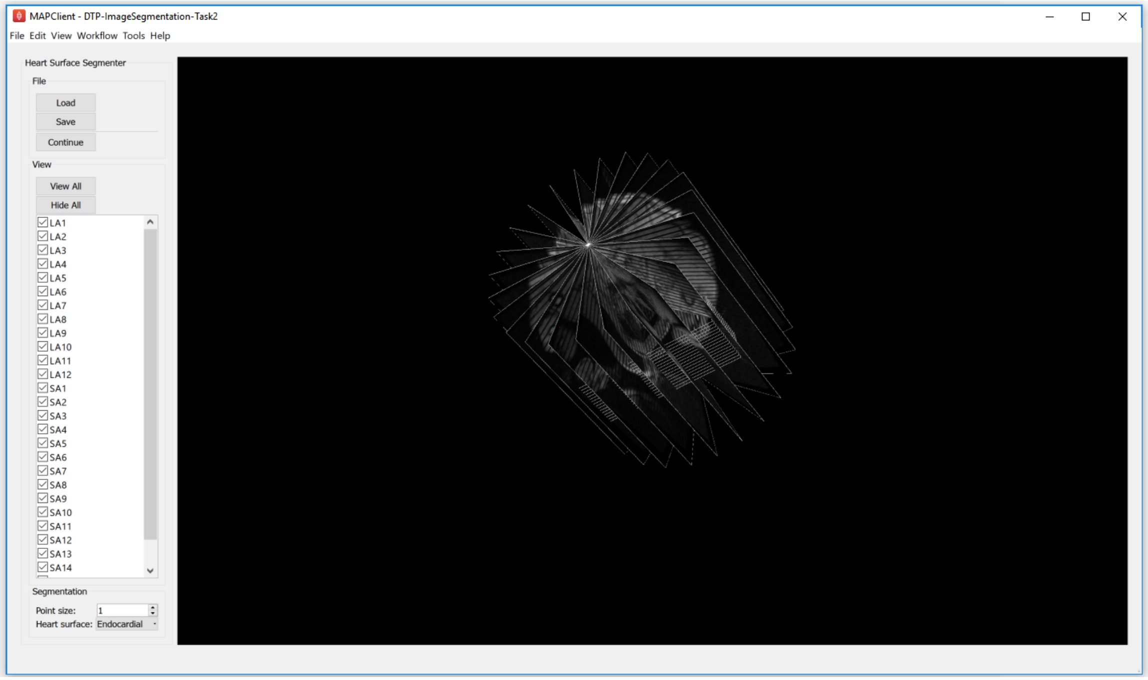
When we execute this workflow we are presented with Fig. 1.6.

In Fig. 1.6 we can see on the left a toolbox that allows us to change the state of this segmentation tool, on the right hand side we can see a three-dimensional view of the two sets of DICOM images. To create this view we have used the the following information from the DICOM header:

- Pixel spacing
- Image orientation patient
- Image position patient
- Rows
- Columns

This has placed each image plane in the machine or magnet coordinate system. In the images we are using, you will see lines across the image picture, this comes from the saturated MR signals so that we can track myocardial motion. In the images that we see we have straight saturated bands indicating that these are the reference images.

From the *View* toolbox on the left-hand side we can show the image planes and from the *File* toolbox we can load and save our progress. The *Continue* button is also in the *File* toolbox for when we are finished segmenting.



Point size: 1

Heart surface: Endocardial

Fig. 1.6: Heart segmentation interface initial state

Using the *View* toolbox first hide all the image planes and then make the 13th short axis image plane visible. You should now be looking at something very similar to Fig. 1.7.

See the *3D View Help* for help on manipulating the view. Move the image plane to a more suitable view for segmentation. We wish to segment both the endocardial and epicardial surfaces of the left ventricle. In the *Segmentation* toolbox we can see which surface of the heart we have set up to segment. In this view the `CTRL` (control) key is used as a modifier for the left mouse button to add segmentation points to the scene. With the left mouse button held down we can drag the segmentation points to the desired location. We can also click on existing segmentation points to adjust their position at a later time. Segmentation points coloured *red* will be put into the *endocardial* set of points, segmentation points coloured *green* will be put into the *epicardial* set of points. Use the *Heart surface* combo box in the *Segmentation* toolbox to change the current point set.

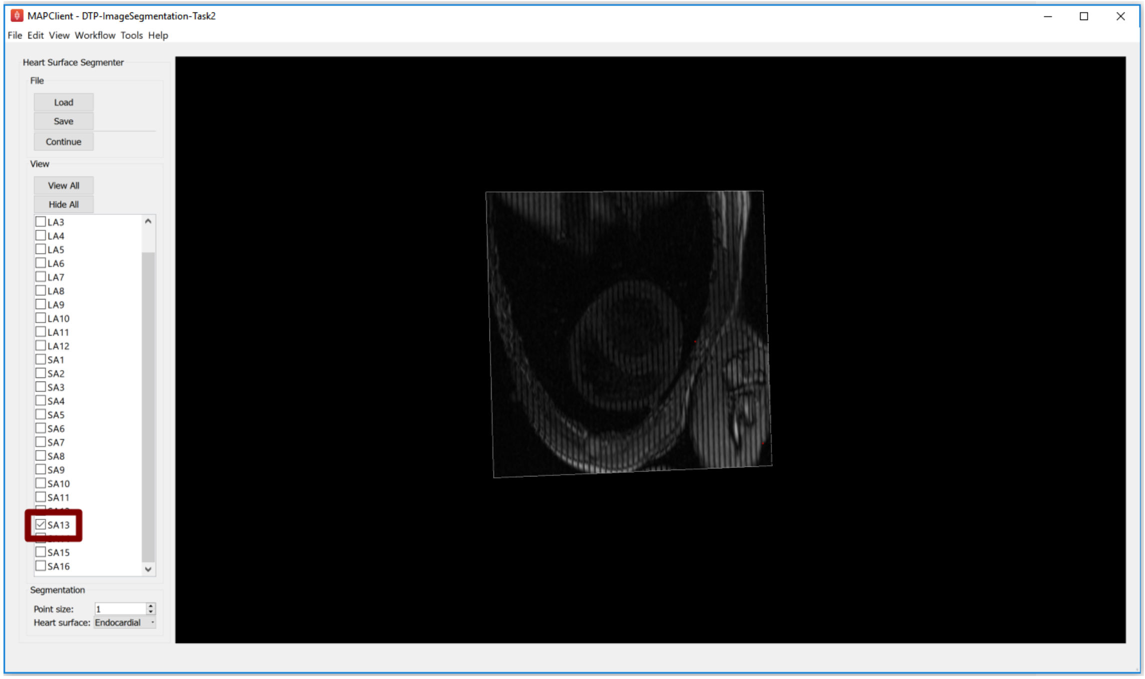
Segmenting this image should result in Fig. 1.8.

Continue segmenting the left ventricle using the long axis images to check for consistency. The end result should look like Fig. 1.9.

Using the *Save* button from the *File* toolbox save your progress and click the *Continue* button to write the two point clouds to disk.

Task 3

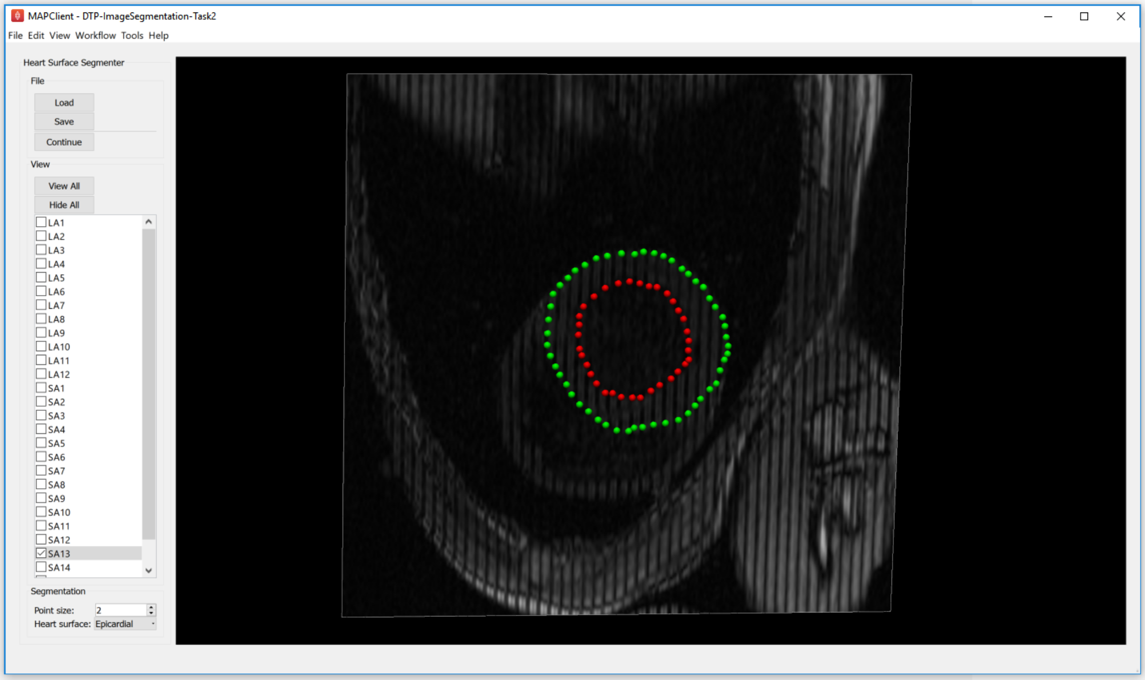
In this task we will use image processing techniques such as edge detection and edge erosion to automatically segment regions of interest. It is often necessary to correct this type of segmentation due to errors in the edge detection or edge errosion process.



Point size: 1

Heart surface: Endocardial

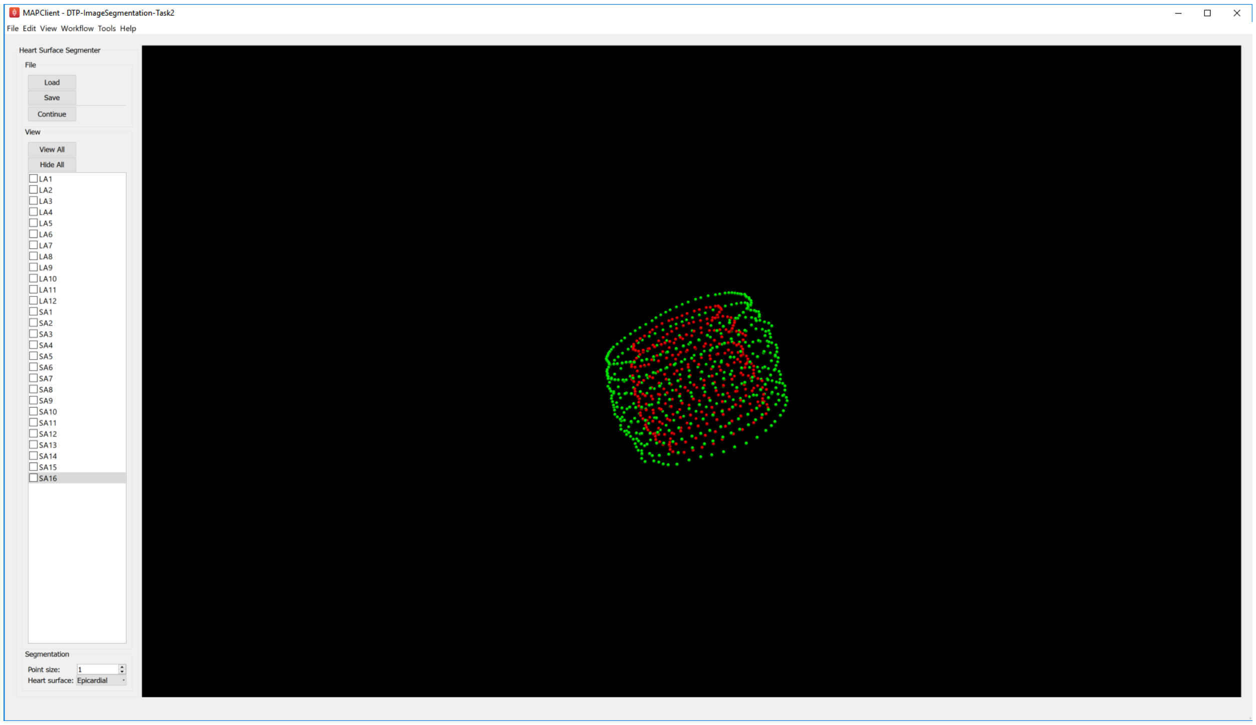
Fig. 1.7: View of the thirteenth short axis image plane



Point size: 2

Heart surface: Epicardial

Fig. 1.8: View of the segmented thirteenth short axis image plane



Point size: 1

Heart surface: Epicardial

Fig. 1.9: View of the segmented left ventricle

Task 4

In this task we want to transform the data created in tasks 2 and 3 from machine coordinates to heart or model coordinates. Open the MAP Client workflow ‘DTP Image Segmentation Task 4’ and execute it. You should see the image planes as before. In this task we need to define the heart coordinate system so that we may contstruct the transformation from machine coordinates to heart coordinates. We can do this by selecting *three* landmark points; the Base point, the Apex point, and the RV point. This will define our heart coordinate system.

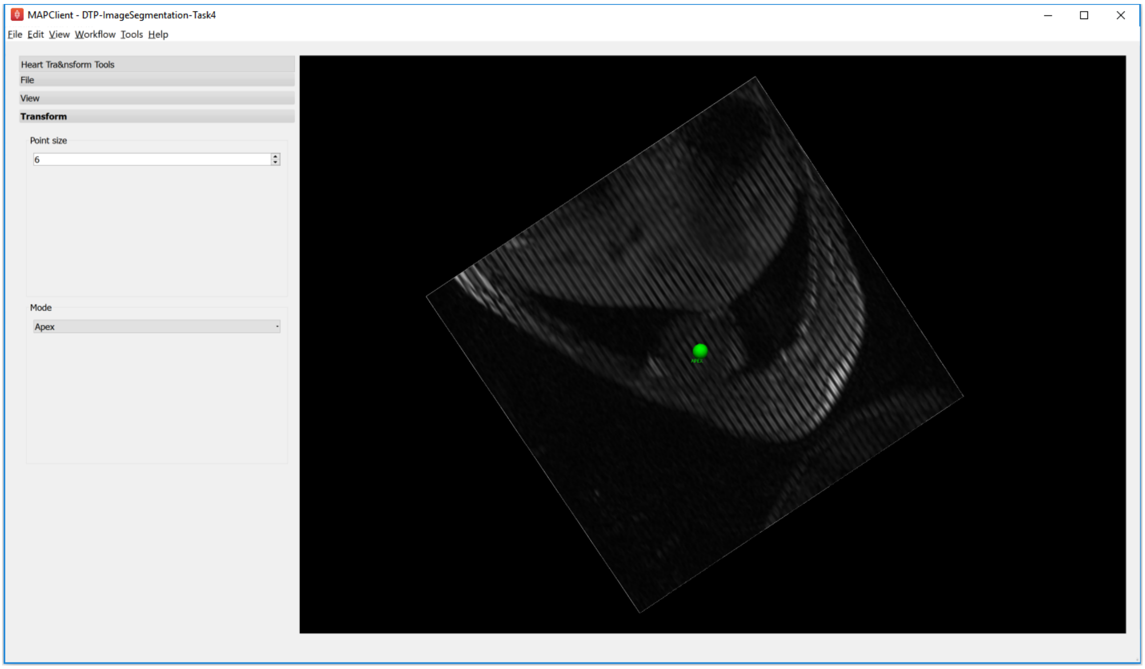
From the *Transform* toolbox we can set the current point we are positioning. Starting with the Apex point find the location at the lower pointed end of the heart which defines the bottom of the left ventricle volume. This can be seen the clearest on the 3rd short axis image plane, Fig. 1.10 shows the apex point.

Make only the 13th short axis image plane visible, on this image plane place the landmarks for the Base point and the RV point(under the *Transform* toolbar). The Base point is the centre of mass of the left ventricle and the RV point is the centre of mass of the right ventricle. See Fig. 1.11 for an example of these locations.

With these three landmarks set, we can determine the heart coordinate system. The origin of this system is one third of the way down the base to apex line. The X axis for the system is increasing from the base point to the apex point the, Y axis is increasing from the base point to the RV point and the cross product of these two vectors defines the Z axis. We make this coordinate system orthogonal by projecting the RV-base line onto the base-apex line.

In Fig. 1.12 we can see an axes glyph to represent the heart coordinate system. This glyph should be consistent with the definition from the previous paragraph.

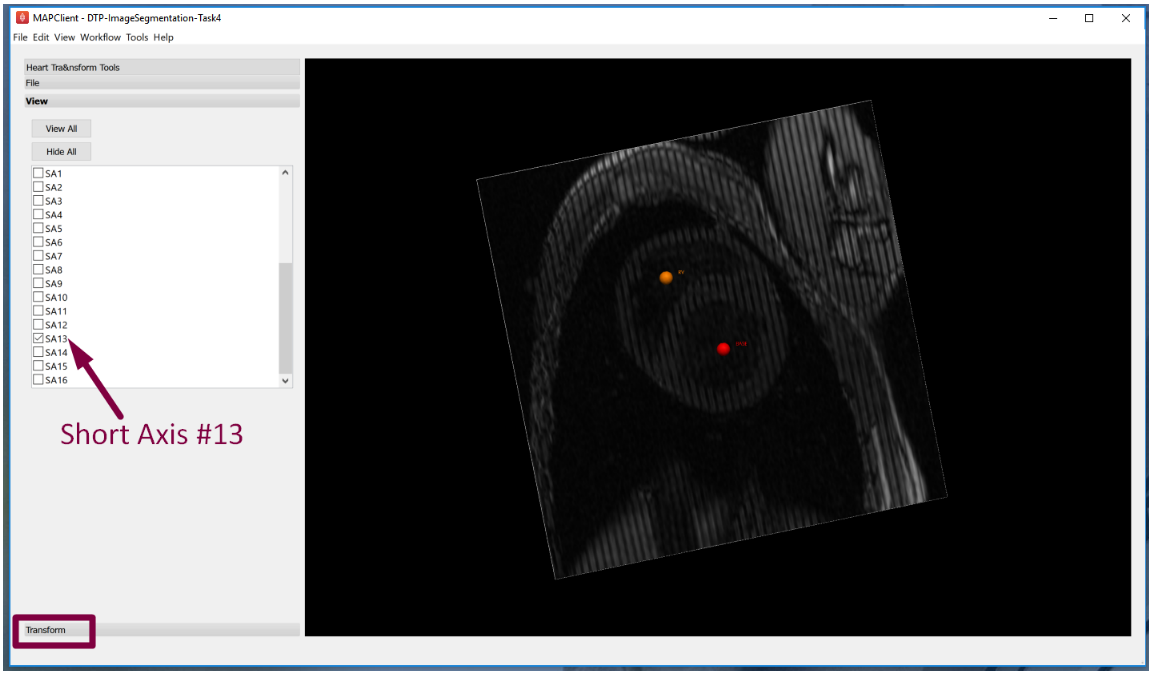
From the *File* toolbox, use the *Save* button to save the location of these points and then click the *Done* button to complete this workflow.


MAPClient - DTP-ImageSegmentation-Task4
File Edit View Workflow Tools Help
Heart Tra&nsform Tools
File
View
Transform

Point size: 6

Mode: Apex

Fig. 1.10: Apex point position in the left ventricle


MAPClient - DTP-ImageSegmentation-Task4
File Edit View Workflow Tools Help
Heart Tra&nsform Tools
File
View
View All
Hide All
Transform
RV

□ SA1
□ SA2
□ SA3
□ SA4
□ SA5
□ SA6
□ SA7
□ SA8
□ SA9
□ SA10
□ SA11
□ SA12
☑ SA13
□ SA14
□ SA15
□ SA16

Short Axis #13

Fig. 1.11: Placement of the Base point and RV point

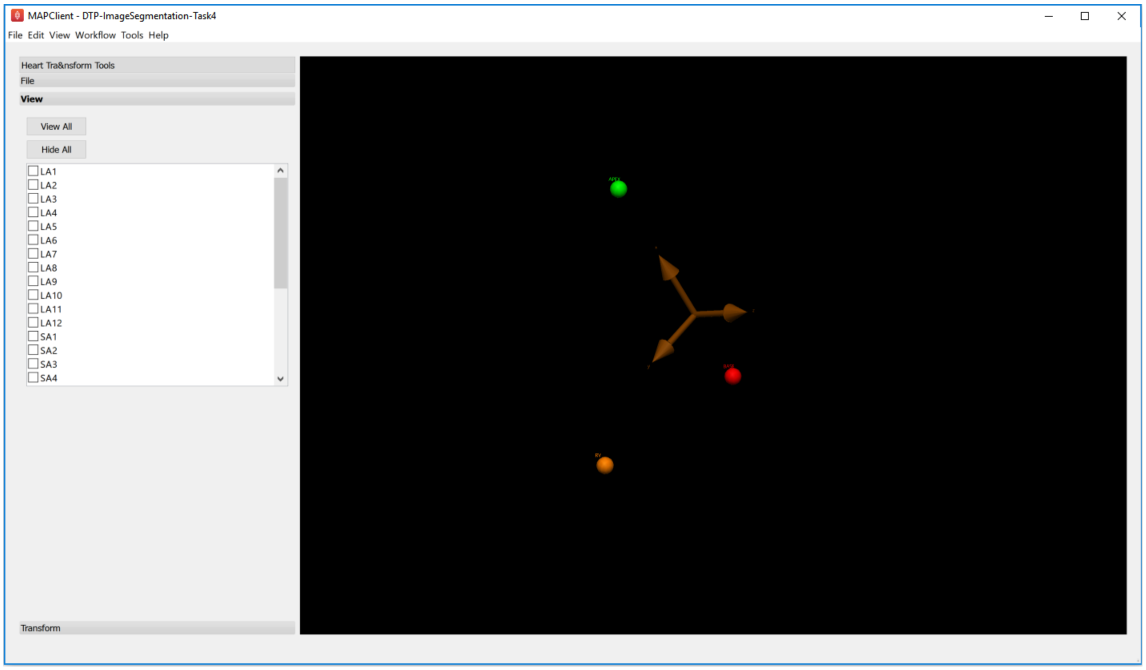


Fig. 1.12: Axes glyph representing the heart coordinate system

Finish

To compete this tutorial we shall put together a complete workflow that will start from DICOM images and result in segmented points of the left ventricle in model coordinates.

3D-View Help

The section describes how you can control the *three-dimensional* view. The three-dimensional view can be manipulated with the mouse and some modifier keys. The following sections describe the effect of the different mouse buttons and modifier keys on manipulating the view.

Left Mouse Button

If you can imagine an invisible sphere around the scene (i.e. everything visible in the view) and then when you use the *left* mouse button and click on a point, the point at which you click is the place where you're setting the starting point for the **rotation** of the sphere. Now when you drag the mouse, the scene is transformed as if you are pulling the sphere around by a piece of string attached to the clicked point.

We can perform a pure *roll* or *yaw* transform of the scene, if we can click outside this invisible sphere (sometimes this is not possible if all of the visible scene is not within the confines of the view) and move the mouse in purely vertical or horizontal manner. In practice, this is quite difficult to do.

Note: By default, there are no modify buttons used in this mode, however, modifier key functionality is often added to suit a particular application.

Middle Mouse Button

The *middle* mouse button can be used to **pan** the scene. Panning the scene results in a translation from the current viewpoint.

Note: By default, there are no modified buttons used in this mode.

Right Mouse Button

The *right* mouse button is used for **zooming**. The zoom applied is what is known as a *fly-by-zoom* where it appears as if the viewer is walking towards or away from the scene. Clicking and dragging down the screen will make it appear as if you are walking away from the scene and clicking and dragging up the screen will make it appear as if you are walking towards the scene.

Note: Using the *shift* key modifier we can change the type zoom. The shift key changes the zoom to a telephoto lens type of zoom in this mode it is as if the viewer is standing still and using the zoom on a camera or pair of binoculars to change the view.

1.2.2 Computational Physiology - Model Construction

Contents:

Model Construction

Model construction consists of:

1. **Mesh Creation**: creating a finite element mesh with a topology and interpolation suited to describing the body of interest to the level of detail required.
2. **Geometric Fitting**: customising the geometry of the mesh to be physically realistic, or tailored to an individual.
3. **Material Field Definition**: describing and fitting additional spatially-varying fields which affect behaviour of the body e.g. material properties such as fibre orientations for anisotropic materials.

Contents:

The central part of this tutorial fits models to breast surface data. This is a very practical application, since customising breast models to individuals is needed to simulate deformation with change in pose, and this in turn helps co-locate regions of possible cancerous tissue from multiple medical images each made with different imaging devices which necessarily use different body poses.

Mesh Creation

We are concerned with constructing models consisting of ‘finite elements’ – simple shapes such as triangles, squares, cubes, etc. - which join together to form a ‘mesh’ which covers the body and describes its topology. Over the elements of the mesh we interpolate coordinates (and eventually other fields of interest) to give the model its 3-D shape and location. Usually mesh creation involves creating the elements and specifying at least initial coordinates for the model.

Common methods for creating a finite element mesh include:

1. Automatic mesh generation to boundaries described by segmented edges, point clouds or CAD models. Automated algorithms are usually limited to creating triangle (in 2-D) or tetrahedron (3-D) elements.
2. Generating part or all the topology from simple, standard shapes such as plates, blocks and tubes, connecting them together then fitting to geometric data. The examples below use this approach.
3. Manually building elements by selecting/creating corner points (‘nodes’) in the correct order. To ease creating a valid model, tools can assist by locking on to points from a surface or point cloud segmented from real medical images.

From a simpler mesh, more complicated models can be created. Surface models can be automatically extruded to 3-D models (triangles become wedges, squares become cubes). Also, different, higher-order basis functions can be used over the same topology to give more degrees of freedom in the model. This is particularly important for studying the human body as it contains few straight lines. In many models of body parts we employ C_1-continuous cubic Hermite basis (interpolation) functions to model their inherent smoothness, and the following examples demonstrate this. There is a downside to using Hermite bases: very careful adjustments are needed to properly connect the mesh in areas where the topology is non-trivial, such as where rounded bodies are closed (top of head, apex of heart), bifurcations (between fingers, legs), and where mesh density needs to be increased.

Mesh creation is a very involved topic; one needs to consider favourable alignment of elements with expected material behaviour, having sufficient density of elements to describe the problem with desired accuracy (the fitting examples below employ 2 different sized meshes to give some indication of the importance of mesh refinement). In contrast, one also wishes to minimise the model size (measured by total number of degrees of freedom) to reduce computation time.

Interactive Graphics Controls

The following tasks use interactive 3-D graphics. In any of the graphical workflow steps you may rotate, pan and zoom the view using the standard controls in the following table, click *View All* to recentre the view and click *Done* to complete/close the workflow step (and save the output model for subsequent workflow steps):

Mouse Button	Transformation
Left	Tumble/Rotate
Middle or Shift+Left	Pan/Translate
Right or Ctrl+Left(Mac)	Fly Zoom
Shift+Right	Camera Zoom

Task 0: Visualising Model Construction

We will jump ahead to look at an example from the visualisation course as it's very illustrative of the process of building a model out of simple shapes. Open the 'DTP-Visualisation-Task1' workflow and execute it. Fig. 1.13 shows a time sequence of constructing a heart model from this example which you will be able to view interactively.

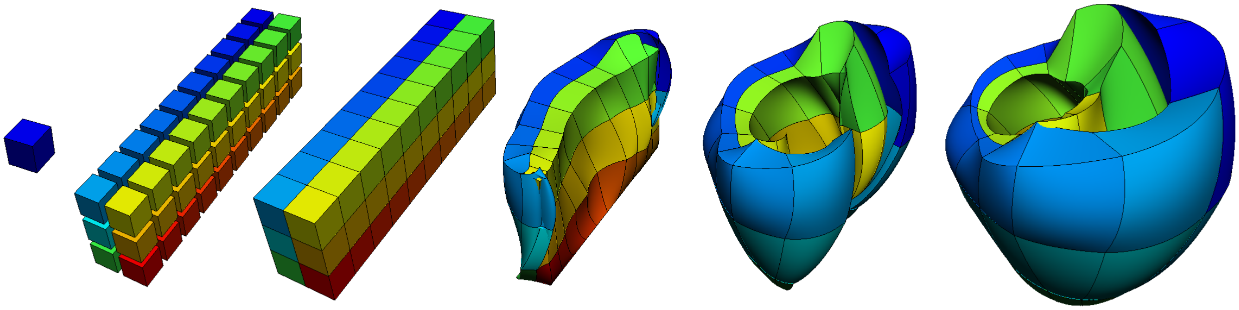


Fig. 1.13: Heart Mesh Construction stages: a single template element; multiple disconnected elements; elements merged into a connected mesh; mesh geometry gradually warped into the shape of the heart, closing up the sides and apex.

This example opens up a SimpleViz viewer for a model that shows stages in constructing a heart model. At the left of the window is a toolbox; switch to the *Time* tab of the toolbox and drag the time slider between 0 and 1. Rotate, pan and zoom into the view using the mouse controls described above.

In most of the model the initially cube-shaped elements are stretched, compressed or distorted, but they keep their essential cube or hexahedral *topology*. However, at the apex (bottom of the heart) the cubes are collapsed into wedge shapes, which while being permissible is an approach which should be minimised.

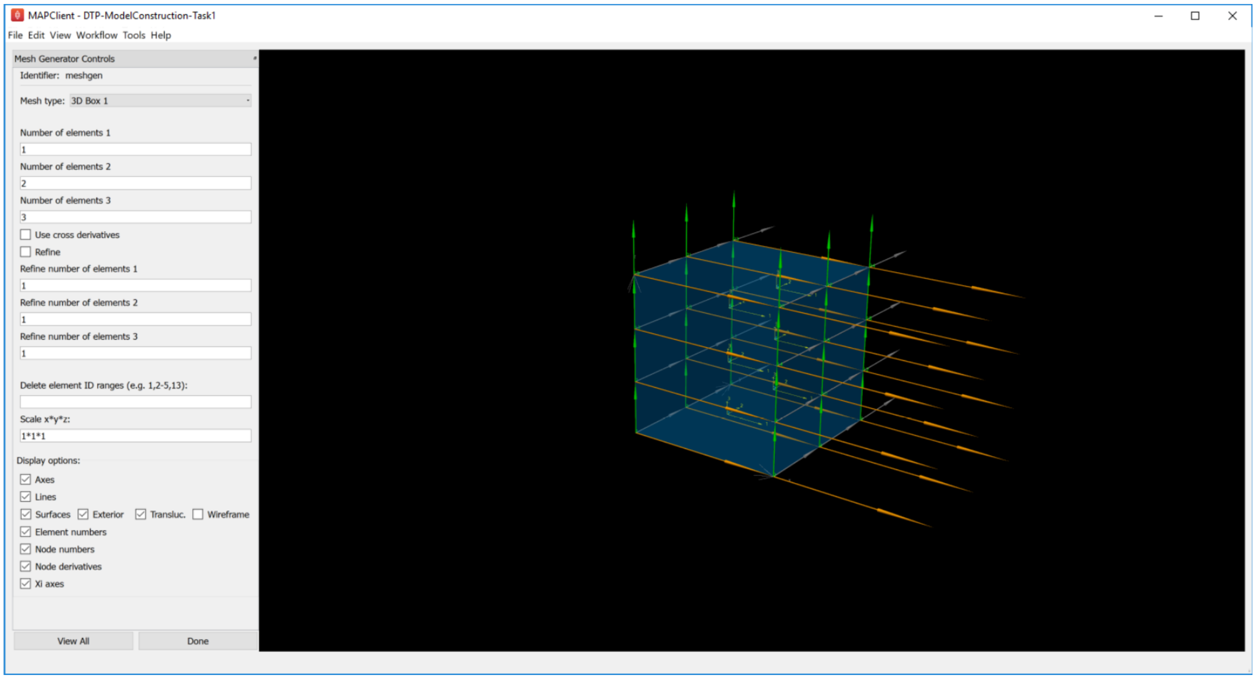
Task 1: Procedural Mesh Generation

Open the 'DTP-ModelConstruction-Task1' workflow and execute it. You will see the *Mesh Generator* interface as shown in Fig. 1.14:

In this task you are encouraged to play: try all mesh types, vary the numbers of elements and options as applicable to the mesh type, turn on and off all graphics, delete elements and scale the mesh.

The mesh types include basic shapes such as 2-D plate, tube and sphere, 3-D box, tube and sphere shell. Other more complicated meshes are being added including whole organ mesh generators. Typical for most finite elements, parameters (for the coordinates field) are held at corner points called 'nodes', and these are interpolated across the elements.

Special to the elements in these generated mesh are node derivative parameters which are interpolated with *Hermite* basis functions to give smooth geometries. If you display *Node derivatives* you will see 2 or 3 arrows showing these derivative parameters which represent tangent vectors at the nodes. The *Xi axes* show the orientation of the coordinate system of each element. Hermite interpolation is simplest when the Xi axes and node derivatives are in-line, and consistent between neighbouring elements; it's best to make this the case over most of a mesh.


MAPClient - DTP-ModelConstruction-Task1
File Edit View Workflow Tools Help
Mesh Generator Controls
View All
Done

Identifier: meshgen

Mesh type: 3D Box 1

Number of elements 1: 1

Number of elements 2: 2

Number of elements 3: 3

☐ Use cross derivatives

☐ Refine

Refine number of elements 1: 1

Refine number of elements 2: 1

Refine number of elements 3: 1

Delete element ID ranges (e.g. 1,2-5,13):

Scale x*y*z: 1*1*1

Display options:

☑ Axes

☑ Lines

☑ Surfaces ☑ Exterior ☑ Transluc. ☐ Wireframe

☑ Element numbers

☑ Node numbers

☑ Node derivatives

☑ Xi axes

Fig. 1.14: Mesh Generator Interface

The elements around the apexes of the sphere meshes use *general linear maps* to sum the apex derivatives weighted by cos and sin terms to smoothly close the mesh at these points. This is generally needed wherever neighbouring elements' coordinates are not aligned.

Ranges of elements can be deleted from the generated mesh, but this is best done after choosing the numbers of elements options for each mesh type. *Note that parts of spheres can be deleted to make e.g. bottom or top hemispheres.*

All the current mesh types make a unit sized mesh by default, but the scale option allows this to be scaled differently in x, y and z.

Task 2: Mesh Generation and Merging

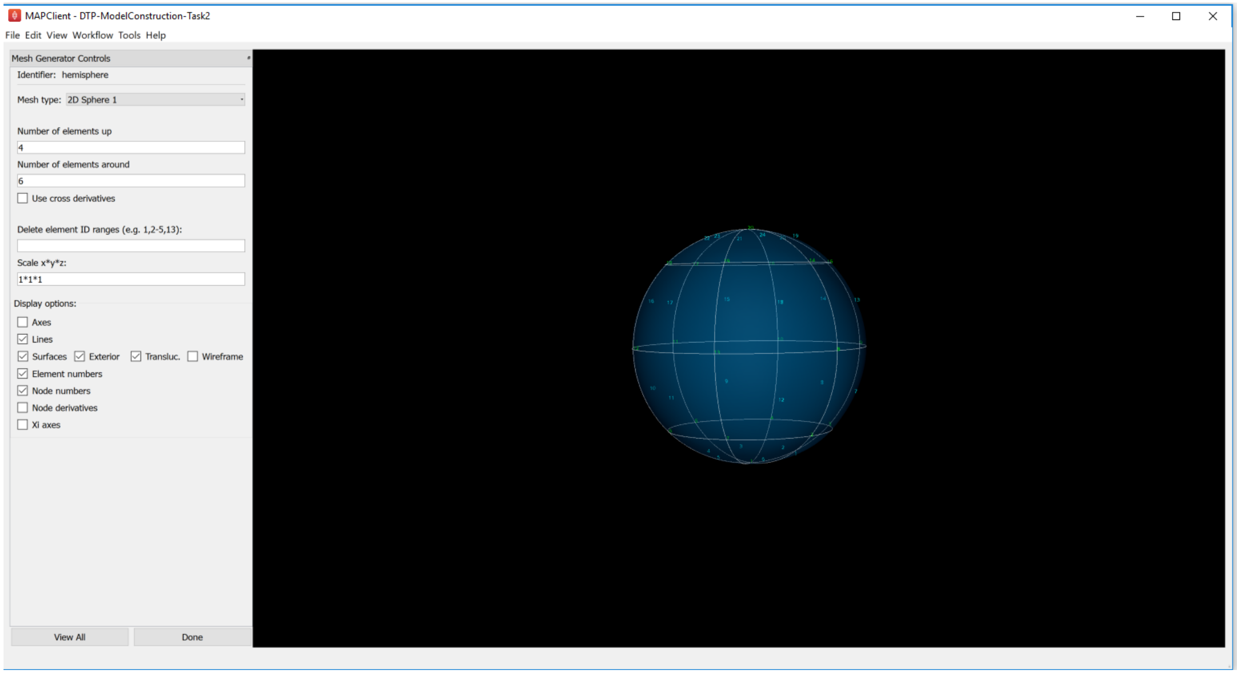
Open the 'DTP-ModelConstruction-Task2' workflow and execute it. A mesh generator interface is displayed with a 'Hemisphere'. Select the *Mesh type* as '2D Sphere 1' from the combo box and input the values for *Number of elements up* (e.g. input number 4) and the *Number of elements around* (e.g. input number 6). The mesh generator interface will be as displayed below:

To get the bottom hemisphere, find the element numbers to be deleted and enter that range in the *Delete element ID ranges* textfield (for this example, the range will be 13-24). The interface will be as shown below after deleting the required element ID range:

Click on *Done*. The next interface is another mesh generator with the 'Tube' element as displayed below:

After the mesh generation of the hemisphere and the tube, clicking on *Done* takes you to the 'Mesh Merger' stage. Observe the 'hemisphere-tube' identifier with merged master and slave nodes(for the ongoing example: map nodes 8=1, 9=2, 10=3 and so on) as shown here:

The first/top input to the Mesh Merger workflow step is the *master* mesh, which appears on the left of the interface, while the second/bottom input is the *slave* mesh, shown on the right. Merging is performed by matching (equating) node numbers from the master mesh with the ones in the slave. The master mesh is so named because it is unmodified by the merge: matched nodes on the slave are replaced by the equivalent master nodes, and the remaining slave nodes


MAPClient - DTP-ModelConstruction-Task2
File Edit View Workflow Tools Help
Mesh Generator Controls
View All
Done

Identifier: hemisphere

Mesh type: 2D Sphere 1

Number of elements up
4

Number of elements around
6

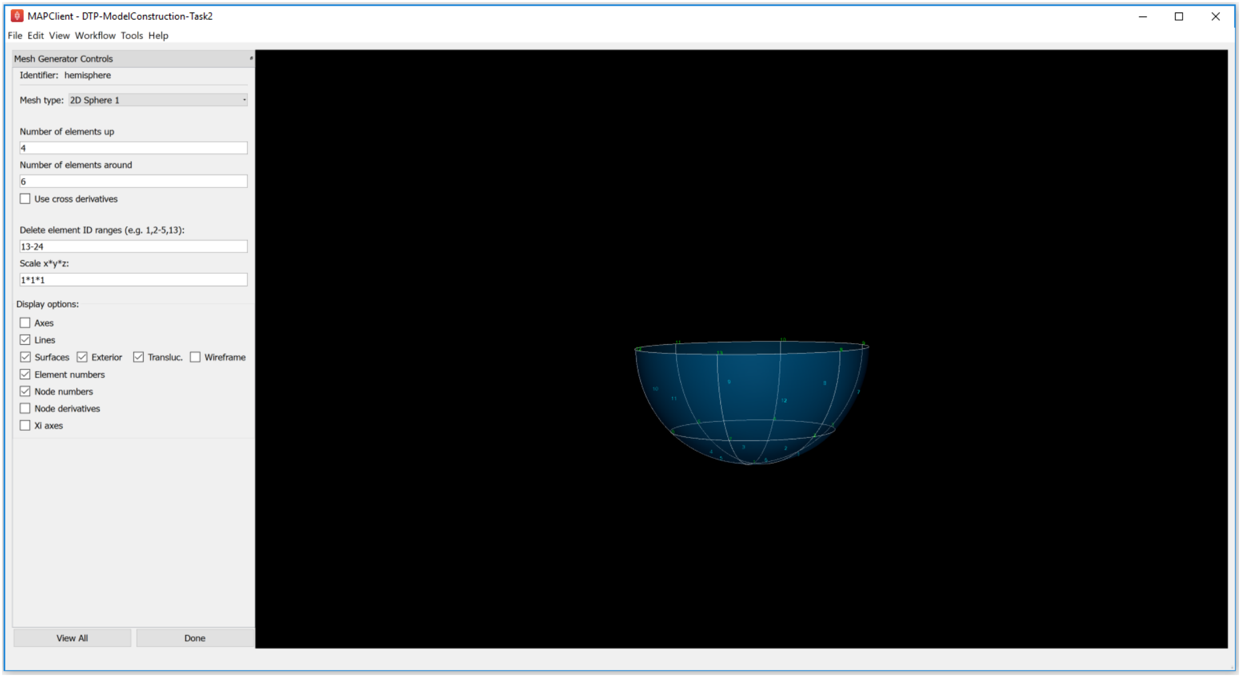
□ Use cross derivatives

Delete element ID ranges (e.g. 1,2-5,13):

Scale x*y*z:
1*1*1

Display options:
□ Axes
☑ Lines
☑ Surfaces ☑ Exterior ☑ Transluc. □ Wireframe
☑ Element numbers
☑ Node numbers
□ Node derivatives
□ Xi axes

Fig. 1.15: Mesh Generator Part 1: Hemisphere


MAPClient - DTP-ModelConstruction-Task2
File Edit View Workflow Tools Help
Mesh Generator Controls
View All
Done

Identifier: hemisphere

Mesh type: 2D Sphere 1

Number of elements up
4

Number of elements around
6

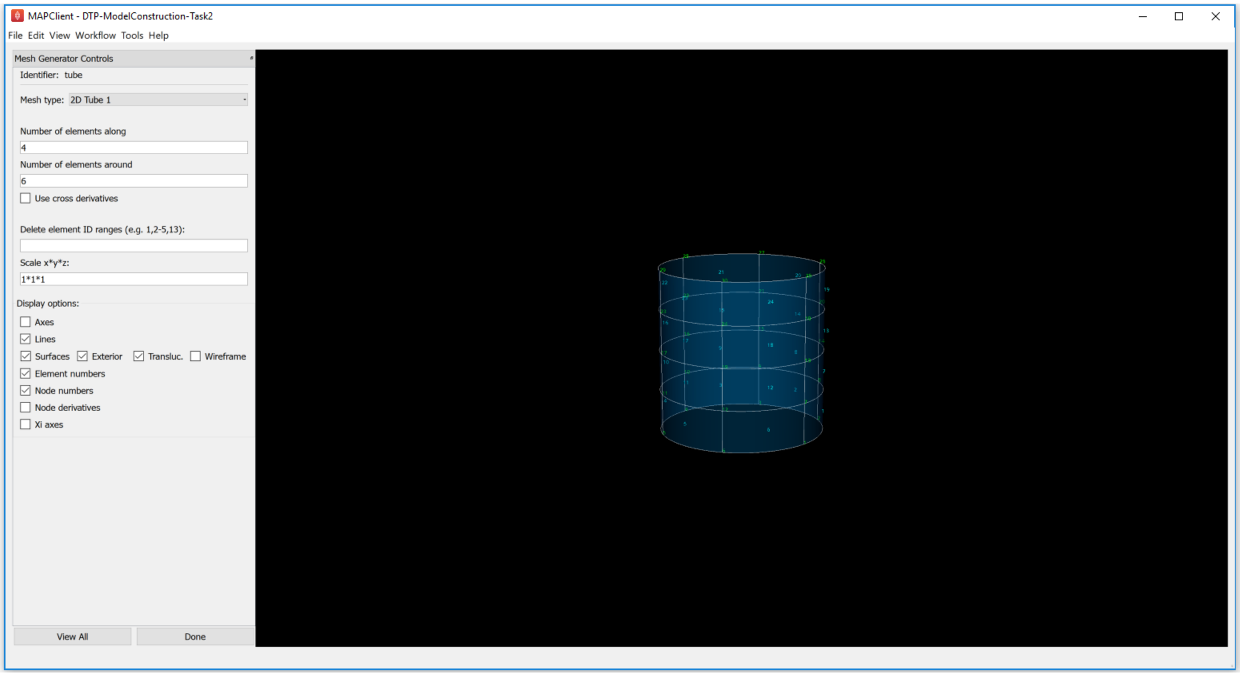
□ Use cross derivatives

Delete element ID ranges (e.g. 1,2-5,13):
13-24

Scale x*y*z:
1*1*1

Display options:
□ Axes
☑ Lines
☑ Surfaces ☑ Exterior ☑ Transluc. □ Wireframe
☑ Element numbers
☑ Node numbers
□ Node derivatives
□ Xi axes

Fig. 1.16: Mesh Generator Part 2: Bottom Hemisphere


MAPClient - DTP-ModelConstruction-Task2
File Edit View Workflow Tools Help
Mesh Generator Controls
View All
Done

Identifier: tube

Mesh type: 2D Tube 1

Number of elements along: 4

Number of elements around: 6

☐ Use cross derivatives

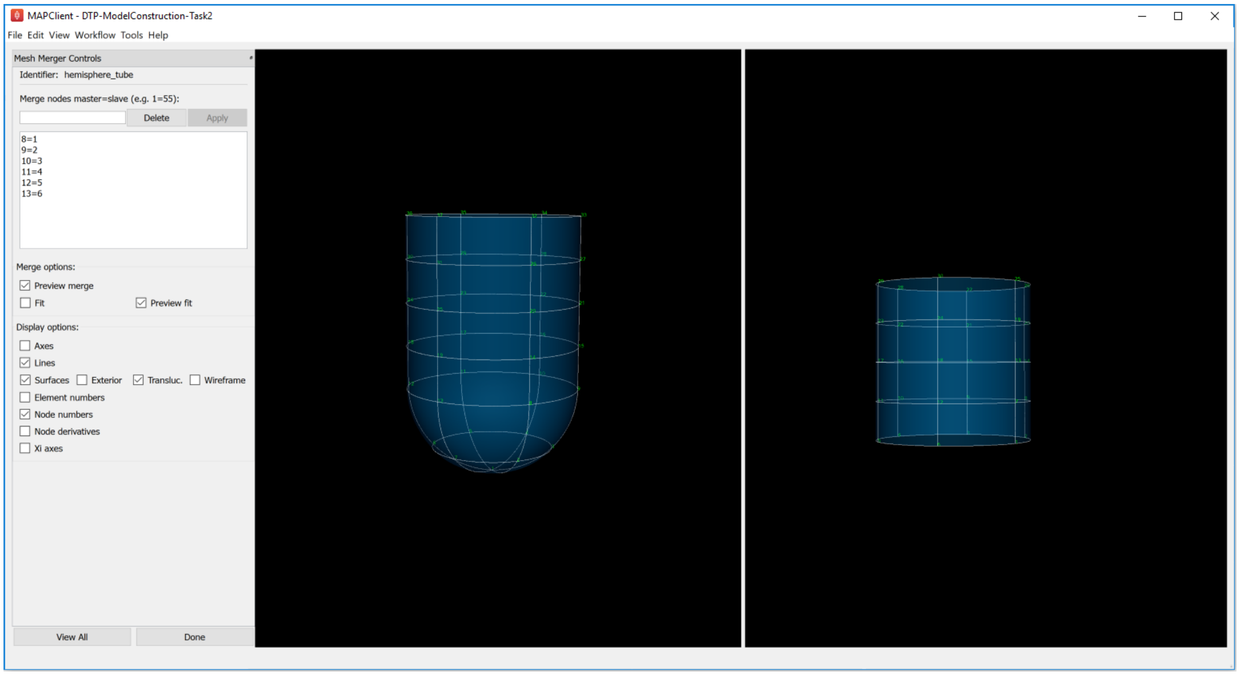
Delete element ID ranges (e.g. 1,2-5,13):

Scale x*y*z: 1*1*1

Display options:

☐ Axes
☑ Lines
☑ Surfaces ☑ Exterior ☑ Transluc. ☐ Wireframe
☑ Element numbers
☑ Node numbers
☐ Node derivatives
☐ Xi axes

Fig. 1.17: Mesh Generator Part 3: Tube


MAPClient - DTP-ModelConstruction-Task2
File Edit View Workflow Tools Help
Mesh Merger Controls
Delete
Apply
View All
Done

Identifier: hemisphere_tube

Merge nodes master=slave (e.g. 1=55):

```
8=1
9=2
10=3
11=4
12=5
13=6
```

Merge options:

☑ Preview merge
☐ Fit ☑ Preview fit

Display options:

☐ Axes
☑ Lines
☑ Surfaces ☐ Exterior ☑ Transluc. ☐ Wireframe
☐ Element numbers
☑ Node numbers
☐ Node derivatives
☐ Xi axes

Fig. 1.18: Mesh Merger Interface, after merging master-slave nodes

are transformed to fit the master, and they with the slave elements are added to the master mesh in the left panel, which is output by the workflow step.

This tool has recently been enhanced to allow master and slave nodes to be interactively selected in the 3-D view, by holding down the 'S' key and clicking on node numbers or node derivatives - one of these must be visible. The slave mesh is now automatically aligned with the master mesh, and there is the option to perform a fit to smooth it out to reduce distortion where the two meshes joined (but beware: it can be slow). Another new feature is that the list of master=slave pairings can be edited, however the changes don't take effect until the *Apply* button is pressed.

Feel free to change the matching nodes (which can be deleted by entering the number and pressing the *Delete* push button, or edited by equating with a different slave node number). No harm is done if nonsense is entered!

Geometric Fitting

The remainder of this tutorial concentrates on directly fitting simple models to data point clouds obtained from an earlier segmentation or other digitisation step. Other types of fitting **not** covered include:

- Fitting to modes from a Principal Component Analysis, where the variation in geometry over a population is reduced to linear combinations of a small number of significant mode shapes (key model poses), and lesser modes are discarded;
- Host-mesh fitting where the body is embedded in a coarse, smooth *host* mesh, data is used to morph the host mesh and the embedded *slave* mesh is moved with it.

In many cases the above methods are used as a first step to get a close approximation before direct geometric fitting.

Smoothfit Tool

This tutorial uses the *Smoothfit* MAP client plugin for interactive fitting. The inputs to Smoothfit in a workflow are a model file and a point cloud file (each currently limited to EX or FieldML formats that can be read by OpenCMISS-Zinc). The *DTP-Smoothfit-Tutorial* workflow in the MAP client is shown in Fig. 1.19, and requires only the input files to be specified (and workflow step identifiers to be named):

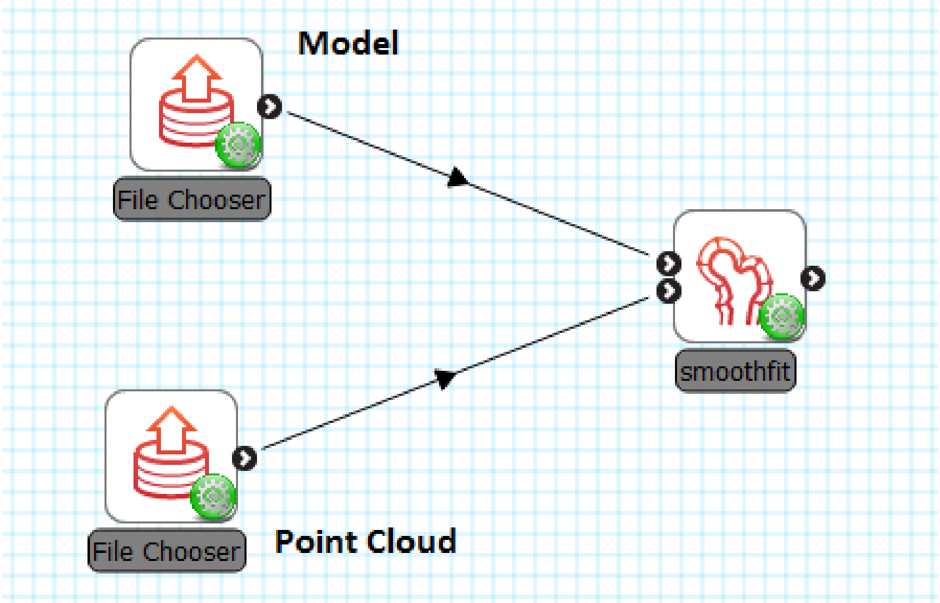
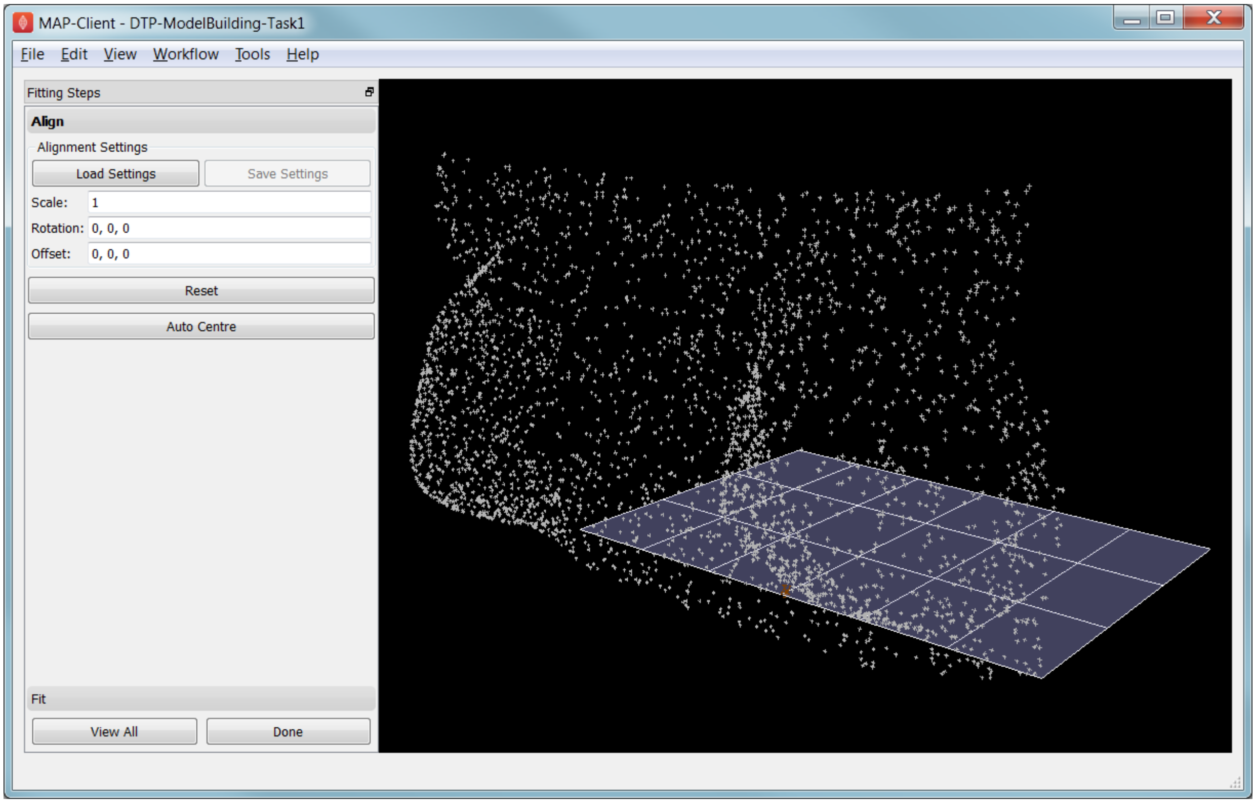


Fig. 1.19: Geometric fitting workflow in the MAP client framework.

When the *DTP-Smoothfit-Tutorial* workflow is executed, the smoothfit interface is displayed showing the model as a semi-transparent surface and the point cloud as a cloud of small crosses. The initial view in Fig. 1.20 shows the interface in its pre-fitting *Align* state.



Scale: 1
Rotation: 0, 0, 0
Offset: 0, 0, 0

Fig. 1.20: Interface for aligning the model with the data point cloud.

Smoothfit uses the interactive view controls defined earlier, including *View All* to recentre the view and click *Done* to close the workflow step and save the output model for subsequent workflow steps.

In the Smoothfit user interface you can hover the mouse pointer over most controls to get help – tool tips – which explain what they do.

Pre-fit: Model Alignment

The first step in fitting is to scale the model and bring it into alignment with the data point cloud prior to projecting data points and fitting. The need to align the model well is explained later with the projection step. To perform alignment you must be on the *Align* page in the toolbar, switched to by clicking on the *Align* label.

To align and scale the model, hold down the ‘A’ key as you left, middle and right mouse button drag in the window (or variant as in the above table): this moves the model relative to the data cloud. Be aware that rotation is a little difficult and may take practice. Other controls include alignment reset, auto centre (in case the model is very far from the data points; may need to click *View All* afterwards) and the *Load Settings* button which will load a saved alignment. (Note that the Save button is disabled in the smoothfit configuration for these tutorials so a pre-saved good alignment is always available for loading.)

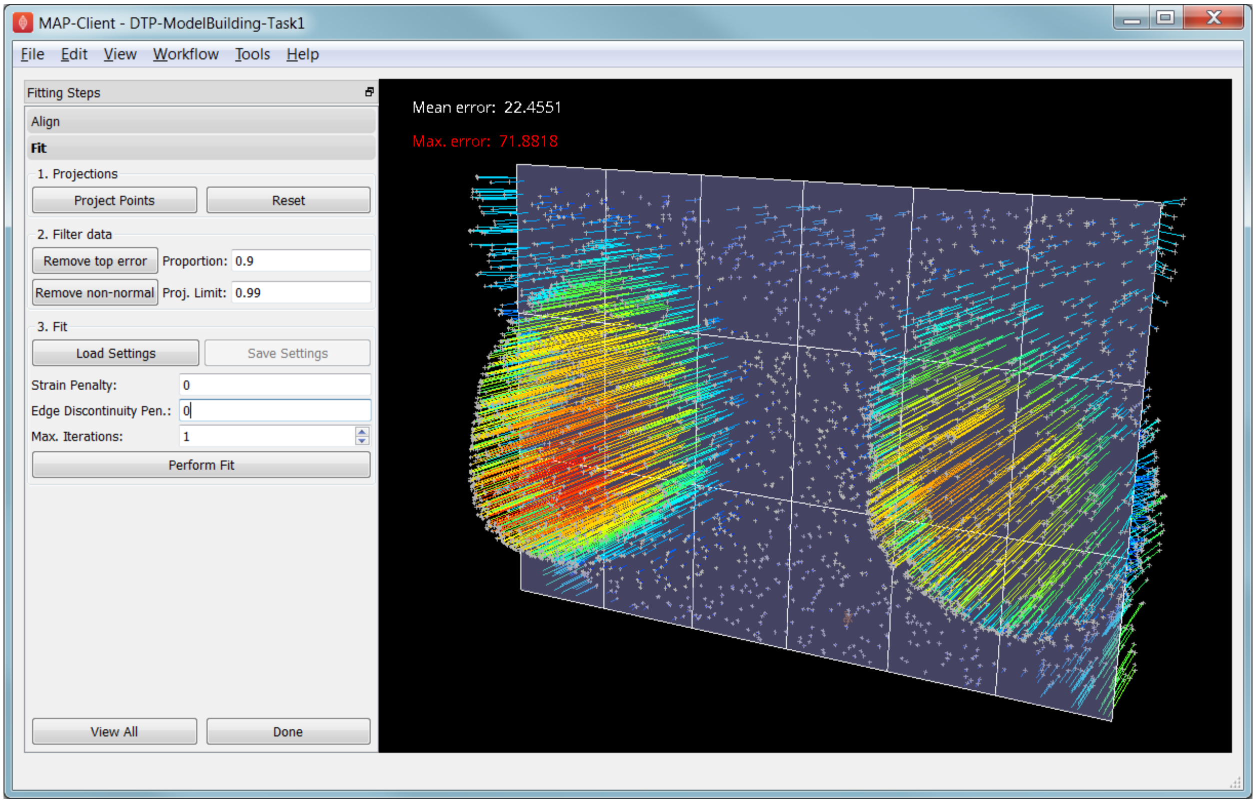
Often the shape of the model and point cloud make it pretty clear where to align to. Smoothfit uses manual alignment, but other tools may make it automatic (based on shape analysis) or semi-automatic (e.g. by identifying 3 or more points on the data cloud as being key points on the model, and automatically transforming to align with them).

Fit stage 1: Projecting Points

Once the model and data points are aligned, switch to the fitting page in the toolbar by clicking on the *Fit* label. These controls show that fitting has three stages: projecting points onto the mesh, filtering bad data, and performing the fit with some user parameters.

Fitting is usually a non-linear task: after initial fits, possibly with multiple iterations, you may need to go back and re-project data points, filter data and re-fit, possibly with different parameters. The trial-and-error nature of fitting, together with the need for judgement on whether a good fit is achieved, make it less a science and more of a dark art!

The first step in fitting is to project the data points onto the nearest locations on the elements of the aligned model, by clicking on the *Project Points* button. In the window you will see projection lines from the data points to the nearest point on the model as shown in Fig. 1.21. These projection lines, interpreted as fitting errors, are coloured by length (blue closest, red furthest away), and there is an on-screen display of the current mean and maximum projection error.



Mean error: 22.4551

Max. error: 71.8818

Proportion: 0.9

Proj. Limit: 0.99

Strain Penalty: 0

Edge Discontinuity Pen.: 0

Max. Iterations: 1

Fig. 1.21: Data points projected onto the initial model.

The key point is that the projections are what the fitting aims to minimise, and if they don't agree on where a point on the mesh should move to, the fit will have problems. It's good if the projection lines are short and/or near parallel, and it's bad if they cross over each other. Two things that help produce good projections are:

1. Good initial alignment of the model. Surfaces should ideally be close to the data points, or at least in a position to produce near-parallel projections.
2. The model should be smoothly curved, i.e. without excessive surface waviness. To help this we use fitting parameters which produce smoother results for initial gross fitting, which we intend to re-project onto for subsequent fine fitting.

In the worst cases, projecting distant data points onto a very wavy model, will produce data which is unusable for fitting.

Note that clicking on the *Reset* button clears all current projections, and restores all points that have been filtered out for subsequent projection.

Fit stage 2: Filtering Data

We often find that some of the data points are not providing useful data for the fit, and we will want to filter these out. The *Filter data* controls shown in Fig. 1.21 allow us to remove data points according to two algorithms.

The first simply removes the data points whose projections are in the specified top proportion of the maximum error, 0.9 (90%) by default. This is mainly used where the data cloud is *noisy* or contains some rogue data points which are best taken out of the solution.

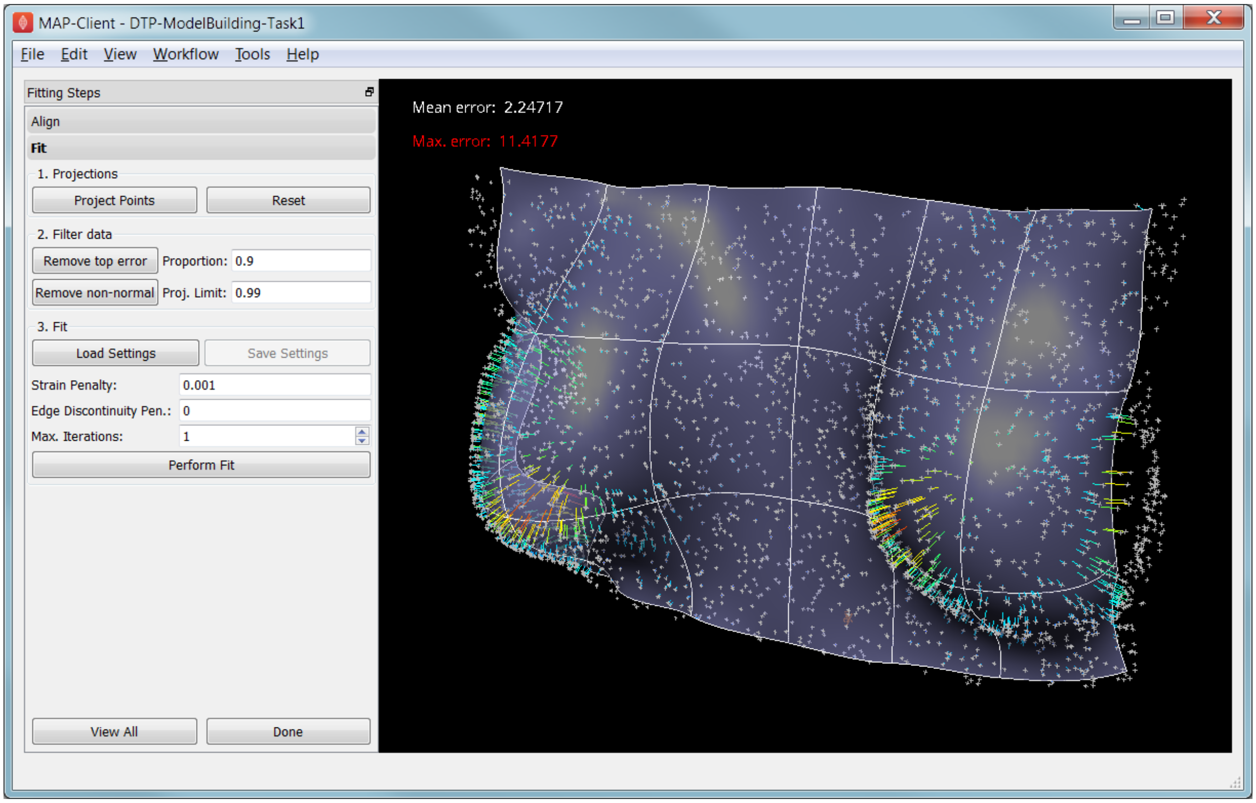
The second filtering tool removes data points whose projections are not normal to the surface. This is only suitable for use with smooth C_1-continuous coordinates (e.g. the Hermite basis meshes used for most of this tutorial) where the surface normal does not suddenly change on element boundaries in the mesh. **Note:** *It is important that you use this only after re-projecting data points since after performing the fit the data point projections will no longer be normal to the surface!*

When fitting a surface model to only a subset of the data points, you will need to use the non-normal filter (and sometimes the top error filter) to eliminate the data points clearly outside of the surface to be fit.

Filtering the data points removes those points from the active set of data points, which gets smaller each time but may be reset to all data points using the *Reset* button.

Fit stage 3: Performing the fit

With data points projected, and bad data filtered out you are ready to fit by clicking on the *Perform Fit* button, however we will usually need to play around with parameters controlling the fit to achieve a good result. Fig. 1.22 shows what the view looks like after 2 iterations of fitting with a moderate strain penalty to keep the solution smooth.



Proportion: 0.9

Proj. Limit: 0.99

Strain Penalty: 0.001

Edge Discontinuity Pen.: 0

Max. Iterations: 1

Mean error: 2.24717

Max. error: 11.4177

Fig. 1.22: Display after gross fitting the breast model.

Fitting may be non-linear so multiple iterations may be needed to converge on a solution. Through the interface one can either re-click on *Perform Fit* button or increase the maximum number of iterations before fitting; **Note**: *fitting stops either when the solution has converged or the maximum iterations is reached*. If the intention is to re-project points later, it is purely up to the user how many iterations to perform before doing this; for typical problems where one wishes to *gross fit* first, it's best to ensure enough iterations have been performed to get the solution close enough for re-projection.

Beware that projections are not recalculated during the fitting: you must manually click on *Project Points* to do this, and you will probably want to filter some more points before re-fitting.

Switching back to the *Align* page clears the fitted solution altogether.

The penalty values allow you to smooth the fit by penalising particular deformations:

1. **Strain Penalty** values limit straining (stretch or compression) in all directions; large values will keep linear dimensions close to the original model.
2. **Curvature Penalty** values limit curvature/waviness of the model, and are very useful for making smooth, attractive fits from noisy data. (Note this is a new feature, not yet shown in the screen shots in this documentation.)
3. **Edge Discontinuity Penalty** values limit angles between adjacent elements across edges, useful for less-than C_1-continuous coordinate fields e.g. the final linear mesh example.

Applying penalties always increases the data point projection error (in a least squares sense, which is the solution method used in the fitting), but generally give a much more attractive result. One should always try to use the smallest value which gives the desired model appearance but still acceptably fits the data points. Low values have a negligible effect on fitting accuracy where there is plenty of data, and the most effect where data is absent, near the edge of the model or noisy.

There is not necessarily a standard value to use for any of the penalty values as it must balance against the model size and number of data points per element. A common strategy is to adjust values by 1-2 orders of magnitude until a likeable result is obtained, then fine-tune. It is often better to use stiffer (higher penalty) values for initial iterations (gross fitting) to prevent waviness from developing in the mesh, then re-projecting and reducing penalties for a final iteration (fine fitting). As for the alignment settings, you can load and save (if enabled) the fitting options.

Performing the fit can take a few seconds, and Smoothfit will appear to hang when fitting is in progress. Processing time is longer with more elements, more complex elements, more data points and when applying penalty terms.

The following tutorial tasks each have a workflow associated with them which should be run in the usual way.

Task 3: Coarse plate model fitted to breast data

Open the 'DTP-ModelConstruction-Task3' workflow and execute it. The breast data was obtained in 'prone' pose (hanging down) as done in MRI scans; this is also the simplest pose to digitise and fit to. Try manually aligning the surface with the breast data using the mouse controls described earlier (hold down 'A' key and the left, middle or right mouse button and drag to rotate, pan or scale the model). Project points and attempt to fit without any smoothing parameters. It takes several seconds to perform the fit: be patient! Try multiple fit iterations until the solution is stable. Re-project and try again.

The result without smoothing even for this example with a coarse mesh and a relatively large number of high quality data points is quite wavy, particularly around the edges. It also has some unusual depressions about the front of the breasts which is not really representative of the data cloud in general.

For a second exercise we'll use a set sequence to obtain a good fit:

1. Switch to the *Align* page to reset the fit, click on *Load Settings* button to load a good alignment.
2. Switch to the *Fit* page, project points and click on *Remove non-normal* button.
3. Click *Load Settings* to load a moderate strain penalty of 0.001 and perform the fit 2 times to get fairly close to the data points.
4. Re-project the data points and click on *Remove non-normal*. (This is the state shown in Fig. 1.22.)

5. Lower the strain penalty to 0.0001 and fit once more. The error bars almost disappear over most of both breasts.
6. Write down the mean and maximum error for comparison later.

While the fit appears to be reasonable over most of the breast area, zoom in close on the tips of the breasts and you will see that the fit is not quite so good there. This is due to the mesh having too few elements to fit the data. The next task uses a slightly denser mesh which can achieve a closer fit, however you will need to wait longer for it to solve.

As an extra exercise switch to the *Align* page to reset the fit, re-project points and fit with a much higher strain penalty (e.g. 0.01) to see how it limits the possible deformation (after several iterations): this is what is considered a 'stiff' model.

Also try fitting with very poor initial alignment to see what happens.

Task 4: Fine plate model fitted to breast data

Open the 'DTP-ModelConstruction-Task4' workflow and execute it. It has the same data point cloud as the first task, but has a mesh with more than twice as many elements and approximately twice as many parameters, so it is more able to attain a close fit with the data, but takes longer to solve.

Try some of the exercises from Task 1 with this model. With more elements the model is more susceptible to wavy solutions so applying appropriate smoothing penalties is more critical.

When performing the second exercise from Task 1, iterate 3 times with the initial strain penalty of 0.001, then re-project points and fit with a strain penalty of 0.0001. Note down the mean and and error: the mean should be under half of the value from Task 1. More importantly, zoom in on the tips of the breasts to see that the fit is much better there.

Reset by switching to Align mode and back, then project and fit with a Curvature Penalty of 10 and Strain Penalty of 0. Reproject and re-fit a second time. You'll find this gives a very attractive result, even after 1 iteration.

Task 5: Fine plate model fitted to noisy data

Open the 'DTP-ModelConstruction-Task5' workflow and execute it. This example uses the same fine plate model (make sure it has 8x5 elements with cross derivatives ticked), however random offsets up to +/- 5mm have been added to all data points. With a large enough number of data points the effect of randomness is diminished however in small areas the randomness can introduce waviness to the solution, so smoothing penalties must be applied.

Try fitting the model without any strain penalty, and fit with several iterations to see the waviness. Reset the fit and try with the regime from task 1: 2 iters at strain penalty 0.001, re-project, 1 iter at strain penalty 0.0001. The overall result is a good fit but there is unattractive waviness on the chest area. If a curvature penalty were available, these issues with noisy data could be better controlled. You may try turning off cross derivatives in the mesh generator; this should slightly help with waviness, and will make solution faster since it reduces the number of degrees of freedom in the problem.

Reset by switching to Align mode and back, then project and fit with a Curvature Penalty of 10 and Strain Penalty of 0. Reproject and re-fit a second time. You'll find this gives a very attractive result that keeps close to the mean surface of these noisy points.

Because of the random noise the mean error will never get very low, but the average fit of the breast surface can be a reasonable 'best fit'.

Task 6: Bilinear model fitted to point cloud

Open the 'DTP-ModelConstruction-Task6' workflow and execute it. This example has a bilinear mesh and needs no alignment with the data point cloud.

Project points and fit with all smoothing penalties set to zero. Rotate the result to see that it has developed a 'ridge' along one side, and the under-constrained corner elements distort unacceptably. Reset the fit (switch to *Align* and back to *Fit* pages), reproject and fit with the *Edge Discontinuity Penalty* set to 1. The result is much smoother. This penalty discourages solutions with differences in surface normals across edges of the mesh. Since the mesh uses bilinear interpolation, exact satisfaction of this condition cannot be met, nevertheless it minimises it as much as possible, and in particular it evens out this discontinuity since it is minimised in a 'least squares' sense.

Experiment with a much higher edge discontinuity penalty (e.g. 10 or even 100) and lower (e.g. 0.1) to see how the fit is affected. Try combining with strain penalty values.

Material Field Fitting

In addition to geometry, bioengineering models often need to include spatially varying data describing the alignment of tissue microstructures, concentrations of cell types, or other differences in material properties. In heart and skeletal muscle, fibre orientations must be described over the body to orient their anisotropic material properties. Similarly, Langer's lines affect properties of the skin, and collagen orientations within other tissue can affect material behaviour.

Each of these properties can be described by spatially-varying fields which interpolate the property of interest over the same elements the coordinates are defined on.

This topic is not covered further in this example, but the concepts of creating and fitting such fields are similar to geometry: one must define the interpolation of the values over the mesh, and fit the field to data obtained from imaging or other techniques. The difference lies mainly in that the data is not coordinates, but orientations when fitting fibres, known concentrations at points for input to cell models etc.

A similar process is often used to obtain solution fields from results. Often the solution technique produces outputs with high accuracy only at certain points in the model. With the Finite Element Method, for example, stress is of highest accuracy at the Gauss points, and fitting can be used to give a better idea of these solution field values away from Gauss points.

DTP-ModelConstruction

Documentation for the Model Construction section of the Computational Physiology module in the MedTech CoRE DTP.

1.2.3 Computational Physiology - Simulation

This tutorial was created as part of the Computational Physiology module in the MedTech CoRE Doctoral Training Programme. The tasks presented in this tutorial are designed to make the reader aware of common key skills required for the application of mathematical models in computational simulation experiments in the context of computational physiology. We will demonstrate these skills across a range of spatial scales and numerical methods.

Contents:

Integrating systems of differential equations

In this tutorial we look into the integration of systems of differential equations, using example models from the domain of computational physiology. We also examine the the main control parameters that can have a significant impact on the performance and accuracy of a given method.

Contents:

Numerical integrators

Euler method

The Euler method to integrate a system of ordinary differential equations is probably the most well known method, particularly popular given its simplicity. Due to its simplicity it is, however, usually not the most appropriate method to use when performing simulation experiments with complex models of biological systems. It is, however, a very useful example to use to demonstrate key concepts that apply to most numerical integration methods.

Given the initial value problem

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0, \tag{1.1}$$

we now want to approximate $y(t)$ over some interval $t_0 \rightarrow t_{final}$. The Euler method approximates $y(t)$ by dividing the interval into a series of steps, of size h, and stepping through the interval. One step of the Euler method from t_n to $t_{n+1} = t_n + h$ is given by

$$y_{n+1} = y_n + hf(t_n, y_n). \tag{1.2}$$

The value of y_n is an approximation of the solution of the ODE (1.1) at time t_n: $y_n \approx y(t_n)$. The Euler method proceeds through the interval in constant steps of size h until $t_n = t_{final}$ and the integration is complete.

Task 1 - The effect of step size

As you'd imagine, the size of h in the Euler method is crucial to the successful application of the method. For the successful (although perhaps inaccurate) integration of a typical physiological model (see the Physiome Repository for a collection of examples), h can be so small that the computational cost of performing the simulation is very high. In some cases, mathematical models may be unsuitable for integration by Euler method, regardless of how small the step size is reduced to.

In the following task, we investigate the effect of altering the size of h on two separate simulation experiments. The first looks at a simple mathematical model where the experiment could be replicated with pen and paper, while the second looks into the application of Euler's method to a biophysical model of cellular electrophysiology.

In this task we use the trivial model:

$$\begin{aligned} x(t) &= sin(t), \\ y'(t) &= cos(t) \quad \text{with} \quad y(0) = 0. \end{aligned} \tag{1.3}$$

As you can see from (1.3), if correctly integrated $x(t)$ and $y(t)$ should be identical. We now use this model to demonstrate the effect of step size (h) on simulating this model using Euler integration over the interval $0 \rightarrow 2\pi$.

1. Run MAP Client, choose *File* → *Open* and select `HOME/projects/mapclient-workflows/DTP-Simulation-Task1`.
2. This simple workflow should look similar to Fig. 1.23. The workflow is pre-configured so there is no configuration required.
3. Click the *Execute* button and you should get a widget displayed as per Fig. 1.24.

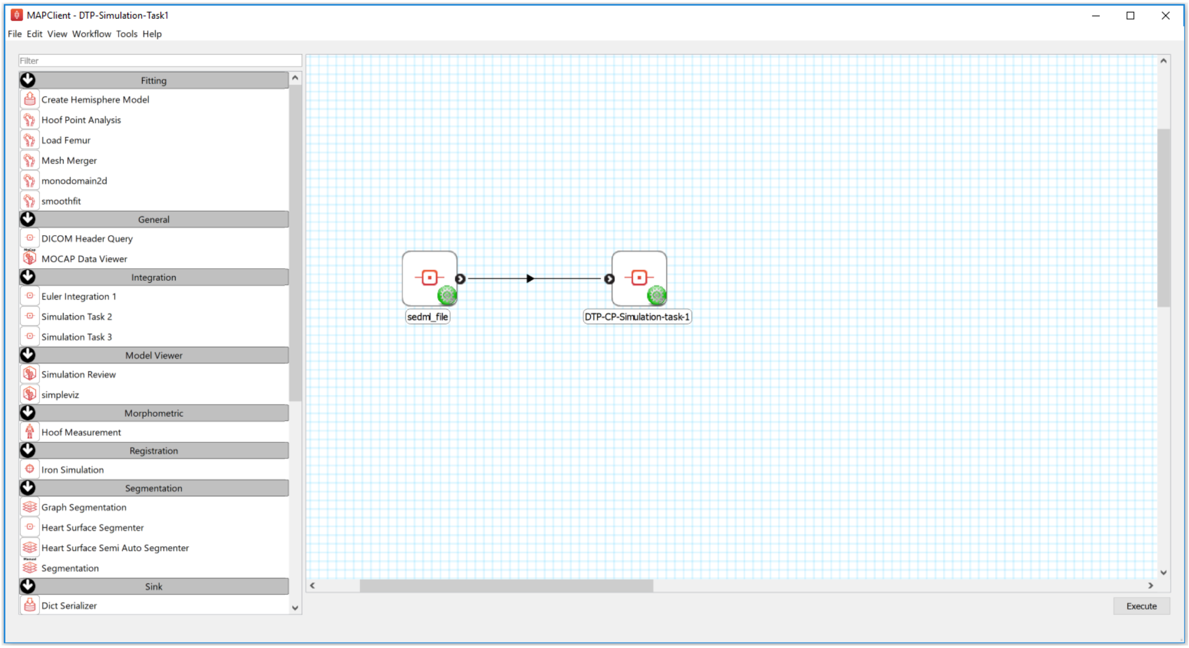
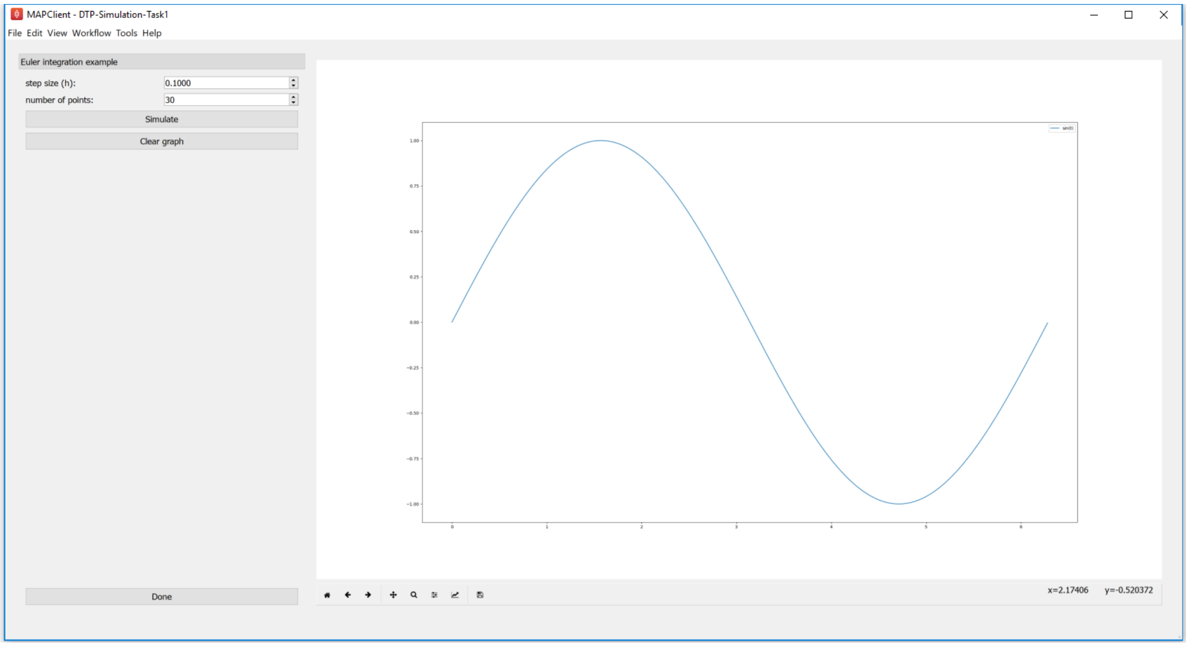


Fig. 1.23: The first Euler example as loaded.


MAPClient - DTP-Simulation-Task1

Euler integration example

step size (h): 0.1000

number of points: 30

x=2.17406 y=-0.520372

Fig. 1.24: The cool Euler integrator interface. In this simple interface, you will see the standard sine function, $sin(t)$, plotted in the right hand panel. The toolbar under the plot is self-explanatory, but provides access to some nifty features. At the top of the left hand panel you will see the control to set the Euler step size for this model, h and also the number of points to be obtained. The *Simulate* button will execute the Euler integration of the model (1.3) and plot the result on the plot to the right. This can be repeated with various values of h. The *Clear graph* button will, surprisingly, clear the current simulation results from the plot panel. The *Done* button will drop you back to the work-flow diagram, where you can get back to the plot by executing the work-flow once more.

4. As described in Fig. 1.24, multiple simulations can be performed with varying values for the step size, h. Shown in Fig. 1.25 you can see that as h reduces in size, the approximation of the model (1.3) by integration using the Euler method gets more accurate.

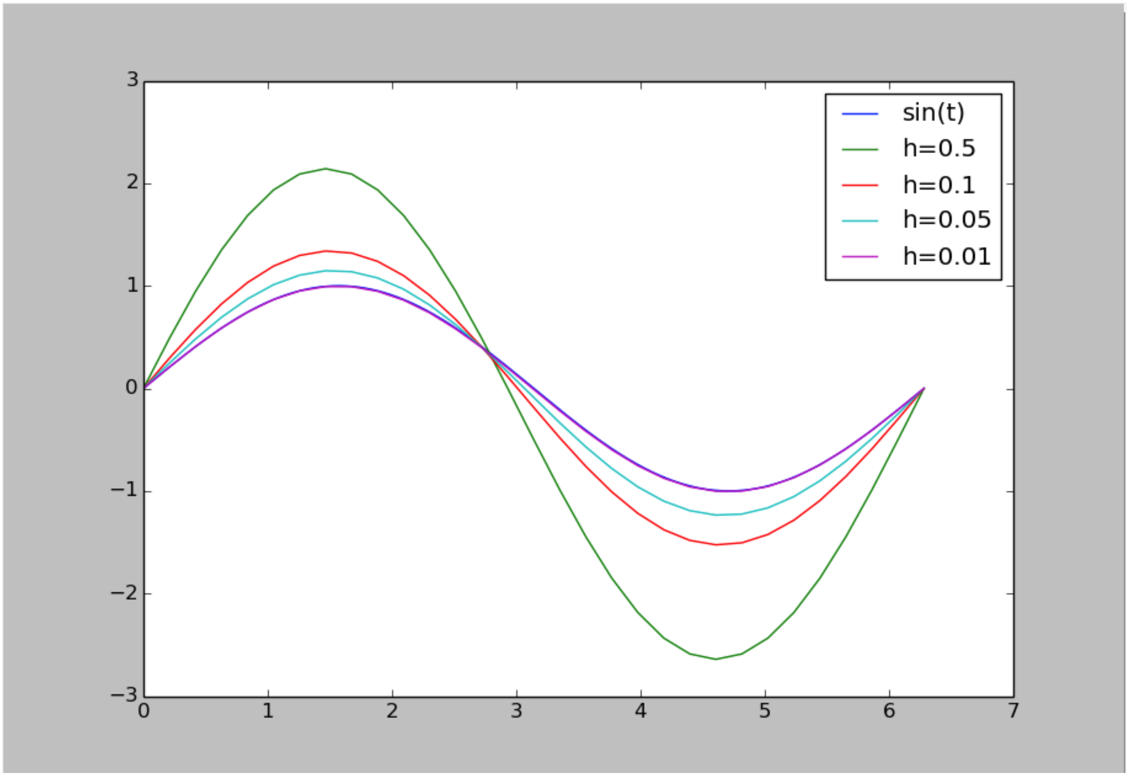


Fig. 1.25: Simulation results demonstrating the effect of step size, h, on the accuracy of Euler's method in approximating the solution of (1.3).

5. Now have a play with combining different values for the step size and the number of points to be obtained. See if you can answer the following.

    1. How small should h be to accurately simulate a sine wave?

    2. What do you think would happen beyond a single cycle?

    3. Given $h = 1$, do you obtain a more accurate solution with a large number of points or a small number of points?

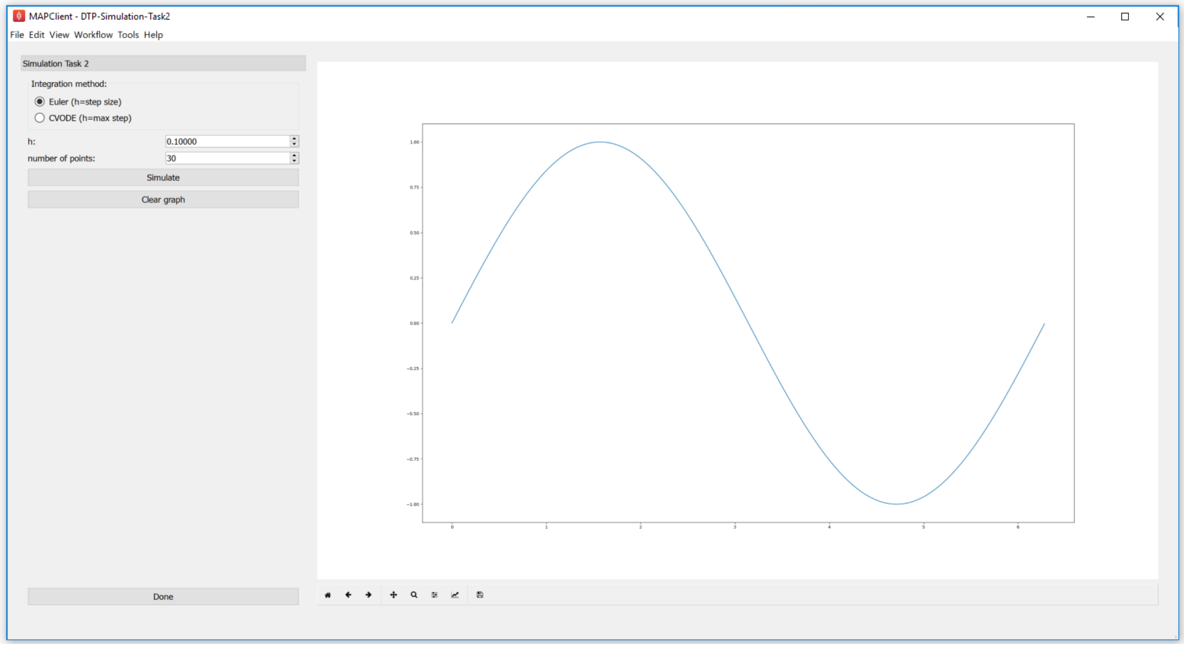
CVODE

From the Sundials suite of tools, CVODE is a solver for stiff and nonstiff ordinary differential equation (ODE) systems (initial value problem) given in explicit form in (1.1) above. CVODE is widely regarded as one of the gold standard implementations of a robust and flexible numerical integrator. One of the advantages of CVODE over Euler's method is that it makes use of adaptive stepping over the interval of integration - rather than taking fixed sized steps through time, for example, CVODE will determine how quickly things are changing and adjust the size of the step accordingly.

Task 2 - Fixed vs Adaptive stepping

In this task we examine the limitations and the computational costs associated with a fixed step method (Euler) compared to an adaptive step method (CVODE). Here we continue with our sine integration demonstration model to help highlight the differences.

1. Run MAP Client, choose *File* → *Open* and select `HOME/projects/mapclient-workflows/DTP-Simulation-Task2`.
2. This simple workflow should look similar to that used in task 1 above (Fig. 1.23). The workflow is pre-configured so there is no configuration required.
3. Click the *Execute* button and you should get a widget displayed as per Fig. 1.26.



MAPClient - DTP-Simulation-Task2

Simulation Task 2

Integration method:

● Euler (h=step size)

○ CVODE (h=max step)

h: 0.10000

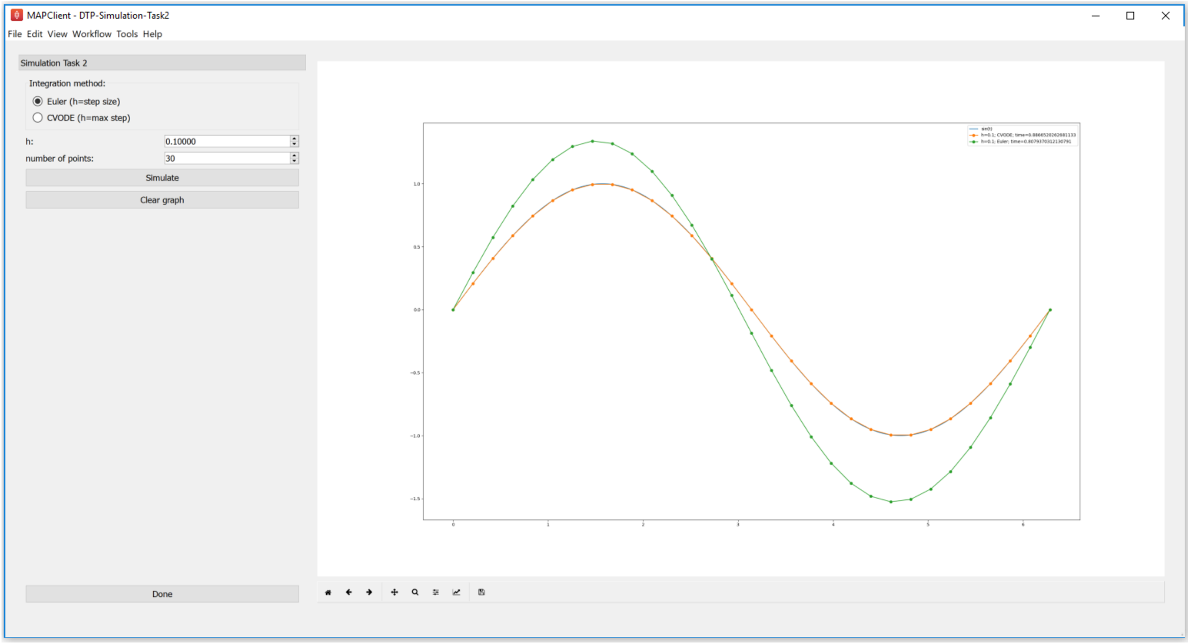
number of points: 30

Fig. 1.26: The user interface in this task is similar to that described in Fig. 1.24, and the common elements behave the same. In addition, there is the ability to choose either the Euler or CVODE numerical integration methods. As the CVODE method is an adaptive stepping method, the value of h is used to limit the maximum step size that the algorithm will use, with $h = 0$ indicating the maximum step size is unlimited.

4. You can now easily see the difference between the two integration methods by directly comparing them, as shown in Fig. 1.27.
5. Now have a play with step sizes, number of points, and integration methods to explore the features of these two integration methods and see if you can address these questions.
   1. What is the largest maximum step size you can use with CVODE to accurately simulate a sine wave with number of points being set to 2?
   2. How small does h need to be to get the same solution with Euler?
   3. Are either of those a useful solution?
   4. What is the minimum number of points required to capture an accurate sine wave?
   5. Can you determine a configuration for Euler and CVODE which demonstrates a cheaper, more accurate, simulation using CVODE with this model?

Task 3 - Error control

In addition to providing adaptive stepping, CVODE is also a very configurable solver. Beyond the maximum step size explored above, a further control parameter of that is often of interest are the tolerances used to control the accumu-


MAPClient - DTP-Simulation-Task2
File Edit View Workflow Tools Help
Simulation Task 2
Simulate
Clear graph
Done
sin(t)
h=0.1; CVODE; time=0.886652026268113
h=0.1; Euler; time=0.807937031213079
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5
6

Integration method:
● Euler (h=step size)
○ CVODE (h=max step)

h: 0.10000

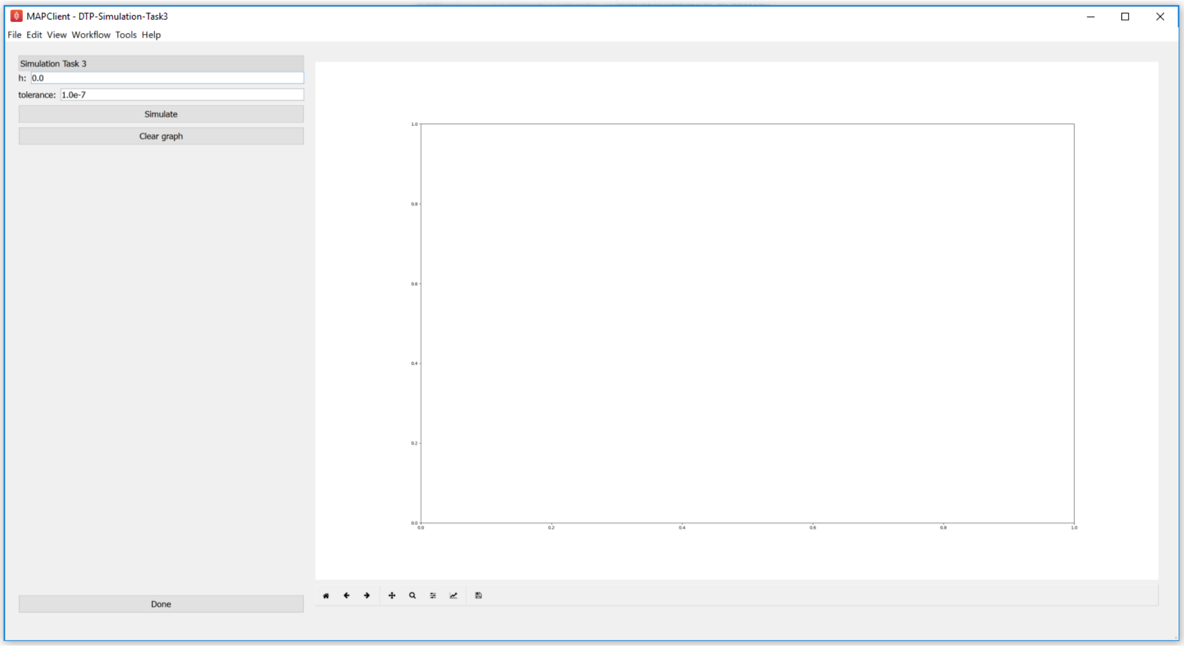
number of points: 30

Fig. 1.27: Simulation results showing the comparison between the Euler and CVODE integrators.

lation of error in the numerical approximation of the mathematical model. This tolerance specifies how accurate we require the solution of the integration to be, and the value used can be very specific to the mathematical model being simulated. In task 2 above, we used a tolerance of 1.0e-7. Depending on the behaviour of your mathematical model, you may need to tighten (reduce) or loosen (increase) the tolerance values, depending on the specific application the model is being used for. Here we explore the effect of the tolerance value on the ICC model introduced above.

We use the recent biophysically based mathematical model of unitary potential activity in interstitial cells of Cajal. The interstitial cells of Cajal (ICC) are the pacemaker cells of the gastrointestinal tract and provide the electrical stimulus required to activate the contraction of smooth muscle cells nescessary for the correct behaviour of the GI tract. This particular model was developed by scientists at the Auckland Bioengineering Institute to investigate a specific hypothesis regarding the biophysical mechanism underlying the pacemaker function of ICCs.

1. Run MAP Client, choose *File* → *Open* and select `HOME/projects/mapclient-workflows/DTP-Simulation-Task3`.
2. This simple workflow should look similar to that used in task 1 above (Fig. 1.23). The workflow is pre-configured so there is no configuration required.
3. Click the *Execute* button and you should get a widget displayed as per Fig. 1.28.
4. You can now investigate the effect of changing the tolerance value and maximum step size on the simulation result. Not all combinations will successfully complete. Example results are shown in Fig. 1.29.
5. After exploring the effects of the integrator parameters and the simulated model behaviour, see if you can answer the following questions.
    1. With $h = 0.0$, how loose can the tolerance be and still get an accurate solution?
    2. How tight can you make the tolerance before the computational cost outweighs any improvement in solution accuracy?
    3. Is there any value of h that will give an accurate solution for a tolerance of 0.01?

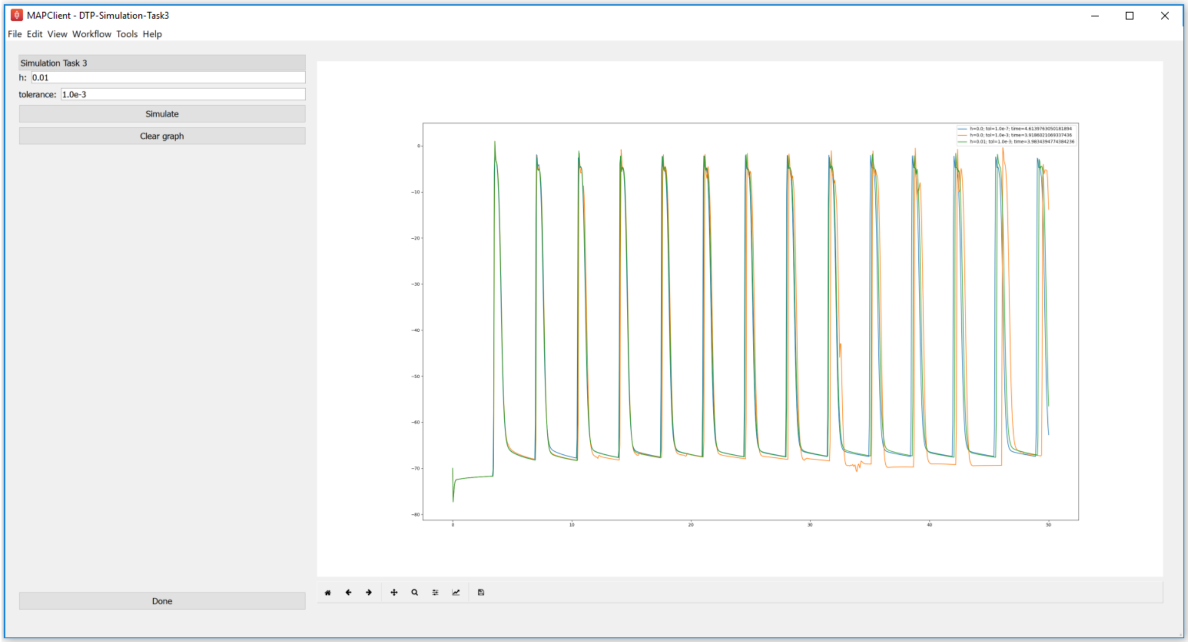

MAPClient - DTP-Simulation-Task3
File Edit View Workflow Tools Help
Simulate
Clear graph
Done
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Simulation Task 3

h: 0.0

tolerance: 1.0e-7

Fig. 1.28: The user interface in this task is similar to that described in Fig. 1.24, and the common elements behave the same. We now are only using the CVODE integration method so h is the maximum step size with $h = 0$ indicating an unlimited step size. The tolerance value for the simulation can also be edited in this interface.


MAPClient - DTP-Simulation-Task3
File Edit View Workflow Tools Help
Simulate
Clear graph
Done
h=0.0; tol=1.0e-7; time=4.6139763050181894
h=0.0; tol=1.0e-3; time=3.9186021069337436
h=0.01; tol=1.0e-3; time=3.9834394774384236
0
−10
−20
−30
−40
−50
−60
−70
−80
0
10
20
30
40
50

Simulation Task 3

h: 0.01

tolerance: 1.0e-3

Fig. 1.29: Simulation results for a selection of simulations of the ICC model using various configurations of the CVODE integratior.

Biological Complexity

Mathematical models are a useful tool for investigating biological systems, but such models only ever approximate the actual biological system and are tuned to specific applications. Depending on your specific requirements, the inclusion of more or less biological complexity may be necessary.

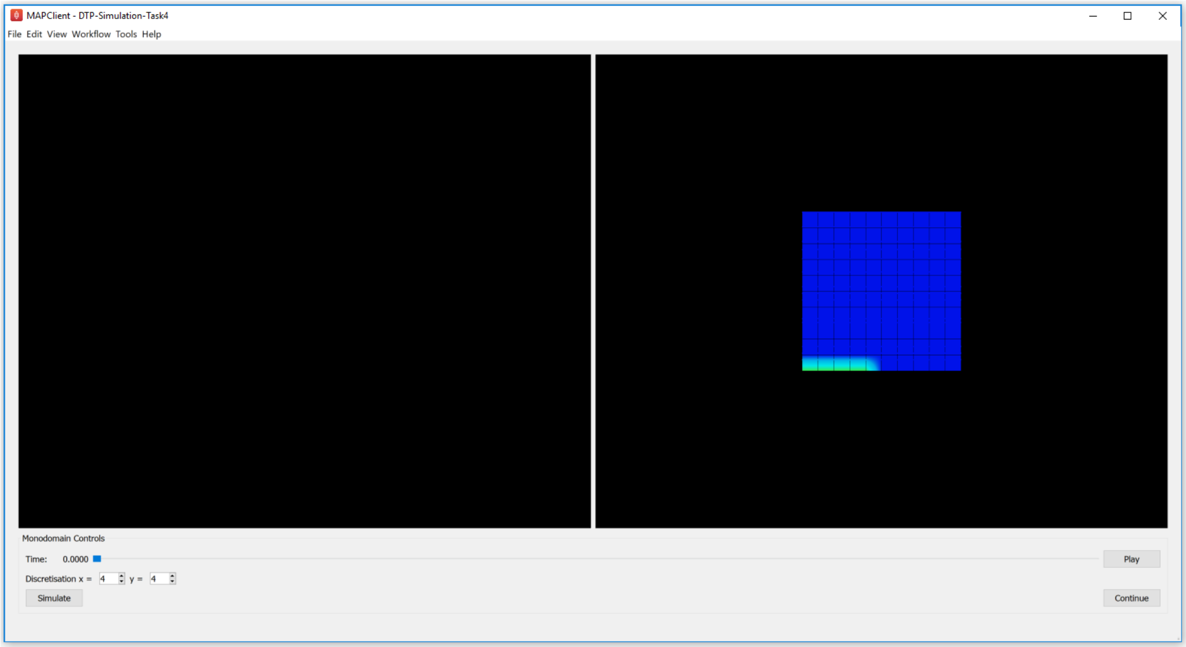
Switch to Powerpoint slides :)

Multiscale simulation

Task 4 - electrophysiological simulation

In this example, we explore the effect of spatial resolution of the numerical method in the simulation of a monodomain electrophysiology simulation.

1. Run MAP Client, choose *File* → *Open* and select `HOME/projects/mapclient-workflows/DTP-Simulation-Task4`.
2. This simple workflow should look similar to that used in task 1 above (Fig. 1.23). The workflow is pre-configured so there is no configuration required.
3. Click the *Execute* button and you should get a widget displayed as per Fig. 1.30.

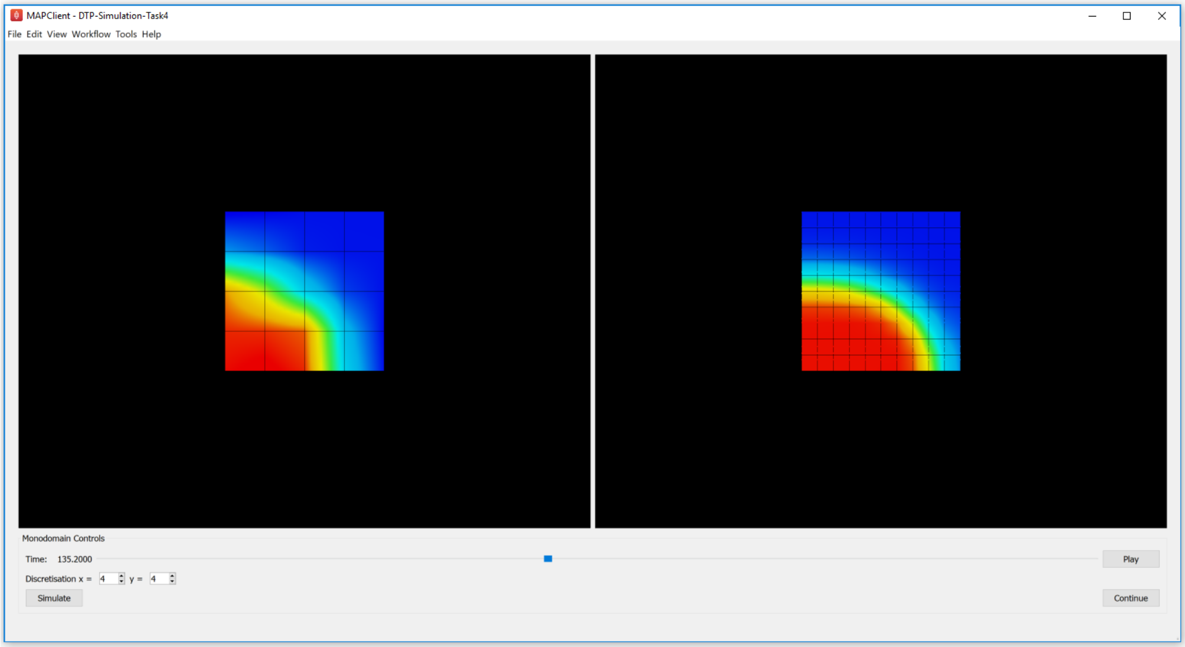

MAPClient - DTP-Simulation-Task4
File Edit View Workflow Tools Help
Monodomain Controls
Play
Simulate
Continue

Time: 0.0000

Discretisation x = 4 y = 4

Fig. 1.30: The user interface in this task initially shows a "converged" solution on the right. The user is able to set the discretisation of the finite element mesh using the widgets at the bottom.

4. You can now investigate the effect of changing the spatial resolution. Example results are shown in Fig. 1.31.
5. You need to be careful in your choice of mesh resolution as it can easily take forever to solve. Have a play and think about the following questions:
    1. How long are you prepared to wait for a suitable simulation result?
    2. Why do you need a higher mesh resolution in the x-direction than the y-direction to achieve a reasonable solution?


MAPClient - DTP-Simulation-Task4
File Edit View Workflow Tools Help
Monodomain Controls
Play
Simulate
Continue

Time: 135.2000

Discretisation x = 4 y = 4

Fig. 1.31: Simulation results for a 2x1 mesh.

3. What is the lowest mesh resolution that gives a reasonable solution compared to the provided converged solution?

Brownie points: Where do these models come from?

Contents:

In the previous steps in this tutorial, you have explored simulations performed using mathematical models encoded in the CellML format. CellML provides a powerful technology to encode these models in a modular and reusable manner which is easily exchangable between tools.

Task 5: Explore CellML using OpenCOR

In this task, we switch from MAP Client to *OpenCOR*. The first step is to get some familarity with using OpenCOR to create models and execute simulations. This can be achieved by working through *Create and run a simple CellML model: editing and simulation*.

Did you notice that link at the end of that section? With a single click you are able to launch the Van der Pol oscillator in OpenCOR and immediately simulate and interact with the model.

1. How cool is that?

Task 6: Find the ICC model

As you will have seen, using the model repository and OpenCOR it is *usually* easy to find useful models and explore them as an aid to understanding some aspect of the model or system represented by the model.

1. Take a look at *Open an existing CellML file from a local directory or the Physiome Model Repository* to learn another way to find models in the repository and load it into OpenCOR.
2. Can you find the ICC model from Task 3 in the repository? Load the ICC model into OpenCOR and see if you can reproduce the simulation behaviours you observed when exploring the simulation settings in Task 3 using the MAP Client.
3. Pretty cool that these two different tools can access the same model description and (hopefully) get the same results, right?

Task 7: The Simulation Experiment Description Markup Language

Once you are able to reproduce the simulations and plots from Task 3, it would be good to save the configuration of the simulation so that you are able to easily re-run the simulation experiment in a manner similar to that Van der Pol oscillator link above.

1. See *SED-ML, functional curation and Web Lab* to learn how to achieve this.
2. Try and export a SED-ML document which reproduces the ICC simulation result you observed back in Task 3 (preferably a working instance, but getting the same errors is also verification!).
3. Share the SED-ML document with a fellow student and see if they are able to get the same result.

Task 8: Explore alternate ICC models

One benefit of computational physiology is that it makes it easy to explore and compare different hypotheses. The ICC model introduced in Task 3 is a model that captures one representation of the ICC electrophysiology, but there are others. To finsh off this part of the module, take a look through the repository to explore some of the other ICC models that are available. And since the ICC cells are primarily driving the contraction of the smooth muscle cells in the GI system, you might want to see if you can discover any smooth muscle cell models that might be relevant to the GI system.

1.2.4 Computational Physiology - Visualisation

Contents:

Visualisation

This tutorial was created as part of the Computational Physiology module in the MedTech CoRE Doctoral Training Programme. The tasks presented in this tutorial are designed to make the reader aware of key visualisation skills used in the context of computational physiology. We will demonstrate these skills across a range of spatial scales and visualisation techniques.

Contents:

Overview

Models, simulation results and image data in Computational Physiology are often three-dimensional, and may vary with time or other material or input parameters. We use visualisation to convert this raw modelling data into visual representations that illustrate the complexities of underlying biophysical behaviour in a manner that is consumable by the target audience.

This exploits humans' astounding visual capabilities, able to perceive three-dimensional objects and their spatial relationships from two-dimensional images of scenes with lighting/depth cues, possibly stereo, particularly when seen from multiple viewpoints.

Researchers need to perform interactive visualisation, alternately varying what is being spatially visualised and moving about the scene to view it from different distances and directions, and viewing changes with time. Such interaction is necessary to discover salient features in the dataset, since they may be deep within a 3-D model and hidden behind other structures, or happen in an instant of time.

One of the main challenges of visualisation is that typical output media such as documents and computer displays are inherently two-dimensional. We employ computer graphics rendering to make 2-D images from 3-D scenes, built up solely from point, line and triangle *primitives*. Visualisation algorithms are used to covert features of a model which may be any dimension into these primitives. Computer graphics rendering applies shading (colouring including with textures/images, lighting, translucency) to draw primitives out of coloured pixels on the output image, and combined with the ability to vary time and viewpoint (at interactive speeds due to the enormous processing power of graphics hardware) gives the resulting output depth and communicates temporal changes.

Often the final objective is to output one or more static 2-D images, a movie or increasingly a shareable interactive 3-D visualisation. These outputs are used for papers and theses, presentations and general publicity. It can't be over-estimated how important it is to develop these visualisation skills when working in Computational Physiology, and how effective they are in communicating results and engaging with your audience: *pretty pictures sell research*.

SimpleViz tool

This tutorial uses the *SimpleViz* MAP client plugin for interactive visualisation. Each tutorial task uses a simple workflow consisting of a File Chooser for specifying a loading script (which loads a model and sets up some initial graphics) which is passed to the SimpleViz step, as shown in Fig. 1.32:

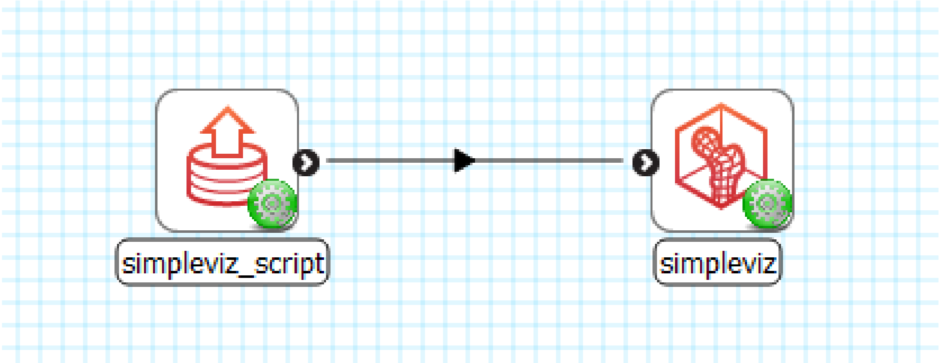


Fig. 1.32: Visualisation workflow using SimpleViz in the MAP client framework.

As the name suggests, SimpleViz presents a simplified interface for performing key aspects of interactive visualisation including results output. As shown in Fig. 1.33 its interface consists of a large 3-D graphics view and a toolbar with a series of pages for performing key functions. These are described in the following tutorial tasks, however it is hoped that many features will be obvious, and you are encouraged to *play* and *have fun*.

Task 1: Viewing

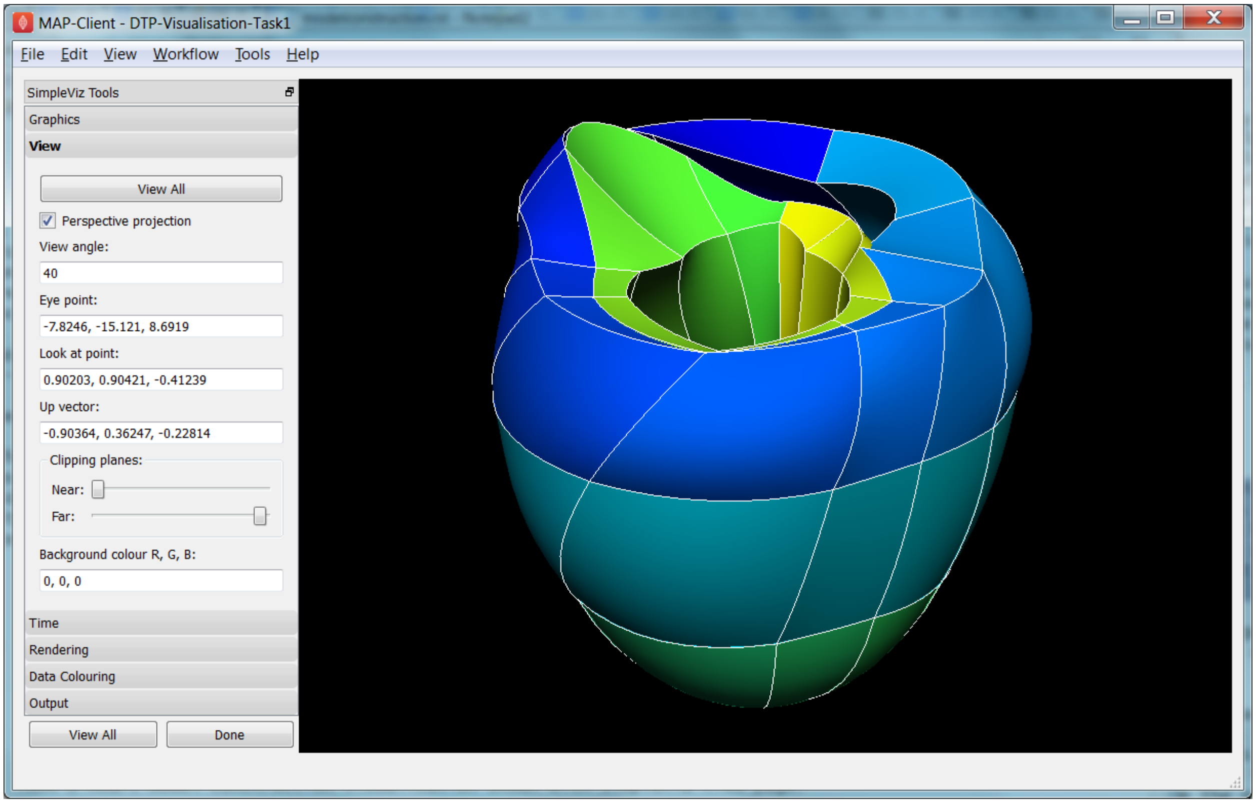
Open the *DTP-Visualisation-Task1* workflow and execute it. This loads the heart model construction visualisation in SimpleViz (from the model construction tutorial) as in Fig. 1.33.

This is a made-up example for demonstrating how complex models are built out of simple shapes (finite elements), in this case cubes. Once you play around with it you will see how a good visualisation can explain complex behaviour with great efficiency.

The example supplies the coordinate locations of 60 elements at 4 times:

1. All elements converged to a single cube (time = 0.0).
2. The elements are exploded into a regular lattice and not connected (time = 0.2).
3. The elements are merged into a block mesh of 10x3x2 elements (time = 0.4). This stage shows that corners, edges and faces of touching elements have merged (except for those eventually on the right ventricle cavity – these open up).
4. The block mesh is deformed into the heart model, merging into a ring where ends touch, closing the apex, and opening the right ventricle (time = 1.0).

At any time switch to the time page and move the time slider to animate the model which smoothly interpolates between the above times. Note that interpolation between times 0.4 and 1.0 is not appropriate for some outside elements which get very distorted, but it is good enough for this demonstration. The following section explains how to change your view of the model which you should be constantly doing when visualising models.



☑ Perspective projection

View angle:
40

Eye point:
-7.8246, -15.121, 8.6919

Look at point:
0.90203, 0.90421, -0.41239

Up vector:
-0.90364, 0.36247, -0.22814

Clipping planes:
Near:
Far:

Background colour R, G, B:
0, 0, 0

Fig. 1.33: SimpleViz heart model construction visualisation, with view controls.

Before proceeding we need to explain some concepts in order to make sense of the following tasks. This model has a **domain** consisting of a mesh of 60 cube-shaped elements which are eventually connected along certain faces. Over the domain we describe the **field** ‘coordinates’ which is a function mapping each of the elements’ local ‘xi’ coordinates to their positions in the 3-D coordinate system; in this case the coordinates are interpolated from coordinates stored at discrete *node points* within the model, which can be visualised and labelled as described in task 2. In computational models there can be any number of fields defined over a domain, representing quantities ranging from material properties to the results of simulations. A key feature of visualisation is that separate fields can be used to set various visualisation attributes, including coordinates, colours, orientation and scaling, labels etc. creating an explosion in the number of permutations of possible graphics that can be displayed.

Manipulating the View

We can manipulate the view with mouse actions: clicking and dragging with the mouse in the graphics window area allows you to rotate, pan and zoom the view. The following table describes which mouse button controls which transformation.

Mouse Button	Transformation
Left	Tumble/Rotate
Middle	Pan/Translate
Right	Fly Zoom
Shift+Right	Camera Zoom

When we transform the view with the mouse you can see the corresponding settings change in SimpleViz’ view page (see Fig. 1.33). You can also directly enter values into the controls. Regular Fly Zoom moves the eye point closer to the lookat point. Camera Zoom changes the angle of view but the eye doesn't move; if you make a very wide angle of view and then move in close, it is like looking through a very wide angle lens. The Tumble/Rotate control rotates about an axis in the scene, like pulling on a tangent to a large sphere filling the window. Play with these controls until they make sense to you. If things start looking too weird, click the *View All* button to restore a normal view.

In real life you can see from in front of your eyes to infinity, albeit not all in focus. In typical 3-D computer graphics everything is in focus, but you can only see a range of distances in front of your eye in the direction of the 'lookat point': between the near and far clipping plane distances. When you view the scene in perspective mode (the default in SimpleViz), the part of space you see is called a *viewing frustum*, which is a pyramid seen from above but with its top chopped off at the near clipping plane. In perspective mode, closer objects are larger, which matches how we see the real world. By turning off perspective you get an *orthographic* or *parallel projection* where sizes of objects are unchanged by distance from the eye, like an extreme telephoto lens effect. Fig. 1.34 illustrates the difference between these two projections, and shows that the near and far clipping planes work the same in both cases. (Note that the 'camera' is termed 'eye' in this documentation.)

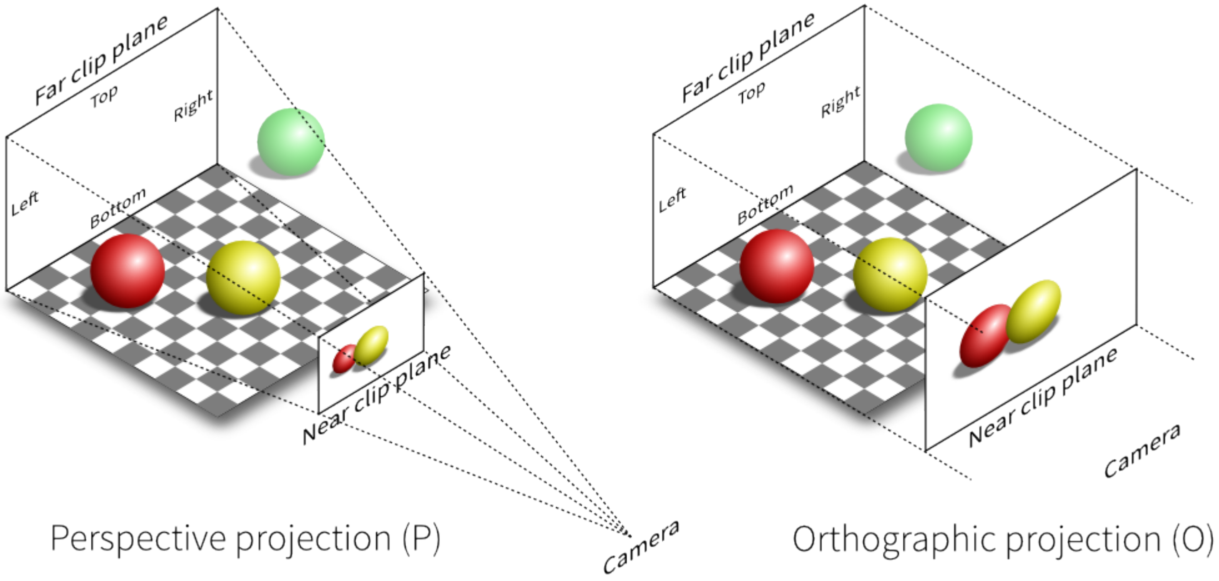


Fig. 1.34: Computer graphics perspective and orthographic/parallel projections. Original illustration from Nicolas P. Rougier, licensed under CC BY 4.0.

Ideally, we want to position the near plane just in front of everything that should be visible and position the far plane just behind everything that should be visible. The better the job we do of this the better the hidden graphics removal will work, which is important when making large high-quality, high-resolution images. SimpleViz sets the range more conservatively than this so that it doesn't need to change the ranges when objects are rotated out-of-plane. (You will notice in this example that multiple graphics drawn at the same depth appear to flash as they battle for which is in front and therefore seen. With lines and surfaces at the same depth the lines look like stitching; under the rendering page is a *perturb lines* option which brings the lines nicely in front. Try it out.)

As their names suggest, the clipping planes can also be used to good effect in hiding graphics that are in the way of what we want to see. Here we will use them to gain an insight into what graphics are actually on the screen.

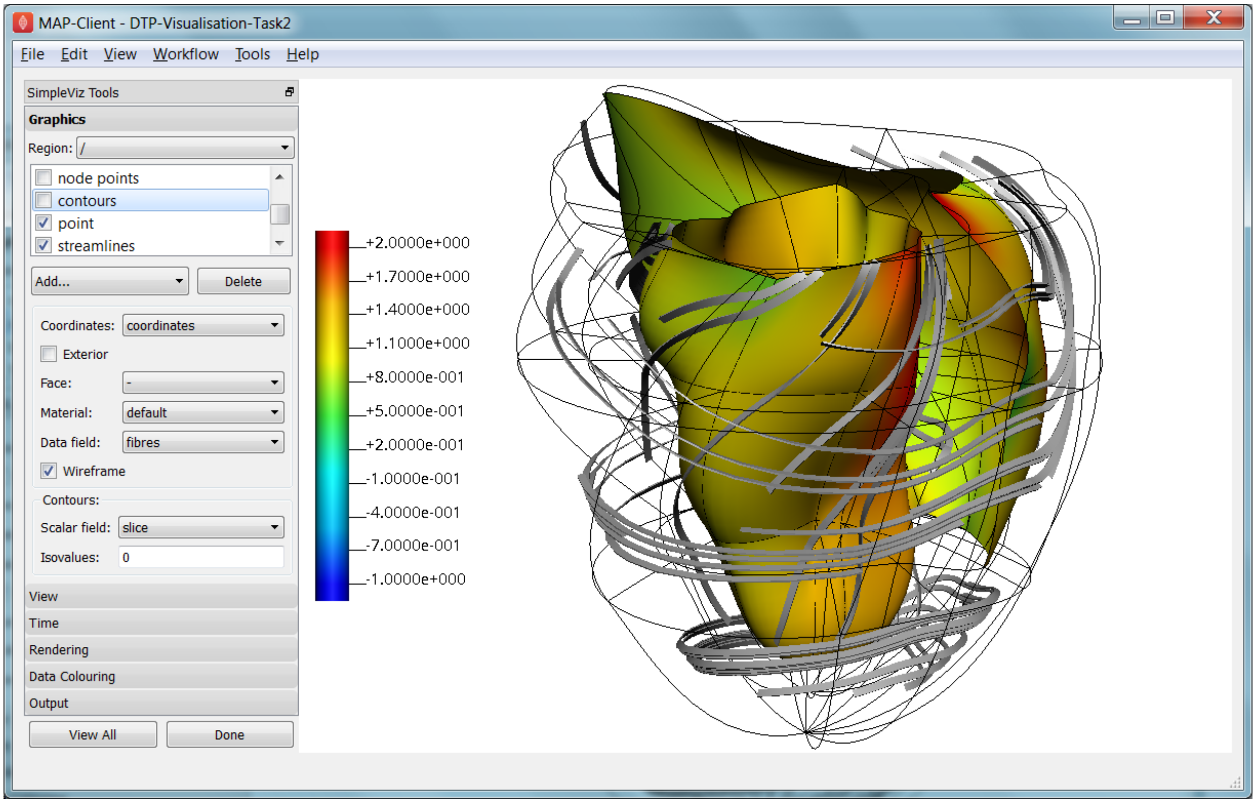
On the *View* page, drag the near clipping plane until close parts of the model disappear; when you are close you can hover over the slider and rotate the mouse wheel which moves it with more precision. Similar clipping occurs if you zoom in close enough to the model since you can't see things behind you. The far clipping plane has a similar effect on the far side of the view.

With the front part of the model being clipped, rotate the view: you will see all the elements are hollow! This reinforces that only points, lines and triangles (surfaces) are ever drawn in computer graphics. Have a look at the list of graphics under the graphics page: it consists of lines and surfaces on the edges and faces of 3-D cube elements. You can assure yourself that the elements are 3-D by making other graphics such as elements points that are calculated in its interior; you'll need to hide the surfaces by un-checking the box next to the surfaces graphics on the list.

For the rest of this task use the viewing controls to look closely at how the bottom of the heart is merged to form an apex, and generally how the initially cube-shaped elements are distorted to make a physically realistic shape.

Task 2: Graphics and Image Output

In this task, we will create basic graphics to visualise a 3-D heart model, and output an image suitable for a publication. Open the *DTP-Visualisation-Task2* workflow and execute it. Fig. 1.35 shows the model visualised how we want at the end of the task, and shows the graphics page controls in SimpleViz.



MAP-Client - DTP-Visualisation-Task2

SimpleViz Tools

Graphics

Region: /

☐ node points
☐ contours
☑ point
☑ streamlines

Coordinates: coordinates

☐ Exterior

Face: -

Material: default

Data field: fibres

☑ Wireframe

Contours:

Scalar field: slice

Isovalues: 0

Fig. 1.35: SimpleViz graphics page showing heart model ready for print output.

The graphics page lists all the individual graphics that make up the visualisation of the model. Each listed graphics item has a square checkbox that controls whether it is visible or not. The heart model is initially visualised with lines, surfaces and node points (drawn as spheres).

Graphics Types

Following are all the main graphics types that can be created with SimpleViz:

- **Lines**: Graphics made from 1-D elements or edges of higher dimensional elements. Drawn by default with line primitives, extra controls allow them to be shown as scaled cylinders.
- **Surfaces**: Surface graphics generated from 2-D elements or faces of 3-D elements.

- **Points**: Visualisations of discrete locations in the model. These are each drawn with the chosen *glyph* (standard shapes including point, sphere, arrow, cone etc.) which can be scaled, oriented and labelled by different fields in the model. Variants include *point* (a single point, e.g. for drawing the axes glyph at the origin), *node points* (points in the model at which parameters are stored for interpolation), *data points* (an additional set of points not used for interpolation), and *element points* (points sampled from the interior of elements, with extra controls for sampling).
- **Contours**: For 3-D models, produces *iso-surfaces* at which the specified scalar field equals a chosen value or values. In 2-D domains, produces *iso-lines*.
- **Streamlines**: visualisations of the path of a fluid particle tracking along a stream vector field specified for the specified length of time. Sampling and line attributes are also settable; different line shapes allow lateral directions or curl to be visualised.

All graphics share some common attributes, for example the field giving their coordinates, the material chosen to colour the graphics, and as appropriate, limiting to exterior or particular faces of parent elements. There is also a *data field* which is used to colour the graphics by the value of the chosen field, as described later.

Select the surfaces and change the material to 'blue'. Experiment with different materials, exterior state and face values for the lines and surfaces.

Point Glyphs and Scaling

Select the *node points* and change the *Glyph* (e.g. to 'cube_solid') and try different values for *Base size*. Glyphs can be oriented and scaled by fields with the final sizes each given by:

```
size = base_size + scaling*scale_field
```

If you want the glyph to be fixed size, give it a base size and either no scale field or zero scaling. If you want the size to be proportional to a field, give it zero base size, choose a scale field and a scaling value which specifies the length/diameter/size of the glyph (since they are all unit sized). If you want to visualise a vector, make the base size '0*width*width' and the scaling 'scale*0*0' to ensure the width is fixed and the length is proportional to the magnitude of the vector.

Create new *element points* graphics via the *Add...* pull down menu and change the *Label field* to 'xi' to show the element's local coordinates at their respective centres. You may need to hide surfaces to see points inside elements. Change the number of divisions to 2 (interpreted as 2*2*2 in 3-D) and change the mode to 'cell_corners'. Be aware that if you label points with the coordinates for this model the values are in a prolate spheroidal coordinate system so will not match the common x, y, z coordinates.

Data Colouring

Click *Done* and restart Task2. Hide the node points. For the surfaces graphics, choose 'fibres' for the *Data field* (we will explain what this field represents later; here we are just treating it as an interesting field to colour graphics by). Go to the *Data Colouring* page and click *Autorange spectrum*, then *Add colour bar* to see the full range of values of 'fibres' over the drawn surfaces. Note that the colour bar is a special *point* graphics added to the graphics list.

Colouring by a field is a key method for visualising variation of solution values across visible parts of a model. You are free to arbitrarily set the range of data values mapped to colours. Enter minimum=0 and maximum=2.

Contours

Add *contours* graphics and choose *Scalar field* 'slice' and *Isovalues* =0. The slice field is defined as a scalar (single component value) given by the plane equation Ax + By + Cz with the right-hand-side given by the isovalues. Experiment with other fields such as lambda, mu, theta (the prolate coordinates) and different isovalues such as 0.7 (or multiple comma-separated isovalues) for these fields.

Drawing contours/isosurfaces is one of the key techniques for visualising the interior of a 3-D model. Often there is a threshold value of a scalar field where interesting or problematic behaviour occurs: where stress exceeds what the material can handle, or where the electric potential of the heart cells rises to a point where the muscle contracts. In such cases a single image can often communicate the main features of what is happing at that time.

Always rotate, zoom and pan around to see what you have created.

Tessellation Quality

On the *contours* graphics, restore the *Scalar field* to ‘slice’ and the *Isovalues* to ‘0’. Set the *Data field* to ‘fibres’. Tick the *Wireframe* check box to see the outline of the actual triangles being drawn for the contours.

Change to the *Rendering* page of the tool bar and inspect the Tessellation divisions. The elements making up the model are divided into linear segments for graphics creation. The Minimum divisions is the number of divisions for a linear element, and these are multiplied by the Refinement factors for non-linear interpolation and coordinate systems. Hence, in this example the heart elements are divided into 4 segments in each dimension.

Type ‘8’ following by `Enter` in the Refinement control. You will see that all curved lines and surfaces suddenly look much smoother. Enter ‘1’ to see how bad linear interpolation looks on these curved elements. Now Enter ‘16’; you will be asked to confirm this number since the 3-D elements are divided into 16*16*16 small cubes for generation of the contours, which for 60 elements requires evaluating the scalar field at 0.25 million locations, and more graphics means it may be considerably slower to generate graphics and even perceptably slower to draw on-screen. Zoom in and look around this fine visualisation.

The divisions are specified as the product of 3 numbers, one for each element ‘xi’ direction. Since the elements of this mesh are thinner and more simply described through the xi3 direction, enter 16*16*4 to see an almost identically high quality result with 1/4 of the calculations.

Tessellation quality is a compromise; use fewer divisions for interactive speed, and raise the number for high quality image output.

Streamlines

Normally, streamlines are used to visualise fluid flow, however, muscle tissue is fibrous and to model its deformation and electrical conduction requires the orientation of these fibres to be described throughout the domain. The *fibres* field describes the orientation of the muscle fibres, but also the lateral sheet direction and sheet normal. This field is suitable for visualisation with streamlines.

Hide the contours and create *streamlines*. Select streamlines *Vector field* ‘fibres’, and set the *Time length* to 100 to see many fibres drawn as lines. You’re free to seed streamlines from multiple sampling points, but we’ll stick with the default centre of each element. Now set the *Base size* to ‘1*0.2’ and the line *Shape* to ‘square extrusion’. Set the *Material* to ‘silver’. This visualises not only the direction of the muscle fibres, but also the planes of muscle fibre sheets, which have different material properties to the sheet normals. Zoom in and have a close look at the resulting graphics.

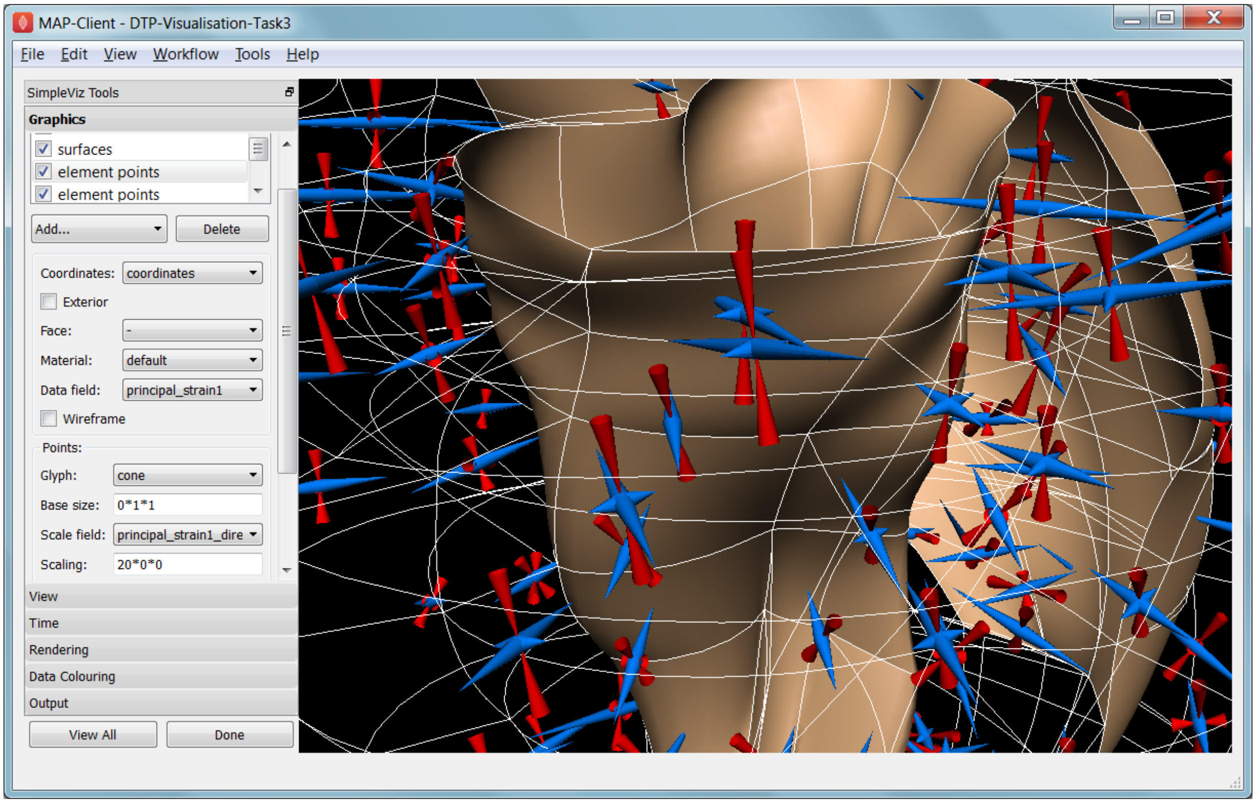
Printed Output

White or coloured graphics on a black background looks great on-screen but terrible on the printed page, plus it is a huge waste of ink/toner! On the *View* page change the *Background colour* to ‘1,1,1’ i.e. white. The problem now is that the white graphics are invisible over the white background! On the *Graphics* page select the *lines* graphics and change the *Material* to ‘black’. Do the same to the *point* graphics used to show the colour bar, so the labels appear in black. We now have what we want on the printed page (admittedly in a more sophisticated graphics package we may want to make the lines thicker, and change the font, however SimpleViz hides these options).

Adjust the window to the size you want, and the orientation of the heart so it looks balanced. From the *Output* page of the toolbar, click on *Save image…* and enter a name, say 'myheart.png'. From outside MAP Client / SimpleViz browse to the file location and have a look at the final output image, which is ready to put in your publication.

Task 3: Deformation Animation

In this task, we read a heart contraction simulation, visualise deformation and strains and output a 3-D animation to the web. Open the *DTP-Visualisation-Task3* workflow and execute it. Fig. 1.36 shows a close up of this model visualising strain tensors.



☑ surfaces
☑ element points
☑ element points

Coordinates: coordinates
□ Exterior
Face: -
Material: default
Data field: principal_strain1
□ Wireframe
Points:
Glyph: cone
Base size: 0*1*1
Scale field: principal_strain1_dire
Scaling: 20*0*0

Fig. 1.36: Visualising strain tensors in the deforming heart.

This model's loader script defines a Lagrangian finite strain field using the rate of change of the coordinate field in deformed versus reference states. Eigenanalysis is performed to get principal strains and their directions, and these are used to scale and orient mirrored cone glyphs. The above figure shows that the first element points' cones are oriented with the first principal strain direction. Not shown in the SimpleViz interface are the mirror and signed scale options use to scale the cones and point them inwards in compression and outwards in extension. A special spectrum is used to show extension in blue and compression in red, using the first principal strain as the data field.

[At the end of this task, advanced users may want to look at the loader script to see how the time-varying model is loaded, how the additional fields are created by expressions, plus how the advanced visualisation options are set up. This example demonstrates that you don't need to be stuck looking at the results exported from your solver; additional fields for visualisation can be created from any mathematical or algorithmic transformation on the exported fields.]

Go to the *Time* page of the toolbar and adjust time to observe the passive inflation and contraction phases of the deformation (the last phase was not solved and just interpolates back to the start). View the changing strains which show how the material deforms at those points. Change the *Glyph* for each element points graphics to ‘arrow_solid’ and see how it looks. On the *Rendering* page change the circle divisions to 4, then 6 and back to 12 to see the effect on the quality of the arrows; the higher the number, the more time it takes to draw the graphics; this may not affect this smallish example, but try increasing the number of sampling divisions on all three element points graphics (to 3*3*3 or higher) to see if it has an effect, particularly when animating.

Making Web Animations

Hide all three element points and view the deformation. Change the surfaces to show all faces, with exterior on. Hide the lines. Look at how the ventricle twists as it contracts.

Traditionally, we’ve produced movies to demonstrate dynamic behaviour, by writing a series of images at different times and using an external movie-maker tool to combine them into a movie file. However, these only show the results from a fixed direction or trajectory.

Here, we are going to export an animated outside surface of the heart into ‘ThreeJS’ format for viewing in a web app (using WebGL). On the Output page, click on *Save WebGL…*, navigate to the ‘export’ folder as instructed by the tutor, choose a filename prefix e.g. ‘defheart’ and click *Save*.

Now open a FireFox browser (other browsers are not yet properly supported) and load the following file from the above export folder, specifying the PATH and the inputprefix of your exported model:

```
file:///PATH/export/sample_export.html?inputprefix=defheart
```

It should display the model as a slowly deforming heart, which you can view from different directions just as in SimpleViz. This technology is relatively new and there is still much to be exploited, but it shows one of the ways visualisations will be shared in the future.

Task 4 Lung Airways Network

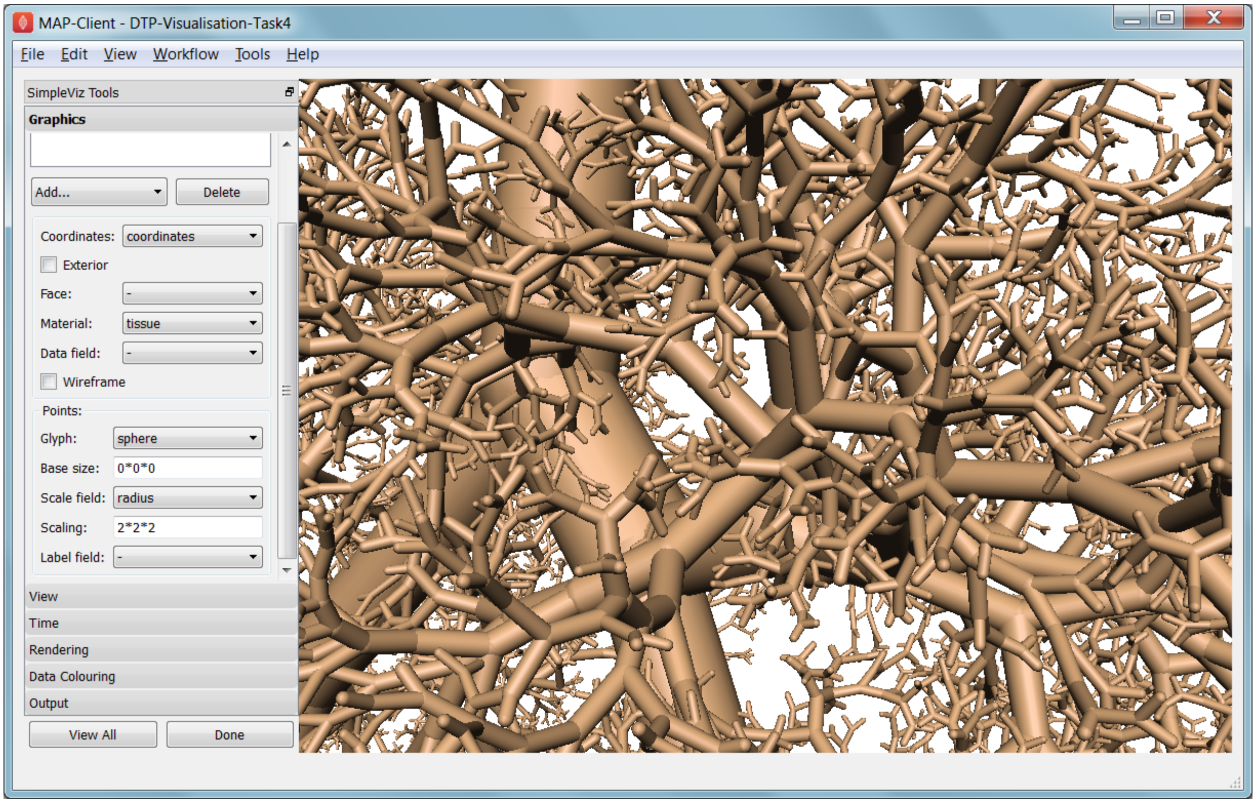
In this task, we read a model of the network of airways in both left and right lungs. The airways are one dimensional elements, but they have a radius field which is used to give them a three dimensional form. Open the *DTP-Visualisation-Task4* workflow and execute it; it’s a large model and can take a while to load. Fig. 1.37 shows a close up of this model at the end of this task.

When initially loaded, the airways are drawn as lines with no indication of how thick they are. On the *View* page change the *Background colour* to 1,1,1 and on the *Graphics* page select the lines and change their *Material* to ‘tissue’. Choose *Scale field* ‘radius’, and set the *Scaling* to ‘2*2’ to use it as a diameter. Change the line shape to ‘circle extrusion’, and after a pause the true-sized airways are shown. Explore the model up close.

One problem with the model is that each airway is a straight tube, which makes for gaps between them when they change direction. A ‘cheap trick’ solution is to draw a sphere at every node point. Add *node points* graphics, set the *Material* to ‘tissue’, the *Scale field* to ‘radius’, the *Scaling* to ‘2*2*2’, and the glyph to ‘sphere’. That should close the gaps reasonably well. Sometimes it’s necessary to be dirty to make a clean image!

For a very attractive view of the airways, select the *lines* graphics and set the *Data field* to ‘radius’. The default range of the spectrum from 0 to 1 looks much nicer than when it is autoranged.

For any of these models it may be helpful to see where the global x, y, z axes are. Add a new *point* graphics, set the *Material* to ‘black’, change the *glyph* to ‘axes_xyz’ and set the *Base size* to 50. Surprisingly, the origin is quite far from the model; you may need to zoom out or click on *View All*’ to see the axes. From the relative size of the axes we can see that coordinate units are in millimetres.



Coordinates: coordinates
□ Exterior
Face: -
Material: tissue
Data field: -
□ Wireframe
Points:
Glyph: sphere
Base size: 0*0*0
Scale field: radius
Scaling: 2*2*2
Label field: -

Fig. 1.37: Close-up of lung airways with spheres plugging gaps.

Task 5 Embedded Airways

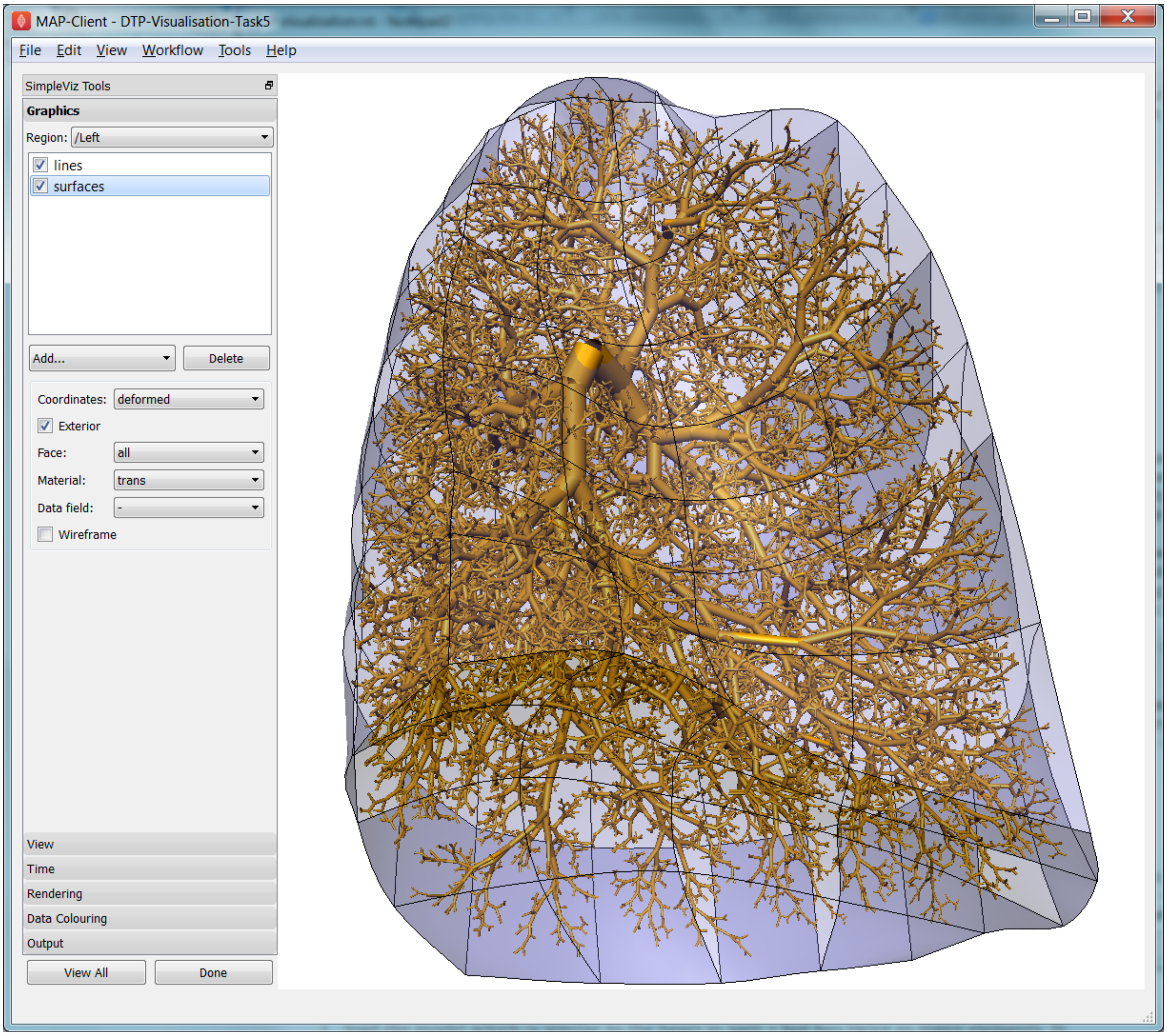
In this task, we visualise a deforming left lung model (deflating from total lung capacity) with embedded airways. Open the *DTP-Visualisation-Task5* workflow and execute it; it's a large model and can take a while to load. Fig. 1.38 shows a close up of this model decorated as part of this task.

On loading, you will see the airways as gold lines inside a lung volume mesh. The model is time-varying, so play with the time slider on the *Time* page to view the deformation (which is not quite as interesting as that of the heart). When looking at the list of graphics you'll be surprised to see an empty list! Above the list of graphics is a 'region chooser'. This model consists of two separate submodels, one for '/AirwaysLeft' and one for '/Left'. Each has its own domains and fields, plus the graphics used to visualise them.

We now decorate the combined model to match the above image. First, on the *View* page, set the *Background colour* to 1,1,1. Next, on the *Graphics* page, switch to *Region* '/AirwaysLeft', select the lines and set *Scale field* 'radius', *Scaling* '2*2' and *Shape* 'circle extrusion'. Switch to *Region* '/Left', select lines and change the *Material* to 'black'. Add surfaces, make them exterior and choose *Material* 'trans', a special semi-transparent material created for this example.

You will find with the fully decorated model that animation with time is much slower, mostly because of the cost of building the 3-D airways. Reducing the circle divisions on the *Rendering* page can speed things up a little at some cost to image quality.

One interesting thing about this visualisation is the fact that the airways move with the deforming lung volume model because they are embedded at fixed element:xi locations within it. This is a technique for reducing the computation and storage costs of multi-scale models: time-varying coordinates need not be stored for the fine airways since they can get them from their host lung model.



Region: /Left

☑ lines

☑ surfaces

Coordinates: deformed

☑ Exterior

Face: all

Material: trans

Data field: -

☐ Wireframe

Fig. 1.38: Left Lung with embedded airways.

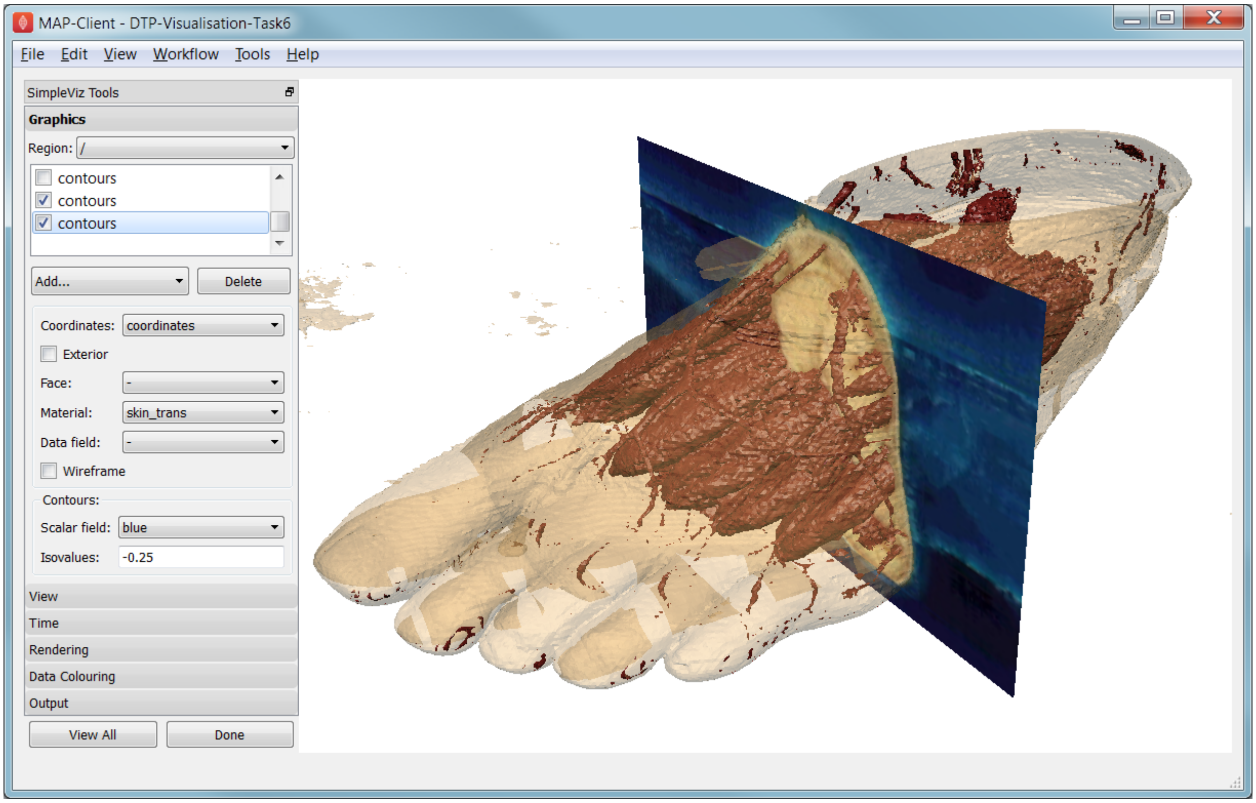
A second interesting point is that the translucency effects are imperfect and ‘patchy’. It actually takes some clever rendering to draw this perfectly, and SimpleViz does not present those options.

There are some other interesting fields in this model. Create contours of z = -100, with *Data field* ‘cmH2O’ a pressure. You will see nothing until you hide the translucent surfaces. The order of drawing is important for simple translucency, so recreating the translucent surfaces after the isosurfaces works better. Once you can see the isosurfaces, autorange the spectrum under the *Data Colouring* page, and display the colour bar (changing its material to ‘black’ on the graphics page, under root region ‘/’). It was a surprise to the researcher that this field drops to zero in the centre of the lung, and may indicate an error. This goes to show how interactive visualisation plays a key role in checking the validity of computational physiology results.

On the ‘Left’ region you can also create *data points* with *Coordinates* field ‘stress_coordinates’ and colour them by *Data field* ‘stressp’. The data points are also embedded in the lungs and field ‘stressp’ varies with time. You may need to hide other graphics to see these well. Play around with adjusting time and autorange the spectrum at different times in the *Data Colouring* page. Data points can be visualised with scaled glyphs just like node points.

Task 6 Image Fields and Texturing

This task demonstrates how images can be used to colour, or *texture* graphics, and how images can be segmented into surfaces as contours of the image field. Open the *DTP-Visualisation-Task6* workflow and execute it. It may take a while to load since it contains a stack of images and some of the contours calculations take some time. Fig. 1.39 shows a view of the model from later in this task. The image data is of the foot, cropped from the NLM Visible Human Project male dataset.



Region: /

□ contours
☑ contours
☑ contours

Coordinates: coordinates

□ Exterior

Face: -

Material: skin_trans

Data field: -

□ Wireframe

Contours:

Scalar field: blue

Isovalues: -0.25

Fig. 1.39: Segmented skin, muscles and vessels of the foot image.

Initially, two perpendicular slices of the 3-D images – contours of x and y – are drawn, plus two contours graphics segmenting surfaces of the skin and interior red tissue including muscles and larger vessels. An initially hidden contours graphics shows segmented bone surfaces, but is not so clean and includes a lot of non-bone surfaces.

First, hide the last two contours graphics and inspect the images drawn in the image block. Try different values of x and y contours, for example, enter isovalues '120, 180' for the first contours (of x), and '120, 180, 240, 300' for the second contours (of y). This shows that the entire volume image is present and able to be shown over graphics. Note you can't currently set up these graphics via the SimpleViz interface as it doesn't have the *texture coordinates* field setting which tells the graphics which part of the image to draw at primitive vertices.

Restore contours to x=128.5, y=185.5, and then show the last two contours in the graphics list, the muscle and skin surfaces. To achieve these visualisations the loader script created fields 'mag_non_muscle' and 'blue' as expressions on the colour (in red, green, blue or RGB space). You can see that it is very clear on the images where the images are red and blue, so these work well.

Now switch to the *Rendering* page of the toolbar and see that only the minimum divisions, at '16*16*16' are used for this model. If you were to create new lines graphics you will see that the isosurfaces are calculated over a separate mesh, sized so that 16 image pixels cross each element. This means that the contours will be as fine as the native resolution of the images. See the result of setting the minimum divisions to 4, then 8, then 16, then enter 32 and wait a while for even finer contours to be seen. Hide the skin and any other distracting graphics and look at the muscles. Transform the view to see them from different angles and up close. Re-display the skin contours, and change their *Material* to 'skin_trans' which is semi-transparent so the muscles can be seen within it.

Zoom right inside this model and change the view angle on the *View* page of the toolbar to a high value e.g. 90 degrees or more. Explore around this amazing 3-D world you have created!

Hide all graphics and turn on the 3rd contours in the list, which correspond to bones. With 32 tessellation divisions they will take a few seconds to be generated. You will see that, indeed, some bones are visible, however there is a great deal of noise and many other structures are falsely shown. This demonstrates some of the difficulty of automatic segmentation on real images, and why additional knowledge including models of the shapes and relative sizes of the parts expected is often needed to extract patient specific models from images.

As an advanced exercise, try tweaking the isovalues for this and other contours to see whether better surfaces can be created. Using a lower tessellation minimum divisions e.g. 16 (on the Rendering page) while exploring.

1.2.5 Computational Physiology - Complete Workflow

Contents:

Complete Workflow

This module covers the complete workflow from the acqusition of medical images through to a clinical outcome or clinical tool.

Overview

At this point we have covered the individual stages that take us from some clinical observation or clinical experiment through to creating clinical outcomes. We shall now turn to looking at a complete workflow, that is starting from some clinical data and finishing with a prediction for clinical use.

In this module we will use the musculoskeletal system to illustrate the complete workflow. Firstly we will consider the partial case of a complete workflow where we will analyse a horses fetlock joint to gain understanding on wear and abrasion and how this relates to injury. Then we will look at a much more complete workflow, the femur morphometric workflow, which starts from a set of patient specific anatomical feducial markers and ends with the femur measurements of a specific patient. The measurements taken from the femur may then be used to provide assistance to a clincian on deciding on a course of action for a positive clinical outcome.

The Musculo-Skeletal System

The musculo-skeletal (MSK) system … has a great deal importance in the clinical setting. With so many people suffering from osteoathritis and other musculo-skeletal diseases the impact of reducing patient suffering is huge. From a computational physiology aspect we are taking patient measurement and constructing a suitable model that can be used to predict behaviour, for example a wear pattern of a joint acheived through computing joint stress and strain.

MSK Introductory Workflow

Task one is to have a look at a workflow for the analysis of the horse fetlock joint. In this workflow we will take a set of measurements that will be used to characterise the joint. With this information we will predict the liklehood of injury and try to determine a training regime that minimises the risk of injury.

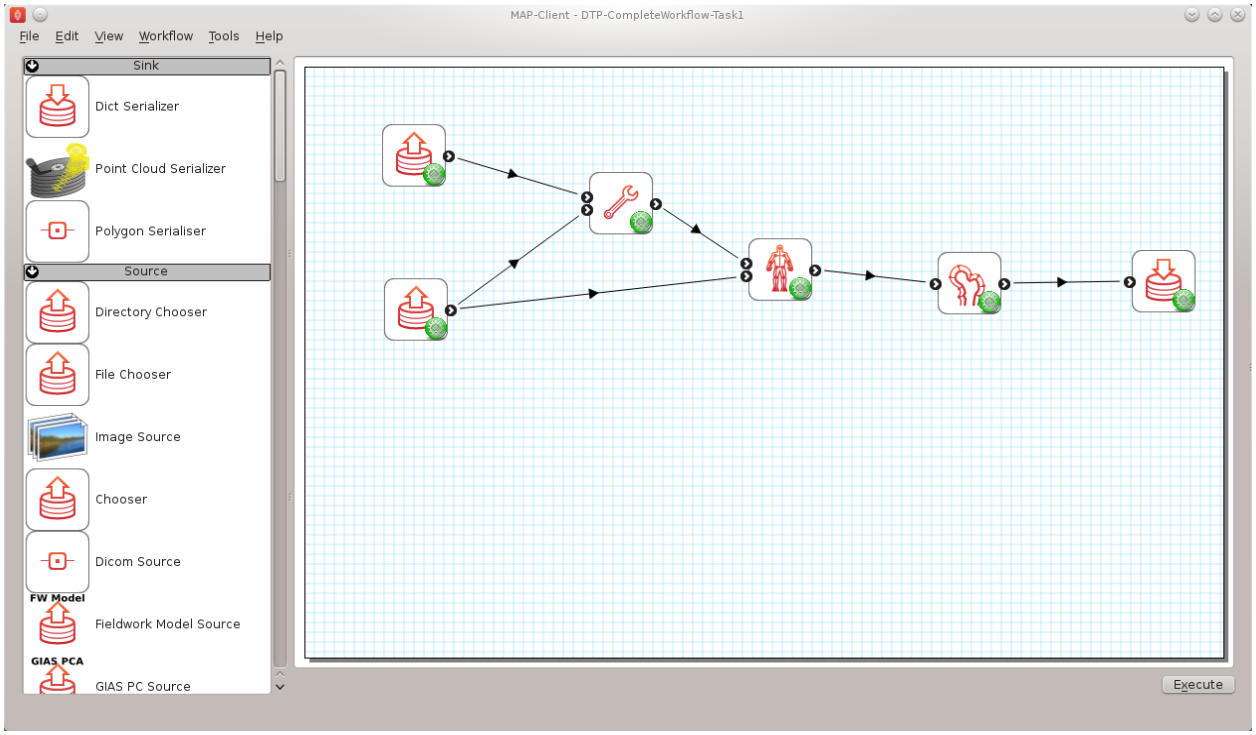


Fig. 1.40: Task 1 workflow horse fetlock characterisation

In this workflow [shown in Fig. 1.40] there are six steps. It starts with reading in a database of horse fetlock mesh coordinate frame definitions and the selection of a fetlock mesh. The corresponding coordinate frame definition is read in for the mesh selected and passed to the 'Hoof Measurement' step along with the fetlock mesh itself. The first three steps in this workflow are non-interactive steps, they are configured proir to executing the workflow. The 'Hoof Measurement' step is an interactive step for making measurements along the sagittal ridge of the fetlock joint.

Fig. 1.41 shows the initial view of the 'Hoof Measurement' step. What we see is some controls on the left handside and on the right a mesh of the fetlock joint cut in half by the segmentation plane. We are going to measure the height of the sagittal ridge. To do this we must place seven points along the line of ridge at three different segmentation plane angles. The seven points must define the line forming the base of the plane and the peak of the ridge. The order that the points are placed on the segmentation plane is not important, however it is important that the segmentation point used to mark the peak of the ridge is the fourth point counted from the left-hand or right-hand side of the plane. Fig. 1.42 shows seven points placed on the segmentation plane with the plane angle set at zero degrees.

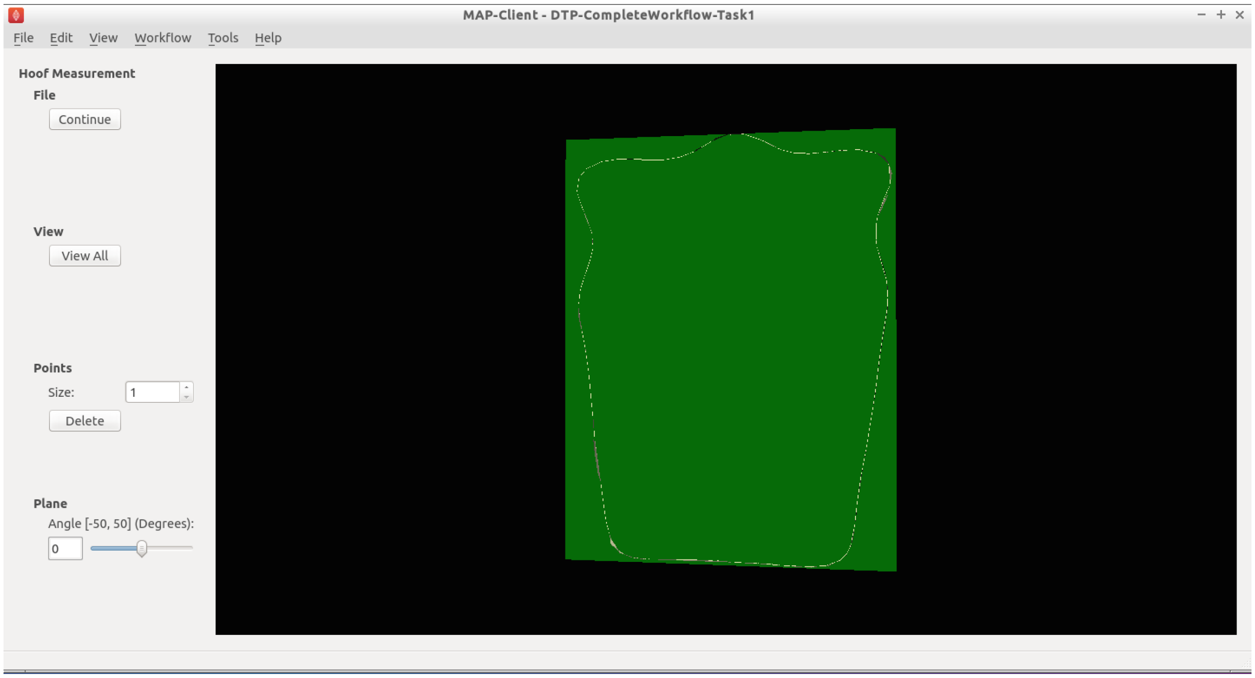


Fig. 1.41: Initial view of the 'Hoof measurement' step

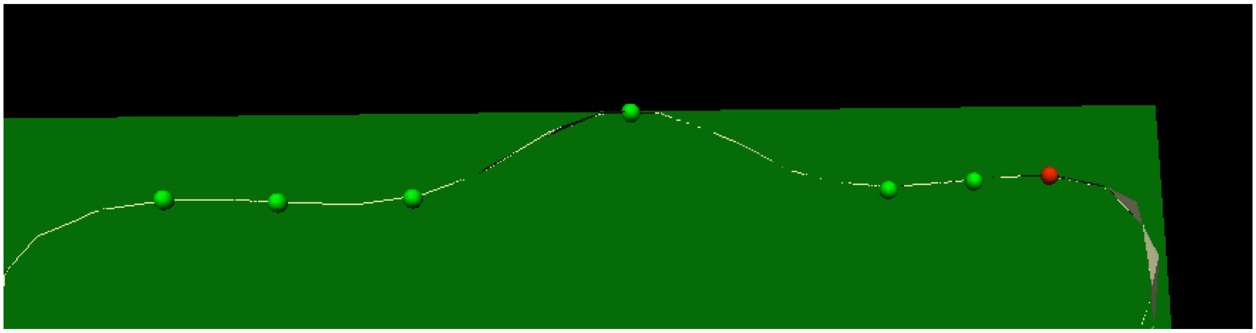


Fig. 1.42: Segmented ridge of fetlock joint at 0 degrees

As shown, six points are used to define the base plane of the joint with the innermost segmentation points of the plane used to mark the start of the rise of the ridge. The middle (or fourth point) is used to mark the peak of the ridge.

For the anaylsis of the ridge we are required to segment the ridge at three different segment plane angles. The angles that we require are +30 degrees, 0 degrees and -30 degrees. The process for the segmentation at each plane angle is the same as outlined above.

Once we have completed segmenting the sagittal ridge for the three plane angles we have completed the interactive part of this workflow. The remaining two steps are the 'Hoof Point Anaylzer' step and the 'Dict Serializer' step.

The 'Hoof Point Analyzer' step takes the segmented data and characterises the hoof throught the height and the angle of the sagittal ridge. This information is then stored to disk in the 'Dict Serializer' step.

At this point the workflow stops but it could carry on to adding the new measurement to a database of such measurements and from there infer from the analysis some risk of injury for the horse. The next workflow in this section will show a much more comprehensive and complex physiological modelling cycle.

Patient Femur Analysis Workflow

Task two is to execute and follow through a (almost) complete computational physiology cycle. For this task we will take a set of MR images of the knee joint and some fudicial markers taken from the patient so that we can make a set of measurements that are consistent across all patients and provide data for a clinician to prepare the best treatment for the patient.

In Fig. 1.43 we have the workflow for analysing a patient's femur. It is a very complex workflow and by far the most advanced that we have seen. We can of course use this workflow to analyse any bone that we have the required population based principal component model data for. At this time we have this data for all the major bones of the lower limb, but here we focus on the femur.

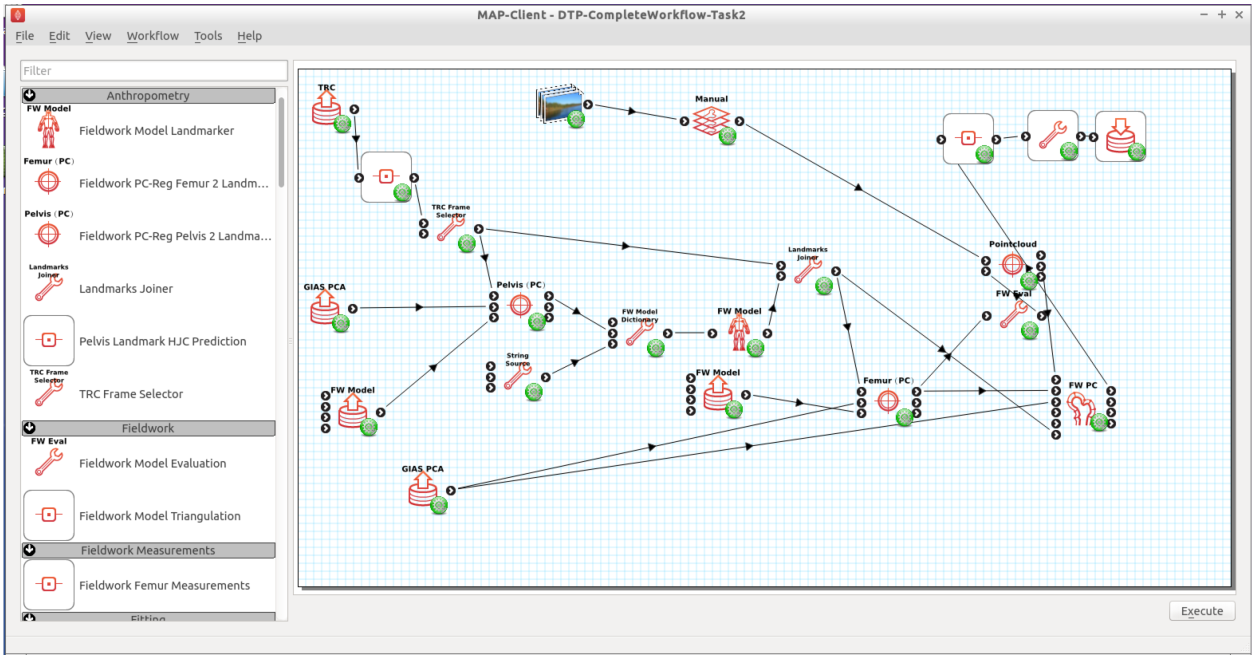


Fig. 1.43: Task 2 patient femur analysis workflow.

This workflow as previously stated is rather complex, it has a number of interactive steps and non-interactive steps. There are a number of entry points and it is unknown which one you will come across first, so the order of the steps that you work through may not be the same order as written here.

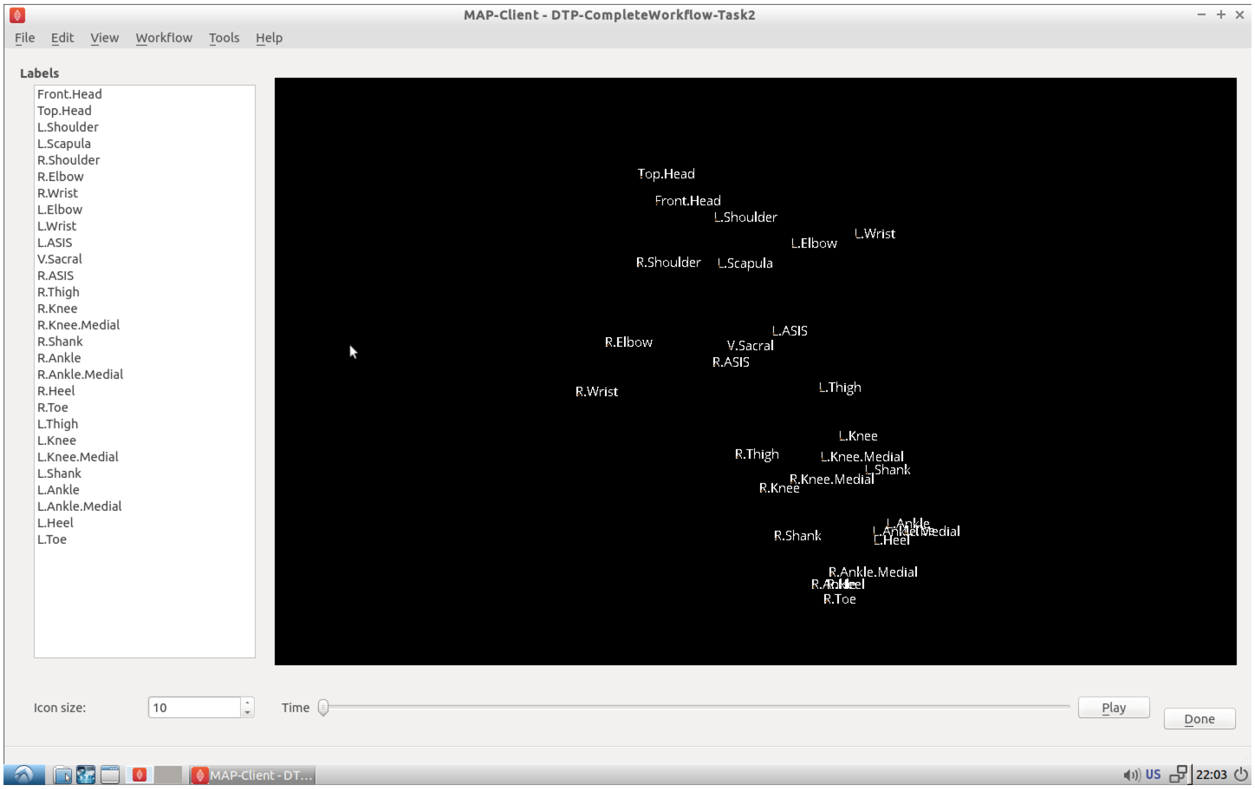
Patient Femur Analysis Overview

The patient femur analysis workflow starts with four inputs; MR images of the knee joint, motion capture data, population mean hip model, population mean femur model. Because MR images are expensive and taking a full MR image stack of the femur for a knee joint problem is not feasable we must augment the MR images with motion capture (MOCAP) data of the patient. In doing this we are able to construct a set of points to fit a femur model to the patient. In the case of the femur we require some internal points that are not available directly from the MOCAP data, but with the help of a hip model we can determine the internal feature points that are required. With a patient specific model we are able to take a set of consistent measurements that a clinician can make use of to determine the best course of treatment for the patient.

Workflow walkthrough

As stated previously the workflow has four entry points, the order in which they are executed is not determinable so here we follow a possible order from which you might have to deviate.

Luckily two of the entry points are non-interactive and are setup when the steps are configured. For these steps we are simply selecting a model and the population principal components from which the model can be modified. We require two models in the case of the femur because the fiducial markers taken from the MOCAP data does not specify the femur hip joint centre that is required for fitting the femur successfully. To satisfy this requirement we first register the pelvis model and then take the hip joint centre landmarks from the fitted model. Thus augmenting the fiducial marker set obtained from the MOCAP data.



Icon size: 10

Fig. 1.44: MOCAP data viewer showing the fiducial markers location and their associated labels.

The first interactive step that occurs when executing the workflow is the MOCAP viewer step (shown in Fig. 1.44). In this step we can check that the MOCAP data is complete and has the correct names for the markers. Correct names are not required but it does help the software automatically select the correct fiducial marker later on in the workflow. It is important however for the fitting of the femur for the MOCAP data to have the knee medial and lateral points marked and anterior superior iliac spine point marked. In our case we are fitting the left femur so we require these fiducial markers on the left side of the subject. Using the MOCAP viewer step check that the 'L.Knee', 'L.Knee.Lateral', 'L.ASIS' and 'V.SACRAL' fiducial markers are present. The list box on the left has a list of all the fiducial markers availble and selecting an entry in this list will highlight that marker in the 3D view of the data.

When you are satisfied that these feducial markers are present continue on to the next step. The next step is the segmentation step, we need to segment the distal end of the femur so that we can fit the model to later. We do not require a lot of segmentation points for the fit an advantage afforded to us when using PCA models.

Segmentation Step

The segmentation step is a reasonably advanced step that affords us the ability of segmenting an image stack with individual segmentation points or besier curves. It also allows us to manipulate the segmentation plane in two ways; the first is the ability to move the plane in the direction of the normal for the plane, the second is the ability to change the orientation of the plane. These four modes are available through the toolbar at the top of the window (Fig. 1.45).

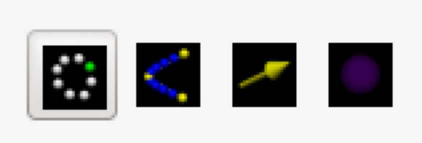


Fig. 1.45: Segmentation toolbar showing the icons for the different tools available to the user.

Segment

In the segment mode segmentation points can be added by using the ctrl key modifier and the left mouse button. Unwanted segmentation points can be removed by selecting them and pressing the delete key or using the delete button on the segmentation panel on the left-hand side. Note that it is not possible to delete a besier curve using the segmentation point delete button or vice versa.

Beseir

In the Beseir mode segmentation points can be added along a curve defined by control points with extra segmentation points placed between the control points automatically, the number of automatically added segmentation points can be changed through the spin box in the Beseir panel on the left-hand side. Besier curves can be deleted by first selecting a curve and pressing the delete key or throught the delete button under the Besier panel. Note that it is not possible to delete a segmentation point when using the Besier curve delete button or vice versa.

Normal

In the normal mode a yellow arrow will be visible this arrow represents the normal of the segmentation plane. With the normal arrow selected (when selected the arrow will be orange) we can move the segmentation plane forwards and backwards in the direction of the segmenation plane normal.

Orientation

In the orientation mode a purple sphere will be visible in the centre of the segmentation plane. In this mode when we attempt to orient the scene with the left mouse button it is the segmentation plane that is oreintated and not the scene.

For our needs in this situation we don't require to segment every feature to the minutest of details. We do need to concentrate on getting the pertinent aspects of the distal end of the femur segmented though. Segmenting the condials and XXXXX parts of the femur is a must, essentially we must add segmentation points over the features of the femur to enable an accurate final fit of the model. Because we are targeting a segmentation for a PCA based model we can segment as little as 20 points and still achieve a satisfactory result.

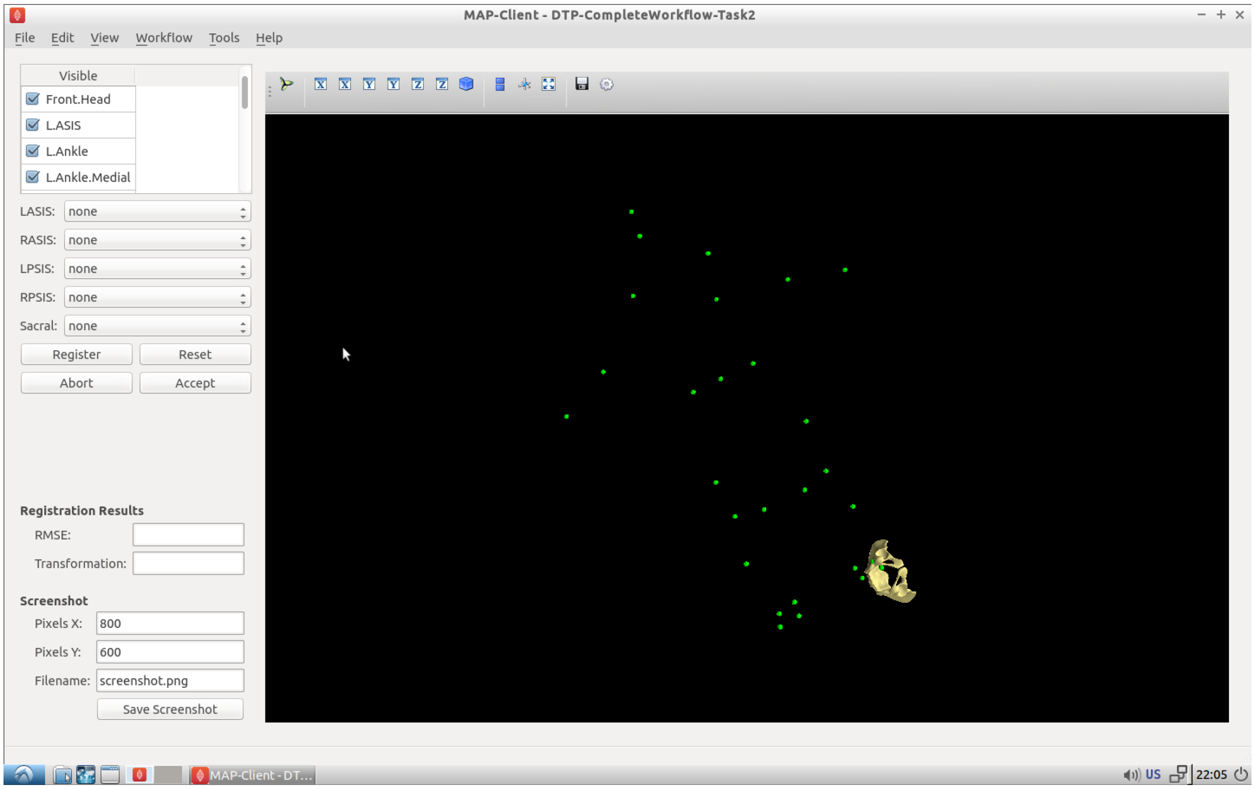
You are able to load a pre-prepared segmentation using the load button on under the file tab. It is also possible to save your own segmentation using the save button under the file tab. But beware that at this point you can only havea single saved segmention, so using the save button will overwrite the pre-prepared segmentation.

Once the segmentation is finished continue on to the registration phase of the workflow.

Registration

The registration phase of the workflow consists of registering the population based model of the hip and femur to the MOCAP fiducial markers, we also need to register the segmentation points defined in image coordinate system to the patient coordinate system.

In Fig. 1.46 we see on the left-hand side a list of check boxes for controlling the visibility of the fiducial markers, five combo boxes to assign model landmark points to fiducial marker points, four buttons to perfom a registration and reject or accept the registration, two boxes that display error measurements of the fitted model and some controls for taking a screen shot. On the right-hand side we see a 3D view of the pelvis model and the fiducial markers.



☑ Front.Head
☑ L.ASIS
☑ L.Ankle
☑ L.Ankle.Medial

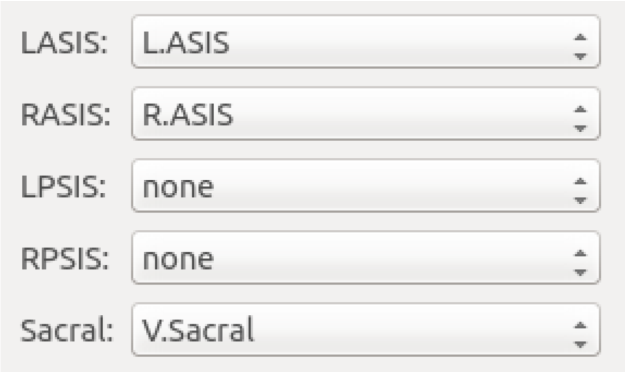
LASIS: none
RASIS: none
LPSIS: none
RPSIS: none
Sacral: none

Registration Results
RMSE:
Transformation:

Screenshot
Pixels X: 800
Pixels Y: 600
Filename: screenshot.png

Fig. 1.46: Initial view of the register pelvis step.

To register the pelvis model to the fiducial markers we must assign the model landmark points to the appropriate fiducial markers. For the registration to work we must choose at least three points that are not co-linear. The landmark points 'L.ASIS', 'R.ASIS' and 'V.SACRAL' satisfy this condition, we need to set the combo boxes to have the correct values. Fig. 1.47 shows the correct associations.



LASIS: L.ASIS

RASIS: R.ASIS

LPSIS: none

RPSIS: none

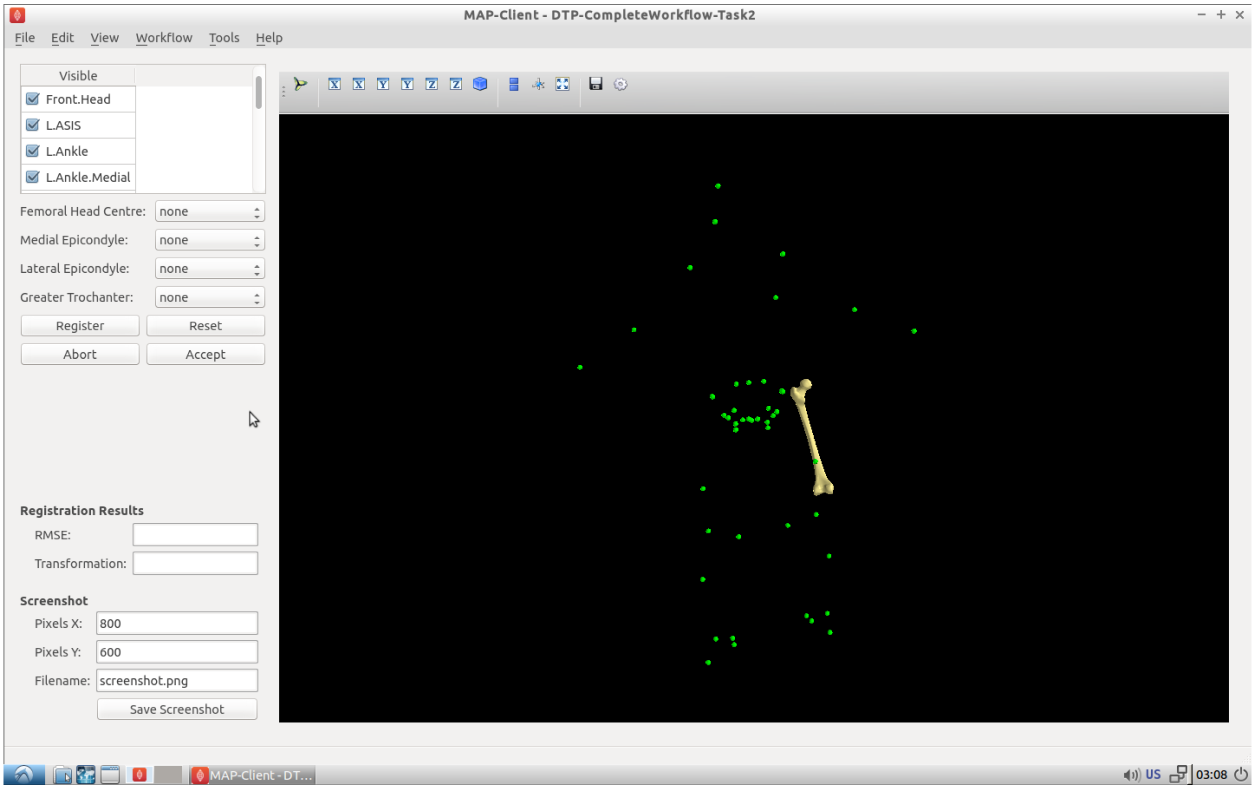
Sacral: V.Sacral

Fig. 1.47: Association of pelvis model landmarks to fiducial markers.

With the correct associations made we can register the model to the markers. The registration process is a three stage process (and if we watch carefully we can see the three stages as they happen); stage one rigid body fit, stage two rigid body fit plus first principal component, stage three rigid body fit plus the first three pricipal components.

Press the register button to perform the registration, if the fit looks correct accept it to continue with the workflow.

Now it is time to register the femur to the subjects feducial markers, in Fig. 1.48 we see a very similar interface to what was seen in the registration of the pelvis (Fig. 1.46). The only difference is that now we have a different set of model landmarks that we are required to associate. If we look at the pelvis region in the 3D view we can see that now there are new feducial markers that were not present in the original MOCAP data. These of course have been generated from the pelvis model so that we can make use of virtual internal markers for fitting the femur.



MAP-Client - DTP-CompleteWorkflow-Task2

Visible

☑ Front.Head

☑ L.ASIS

☑ L.Ankle

☑ L.Ankle.Medial

Femoral Head Centre: none

Medial Epicondyle: none

Lateral Epicondyle: none

Greater Trochanter: none

Registration Results

RMSE:

Transformation:

Screenshot

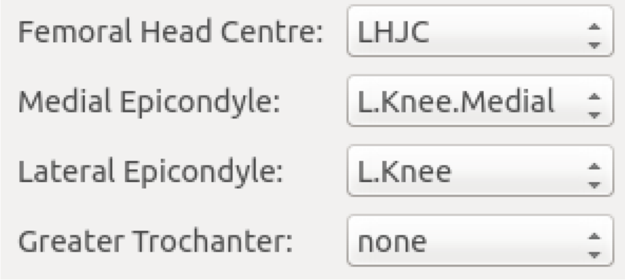
Pixels X: 800

Pixels Y: 600

Filename: screenshot.png

Fig. 1.48: Initial view of the register femur step.

Again we need to associate at least three points that are not co-linear, which is why we required the pelvis model. The first two points that we can identify for the femur registration are the knee medial and knee lateral feducial markers. But we do not have a third feducial marker which we can choose to associate with the femur. However, from the pelvis model we can use the femoral head joint centre landmark as the third point for the registration. Fig. 1.49 shows the correct associations for the left femur model.



Femoral Head Centre: LHJC

Medial Epicondyle: L.Knee.Medial

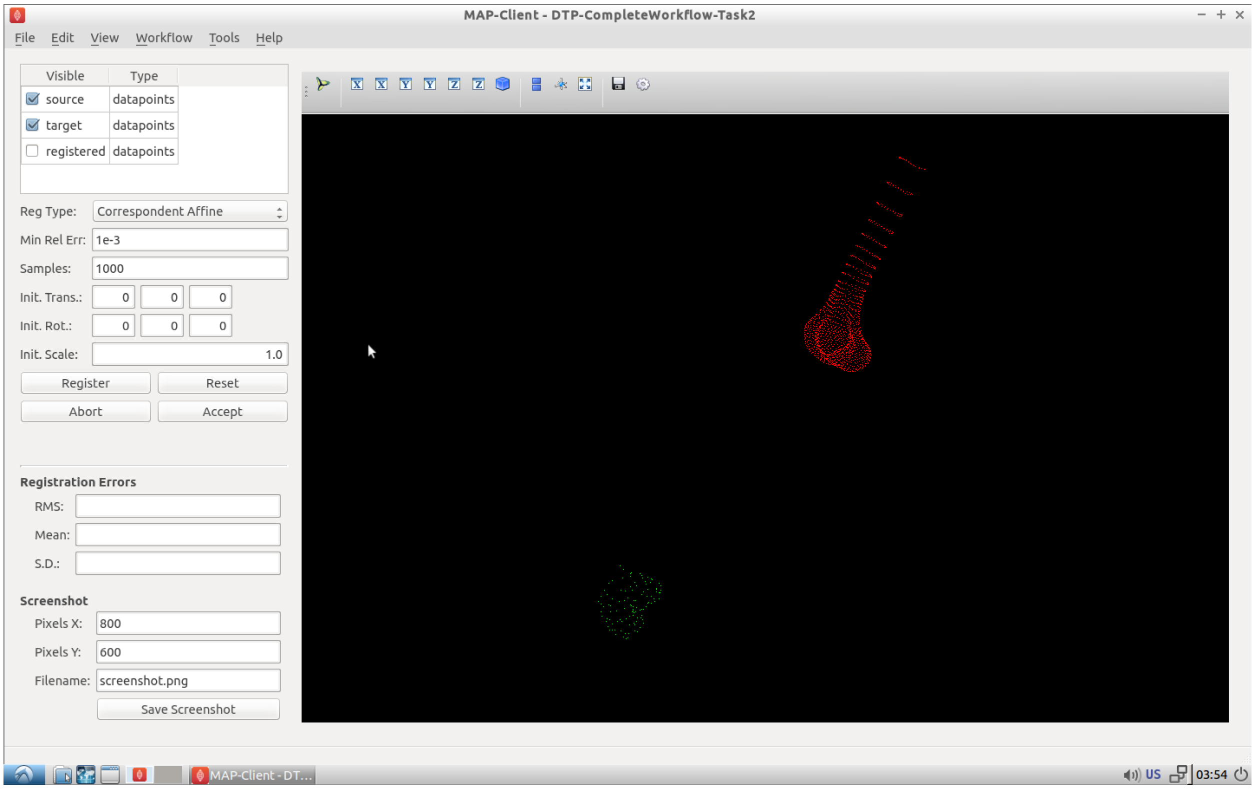
Lateral Epicondyle: L.Knee

Greater Trochanter: none

Fig. 1.49: Association of femur model landmarks to fiducial markers.

Press the register button to perform the registration, if the fit looks correct accept it to continue with the workflow.

At this point in the workflow we need to register the segmented point cloud in magnet coordinates of the images to the subject feducial marker coordinate system. Fig. 1.50



MAP-Client - DTP-CompleteWorkflow-Task2

Visible	Type
☑ source	datapoints
☑ target	datapoints
□ registered	datapoints

Reg Type: Correspondent Affine

Min Rel Err: 1e-3

Samples: 1000

Init. Trans.: 0 0 0

Init. Rot.: 0 0 0

Init. Scale: 1.0

Registration Errors

RMS:

Mean:

S.D.:

Screenshot

Pixels X: 800

Pixels Y: 600

Filename: screenshot.png

Fig. 1.50: Initial view of the register segmented point cloud step.

Again we have a related interface to the two previous registration steps with a few differences. Now instead of five comboboxes we have only one, the only combobox allows us to choose the type of registration to perform. In this situation we want to perform an iterative closest point (ICP) source to target registration. If we were to set this registration type in the registration type combobox and push the register button we get the result as seen in Fig. 1.51.

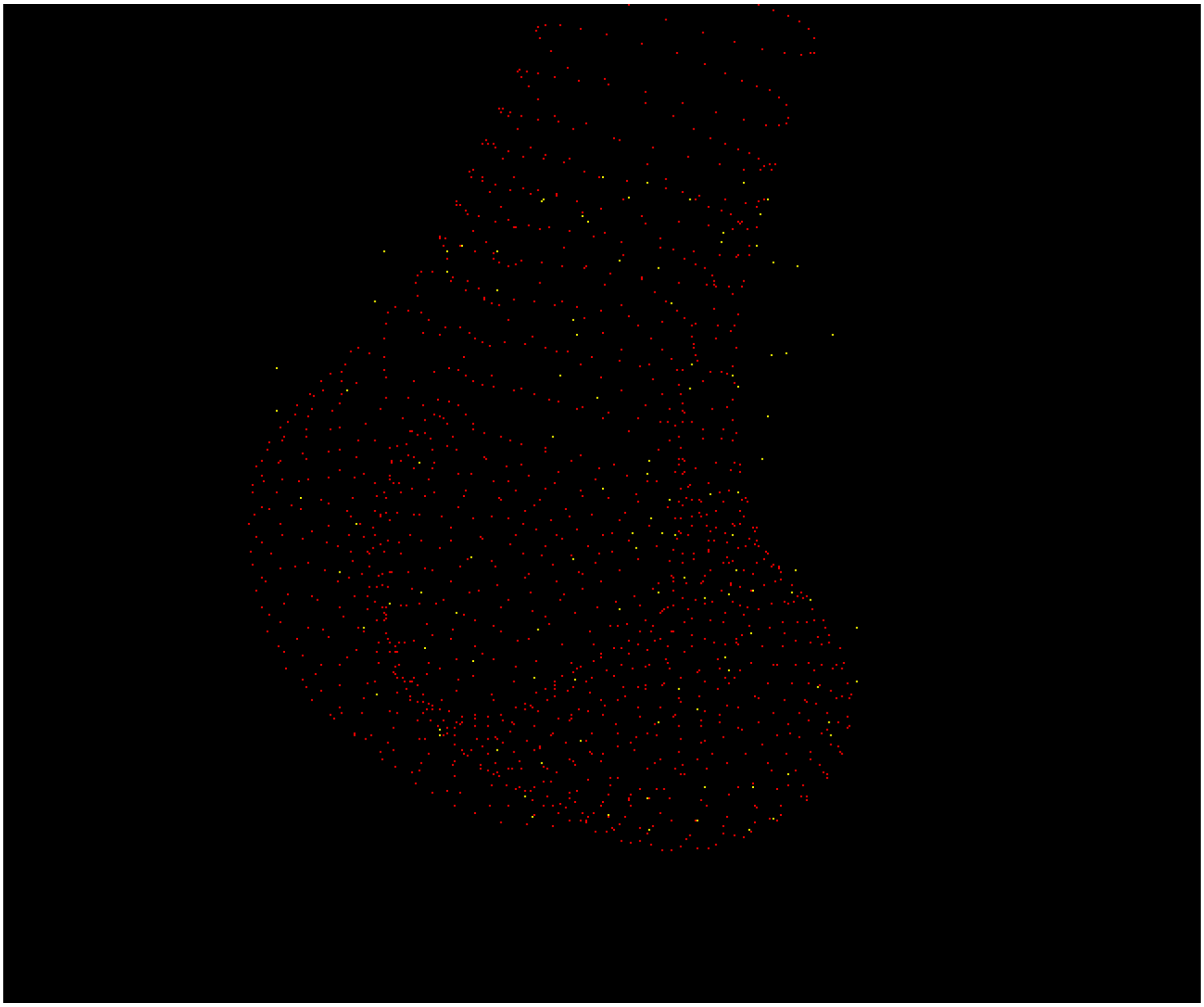
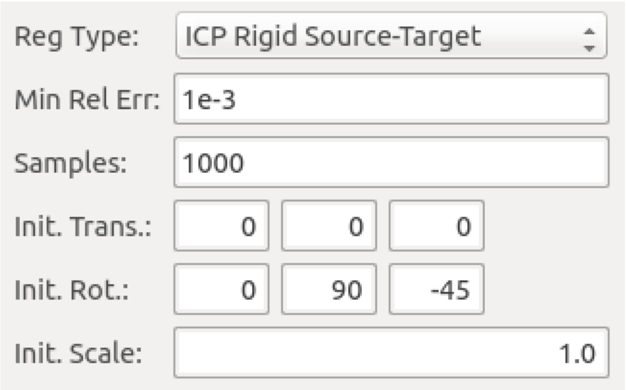


Fig. 1.51: Result of registering the segmented point cloud using ICP source-target registration method with the default settings.

We can see here that some of the segmented points do not correspond very well to the point cloud generated from the femur model, this should not be the case we should have a good correspondence between segmented points and model points. The reason for this discrepency is because the ICP algorithm is very sensitive to initial alignment and the minimum value the algorithm has converged to is a local minimum which is not the global minimum in the solution space. To get a better registration we need to set the initial rotation of the segmented point cloud. In the initial rotation line edit boxes set the values to 0, 90, -45. If you haven't already, use the reset push button to reset the registration and push the register button to perform the registration with the new initial values (see Fig. 1.52 for the correct settings). We should now see a much better alignment of the two point clouds



Reg Type: ICP Rigid Source-Target

Min Rel Err: 1e-3

Samples: 1000

Init. Trans.: 0 0 0

Init. Rot.: 0 90 -45

Init. Scale: 1.0

Fig. 1.52: Settings for the ICP source-target registration method.

Have a look at the fit, if we do not have enough segmented points it will be difficult for the ICP registration to find a satisfactory fit to the model point cloud. In this situation, to get a satisfactory fit, we need to set the initial values so that they are very close to the final values making the registration via ICP redundant.

When you have a satisfactory registration push the accept button.

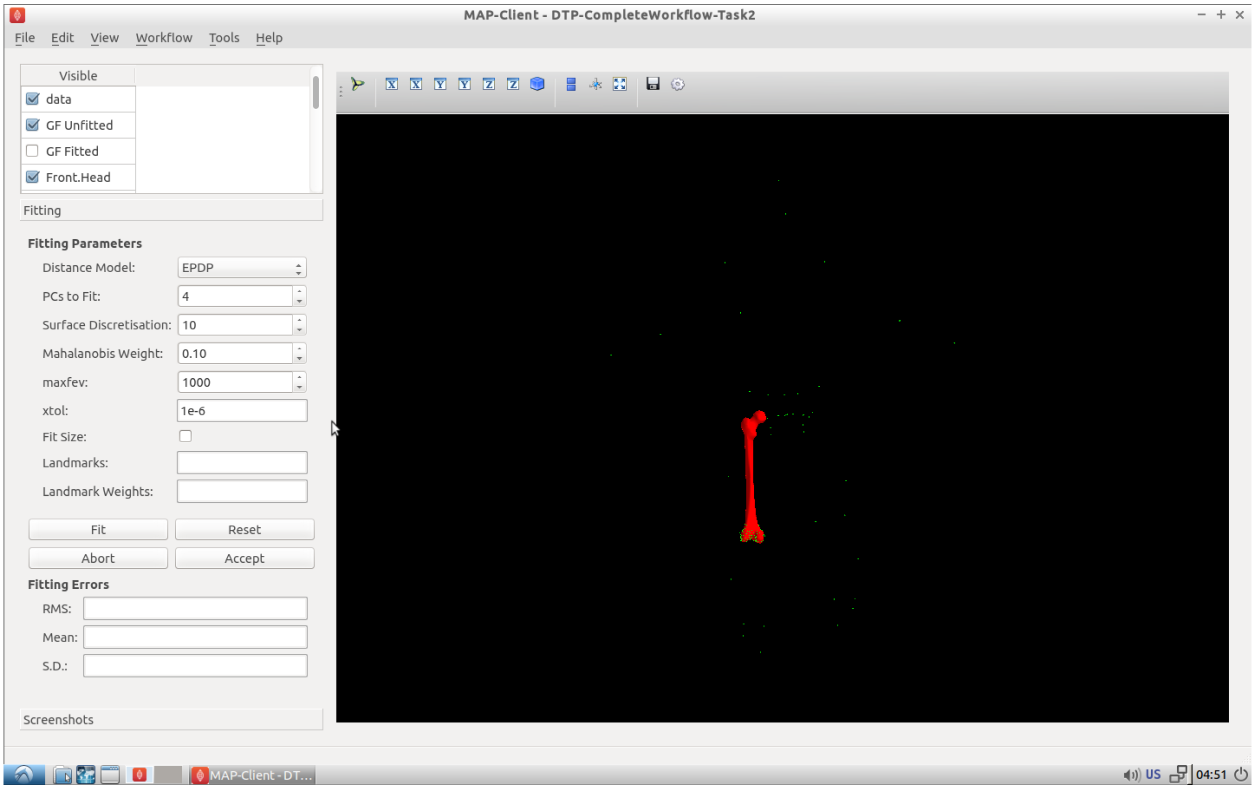
We have now arrived at the last interactive step in this workflow (Fig. 1.53). We need to fit the PCA femur model to the segmented point cloud. Again we see a very similar interface as we have seen previously in the registration steps the difference is that we now have a fitting parameters section. For the fitting we want to fit the datapoints to the element points (DPEP), set the distance model combobox to DPEP to make sure that the we are doing the fit in the correct direction.

If we now fit the data to the model, using the fit push button, we see that the model has been fitted to the segmented point cloud. We can also see that the top of the femur has moved quite a bit as well, this is because we have not constrained that part of the model with any datapoints and thus it is free to move. We can, if so desired, pin the femoral head to the hip joint centre landmark to change this behaviour. For the purposes of this exercise we won't do this but it is something to keep in mind. Fig. 1.54 shows the final fitted model in yellow.

When we push the accept button the workflow will finish to it's conclusion. The last few remaining steps take measurements from the femur model and save them to disk. The idea here is that the measurements will be used to inform a clinician or be used in a tool to aide in the subjects treatment.

1.3 Introduction to (ODE) Modelling Best Practices

Contents:



☑ data
☑ GF Unfitted
□ GF Fitted
☑ Front.Head

Fitting Parameters

Distance Model: EPDP

PCs to Fit: 4

Surface Discretisation: 10

Mahalanobis Weight: 0.10

maxfev: 1000

xtol: 1e-6

Fit Size: □

Landmarks:

Landmark Weights:

Fitting Errors

RMS:

Mean:

S.D.:

Fig. 1.53: Initial view of the model fitting step.

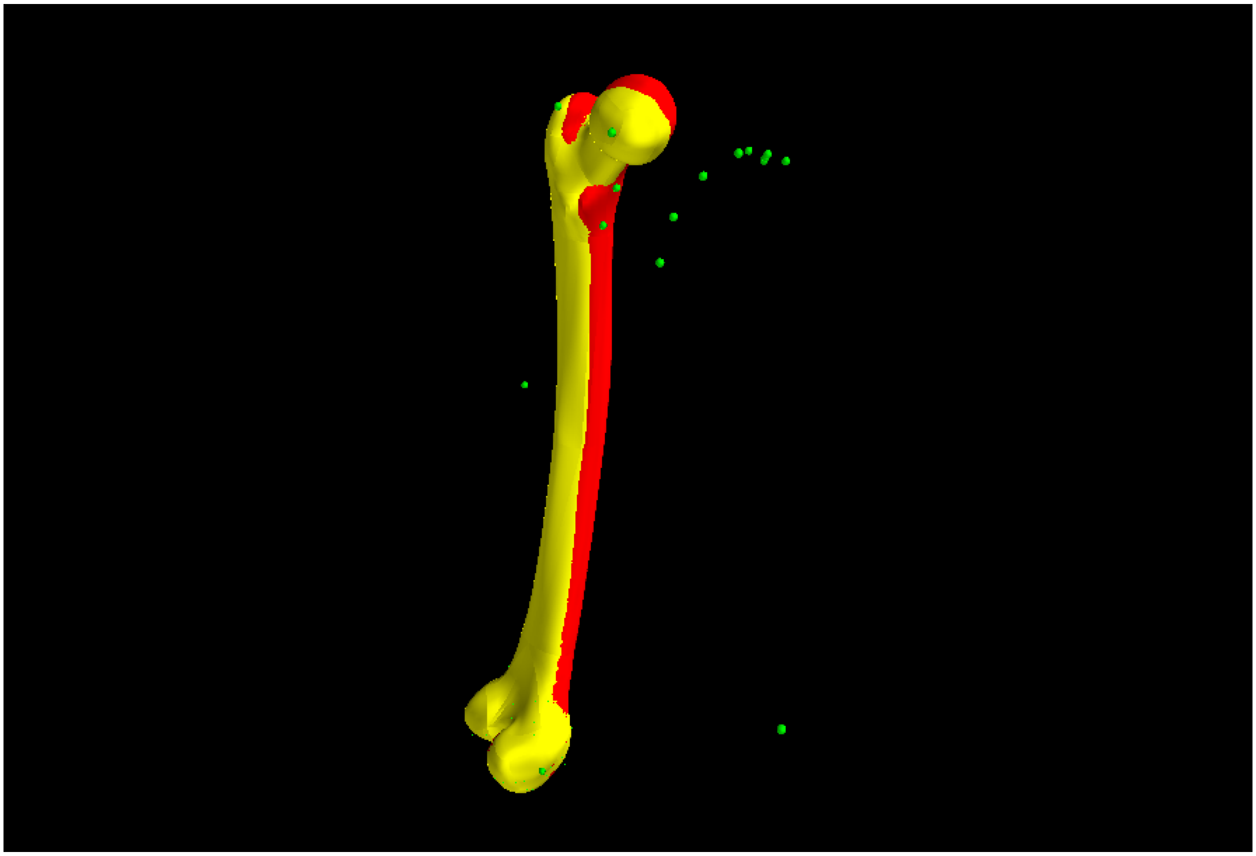


Fig. 1.54: The reference model (red) and the final fitted model (yellow).

In this part of the computational physiology module we demonstrate some modelling practices that we recommend you follow when developing any mathematical model. The key principles are **modularity**, **reuse**, and **reproducibility** - we demonstrate those here using the tool OpenCOR and the CellML and SED-ML formats, but the principles are valid across the spectrum of computational physiology. It is also important to be able to share and collaborate with other scientists and we demonstrate this using the Physiome Model Repository (PMR).

Here are a couple of links in case you need a refer back to some of the material covered earlier in the module.

- *Create and run a simple CellML model: editing and simulation*
- *Open an existing CellML file from a local directory or the Physiome Model Repository*
- *SED-ML, functional curation and Web Lab*

1.3.1 Reproducible and reusable model descriptions

In this section we introduce some of the *key concepts* and *best practices* for ensuring your mathematical models are reproducible and reusable.

Units

A model of ion channel gating and current: Introducing CellML units

A good example of a model based on a first order equation is the one used by Hodgkin and Huxley [AAF52] to describe the gating behaviour of an ion channel (see also next three sections). Before we describe the gating behaviour of an ion channel, however, we need to explain the concepts of the 'Nernst potential' and channel conductance.

An ion channel is a protein or protein complex embedded in the bilipid membrane surrounding a cell and containing a pore through which an ion Y^+ (or Y^-) can pass when the channel is open. If the concentration of this ion is $[Y^+]_o$ outside the cell and $[Y^+]_i$ inside the cell, the force driving an ion through the pore is calculated from the change in *entropy*.

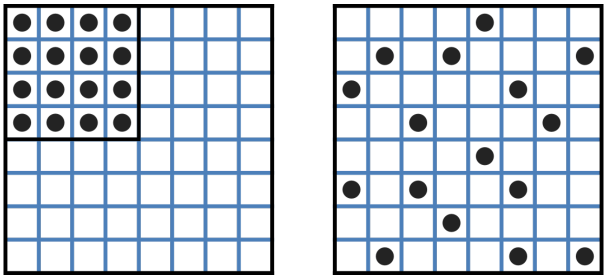


Fig. 1.55: Distribution of microstates in a system [J97]. The 16 particles in a confined region (left) have only one possible arrangement (W = 1) and therefore zero entropy ($k_B$lnW $= 0$). When the barrier is removed and the number of possible locations for each particle increases 4x (right), the number of possible arrangements for the 16 particles increases by 416 and the increase in entropy is therefore $ln(416)$ or $16ln4$. The thermal energy (temperature) of the previously confined particles on the left has been redistributed in space to achieve a more probable (higher entropy) state. If we now added more particles to the container on the right, the concentration would increase and the entropy would decrease.

Entropy S ($J.K^{-1}$) is a measure of the number of microstates available to a system, as defined by Boltzmann's equation $S = k_B$lnW, where W is the number of ways of arranging a given distribution of microstates of a system and k_B is Boltzmann's constant[1]. The driving force for ion movement is the dispersal of energy into a more probable distribution (see Fig. 1.55; cf. the second law of thermodynamics[2]).

The energy change Δq associated with this change of entropy ΔS at temperature T is $\Delta q = T\Delta S$ (J).

For a given volume of fluid the number of microstates W available to a solute (and hence the entropy of the solute) at a high concentration is less than that for a low concentration[3]. The energy difference driving ion movement from

[1] The Brownian motion of individual molecules has energy k_BT (J), where the Boltzmann constant k_B is approximately $1.34x10^{-23}$ ($J.K^{-1}$). At 25°C, or 298K, k_BT = 4.10^{-21} (J) is the minimum amount of energy to contain a 'bit' of information at that temperature.

[2] The *first law of thermodynamics* states that energy is conserved, and the *second law* (that natural processes are accompanied by an increase in entropy of the universe) deals with the distribution of energy in space.

[3] At infinitely high concentration the specified volume is jammed packed with solute and the entropy is zero.

a high ion concentration $[Y^+]_i$ (lower entropy) to a lower ion concentration $[Y^+]_o$ (higher entropy) is therefore

$\Delta q = T\Delta S = k_B T\left(\ln [Y^+]_o - \ln [Y^+]_i\right) = k_B T \ln \frac{[Y^+]_o}{[Y^+]_i}\ (J.ion^{-1})$

or

$\Delta Q = RT \ln \frac{[Y^+]_o}{[Y^+]_i}\ (J.mol^{-1})$.

$R = k_B N_A \approx 1.34x10^{-23}(J.K^{-1})x6.02x10^{23}(mol^{-1}) \approx 8.4(J.mol^{-1}K^{-1})$ is the ‘universal gas constant’[4]. At 25°C (298K), RT $\approx 2.5kJ.mol^{-1}$.

Every positively charged ion that crosses the membrane raises the potential difference and produces an electrostatic driving force that opposes the entropic force (see Fig. 1.56). To move an electron of charge *e* ($\approx 1.6x10^{-19}$ C) through a voltage change of $\Delta\phi$ (V) requires energy $e\Delta\phi$ (J) and therefore the energy needed to move an ion Y^+ of valence *z=1* (the number of charges per ion) through a voltage change of $\Delta\phi$ is ze$\Delta\phi$ $(J.ion^{-1})$ or ze$N_A\Delta\phi$ $(J.mol^{-1})$. Using Faraday's constant $F = eN_A$, where $F \approx 0.96x10^5 C.mol^{-1}$, the change in energy density at the macroscopic scale is zF$\Delta\phi$ $(J.mol^{-1})$.

No further movement of ions takes place when the force for entropy driven ion movement exactly equals the opposing electrostatic driving force associated with charge movement:

zF$\Delta\phi$ = RT $\ln \frac{[Y^+]_o}{[Y^+]_i}$ $(J.mol^{-1})$ or $\Delta\phi = E_Y = \frac{\text{RT}}{\text{zF}} \ln \frac{[Y^+]_o}{[Y^+]_i}$ $(J.C^{-1}$ or V)

where E_Y is the ‘equilibrium’ or ‘Nernst’ potential for Y^+. At 25°C (298K), $\frac{\text{RT}}{F} = \frac{2.5x10^3}{0.96x10^5}(J.C^{-1}) \approx 25mV$.

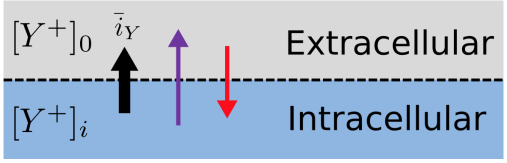


Fig. 1.56: The balance between entropic and electrostatic forces determines the Nernst potential.

For an open channel the electrochemical current flow is driven by the open channel conductance g_Y times the difference between the transmembrane voltage V and the Nernst potential for that ion:

i_Y=g_Y $(V - E_Y)$.

This defines a linear current-voltage relation (‘Ohms law’) as shown in Fig. 1.57. The gates to be discussed below modify this open channel conductance.

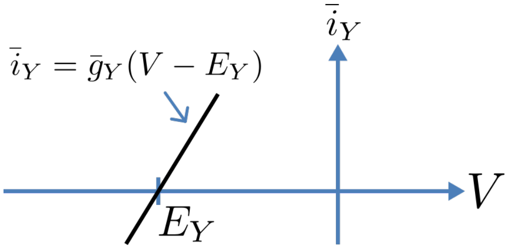


Fig. 1.57: Open channel linear current-voltage relation

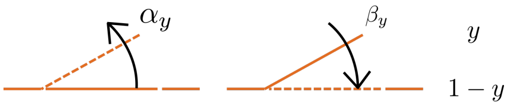
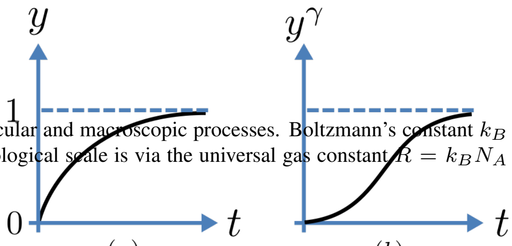


Fig. 1.58: Ion channel gating kinetics. y is the fraction of gates in the open state. α_y and β_y are the rate constants for opening and closing, respectively.

To describe the time dependent transition between the closed and open states of the channel, Hodgkin and Huxley introduced the idea of channel gates that control the passage of ions through a membrane ion channel. If the fraction of gates that are open is y, the fraction of



(a) (b)

Fig. 1.59: Transient behaviour for one gate (left) and γ gates in series (right). Note that the right hand graph has an initial S-shaped increase, reflecting the multiple gates in series.

[4] N_A is Avogadro's number ($6.023x10^{23}$) and is the scaling factor between molecular and macroscopic processes. Boltzmann's constant k_B and electron charge e operate at the atomic/molecular scale. Their effect at the physiological scale is via the universal gas constant $R = k_B N_A$ and Faraday's constant $F = eN_A$.

gates that are closed is $1 - y$, and a first order ODE can be used to describe the transition between the two states (see Fig. 1.58):

$$\frac{\mathrm{dy}}{\mathrm{dt}} = \alpha_y (1 - y) - \beta_y.\mathrm{y}$$

where $\alpha_y$is the opening rate and β_y is the closing rate.

The solution to this ODE is

$$y = \frac{\alpha_y}{\alpha_y + \beta_y} + Ae^{-(\alpha_y + \beta_y)t}$$

The constant A can be interpreted as $A = y(0) - \frac{\alpha_y}{\alpha_y + \beta_y}$ as in the previous example and, with $y(0) = 0$ (i.e. all gates initially shut), the solution looks like Fig. 1.59(a).

The experimental data obtained by Hodgkin and Huxley for the squid axon, however, indicated that the initial current flow began more slowly (Fig. 1.59(b)) and they modelled this by assuming that the ion channel had γ gates in series so that conduction would only occur when all gates were at least partially open. Since y is the probability of a gate being open, y^γ is the probability of all γ gates being open (since they are assumed to be independent) and the current through the channel is

$$i_Y = i_Y y^\gamma = y^\gamma g_Y (V - E_Y)$$

where i_Y= $g_Y (V - E_Y)$ is the steady state current through the open gate.

We can represent this in OpenCOR with a simple extension of the first order ODE model, but in developing this model we will also demonstrate the way in which CellML deals with units.

Note that the decision to deal with units in CellML, rather than just ignoring them or insisting that all equations are represented in dimensionless form, was made in order to be able to check the physical consistency of all terms in each equation[5].

There are seven base physical quantities defined by the *International d'Unités* (SI)[6]. These are (with their SI units):

- **length** (meter or m)
- **time** (second or s)
- **amount of substance** (mole)
- **temperature** (K)
- **mass** (kilogram or kg)
- **current** (amp or A)
- **luminous intensity** (candela).

All other units are derived from these seven. Additional derived units that CellML defines intrinsically (with their dependence on previously defined units) are: **Hz** (s^{-1}); **Newton**, N ($kg.m.s^{-1}$); **Joule**, J ($N.m$); **Pascal**, Pa ($N.m^{-2}$); **Watt**, W ($J.s^{-1}$); **Volt**, V ($W.A^{-1}$); **Siemen**, S ($A.V^{-1}$); **Ohm**, Ω ($V.A^{-1}$); **Coulomb**, C ($s.A$); **Farad**, F ($C.V^{-1}$); **Weber**, Wb ($V.s$); and **Henry**, H ($Wb.A^{-1}$). Multiples and fractions of these are defined as follows:

[5] It is well accepted in engineering analysis that thinking about and dealing with units is a key aspect of modelling. Taking the ratio of dimensionally consistent terms provides non-dimensional numbers which can be used to decide when a term in an equation can be omitted in the interests of modelling simplicity. We investigate this idea further in a later section.

[6] http://en.wikipedia.org/wiki/International_System_of_Units

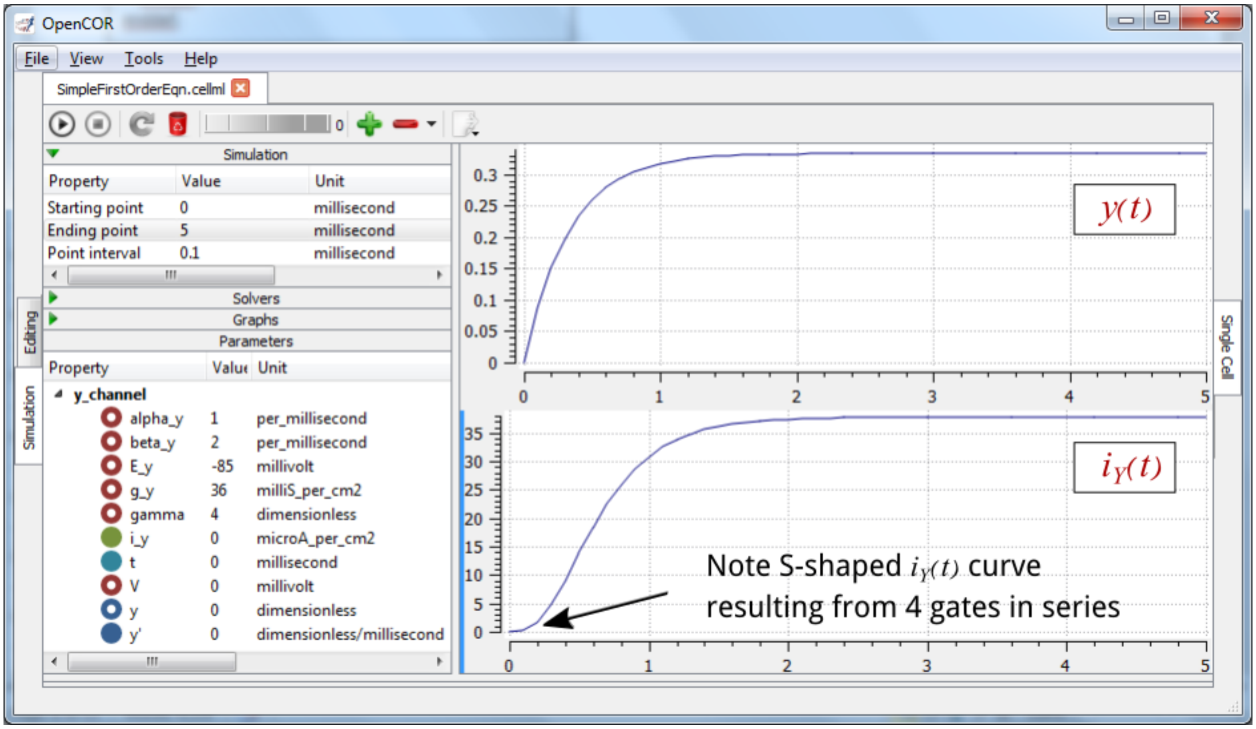
	Prefix		deca	hecto	kilo	mega	giga	tera	peta	exa	zetta	yotta
Multiples	Symbol		da	h	k	M	G	T	P	E	Z	Y
	Factor	10^0	10^1	10^2	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}
	Prefix		deci	centi	milli	micro	nano	pico	femto	atto	zepto	yocto
Fractions	Symbol		d	c	m	μ	n	p	f	a	z	y
	Factor	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}

Units for this model, with multiples and fractions, are illustrated in the following *CellML Text* code:

```
1  def model first_order_model as
2      def unit millisec as
3          unit second {pref: milli};
4      enddef;
5      def unit per_millisec as
6          unit second {pref: milli, expo: -1};
7       enddef;
8      def unit millivolt as
9          unit volt {pref: milli};
10     enddef;
11     def unit microA_per_cm2 as
12         unit ampere {pref: micro};
13         unit metre {pref: centi, expo: -2};
14     enddef;
15     def unit milliS_per_cm2 as
16         unit siemens {pref: milli};
17         unit metre {pref: centi, expo: -2};
18     enddef;
19     def comp ion_channel as
20         var V: millivolt {init: 0};
21         var t: millisec {init: 0};
22         var y: dimensionless {init: 0};
23         var E_y: millivolt {init: -85};
24         var i_y: microA_per_cm2;
25         var g_y: milliS_per_cm2 {init: 36};
26         var gamma: dimensionless {init: 4};
27         var alpha_y: per_millisec {init: 1};
28         var beta_y: per_millisec {init: 2};
29         ode(y, t) = alpha_y*(1{dimensionless}-y)-beta_y*y;
30         i_y = g_y*pow(y, gamma)*(V-E_y);
31      enddef;
32 enddef;
```

Line 2: Define units for time as millisecs
Line 5: Define per_millisec units
Line 8: Define units for voltage as millivolts
Line 11: Define units for current as microAmps per cm^2
Line 15: Define units for conductance as milliSiemens per cm^2
Lines 20-28: Define units and initial conditions for variables
Line 29: Define ODE for gating variable y
Line 30: Define channel current

The solution of these equations for the parameters indicated above is illustrated in Fig. 1.60.



Property	Value	Unit
Starting point	0	millisecond
Ending point	5	millisecond
Point interval	0.1	millisecond

Property	Value	Unit
y_channel		
alpha_y	1	per_millisecond
beta_y	2	per_millisecond
E_y	-85	millivolt
g_y	36	milliS_per_cm2
gamma	4	dimensionless
i_y	0	microA_per_cm2
t	0	millisecond
V	0	millivolt
y	0	dimensionless
y'	0	dimensionless/millisecond

Note S-shaped $i_Y(t)$ curve resulting from 4 gates in series

Fig. 1.60: The behaviour of an ion channel with $\gamma = 4$ gates transitioning from the closed to the open state at a membrane voltage $V = 0$ (OpenCOR link). The opening and closing rate constants are $\alpha_y = 1$ ms^{-1} and $\beta_y = 2$ ms^{-1}. The ion channel has an open conductance of $g_Y = 36$ mS.cm^{-2} and an equilibrium potential of $E_Y = -85$ mV. The upper transient is the response $y(t)$ for each gate and the lower trace is the current through the channel. Note the slow start to the current trace in comparison with the single gate transient $y(t)$.

The model of a gated ion channel presented here is used in the next two sections for the neural potassium and sodium channels and then in Section 11 for cardiac ion channels. The gates make the channel conductance time dependent and, as we will see in the next section, the experimentally observed voltage dependence of the gating rate constants α_y and β_y means that the channel conductance (including the open channel conductance) is voltage dependent. For a partially open channel ($y < 1$), the steady state conductance is $(y_\infty)^\gamma .g_Y$, where $y_\infty = \frac{\alpha_y}{\alpha_y+\beta_y}$. Moreover the gating time constants $\tau = \frac{1}{\alpha_y+\beta_y}$ are therefore also voltage dependent. Both of these voltage dependent factors of ion channel gating are important in explaining channel properties, as we show now for the neural potassium and sodium ion channels.

The **first concept** is that everything in your model should have clearly defined physical dimensions (or at least a clear statement that the quantity has no physical dimension). This ensures that when a scientist reuses a model they are able to use software tools to check for dimensional consistency and even potentially automatically convert quantities as required.

- Take a look through the section *A model of ion channel gating and current: Introducing CellML units* for a discussion and some examples demonstrating the use of units in CellML and OpenCOR.

Modularity

A model of the potassium channel: Introducing CellML components and connections

We now deal specifically with the application of the previous model to the Hodgkin and Huxley (HH) potassium channel. Following the convention introduced by Hodgkin and Huxley, the gating variable for the potassium channel is n and the number of gates in series is $\gamma = 4$, therefore

$$i_K = \bar{i}_K n^4 = n^4 \bar{g}_K \left(V - E_K \right)$$

where $\bar{g}_K = 36\text{mS.cm}^{-2}$, and with intra- and extra-cellular concentrations $\left[K^+\right]_i = 90\text{mM}$ and $\left[K^+\right]_o = 3\text{mM}$, respectively, the Nernst potential for the potassium channel ($z = 1$ since one +ve charge on K^+) is

$$E_k = \frac{\text{RT}}{\text{zF}} \, ln \frac{\left[K^+\right]_o}{\left[K^+\right]_i} = 25 \, ln \frac{3}{90} = -85\text{mV}.$$

As noted above, this is called the *equilibrium potential* since it is the potential across the cell membrane when the channel is open but no current is flowing because the electrostatic driving force from the potential (voltage) difference between internal and external ion charges is exactly matched by the entropic driving force from the ion concentration difference. $n^4 \bar{g}_K$ is the channel conductance.

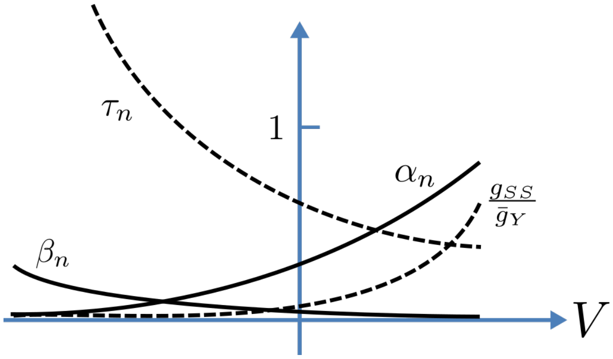


Fig. 1.61: Voltage dependence of rate constants α_n and β_n (ms^{-1}), time constant τ_n (ms) and relative conductance $\frac{g_{SS}}{\bar{g}_Y}$.

The gating kinetics are described (as before) by

$$\frac{\text{dn}}{\text{dt}} = \alpha_n \left(1 - n \right) - \beta_n . \text{n}$$

with time constant $\tau_n = \frac{1}{\alpha_n + \beta_n}$ (see *A simple first order ODE*).

The main difference from the gating model in our previous example is that Hodgkin and Huxley found it necessary to make the rate constants functions of the membrane potential V (see Fig. 1.61) as follows[1]:

$$\alpha_n = \frac{-0.01(V+65)}{e^{\frac{-(V+65)}{10}} - 1}; \beta_n = 0.125 e^{\frac{-(V+75)}{80}} .$$

Note that under steady state conditions when $t \to \infty$ and $\frac{\text{dn}}{\text{dt}} \to 0, \quad n|_{t=\infty} = n_\infty = \frac{\alpha_n}{\alpha_n + \beta_n}$.

The voltage dependence of the steady state channel conductance is then

$$g_{\text{SS}} = \left(\frac{\alpha_n}{\alpha_n + \beta_n} \right)^4 . \bar{g}_Y .$$

(see Fig. 1.61). The steady state current-voltage relation for the channel is illustrated in Fig. 1.62.

These equations are captured with OpenCOR *CellML Text* view (together with the previous unit definitions) below. But first we need to explain some further CellML concepts.

[1] The original expression in the HH paper used $\alpha_n = \frac{0.01(v+10)}{e^{\frac{(v+10)}{10}} - 1}$ and $\beta_n = 0.125 e^{\frac{v}{80}}$, where v is defined relative to the resting potential (-75mV) with +ve corresponding to +ve *inward* current and $v = -(V + 75)$.

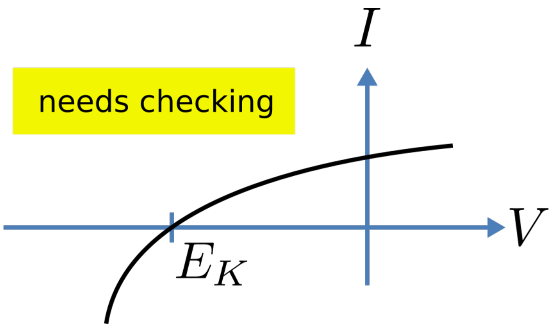


Fig. 1.62: The steady-state current-voltage relation for the potassium channel.

We introduced CellML units above. We now need to introduce three more CellML constructs: components, connections (mappings between components) and groups. For completeness we also show one other construct in Fig. 1.63, imports, that will be used later in *A model of the nerve action potential: Introducing CellML imports*.

Defining components serves two purposes: it preserves a modular structure for CellML models, and allows these component modules to be imported into other models, as we will illustrate later [DPPJ03]. For the potassium channel model we define components representing (i) the environment, (ii) the potassium channel conductivity, and (iii) the dynamics of the n-gate.

Since certain variables (t, V and n) are shared between components, we need to also define the component maps as indicated in the *CellML Text* view below.

The *CellML Text* code for the potassium ion channel model is as follows[2]:

Potassium_ion_channel.cellml

```
def model potassium_ion_channel as
   def unit millisec as
      unit second {pref: milli};
   enddef;
   def unit per_millisec as
      unit second {pref: milli, expo: -1};
   enddef;
   def unit millivolt as
      unit volt {pref: milli};
   enddef;
   def unit per_millivolt as
      unit millivolt {expo: -1};
   enddef;
   def unit per_millivolt_millisec as
      unit per_millivolt;
      unit per_millisec;
   enddef;
```

(continues on next page)

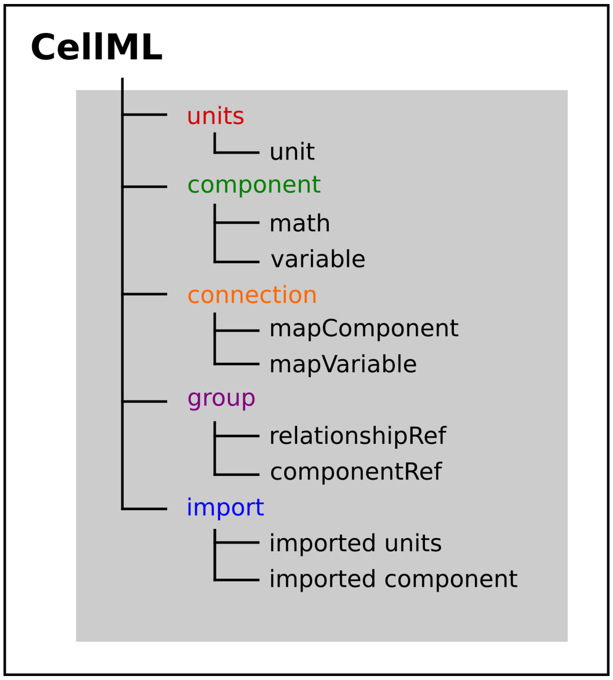


Fig. 1.63: Key entities in a CellML model.

[2] From here on we use a coloured background to identify code blocks that relate to a particular CellML construct: units, components, mappings and encapsulation groups and later imports.

(continued from previous page)

```
  def unit microA_per_cm2 as
     unit ampere {pref: micro};
     unit metre {pref: centi, expo: -2};
  enddef;
  def unit milliS_per_cm2 as
     unit siemens {pref: milli};
     unit metre {pref: centi, expo: -2};
  enddef;
  def unit mM as
     unit mole {pref: milli};
  enddef;
  def comp environment as
     var V: millivolt { pub: out};
     var t: millisec {pub: out};
     V = sel
     case (t␣
↪> 5 {millisec}) and (t < 15 {millisec}):
        -85.0 {millivolt};
     otherwise:
        0.0 {millivolt};
     endsel;
  enddef;
  def group as encapsulation for
     comp potassium_channel incl
        comp potassium_channel_n_gate;
     endcomp;
  enddef;
  def comp potassium_channel as
   ␣
↪ var V: millivolt {pub: in , priv: out};
 ␣
↪   var t: millisec {pub: in, priv: out};
     var n: dimensionless {priv: in};
     var i_K: microA_per_cm2 {pub: out};
     var g_K: milliS_per_cm2 {init: 36};
     var Ko: mM {init: 3};
     var Ki: mM {init: 90};
     var RTF: millivolt {init: 25};
     var E_K: millivolt;
     var K_
↪conductance: milliS_per_cm2 {pub: out};
     E_K=RTF*ln(Ko/Ki);
     K_conductance␣
↪= g_K*pow(n, 4{dimensionless});
     i_K = K_conductance*(V-E_K);
  enddef;
  def comp potassium_channel_n_gate as
     var V: millivolt {pub: in};
     var t: millisec {pub: in};
     var n:␣
↪dimensionless {init: 0.325, pub: out};
     var alpha_n: per_millisec;
     var beta_n: per_millisec;
     alpha_n = 0.01{per_
↪millivolt_millisec}*(V+10{millivolt})
        /(exp((V+10{millivolt}
↪)/10{millivolt})-1{dimensionless});
```

(continues on next page)

(continued from previous page)

```
67       beta_n = 0.
   ↪125{per_millisec}*exp(V/80{millivolt});
68       ode(n, t)␣
   ↪= alpha_n*(1{dimensionless}-n)-beta_n*n;
69    enddef;
70    def map between␣
   ↪environment and potassium_channel for
71       vars V and V;
72       vars t and t;
73    enddef;
74    def map between potassium_channel and
75      potassium_channel_n_gate for
76       vars V and V;
77       vars t and t;
78       vars n and n;
79    enddef;
80 enddef;
```

Lines 2-28: Define units.
Lines 29-38: Define component 'environment'.
Lines 32-37: Define **voltage step**.
Lines 39-43: Define encapsulation of 'n_gate'.
Lines 44-58: Define component 'potassium_channel'.
Lines 59-69: Define component 'potassium_channel_n_gate'.
Lines 70-79: Define mappings between components for variables that are shared between these components.

Note that several other features have been added:

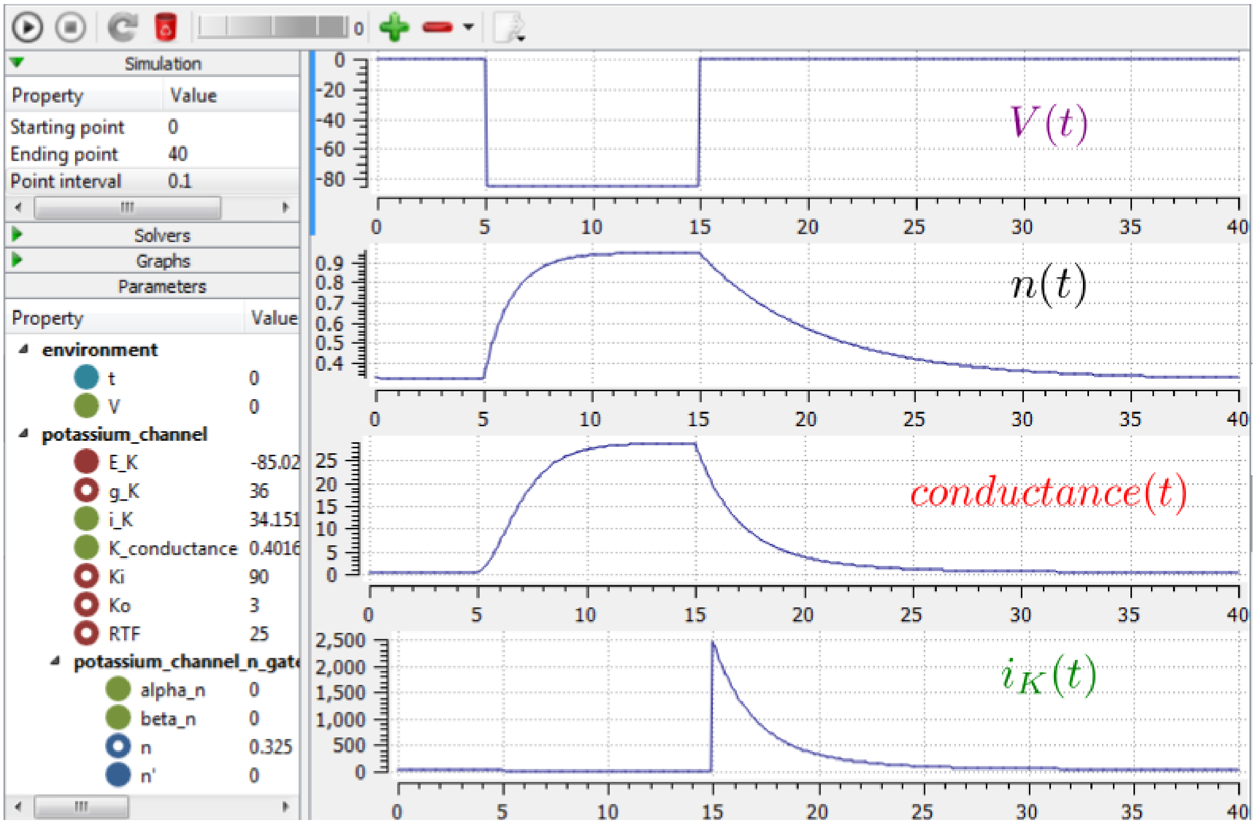
- the event control *select case* which indicates that the voltage is specified to jump from 0 mV to -85 mV at t = 5 ms then back to 0 mV at t = 15 ms. This is only used here in order to test the K channel model; when the potassium_channel component is later imported into a neuron model, **the environment component is not imported**.
- the use of encapsulation to embed the **potassium_channel_n_gate** inside the **potassium_channel**. This avoids the need to establish mappings from **environment** to **potassium_channel_n_gate** since the gate component is entirely within the channel component.
- the use of $\{pub : in\}$ and $\{pub : out\}$ to indicate which variables are either supplied as inputs to a component or produced as outputs from a component[3]. Any variables not labelled as *in* or *out* are local variables or parameters defined and used only within that component. Public (and private) interfaces are discussed in more detail in the next section.

We now use OpenCOR, with *Ending point* 40 and *Point interval* 0.1, to solve the equations for the potassium channel under a voltage step condition in which the membrane voltage is clamped initially at 0mV and then stepped down to -85mV for 10ms before being returned to 0mV. At 0mV, the steady state value of the n gate is $n_\infty = \frac{\alpha_n}{\alpha_n+\beta_n} = 0.324$ and, at -85mV, $n_\infty = 0.945$.

The voltage traces are shown at the top of Figure 21. The n-gate response, shown next, is to open further from its partially open value of n =0.324 at 0mV and then plateau at an almost fully open state of n =0.945 at the Nernst

[3] Note that a later version of CellML will remove the terms in and out since it is now thought that the direction of information flow should not be constrained.

potential -85mV before closing again as the voltage is stepped back to 0mV. Note that the gate opening behaviour (set by the voltage dependence of the α_n opening rate constant) is faster than the closing behaviour (set by the voltage dependence of the β_n closing rate constant). The channel conductance ($= n^4\bar{g}_K$) is shown next – note the initial s-shaped conductance increase caused by the n^4 (four gates in series) effect on conductance. Finally the channel current i_K = conductance x $(V - E_K)$ is shown at the bottom. Because the voltage is clamped at the Nernst potential (-85mV) during the period when the gate is opening, there is no current flow, but when the voltage is stepped back to 0mV, the open gates begin to close and the conductance declines but now there is a voltage gradient to drive an outward (positive) current flow through the partially open channel – albeit brief since the channel is closing.



Property	Value
Starting point	0
Ending point	40
Point interval	0.1

Property	Value
t	0
V	0
E_K	-85.02
g_K	36
i_K	34.151
K_conductance	0.4016
Ki	90
Ko	3
RTF	25
alpha_n	0
beta_n	0
n	0.325
n'	0

Fig. 1.64: Kinetics of the potassium channel gates for a voltage step from 0mV to -85mV (OpenCOR link). The voltage clamp step is shown at the top, then the n gate first order response, then the channel conductance, then the channel current. Notice how the conductance is slightly slower to turn on (due to the four gates in series) but fast to inactivate. Current only flows when there is a non-zero conductance and a non-zero voltage gradient. This is called the 'tail current'.

Note that the *CellML Text* code above includes the Nernst equation with its dependence on the concentrations $[K^+]_i$= 90mM and $[K^+]_o$= 3mM. Try raising the external potassium concentration to $[K^+]_o$= 10mM – you will then see the Nernst potential increase from -85mV to -55mV and a negative (inward) current flowing during the period when the membrane voltage is clamped to -85mV. The cell is now in a 'hyperpolarised' state because the potential is less than the equilibrium potential.

Note that you can change a model parameter such as $[K^+]_o$ either by changing the value in the left hand *Parameters* window (which leaves the file unchanged) or by editing the *CellML Text* code (which does change the file when you save from *CellML Text* view – which you have to do to see the effect of that change.

This potassium channel model will be used later, along with a sodium channel model and a leakage channel model, to form the Hodgkin-Huxley neuron model, where the membrane ion channel current flows are coupled to the equations governing current flow along the axon to generate an action potential.

A model of the sodium channel: Introducing CellML encapsulation and interfaces

The HH sodium channel has two types of gate, an m gate (of which there are 3) that is initially closed ($m = 0$) before activating and inactivating back to the closed state, and an h gate that is initially open ($h = 1$) before activating and inactivating back to the open state. The short period when both types of gate are open allows a brief window current to pass through the channel. Therefore,

$$i_{\text{Na}} = \bar{i}_{\text{Na}} m^3 h = m^3 \text{h}.\bar{g}_{\text{Na}} \left(V - E_{\text{Na}}\right)$$

where $\bar{g}_{\text{Na}} = 120$ mS.cm^{-2}, and with $\left[\text{Na}^+\right]_i$= 30mM and $\left[\text{Na}^+\right]_o$= 140mM, the Nernst potential for the sodium channel (z=1) is

$$E_{\text{Na}} = \frac{\text{RT}}{\text{zF}} ln \frac{\left[\text{Na}^+\right]_o}{\left[\text{Na}^+\right]_i} = 25 \; ln \frac{140}{30} = 35\text{mV}.$$

The gating kinetics are described by

$$\frac{\text{dm}}{\text{dt}} = \alpha_m \left(1 - m\right) - \beta_m.\text{m}; \frac{\text{dh}}{\text{dt}} = \alpha_h \left(1 - h\right) - \beta_h.\text{h}$$

where the voltage dependence of these four rate constants is determined experimentally to be[1]

$$\alpha_m = \frac{-0.1\left(V + 50\right)}{e^{\frac{-(V+50)}{10}} - 1}; \beta_m = 4e^{\frac{-(V+75)}{18}}; \alpha_h = 0.07e^{\frac{-(V+75)}{20}}; \beta_h = \frac{1}{e^{\frac{-(V+45)}{10}} + 1}.$$

Before we construct a CellML model of the sodium channel, we first introduce some further CellML concepts that help deal with the complexity of biological models: first the use of *encapsulation groups* and *public* and *private interfaces* to control the visibility of information in modular CellML components. To understand encapsulation, it is useful to use the terms ‘parent’, ‘child’ and ‘sibling’.

```
def group as encapsulation for
   comp sodium_channel incl
      comp sodium_channel_m_gate;
      comp sodium_channel_h_gate;
   endcomp;
enddef;
```

We define the CellML components **sodium_channel_m_gate** and **sodium_channel_h_gate** below. Each of these components has its own equations (voltage-dependent gates and first order gate kinetics) but they are both parts of one protein – the sodium channel – and it is useful to group them into one **sodium_channel** component as shown above:

We can then talk about the sodium channel as the parent of two children: the m gate and the h gate, which are therefore siblings. A *private interface* allows a parent to talk to its children and a *public interface* allows siblings to talk among themselves and to their parents (see Fig. 1.65).

The OpenCOR *CellML Text* for the HH sodium ion channel is given below.

Sodium_ion_channel.cellml

```
def model sodium_ion_channel as
   def unit millisec as
      unit second {pref: milli};
   enddef;
   def unit per_millisec as
      unit second {pref: milli, expo: -1};
```

(continues on next page)

[1] The HH paper used $\alpha_m = \frac{0.1(v+25)}{e^{\frac{(v+25)}{10}} - 1}; \beta_m = 4e^{\frac{v}{18}}; \alpha_h = 0.07e^{\frac{v}{20}}; \beta_h = \frac{1}{e^{\frac{(v+30)}{10}} + 1};$.

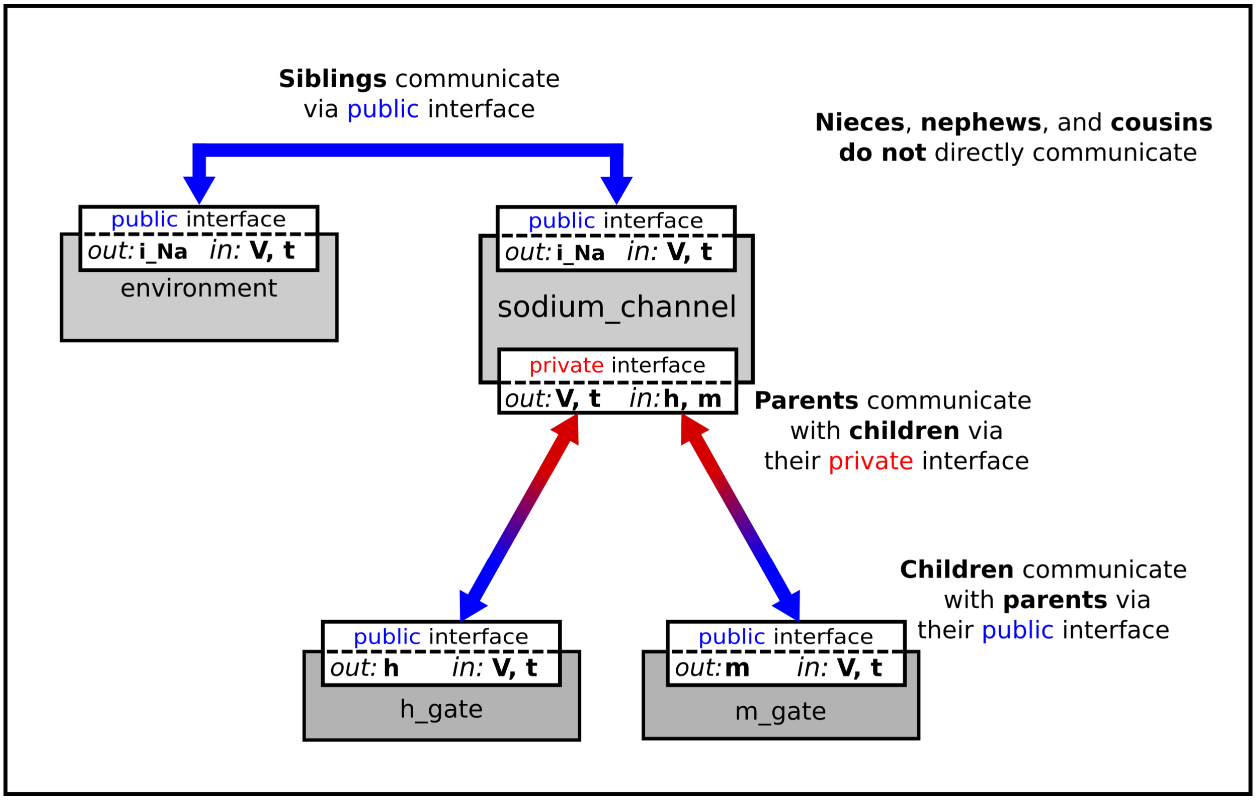


Fig. 1.65: Children talk to each other as siblings, and to their parents, via *public interfaces*. But the outside world can only talk to children through their parents via a *private interface*. Note that the siblings **m_gate** and **h_gate** could talk via a *public interface* but only if a mapping is established between them (not needed here).

(continued from previous page)

```
enddef;
def unit millivolt as
   unit volt {pref: milli};
enddef;
def unit per_millivolt as
   unit millivolt {expo: -1};
enddef;
def unit per_millivolt_millisec as
   unit per_millivolt;
   unit per_millisec;
enddef;
def unit microA_per_cm2 as
   unit ampere {pref: micro};
   unit metre {pref: centi, expo: -2};
enddef;
def unit milliS_per_cm2 as
   unit siemens {pref: milli};
   unit metre {pref: centi, expo: -2};
enddef;
def unit mM as
   unit mole {pref: milli};
enddef;
def comp environment as
   var V: millivolt {pub: out};
   var t: millisec {pub: out};
   V = sel
   case (t > 5 {millisec}) and (t < 15 {millisec}):
      -20.0 {millivolt};
   otherwise:
      -85.0 {millivolt};
   endsel;
enddef;
def group as encapsulation for
   comp sodium_channel incl
      comp sodium_channel_m_gate;
      comp sodium_channel_h_gate;
   endcomp;
enddef;
def comp sodium_channel as
   var V: millivolt {pub: in, priv: out};
   var t: millisec {pub: in, priv: out };
   var m: dimensionless {priv: in};
   var h: dimensionless {priv: in};
   var g_Na: milliS_per_cm2 {init: 120};
   var i_Na: microA_per_cm2 {pub: out};
   var Nao: mM {init: 140};
   var Nai: mM {init: 30};
   var RTF: millivolt {init: 25};
   var E_Na: millivolt;
   var Na_conductance: milliS_per_cm2 {pub: out};

   E_Na=RTF*ln(Nao/Nai);
   Na_conductance = g_Na*pow(m, 3{dimensionless})*h;
   i_Na= Na_conductance*(V-E_Na);
enddef;
def comp sodium_channel_m_gate as
   var V: millivolt {pub: in};
```

(continues on next page)

(continued from previous page)

```
      var t: millisec {pub: in};
      var alpha_m: per_millisec;
      var beta_m: per_millisec;
      var m: dimensionless {init: 0.05, pub: out};
      alpha_m = 0.1{per_millivolt_millisec}*(V+25{millivolt})
         /(exp((V+25{millivolt})/10{millivolt})-1{dimensionless});
      beta_m = 4{per_millisec}*exp(V/18{millivolt});
      ode(m, t) = alpha_m*(1{dimensionless}-m)-beta_m*m;
   enddef;
   def comp sodium_channel_h_gate as
      var V: millivolt {pub: in};
      var t: millisec {pub: in};
      var alpha_h: per_millisec;
      var beta_h: per_millisec;
      var h: dimensionless {init: 0.6, pub: out};
      alpha_h = 0.07{per_millisec}*exp(V/20{millivolt});
      beta_h = 1{per_millisec}/(exp((V+30{millivolt})/10{millivolt})+1{dimensionless}
↪);
      ode(h, t) = alpha_h*(1{dimensionless}-h)-beta_h*h;
   enddef;
   def map between environment and sodium_channel for
      vars V and V;
      vars t and t;
   enddef;
   def map between sodium_channel and sodium_channel_m_gate for
      vars V and V;
      vars t and t;
      vars m and m;
   enddef;
   def map between sodium_channel and sodium_channel_h_gate for
      vars V and V;
      vars t and t;
      vars h and h;
   enddef;
enddef;
```

The results of the OpenCOR computation, with *Ending point* 40 and *Point interval* 0.1, are shown in Fig. 1.66 with plots $V(t)$, $m(t)$, $h(t)$, $g_{\text{Na}}(t)$ and $i_{\text{Na}}(t)$ for voltage steps from (a) -85mV to -20mV, (b) -85mV to 0mV and (c) -85mV to 20mV. There are several things to note:

i. The kinetics of the m-gate are much faster than the h-gate.

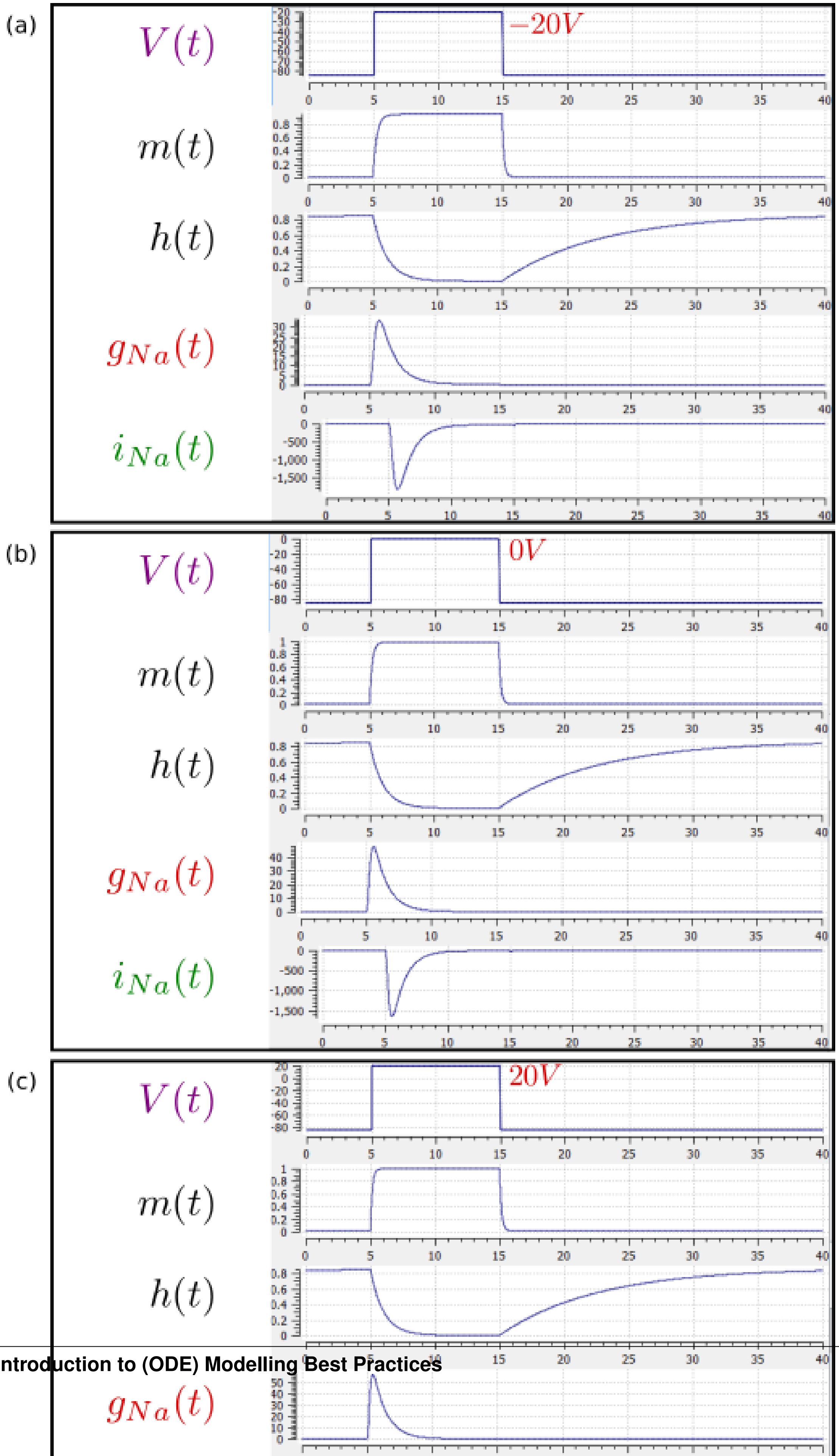
ii. The opening behaviour is faster as the voltage is stepped to higher values since $\tau = \frac{1}{\alpha_n+\beta_n}$ reduces with increasing V (see Fig. 1.61).

iii. The sodium channel conductance rises (*activates*) and then falls (*inactivates*) under a positive voltage step from rest since the three m-gates turn on but the h-gate turns off and the conductance is a product of these. Compare this with the potassium channel conductance shown in Fig. 1.64 which is only reduced back to zero by stepping the voltage back to its resting value – i.e. *deactivating* it.

iv. The only time current i_{Na} flows through the sodium channel is during the brief period when the m-gate is rapidly opening and the much slower h-gate is beginning to close. A small current flows during the reverse voltage step but this is at a time when the h-gate is now firmly off so the magnitude is very small.

v. The large sodium current i_{Na} is an inward current and hence negative.

Note that the bottom trace does not quite line up at t=0 because the values shown on the axes are computed automatically and hence can take more or less space depending on their magnitude.


(a)
V(t)
−20V
m(t)
h(t)
gNa(t)
iNa(t)
0
-500
-1,000
-1,500
0 5 10 15 20 25 30 35 40
(b)
V(t)
0V
m(t)
h(t)
gNa(t)
iNa(t)
0 5 10 15 20 25 30 35 40
(c)
V(t)
20V
m(t)
h(t)
gNa(t)
0 5 10 15 20 25 30 35 40

The **second concept** is that of modularity. By defining your mathematical model in a modular fashion with clearly defined interfaces you ensure that other scientists (and even yourself!) are able to make sure of the modules in future novel work.

- The section *A model of the potassium channel: Introducing CellML components and connections* provides an introduction to the syntax for defining modules in CellML and the mechanism for defining the communication between the modules.
- *A model of the sodium channel: Introducing CellML encapsulation and interfaces* then introduces the concept of hierarchically grouping modules as an aide to reuse and abstraction, as well as definition of module interfaces.

Reuse

A model of the nerve action potential: Introducing CellML imports

Here we describe the first (and most famous) model of nerve fibre electrophysiology based on the membrane ion channels that we have discussed in the last two sections. This is the work by Alan Hodgkin and Andrew Huxley in 1952 [AAF52] that won them (together with John Eccles) the 1963 Noble prize in Physiology or Medicine for *"their discoveries concerning the ionic mechanisms involved in excitation and inhibition in the peripheral and central portions of the nerve cell membrane"*.

Cable equation

The *cable equation* was developed in 1890[1] to predict the degradation of an electrical signal passing along the transatlantic cable. It is derived as follows:

If the voltage is raised at the left hand end of the cable (shown by the deep red in Fig. 1.67), a current i_a (A) will flow that depends on the voltage gradient, given by $\frac{\partial V}{\partial x}$ $(V.m^{-1})$ and the resistance r_a $(\Omega.m^{-1})$, Ohm's law gives $-\frac{\partial V}{\partial x} = r_a i_a$. But if the cable leaks current i_m $(A.m^{-1})$ per unit length of cable, conservation of current gives $\frac{\partial i_a}{\partial x} = i_m$ and therefore, substituting for i_a , $\frac{\partial}{\partial x}\left(-\frac{1}{r_a}\frac{\partial V}{\partial x}\right) = i_m$. There are two sources of membrane current i_m , one associated with the capacitance C_m ($\approx 1\mu F/\text{cm}^2$) of the membrane, $C_m \frac{\partial V}{\partial t}$, and one associated with holes or channels in the membrane, i_{leak}. Inserting these into the RHS gives

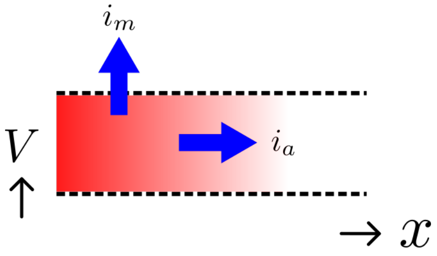


Fig. 1.67: Current flow in a leaky cable.

$$\frac{\partial}{\partial x}\left(-\frac{1}{r_a}\frac{\partial V}{\partial x}\right) = i_m = C_m\frac{\partial V}{\partial t} + i_{\text{leak}}$$

Rearranging gives the *cable equation* (for constant r_a):

$$C_m\frac{\partial V}{\partial t} = -\frac{1}{r_a}\frac{\partial^2 V}{\partial x^2} - i_{\text{leak}}$$

where all terms represent *current density* (current per membrane area) and have units of $\mu A/\text{cm}^2$.

Action potentials

[1] http://en.wikipedia.org/wiki/Cable_theory

The cable equation can be used to model the propagation of an action potential along a neuron or any other excitable cell. The ‘leak’ current is associated primarily with the inward movement of sodium ions through the membrane ‘sodium channel’, giving the **inward** membrane current i_{Na}, and the outward movement of potassium ions through a membrane ‘potassium channel’, giving the **outward** current i_K (see Fig. 1.68). A further small leak current $i_L = g_L\left(V - E_L\right)$ associated with chloride and other ions is also included.

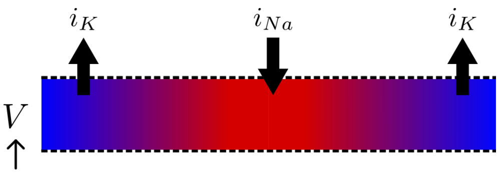


Fig. 1.68: Current flow in a neuron.

When the membrane potential V rises due to axial current flow, the Na channels open and the K channels close, such that the membrane potential moves towards the Nernst potential for sodium. The subsequent decline of the Na channel conductance and the increasing K channel conductance as the voltage drops rapidly repolarises the membrane to its resting potential of -85mV (see Fig. 1.69).

We can neglect[2] the term $(-\frac{1}{r_a}\frac{\partial^2 V}{\partial x^2})$ (the rate of change of axial current along the cable) for the present models since we assume the whole cell is clamped with an axially uniform potential. We can therefore obtain the membrane potential V by integrating the first order ODE

$$\frac{\mathrm{dV}}{\mathrm{dt}} = -\left(i_{\text{Na}} + i_K + i_L\right)/C_m.$$

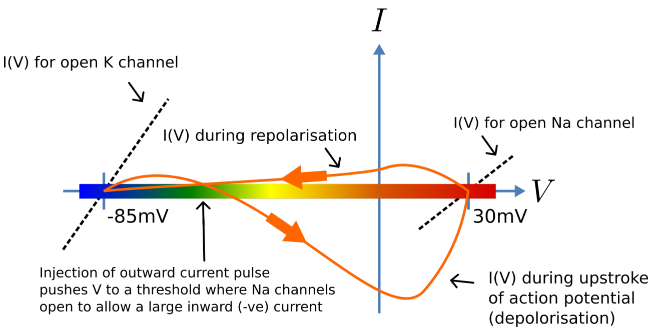

I
I(V) for open K channel
I(V) during repolarisation
I(V) for open Na channel
V
-85mV
30mV
Injection of outward current pulse pushes V to a threshold where Na channels open to allow a large inward (-ve) current
I(V) during upstroke of action potential (depolorisation)

Fig. 1.69: Current-voltage trajectory during an action potential.

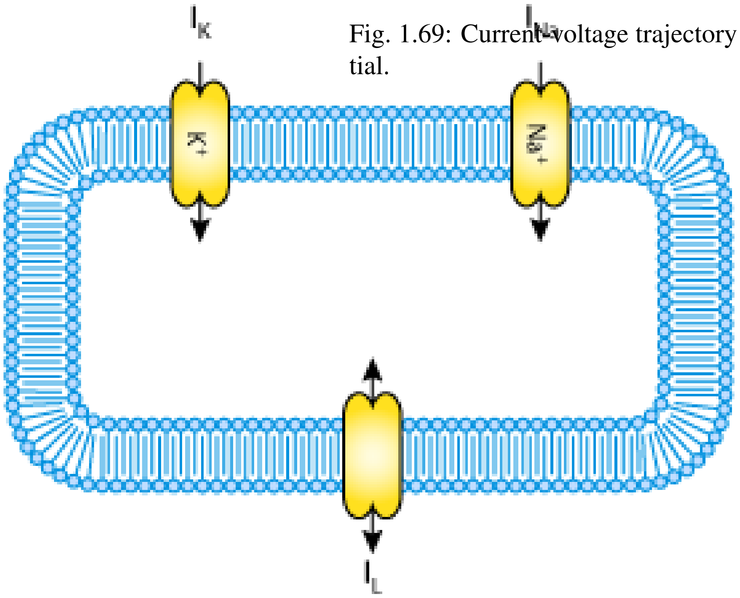


Fig. 1.70: A schematic cell diagram describing the current flows across the cell bilipid membrane that are captured in the Hodgkin-Huxley model. The membrane ion channels are a sodium (Na$^+$) channel, a potassium (K$^+$) channel, and a leakage (L) channel (for chloride and other ions) through which the currents $\mathrm{I_{Na}}$, $\mathrm{I_K}$ and $\mathrm{I_L}$ flow, respectively.

We use this example to demonstrate the importing feature of CellML. CellML *imports* are used to bring a previously defined CellML model of a component into

[2] This term is needed when determining the propagation of the action potential, including its wave speed.

the new model (in this case the Na and K channel components defined in the previous two sections, together with a leakage ion channel model specified below). Note that importing a component brings the children components with it along with their connections and units, but it does not bring the siblings of that component with it.

To establish a CellML model of the HH equations we first lay out the model components with their public and private interfaces (Fig. 1.71).

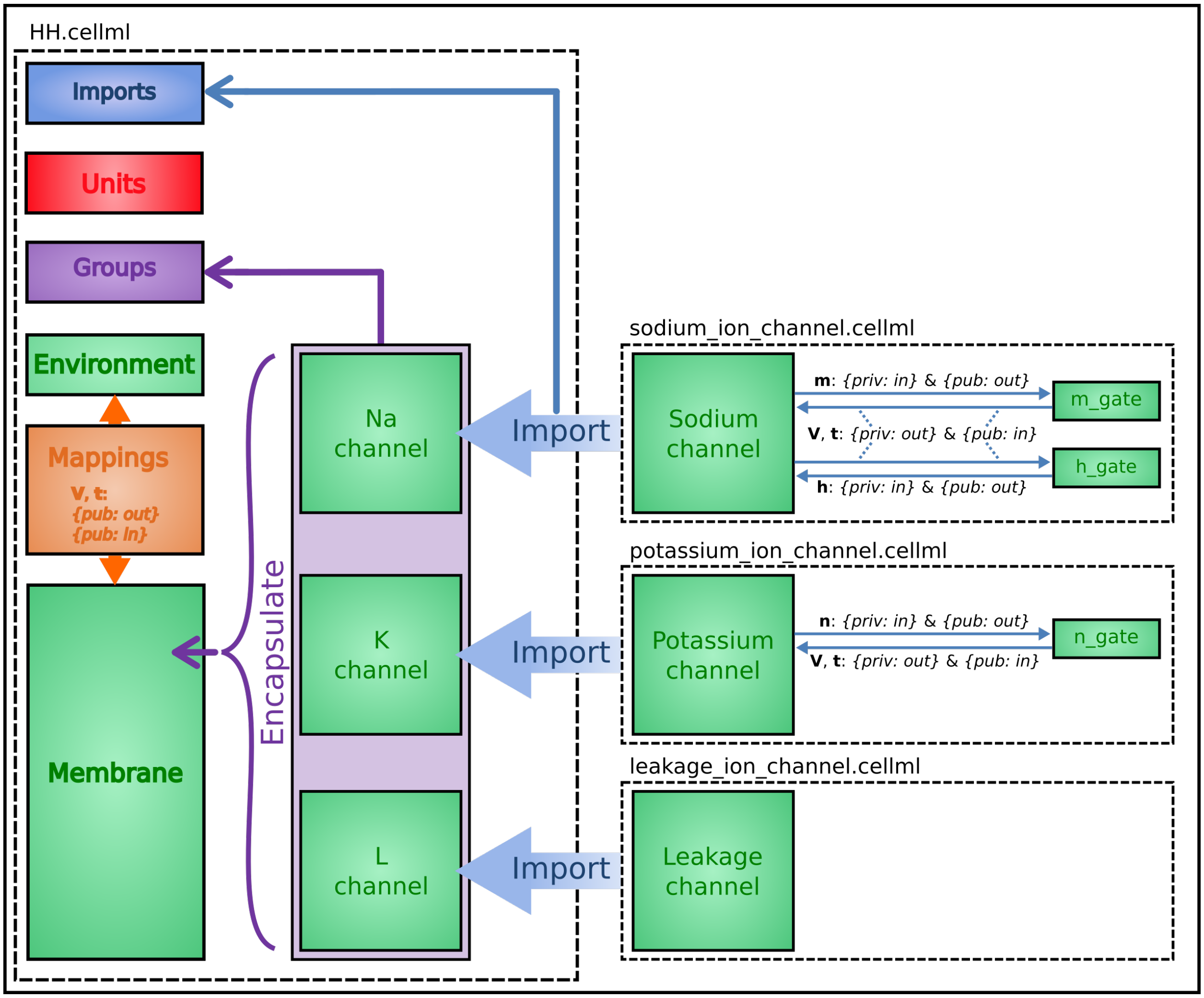

HH.cellml
Imports
Units
Groups
Environment
Mappings
V, t:
{pub: out}
{pub: in}
Membrane
Encapsulate
Na
channel
K
channel
L
channel
Import
Import
Import
sodium_ion_channel.cellml
Sodium
channel
m: {priv: in} & {pub: out}
m_gate
V, t: {priv: out} & {pub: in}
h_gate
h: {priv: in} & {pub: out}
potassium_ion_channel.cellml
Potassium
channel
n: {priv: in} & {pub: out}
n_gate
V, t: {priv: out} & {pub: in}
leakage_ion_channel.cellml
Leakage
channel

Fig. 1.71: Overall structure of the HH CellML model showing the encapsulation hierarchy (purple), the CellML model imports (blue) and the other key parts (units, components, and mappings) of the top level CellML model.

The HH model is the top level model. The *CellML Text* code for the HH model, together with the leakage_channel model, is given below. The imported potassium_ion_channel model and sodium_ion_channel model are unchanged from the previous sections

HH.cellml

```
def model HH as
   def import using "sodium_ion_channel.cellml" for
      comp Na_channel using comp sodium_channel;
   enddef;
   def import using "potassium_ion_channel.cellml" for
```

(continues on next page)

(continued from previous page)

```
      comp K_channel using comp potassium_channel;
   enddef;
   def import using "leakage_ion_channel.cellml" for
      comp L_channel using comp leakage_channel;
   enddef;
   def unit millisec as
      unit second {pref: milli};
   enddef;
   def unit millivolt as
      unit volt {pref: milli};
   enddef;
   def unit microA_per_cm2 as
      unit ampere {pref: micro};
      unit metre {pref: centi, expo: -2};
   enddef;
   def unit microF_per_cm2 as
      unit farad {pref: micro};
      unit metre {pref: centi, expo: -2};
   enddef;
   def group as encapsulation for
      comp membrane incl
         comp Na_channel;
         comp K_channel;
         comp L_channel;
      endcomp;
   enddef;
   def comp environment as
      var V: millivolt {init: -85, pub: out};
      var t: millisec {pub: out};
   enddef;
   def map between environment and membrane for
      vars V and V;
      vars t and t;
   enddef;
   def map between membrane and Na_channel for
      vars V and V;
      vars t and t;
      vars i_Na and i_Na;
   enddef;
   def map between membrane and K_channel for
      vars V and V;
      vars t and t;
      vars i_K and i_K;
   enddef;
   def map between membrane and L_channel for
      vars V and V;
      vars i_L and i_L;
   enddef;
   def comp membrane as
      var V: millivolt {pub: in, priv: out};
      var t: millisec {pub: in, priv: out};
      var i_Na: microA_per_cm2 {pub: out, priv: in};
      var i_K: microA_per_cm2 {pub: out, priv: in};
      var i_L: microA_per_cm2 {pub: out, priv: in};
      var Cm: microF_per_cm2 {init: 1};
      var i_Stim: microA_per_cm2;
      var i_Tot: microA_per_cm2;
```

(continues on next page)

(continued from previous page)

```
      i_Stim = sel
      case (t >= 1{millisec}) and (t <= 1.2{millisec}):
         100{microA_per_cm2};
      otherwise:
         0{microA_per_cm2};
      endsel;
      i_Tot = i_Stim + i_Na + i_K + i_L;
      ode(V,t) = -i_Tot/Cm;
   enddef;
enddef;
```

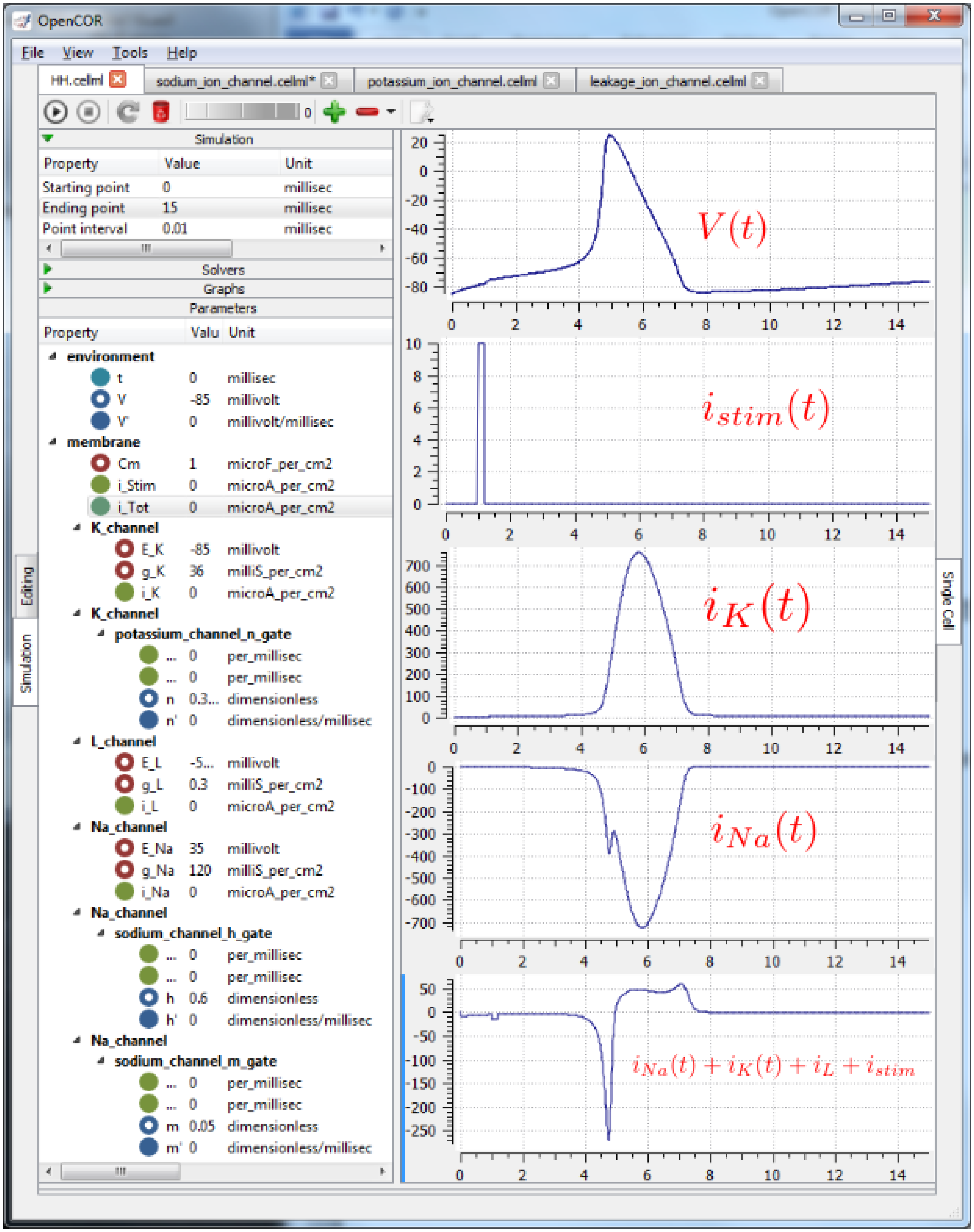
`Leakage_ion_channel`

```
def model leakage_ion_channel as
   def unit millisec as
      unit second {pref: milli};
   enddef;
   def unit millivolt as
      unit volt {pref: milli};
   enddef;
   def unit per_millivolt as
      unit millivolt {expo: -1};
   enddef;
   def unit microA_per_cm2 as
      unit ampere {pref: micro};
      unit metre {pref: centi, expo: -2};
   enddef;
   def unit milliS_per_cm2 as
      unit siemens {pref: milli};
      unit metre {pref: centi, expo: -2};
   enddef;
   def comp environment as
      var V: millivolt {init: 0, pub: out};
      var t: millisec {pub: out};
   enddef;
   def map between leakage_channel and environment for
      vars V and V;
   enddef;
   def comp leakage_channel as
      var V: millivolt {pub: in};
      var i_L: microA_per_cm2 {pub: out};
      var g_L: milliS_per_cm2 {init: 0.3};
      var E_L: millivolt {init: -54.4};
      i_L = g_L*(V-E_L);
   enddef;
enddef;
```

Note that the *CellML Text* code for the potassium channel is *Potassium_ion_channel.cellml* and for the sodium channel is *Sodium_ion_channel.cellml*.

Note that the only units that need to be defined for this top level HH model are the ones explicitly required for the membrane component. All the other units, required for the various imported sub-models, are imported along with the imported components.

The results generated by the HH model are shown in Fig. 1.72.



Simulation

Property	Value	Unit
Starting point	0	millisec
Ending point	15	millisec
Point interval	0.01	millisec

Parameters

Property	Valu	Unit
environment		
● t	0	millisec
○ V	-85	millivolt
● V'	0	millivolt/millisec
membrane		
○ Cm	1	microF_per_cm2
● i_Stim	0	microA_per_cm2
● i_Tot	0	microA_per_cm2
K_channel		
○ E_K	-85	millivolt
○ g_K	36	milliS_per_cm2
● i_K	0	microA_per_cm2
K_channel		
potassium_channel_n_gate		
● ...	0	per_millisec
● ...	0	per_millisec
○ n	0.3...	dimensionless
● n'	0	dimensionless/millisec
L_channel		
○ E_L	-5...	millivolt
○ g_L	0.3	milliS_per_cm2
● i_L	0	microA_per_cm2
Na_channel		
○ E_Na	35	millivolt
○ g_Na	120	milliS_per_cm2
● i_Na	0	microA_per_cm2
Na_channel		
sodium_channel_h_gate		
● ...	0	per_millisec
● ...	0	per_millisec
○ h	0.6	dimensionless
● h'	0	dimensionless/millisec
Na_channel		
sodium_channel_m_gate		
● ...	0	per_millisec
● ...	0	per_millisec
○ m	0.05	dimensionless
● m'	0	dimensionless/millisec

Fig. 1.72: Results from OpenCOR for the Hodgkin Huxley (HH) CellML model. The top panel shows the generated action potential. Note that the stimulus current is not really needed as the background outward leakage current is enough to drive the membrane potential up to the threshold for sodium channel opening.

Important note

It is often convenient to have the sub-models – in this case the sodium_ion_channel.cellml model, the potassium_ion_channel.cellml model and the leakage_ion_channel.cellml model - loaded into OpenCOR at the same time as the high level model (HH.cellml), as shown in Fig. 1.73 . If you make changes to a model in the *CellML Text* view, you must save the file (*CTRL-S*) before running a new simulation since the simulator works with the saved model. Furthermore, a change to a sub-model will only affect the high level model which imports it if you also save the high level model (or use the *Reload* option under the File menu). An asterisk appears next to the name of a file when a change has been made and the file has not been saved. The asterisk disappears when the file is saved.

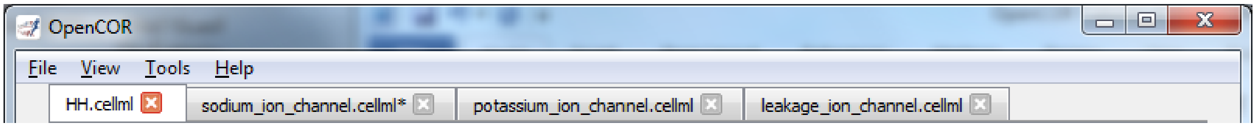

OpenCOR
File View Tools Help
HH.cellml
sodium_ion_channel.cellml*
potassium_ion_channel.cellml
leakage_ion_channel.cellml

Fig. 1.73: The HH.cellml model and its three sub-models are available under separate tabs in OpenCOR.

A model of the cardiac action potential: Importing units and parameters

We now examine the Noble 1962 model [D62] that applied the Hodgkin-Huxley approach to cardiac cells and thereby initiated the development of a long line of cardiac cell models that, in their human cell formulation, are now used clinically and are the most sophisticated models of any cell type. It was the incorporation of these models into whole heart bioengineering models that initiated the Physiome Project. We also illustrate the use of imported units and imported parameter sets.

Cardiac cells have similar gradients of potassium and sodium ions that operate in a similar way to neurons (as do all electrically active cells). There is one major difference, however, in the potassium channel that holds the cells in their resting state at -85mV (HH neuron) or -100mV (cardiac Purkinje cells). This difference is illustrated in Fig. 1.74(a). When the membrane potential is raised above the equilibrium potential for potassium, the cardiac channel conductance shown by the dashed line drops to nearly zero – i.e. it is an *inward rectifier* since it rectifies ('cuts off') the outward current that otherwise would have flowed through the channel at that potential. This is an evolutionary adaptation of the potassium channel to avoid loss of potassium ions out of the cell during the long plateau phase of the cardiac action potential (Fig. 1.74(b)) needed to give the heart time to contract. This evolutionary change saves the additional energy that would otherwise be needed to pump potassium ions back into the cell, but this Faustian "pact with the devil" is also the reason the heart is so susceptible to conduction failure (more on this later). To explain his data on Purkinje cells Noble [D62] postulated the existence of two inward rectifier potassium channels, one with a conductance g_{K1} that showed voltage dependence but no significant time dependence and another with conductance g_{K2} that showed less severe rectification with time dependent gating similar to the HH four-gated potassium channel.

To model the cardiac action potential in Purkinje fibres (a cardiac cell specialised for rapid conduction from the atrio-ventricular node to the apical ventricular myocardial tissue), Noble [D62] proposed two potassium channels (one of these being the inwardly rectifying potassium channel described above and the other called the delayed potassium channel), one sodium channel (very similar to the HH neuronal sodium channel) and one leakage channel (also similar to the HH one).

The equations for these are as follows: (as for the HH model, time is in ms, voltages are in mV, concentrations are in mM, conductances are in mS, currents are in µA and capacitance is in µF).

Inward rectifying $\mathbf{i_{K1}}$ potassium channel (voltage dependent only)

$$i_{K1} = g_{K1}\left(V - E_K\right), \text{ with } E_K = \frac{\text{RT}}{\text{zF}} ln\frac{\left[K^+\right]_o}{\left[K^+\right]_i} = 25ln\frac{2.5}{140} = -100\text{mV}.$$

$$g_{K1} = 1.2e^{\frac{-(V+90)}{50}} + 0.015e^{\frac{(V+90)}{60}}$$

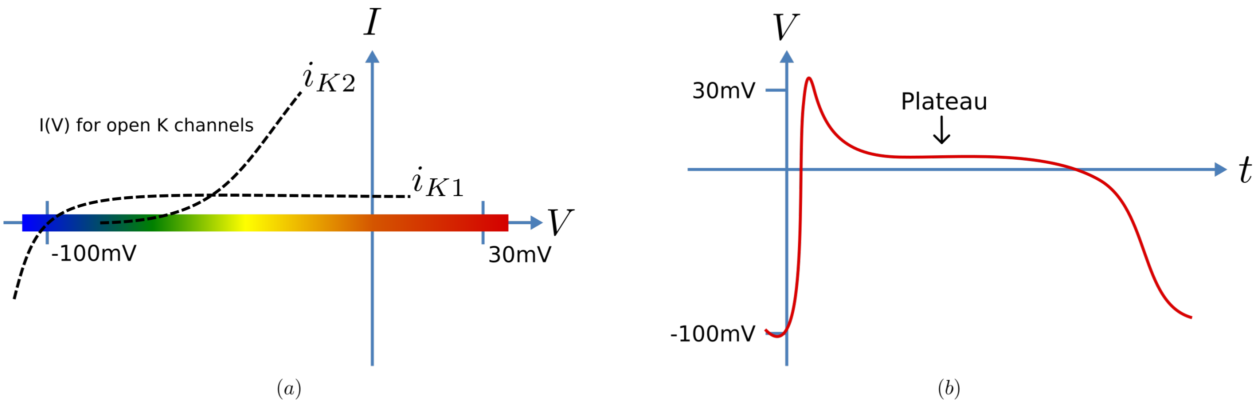


Fig. 1.74: Current-voltage relations (a) around the equilibrium potentials for the potassium and sodium channels in cardiac cells. The sodium channel is similar to the one in neurons but the two potassium channels have an inward rectifying property that stops leakage of potassium ions out of the cell when the membrane potential (illustrated in (b)) is high during the plateau phase of the cardiac action potential.

Inward rectifying $\mathbf{i_{K2}}$ potassium channel (voltage and time dependent)[1]

$$\begin{aligned} i_{K2} &= g_{K2}\left(V - E_K\right) \\ g_{K2} &= 1.2n^4 \\ \frac{\mathrm{dn}}{\mathrm{dt}} &= \alpha_n\left(1-n\right) - \beta_n.\mathrm{n},\ \text{where } \alpha_n = \frac{-0.0001\left(V+50\right)}{e^{\frac{-(V+50)}{10}}-1} \text{ and } \beta_n = 0.002e^{\frac{-(V+90)}{80}}. \end{aligned}$$

Note that the rate constants here reflect a much slower onset of the time dependent change in conductance than in the HH potassium channel.

Sodium channel

$$\begin{aligned} i_{\mathrm{Na}} &= \left(g_{\mathrm{Na}} + 140\right)\left(V - E_{\mathrm{Na}}\right),\ \text{with } E_{\mathrm{Na}} = \frac{\mathrm{RT}}{\mathrm{zF}} ln \frac{\left[\mathrm{Na}^+\right]_o}{\left[\mathrm{Na}^+\right]_i} = 25ln\frac{140}{30} = 35\mathrm{mV}. \\ g_{\mathrm{Na}} &= m^3\mathrm{h}.g_{Na\_max} \text{ where } g_{Na\_max} = 400\mathrm{mS}. \\ \frac{\mathrm{dm}}{\mathrm{dt}} &= \alpha_m\left(1-m\right) - \beta_m.\mathrm{m},\ \text{where } \alpha_m = \frac{-0.1\left(V+48\right)}{e^{\frac{-(V+48)}{15}}-1} \text{ and } \beta_m = \frac{0.12\left(V+8\right)}{e^{\frac{(V+8)}{5}}-1} \\ \frac{\mathrm{dh}}{\mathrm{dt}} &= \alpha_h\left(1-h\right) - \beta_h.\mathrm{h},\ \text{where } \alpha_h = 0.17e^{\frac{-(V+90)}{20}} \text{ and } \beta_h = \frac{1}{1+e^{\frac{-(V+42)}{10}}} \end{aligned}$$

Leakage channel

$$i_{\mathrm{leak}} = g_L\left(V - E_L\right),\ \text{with } E_L = -60mV \text{ and } g_L = 0.075\mathrm{mS}.$$

Membrane equation[2]

$$\frac{\mathrm{dV}}{\mathrm{dt}} = -\left(i_{\mathrm{Na}} + i_{K1} + i_{K2} + i_{\mathrm{leak}}\right)/C_m \text{ where } C_m = 12\mu\mathrm{F}.$$

Fig. 1.75 shows the structure of the model, including separate files for units, parameters, and the three ion channels (the two potassium channels are lumped together). We include the Nernst equations dependence on potassium and sodium ion concentrations in order to demonstrate the use of parameter values, defined in a separate parameters file, that are read in at the top (whole cell model) level and passed down to the individual ion channel models.

[1] The second inwardly rectifying channel model was later replaced with two currents and , so that modern cardiac cell models do not include but they do include the inward rectifier (see later section).

[2] The Purkinje fibre membrane capacitance is 12 times higher than that found for squid axon. The use of μF ensures unit consistency with ms, mV and A since F is equivalent to C.V^{-1} or s.A.V^{-1} and therefore A/ F or A/(ms. A. mV^{-1}) on the RHS matches mV/ms on the LHS).

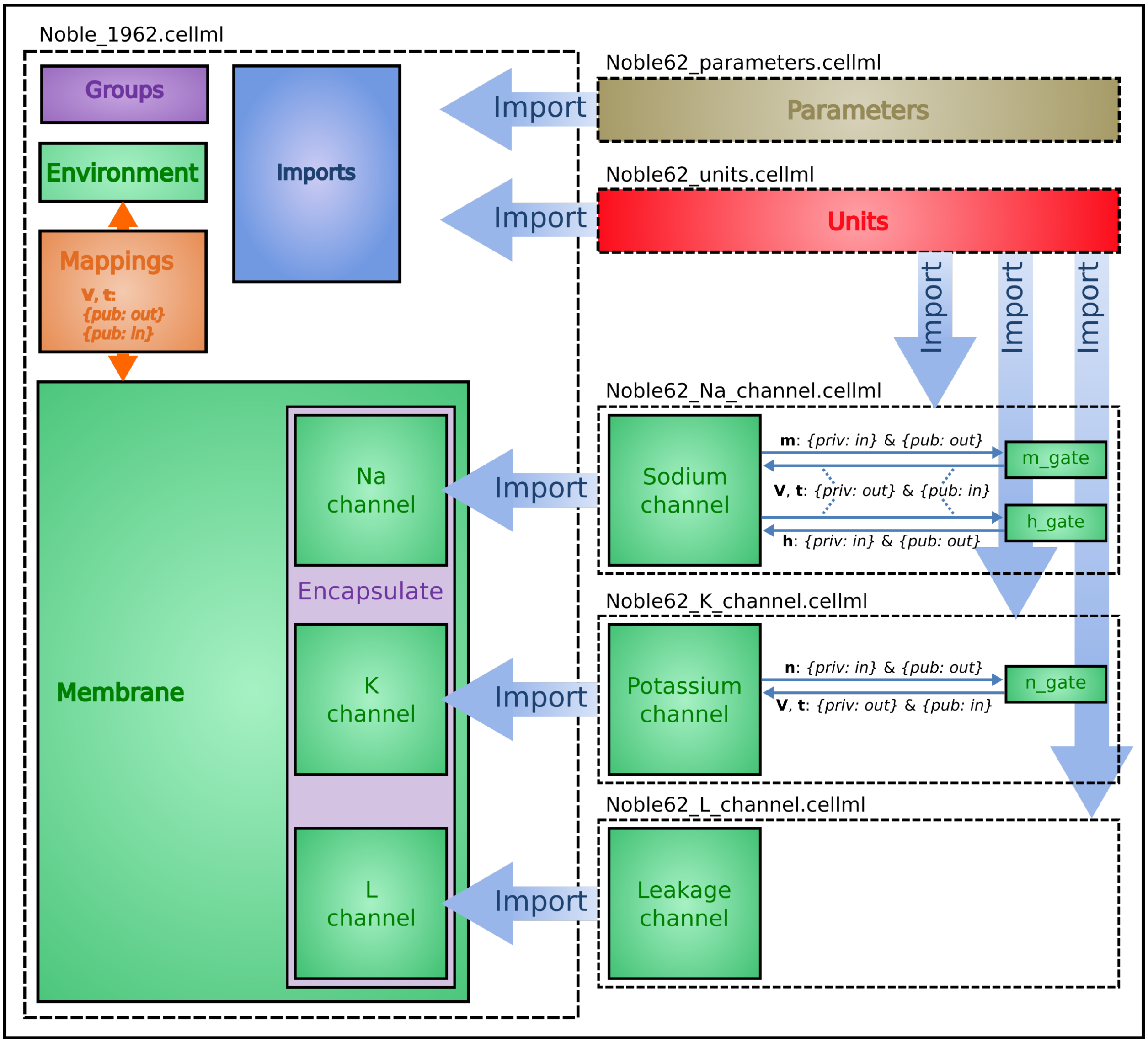


Fig. 1.75: Overall structure of the Noble62 CellML model showing the encapsulation hierarchy (purple), the CellML model imports (blue) and the other key parts (units, components & mappings) of the top level CellML model. Note that the overall structure of the Noble62 model differs from that of the earlier HH model in that all units are defined in a units file and imported where needed (shown by the import arrows). Also the ion concentration parameters are defined in a parameters file and imported into the top level file but passed down to the modules that use them via the mappings.

The CellML Text code for all six files is shown on the following two pages. The arrows indicate the imports (appropriately colour coded for units, components, and parameters).

Graphical outputs from solution of the Noble 1962 model with OpenCOR for 5000ms are shown in Fig. 1.75. Interpretation of the model outputs is given in the Fig. 1.75 legend. The Noble62 model was developed further by Noble and others to include additional sodium and potassium channels, calcium channels (needed for excitation-contraction coupling), chloride channels and various ion exchange mechanisms (Na/Ca, Na/H), co-transporters (Na/Cl, K/Cl) and energy (ATP)-dependent pumps (Na/K, Ca) needed to model the observed beat by beat changes in intracellular ion concentrations. These are discussed further in Section 15.

Note: The downloadable links below are links to the raw text file that may be used for copying and pasting into OpenCOR. For the underlying CellML file that is suitable for opening with OpenCOR from disk obtain the xml file.

Raw text: `Noble_1962.txt`, XML file: Noble_1962.cellml.

```
def model Noble_1962 as
   def import using "Noble62_Na_channel.xml" for
      comp Na_channel using comp sodium_channel;
   enddef;
   def import using "Noble62_K_channel.xml" for
      comp K_channel using comp potassium_channel;
   enddef;
   def import using "Noble62_L_channel.xml" for
      comp L_channel using comp leakage_channel;
   enddef;
   def import using "Noble62_units.xml" for
      unit mV using unit mV;
      unit ms using unit ms;
      unit nanoF using unit nanoF;
      unit nanoA using unit nanoA;
   enddef;
   def import using "Noble62_parameters.xml" for
      comp parameters using comp parameters;
   enddef;
   def map between parameters and membrane for
      vars Ki and Ki;
      vars Ko and Ko;
      vars Nai and Nai;
      vars Nao and Nao;
   enddef;
   def comp environment as
      var t: ms {init: 0, pub: out};
   enddef;
   def group as encapsulation for
      comp membrane incl
         comp Na_channel;
         comp K_channel;
         comp L_channel;
      endcomp;
   enddef;
   def comp membrane as
      var V: mV {init: -85, pub: out, priv: out};
      var t: ms {pub: in, priv: out};
      var Cm: nanoF {init: 12000};
      var Ki: mM {pub: in, priv: out};
      var Ko: mM {pub: in, priv: out};
```

(continues on next page)

(continued from previous page)

```
      var Nai: mM {pub: in, priv: out};
      var Nao: mM {pub: in, priv: out};
      var i_Na: nanoA {pub: out, priv: in};
      var i_K: nanoA {pub: out, priv: in};
      var i_L: nanoA {pub: out, priv: in};
      ode(V, t) = -(i_Na+i_K+i_L)/Cm;
   enddef;
   def map between environment and membrane for
      vars t and t;
   enddef;
   def map between membrane and Na_channel for
      vars V and V;
      vars t and t;
      vars Nai and Nai;
      vars Nao and Nao;
      vars i_Na and i_Na;
   enddef;
   def map between membrane and K_channel for
      vars V and V;
      vars t and t;
      vars Ki and Ki;
      vars Ko and Ko;
      vars i_K and i_K;
   enddef;
   def map between membrane and L_channel for
      vars V and V;
      vars i_L and i_L;
   enddef;
enddef;
```

Raw text: `Noble62_units.txt`, XML file Noble62_units.cellml.

```
def model Noble62_units as
   def unit ms as
      unit second {pref: milli};
   enddef;
   def unit per_ms as
      unit second {pref: milli, expo: -1};
   enddef;
   def unit mV as
      unit volt {pref: milli};
   enddef;
   def unit mM as
      unit mole {pref: milli};
   enddef;
   def unit per_mV as
      unit volt {pref: milli, expo: -1};
   enddef;
   def unit per_mV_ms as
      unit mV {expo: -1};
      unit ms {expo: -1};
   enddef;
   def unit microS as
      unit siemens {pref: micro};
   enddef;
   def unit nanoF as
      unit farad {pref: nano};
```

(continues on next page)

(continued from previous page)

```
   enddef;
   def unit nanoA as
      unit ampere {pref: nano};
   enddef;
enddef;
```

Raw text: `Noble62_parameters.txt`, XML file Noble62_parameters.cellml.

```
def model Noble62_parameters as
   def import using "Noble62_units.xml" for
      unit mM using unit mM;
   enddef;
   def comp parameters as
      var Ki: mM {init: 140, pub: out};
      var Ko: mM {init: 2.5, pub: out};
      var Nai: mM {init: 30, pub: out};
      var Nao: mM {init: 140, pub: out};
   enddef;
enddef;
```

Raw text: `Noble62_Na_channel.txt`, XML file Noble62_Na_channel.cellml.

```
def model sodium_ion_channel as
   def import using "Noble62_units.xml" for
      unit mV using unit mV;
      unit ms using unit ms;
      unit mM using unit mM;
      unit per_ms using unit per_ms;
      unit per_mV using unit per_mV;
      unit per_mV_ms using unit per_mV_ms;
      unit microS using unit microS;
      unit nanoA using unit nanoA;
   enddef;
   def group as encapsulation for
      comp sodium_channel incl
      comp sodium_channel_m_gate;
      comp sodium_channel_h_gate;
   endcomp;
   enddef;
   def comp sodium_channel as
      var V: mV {pub: in, priv: out};
      var t: ms {pub: in, priv: out};
      var g_Na_max: microS {init: 400000};
      var g_Na: microS;
      var E_Na: mV;
      var m: dimensionless {priv: in};
      var h: dimensionless {priv: in};
      var Nai: mM {pub: in};
      var Nao: mM {pub: in};
      var RTF: mV {init: 25};
      var i_Na: nanoA {pub: out};
      E_Na = RTF*ln(Nao/Nai);
      g_Na = pow(m, 3{dimensionless})*h*g_Na_max;
      i_Na = (g_Na+140{microS})*(V-E_Na);
   enddef;
   def comp sodium_channel_m_gate as
      var V: mV {pub: in};
```

(continues on next page)

(continued from previous page)

```
      var t: ms {pub: in};
      var m: dimensionless {init: 0.01, pub: out};
      var alpha_m: per_ms;
      var beta_m: per_ms;
      alpha_m = -0.10{per_mV_ms}*(V+48{mV})
        /(exp(-(V+48{mV})/15{mV})-1{dimensionless});
      beta_m = 0.12{per_mV_ms}*(V+8{mV})
        /(exp((V+8{mV})/5{mV})-1{dimensionless});
      ode(m, t)=alpha_m*(1{dimensionless}-m)-beta_m*m;
   enddef;
   def comp sodium_channel_h_gate as
      var V: mV {pub: in};
      var t: ms {pub: in};
      var h: dimensionless {init: 0.8, pub: out};
      var alpha_h: per_ms;
      var beta_h: per_ms;
      alpha_h = 0.17{per_ms}*exp(-(V+90{mV})/20{mV});
      beta_h = 1.00{per_ms}
        /(1{dimensionless}+exp(-(V+42{mV})/10{mV}));
      ode(h, t) = alpha_h*(1{dimensionless}-h)-beta_h*h;
   enddef;
   def map between sodium_channel
     and sodium_channel_m_gate for
      vars V and V;
      vars t and t;
      vars m and m;
   enddef;
   def map between sodium_channel
     and sodium_channel_h_gate for
      vars V and V;
      vars t and t;
      vars h and h;
   enddef;
enddef;
```

Raw text: `Noble62_K_channel.txt`, XML file Noble62_K_channel.cellml.

```
def model potassium_ion_channel as
   def import using "Noble62_units.xml" for
      unit mV using unit mV;
      unit ms using unit ms;
      unit mM using unit mM;
      unit per_ms using unit per_ms;
      unit per_mV using unit per_mV;
      unit per_mV_ms using unit per_mV_ms;
      unit microS using unit microS;
      unit nanoA using unit nanoA;
   enddef;
   def group as encapsulation for
      comp potassium_channel incl
         comp potassium_channel_n_gate;
      endcomp;
   enddef;
   def comp potassium_channel as
      var V: mV {pub: in, priv: out};
      var t: ms {pub: in, priv: out};
      var n: dimensionless {priv: in};
```

(continues on next page)

(continued from previous page)

```
      var Ki: mM {pub: in};
      var Ko: mM {pub: in};
      var RTF: mV {init: 25};
      var E_K: mV;
      var g_K1: microS;
      var g_K2: microS;
      var i_K: nanoA {pub: out};
      E_K = RTF*ln(Ko/Ki);
      g_K1 = 1200{microS}*exp(-(V+90{mV})/50{mV})
        +15{microS}*exp((V+90{mV})/60{mV});
      g_K2 = 1200{microS}*pow(n, 4{dimensionless});
      i_K = (g_K1+g_K2)*(V-E_K);
   enddef;
   def comp potassium_channel_n_gate as
      var V: mV {pub: in};
      var t: ms {pub: in};
      var n: dimensionless {init: 0.01, pub: out};
      var alpha_n: per_ms;
      var beta_n: per_ms;
      alpha_n = -0.0001{per_mV_ms}*(V+50{mV})
        /(exp(-(V+50{mV})/10{mV})-1{dimensionless});
      beta_n = 0.0020{per_ms}*exp(-(V+90{mV})/80{mV});
      ode(n,t)= alpha_n*(1{dimensionless}-n)-beta_n*n;
   enddef;
   def map between environment
     and potassium_channel for
      vars V and V;
      vars t and t;
   enddef;
   def map between potassium_channel and
     potassium_channel_n_gate for
      vars V and V;
      vars t and t;
      vars n and n;
   enddef;
enddef;
```

Raw text: `Noble62_L_channel.txt`, XML file Noble62_L_channel.cellml.

```
def model leakage_ion_channel as
   def import using "Noble62_units.xml" for
      unit mV using unit mV;
      unit ms using unit ms;
      unit microS using unit microS;
      unit nanoA using unit nanoA;
   enddef;
   def comp leakage_channel as
      var V: mV {pub: in};
      var g_L: microS {init: 75};
      var E_L: mV {init: -60};
      var i_L: nanoA {pub: out};
      i_L = g_L*(V-E_L);
   enddef;
enddef;
```

We have now covered all existing features of CellML and OpenCOR. But, most importantly, you have learned 'best practice' for building CellML models, including encapsulation of sub-components and a modular approach in which

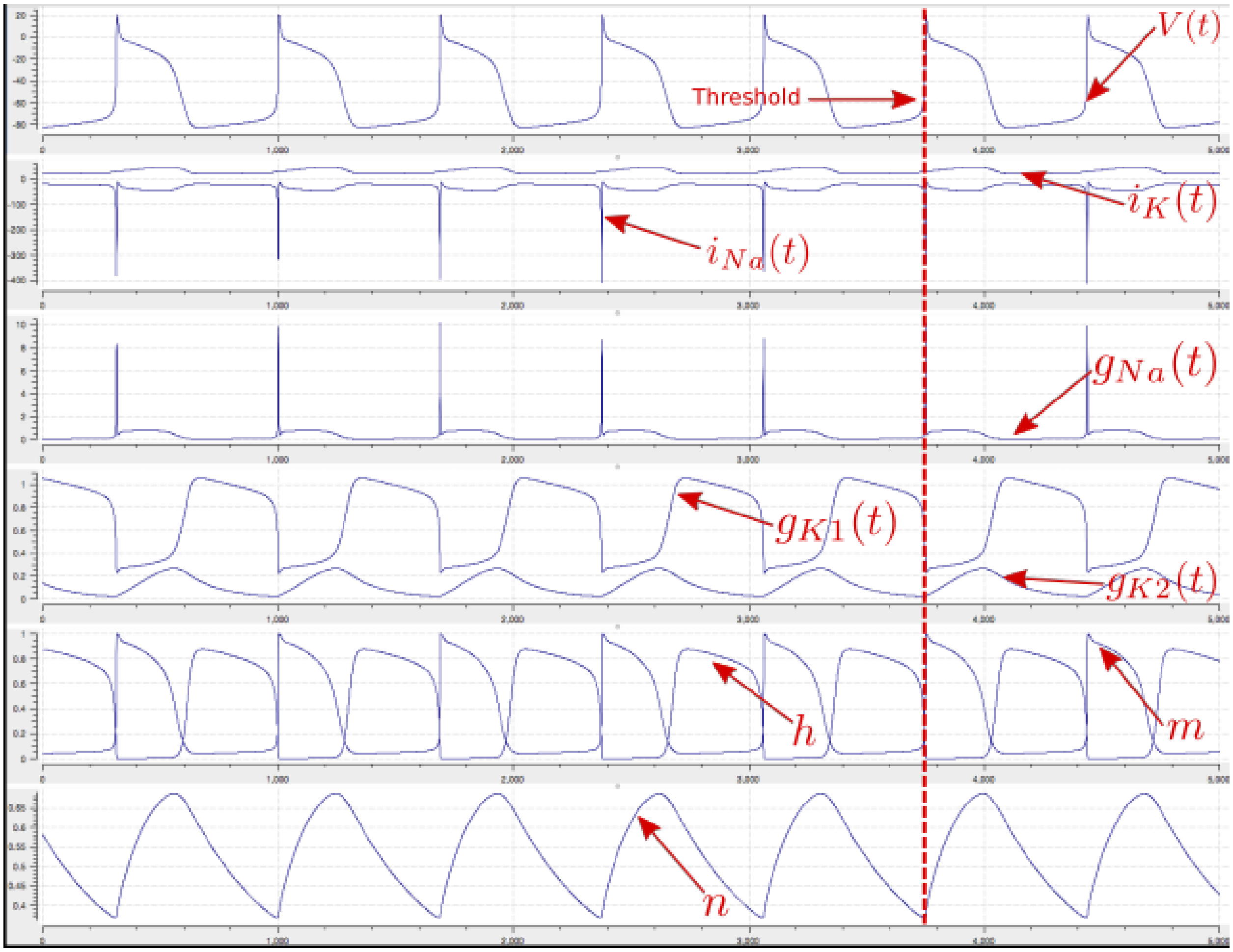


Fig. 1.76: Output from the Noble62 model (OpenCOR link). Top panel is $V(t)$, the cardiac action potential. The next panel has the two membrane ion channel currents $i_{\mathrm{Na}}(t)$ and $i_K(t)$. Note that $i_{\mathrm{Na}}(t)$ has a very brief downward (i.e. inward current) spike that is triggered when the membrane voltage reaches about -70mV. This is caused by the huge increase in sodium channel conductance $g_{\mathrm{Na}}(t)$ shown in the panel below associated with the simultaneous opening of the *m*-gate and closing of the *h*-gate (5th panel down). The resting state of about -80mV in the top panel is set by the potassium equilibrium (Nernst) potential via the open potassium channels. As can be seen from the 4th and bottom panels, it is the closing of the time-dependent potassium *n*-gate and the corresponding decline of potassium conductance that, with a small background leakage current $i_L(t)$, leads to the membrane potential rising from -80mV to the threshold for activation of the sodium channel (note the dotted red line showing the point when *n(t)* reaches a minimum). Later cardiac cell models include additional ion channels that directly affect the heart rate by controlling this rise.

units, parameters and model components are defined in separate files that are imported into a composite model.

The final **key concept** is that of reusing existing modules, defined using the previous concepts.

- *A model of the nerve action potential: Introducing CellML imports* introduces the mechanism by which existing modules can be reused in CellML models.

When describing your model it is useful to separate the mathematical model from its instaniation with a specific set of parameter values. This allows the model to be reused with different parameter sets for different purposes, including different simulation protocols, etc. Furthermore, rather than continually redefining the same set of units, it is better practice to simply define your units once and reuse their definition as required. Potentially even defining a standard set of units for a research group or collaboration to use and storing that in the repository for everyone to make use of.

- In *A model of the cardiac action potential: Importing units and parameters* we demonstrate the separation of model parameters and reuse of a common units definition.

1.3.2 Collaboration, versioning, discovery

The Physiome Model Repository and the link to bioinformatics

The Physiome Model Repository (PMR) [LCPF08] is the main online repository for the IUPS Physiome Project, providing version and access controlled repositories, called *workspaces*, for users to store their data. Currently there are over 700 public workspaces and many private workspaces in the repository. PMR also provides a mechanism to create persistent access to specific revisions of a workspace, termed *exposures*. Exposure plugins are available for specific types of data (e.g. CellML or FieldML documents) which enable customizable views of the data when browsing the repository via a web browser, or an application accessing the repository's content via web services.

More complete documentation describing how to use PMR is available in the PMR documentation: https://models.physiomeproject.org/docs.

The CellML models on models.physiomeproject.org are listed under 20 categories, shown below: (numbers of exposures in each category are given besides the bar graph, correct as at early 2016)

Browse by category

Calcium Dynamics	140
Cardiovascular Circulation	60
Cell Cycle	38
Cell Migration	2
Circadian Rhythms	22
Electrophysiology	230
Endocrine	60
Excitation-Contraction Coupling	22
Gene Regulation	12
Hepatology	29
Immunology	55
Ion transport	13
Mechanical Constitutive Laws	19
Metabolism	86
Myofilament Mechanics	22
Neurobiology	33
pH regulation	2
PKPD	11
Signal Transduction	120
Synthetic Biology	6

Note that searching of models can be done anywhere on the site using the search box on the upper right hand corner. An important benefit of ensuring that the models on the PMR are annotated is that models can then be retrieved by a web-search using any of the annotated terms in the models.

To illustrate the features of PMR, click on the Hund, Rudy 2004 (Basic) model in the alphabetic listing of models under the *Electrophysiology* category.

The section labelled 'Model Structure' contains the journal paper abstract and often a diagram of the model[1]. This is shown for the Hund-Rudy 2004 model in Fig. 1.78. This model, with over 22 separate protein model components, is also a good example of why it is important to build models from modular components [CMEJ08], and in particular the individual ion channels for electrophysiology models.

There is a list of 'Views Available' for the CellML model on the right hand side of the exposure page. The function of each of these views is as follows:

Views Available

Documentation - Takes you to the main exposure page.

Model Metadata - Lists metadata including authors, title, journal, Pubmed ID and model annotations.

Model Curation - Provides the curation status of the model. Note: this is soon to be updated.

Mathematics - Displays all the mathematical equations contained in the model.

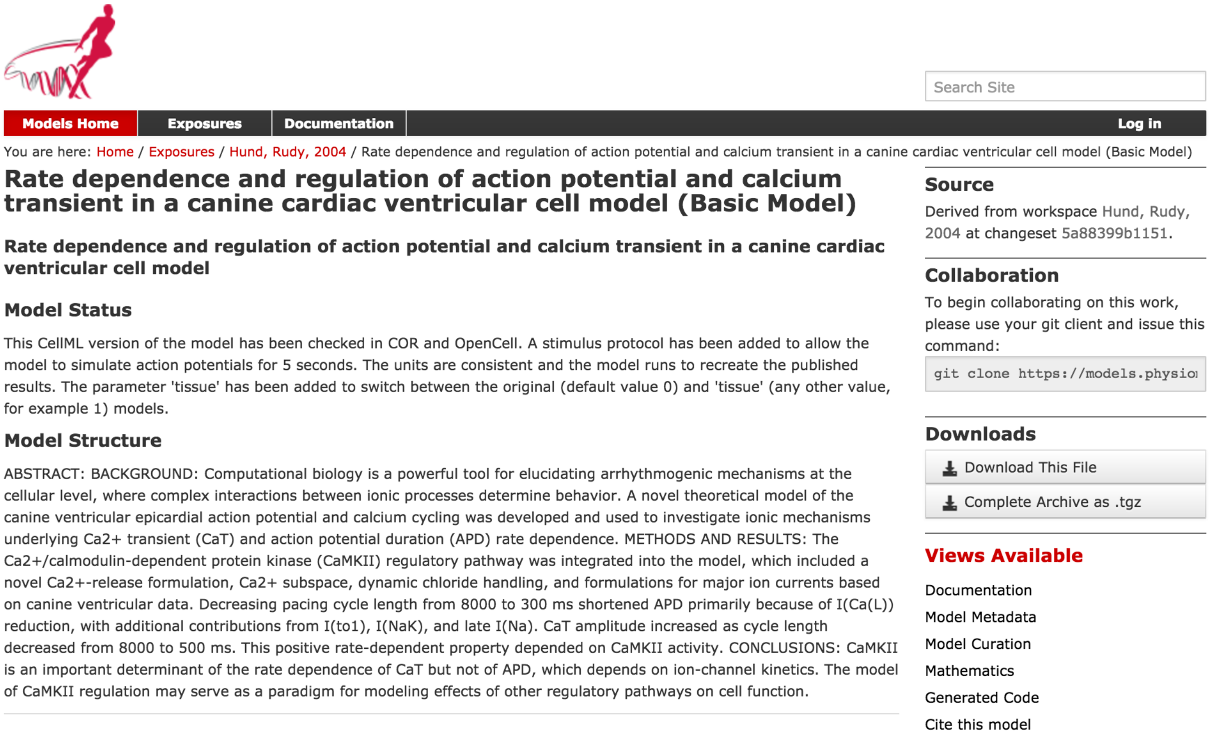
Generated Code - Various codes (C, C-IDA, F77, MATLAB or Python) generated from the model.

Cite this model - Provides details on how to cite use of the CellML model.

Source view - Gives a full listing of the XML code for the model.

Launch with OpenCOR - Opens the model (or simulation experiment) in OpenCOR.

[1] These are currently hand drawn SVG diagrams but the plan is to automatically generate them from the model annotation and also (at some stage!) to animate them as the model is executed.



Search Site

Models Home | Exposures | Documentation | Log in

You are here: Home / Exposures / Hund, Rudy, 2004 / Rate dependence and regulation of action potential and calcium transient in a canine cardiac ventricular cell model (Basic Model)

Rate dependence and regulation of action potential and calcium transient in a canine cardiac ventricular cell model (Basic Model)

Rate dependence and regulation of action potential and calcium transient in a canine cardiac ventricular cell model

Model Status

This CellML version of the model has been checked in COR and OpenCell. A stimulus protocol has been added to allow the model to simulate action potentials for 5 seconds. The units are consistent and the model runs to recreate the published results. The parameter 'tissue' has been added to switch between the original (default value 0) and 'tissue' (any other value, for example 1) models.

Model Structure

ABSTRACT: BACKGROUND: Computational biology is a powerful tool for elucidating arrhythmogenic mechanisms at the cellular level, where complex interactions between ionic processes determine behavior. A novel theoretical model of the canine ventricular epicardial action potential and calcium cycling was developed and used to investigate ionic mechanisms underlying Ca2+ transient (CaT) and action potential duration (APD) rate dependence. METHODS AND RESULTS: The Ca2+/calmodulin-dependent protein kinase (CaMKII) regulatory pathway was integrated into the model, which included a novel Ca2+-release formulation, Ca2+ subspace, dynamic chloride handling, and formulations for major ion currents based on canine ventricular data. Decreasing pacing cycle length from 8000 to 300 ms shortened APD primarily because of I(Ca(L)) reduction, with additional contributions from I(to1), I(NaK), and late I(Na). CaT amplitude increased as cycle length decreased from 8000 to 500 ms. This positive rate-dependent property depended on CaMKII activity. CONCLUSIONS: CaMKII is an important determinant of the rate dependence of CaT but not of APD, which depends on ion-channel kinetics. The model of CaMKII regulation may serve as a paradigm for modeling effects of other regulatory pathways on cell function.

Source

Derived from workspace Hund, Rudy, 2004 at changeset 5a88399b1151.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone https://models.physio
```

Downloads

Download This File

Complete Archive as .tgz

Views Available

Documentation

Model Metadata

Model Curation

Mathematics

Generated Code

Cite this model

Fig. 1.77: The Physiome Model Repository exposure page for the basic Hund-Rudy 2004 model.

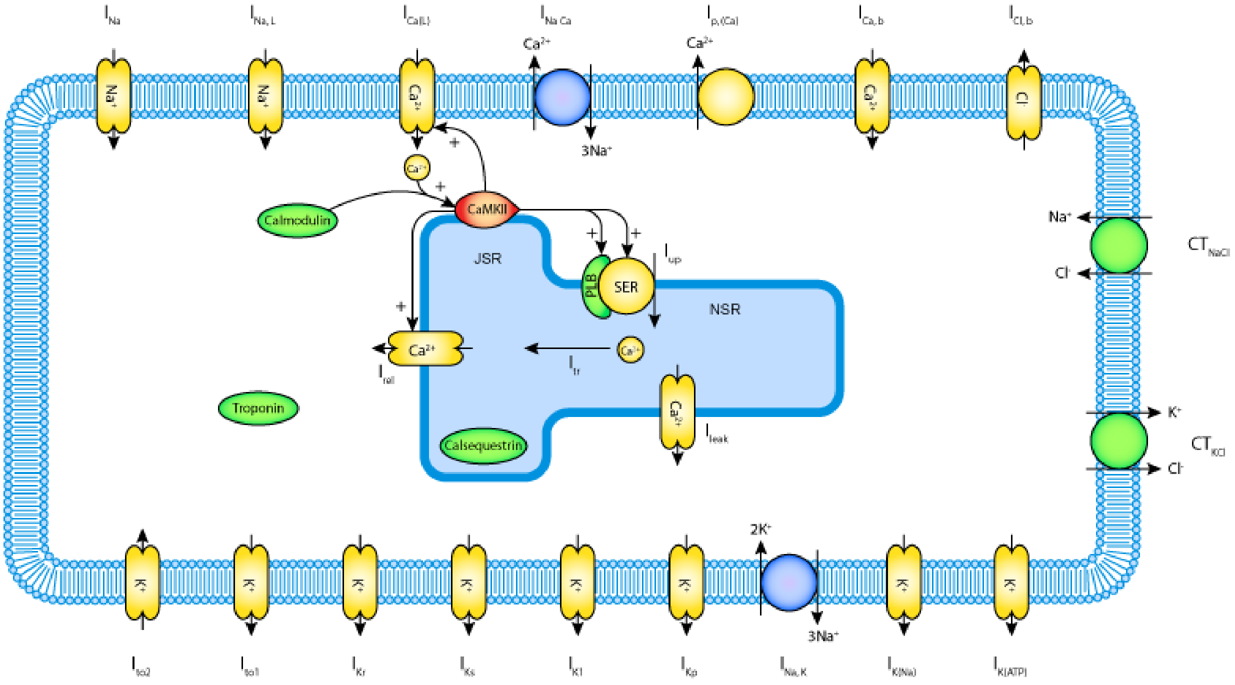


Fig. 1.78: A diagrammatic representation of the Hund-Rudy 2004 model.

Note that CellML models are available under a Creative Commons Attribution 3.0 Unported License[2]. This means that you are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

for any purpose, including commercial use.

The next stage of content development for PMR is to provide a list of the modular components of these models each with their own exposure. For example, models for each of the individual ion channels used in the publication-based electrophysiological models will be available as standalone models that can then be imported as appropriate into a new composite model. Similarly for enzymes in metabolic pathways and signalling complexes in signalling pathways, etc. Some examples of these protein modules are:

Sodium/hydrogen exchanger 3 https://models.physiomeproject.org/e/236/

Thiazide-sensitive Na-Cl cotransporter https://models.physiomeproject.org/e/231/

Sodium/glucose cotransporter 1 https://models.physiomeproject.org/e/232/

Sodium/glucose cotransporter 2 https://models.physiomeproject.org/e/233/

Note that in each case, as well as the CellML-encoded mathematical model, links are provided (see Fig. 1.79) to the UniProt Knowledgebase for that protein, and to the Foundational Model of Anatomy (FMA) ontology (via the EMBLE-EBI Ontology Lookup Service) for information about tissue regions relevant to the expression of that protein (e.g. *Proximal convoluted tubule*, *Apical plasma membrane*; *Epithelial cell of proximal tubule*; *Proximal straight tubule*). Similar facilities are available for SMBL-encoded biochemical reaction models through the Biomodels database [AYY].

Using PMR with OpenCOR

In addition to the *PMR* window for browsing public exposures directly in OpenCOR (*PMR window*) OpenCOR has the ability for users to directly create and access their workspaces in PMR.

Note: It is a feature of PMR that all data is persistent and permanent. As such, any workspaces created on the main instance of PMR (https://models.physiomeproject.org/) can not be deleted. For the purposes of teaching, we have an alternate instance of PMR (https://teaching.physiomeproject.org/) which is periodically cleared out and synschronised from the main instance. Using the teaching instance allows you to play around without the worry of things being permanent.

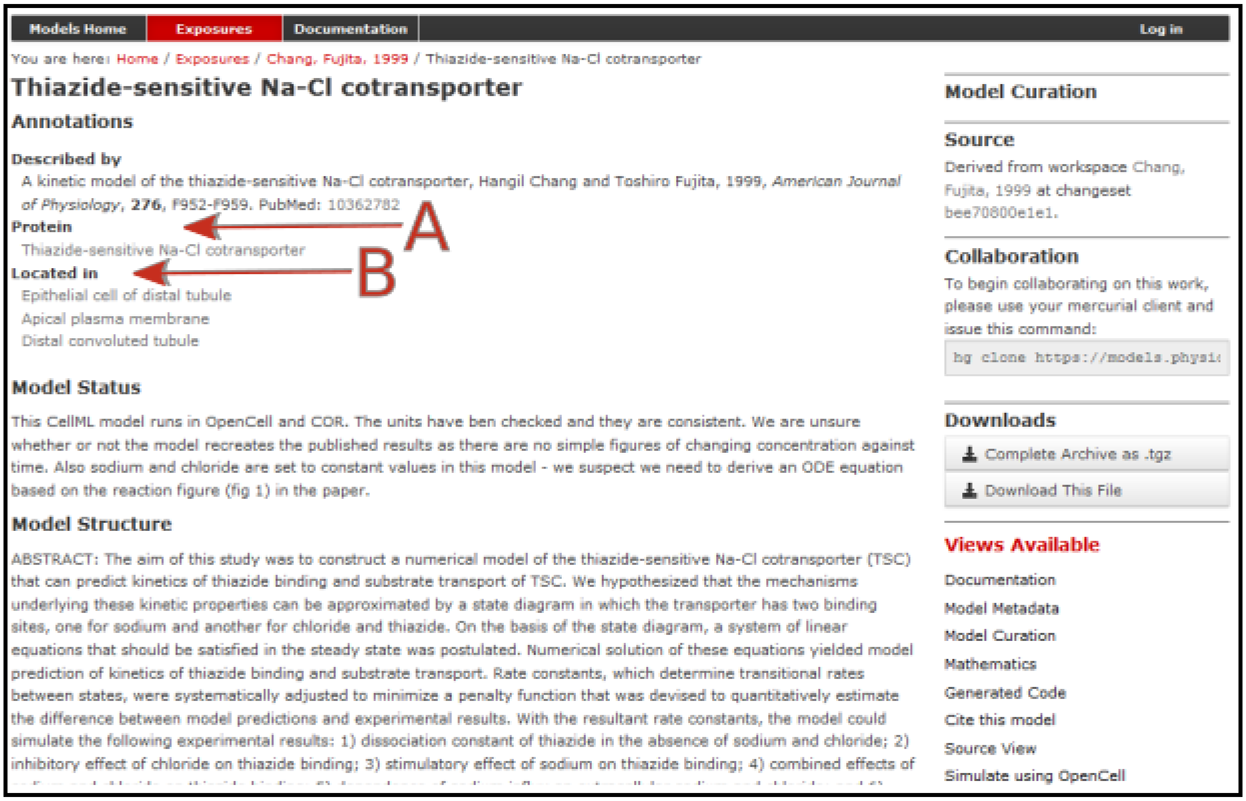
1. Register for a user account on the teaching instance of PMR.

In order to make use of the teaching instance of PMR, you must first have an account for that instance of the repository. For teaching purposes it is best to register a new account. This can be done by first opening this link in your browser: https://teaching.physiomeproject.org. Then click the *Log in* button on (shown in Fig. 1.80) then the *registration form* link.

After filling in the names and email fields and clicking *Register* you will receive an email inviting you to confirm and set a password. Once that is completed you can then log in. Clicking on *My Workspaces* will take you to a listing of all your workspaces and provides access to the the *Workspace creation form*.

2. Create a new workspace, in this example the title ‘Test workspace’ has been used.

[2] https://creativecommons.org/licenses/by/3.0/



You are here: Home / Exposures / Chang, Fujita, 1999 / Thiazide-sensitive Na-Cl cotransporter

Thiazide-sensitive Na-Cl cotransporter

Annotations

Described by

A kinetic model of the thiazide-sensitive Na-Cl cotransporter, Hangil Chang and Toshiro Fujita, 1999, *American Journal of Physiology*, **276**, F952-F959. PubMed: 10362782

Protein

Thiazide-sensitive Na-Cl cotransporter

Located in

Epithelial cell of distal tubule
Apical plasma membrane
Distal convoluted tubule

Model Status

This CellML model runs in OpenCell and COR. The units have ben checked and they are consistent. We are unsure whether or not the model recreates the published results as there are no simple figures of changing concentration against time. Also sodium and chloride are set to constant values in this model - we suspect we need to derive an ODE equation based on the reaction figure (fig 1) in the paper.

Model Structure

ABSTRACT: The aim of this study was to construct a numerical model of the thiazide-sensitive Na-Cl cotransporter (TSC) that can predict kinetics of thiazide binding and substrate transport of TSC. We hypothesized that the mechanisms underlying these kinetic properties can be approximated by a state diagram in which the transporter has two binding sites, one for sodium and another for chloride and thiazide. On the basis of the state diagram, a system of linear equations that should be satisfied in the steady state was postulated. Numerical solution of these equations yielded model prediction of kinetics of thiazide binding and substrate transport. Rate constants, which determine transitional rates between states, were systematically adjusted to minimize a penalty function that was devised to quantitatively estimate the difference between model predictions and experimental results. With the resultant rate constants, the model could simulate the following experimental results: 1) dissociation constant of thiazide in the absence of sodium and chloride; 2) inhibitory effect of chloride on thiazide binding; 3) stimulatory effect of sodium on thiazide binding; 4) combined effects of

Model Curation

Source

Derived from workspace Chang, Fujita, 1999 at changeset bee70800e1e1.

Collaboration

To begin collaborating on this work, please use your mercurial client and issue this command:

```
hg clone https://models.physi
```

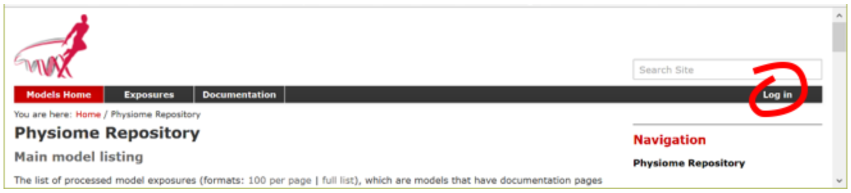
Downloads

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

Fig. 1.79: The PMR workspace for the Thiazide-sensitive Na-Cl cotransporter. Bioinformatic data for this model is accessed via the links under the headings highlight by the arrows and include **Protein** (labelled A) and the model **Location** (labelled B). Other information is as already described for the Hund-Rudy 2004 model.



You are here: Home / Physiome Repository

Physiome Repository

Main model listing

The list of processed model exposures (formats: 100 per page | full list), which are models that have documentation pages

Navigation

Physiome Repository

Fig. 1.80: The log in button for the teaching instance of PMR.

The PMR Workspaces window

A window labelled *PMR workspaces* is available in OpenCOR (see Fig. 1.81). If it is not currently visible it can be selected via *View* → *Windows* → *PMR workspaces* (or perhaps the `Ctrl-space` shortcut).

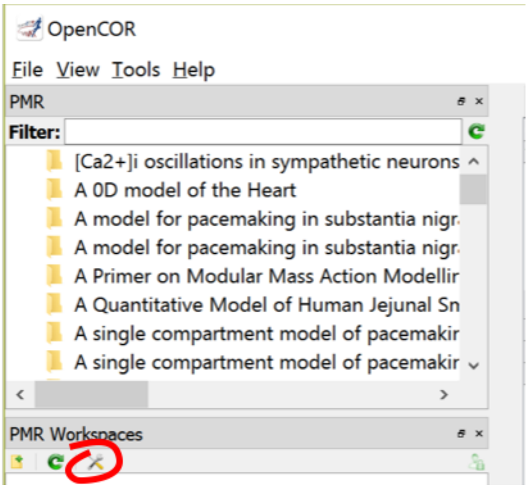

OpenCOR
File View Tools Help
PMR
Filter:
[Ca2+]i oscillations in sympathetic neurons
A 0D model of the Heart
A model for pacemaking in substantia nigr
A model for pacemaking in substantia nigr
A Primer on Modular Mass Action Modellir
A Quantitative Model of Human Jejunal Sn
A single compartment model of pacemakir
A single compartment model of pacemakir
PMR Workspaces

Fig. 1.81: PMR workspace shown on the left hand panel in OpenCOR. The preferences button is highlighted.

3. Set preferences.

Clicking the preferences button (Fig. 1.81) presents a *Preferences* dialog box with three settings: PMR instance, Name and Email. For the current purpose choose *https://teaching.physiomeproject.org* for the first and enter your name and email. These are used to identify you as the author of changes you submit back to the repository (view an example history).

4. Log into PMR from OpenCOR.

Before you can view private information or submit changes to PMR you must first log in to PMR from OpenCOR and grant OpenCOR permission to use your account. You accomplish this by clicking on the top right button in the *PMR Workspaces* window and then logging in with your new user name and password (created in step 1). Then grant access for OpenCOR to gain access to your PMR workspaces. The PMR workspaces window will then show all your workspaces, which should currently consist of the new workspace created in step 2. Note that using the same top right button you can log off - and when you next authenticate you will again be asked to grant access but this time without needing to login with your password.

Right clicking on the workspace name brings up a list of options for that workspace, the first being to view the workspace within PMR (in the web browser). Another option allows you to make a local copy of a workspace on your local disk - this will create a copy of the workspace on your local computer in which you are able to make changes.

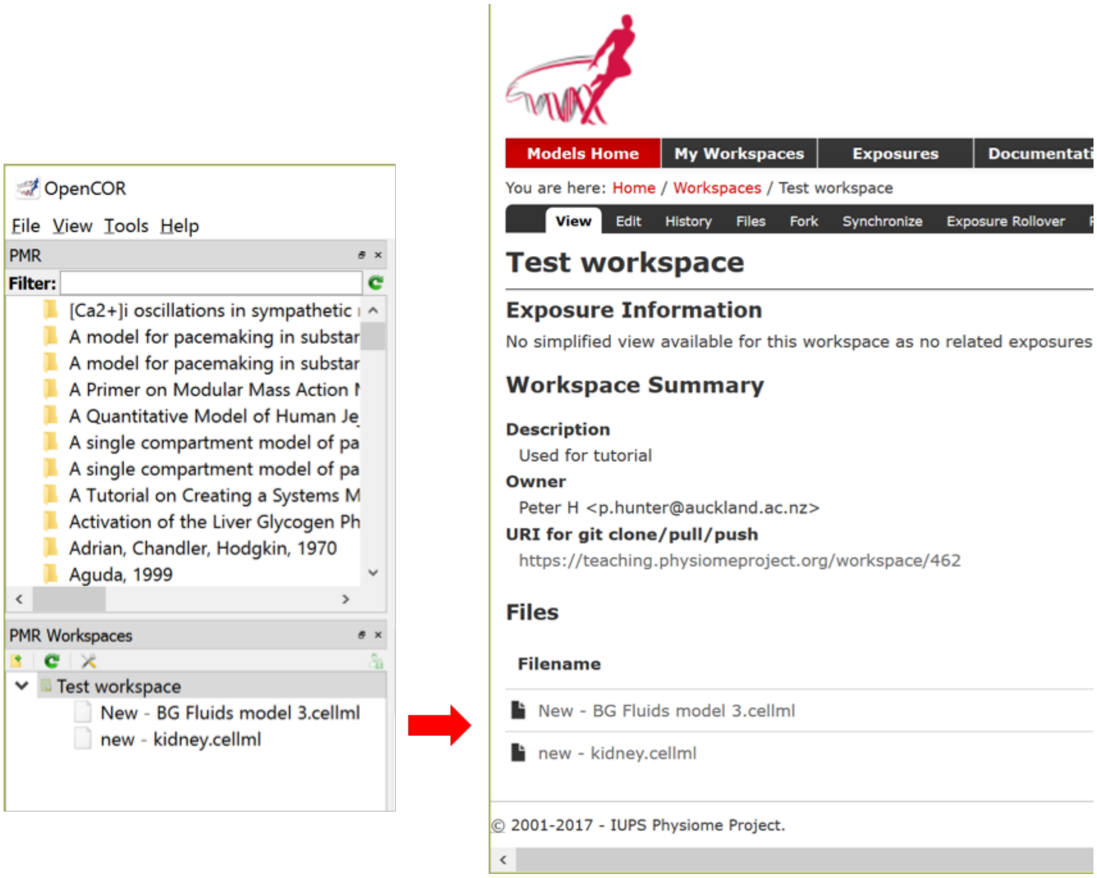
5. Make a local copy of your *test workspace*

Using the *Make Local Workspace Copy…* option from the right-click menu on the workspace you created in step 2, clone the workspace to your PC. When doing this you will need to provide the folder in which you want to store the workspace contents - make sure you remember where this folder is!

6. Save a CellML model to your workspace.

A CellML file opened in OpenCOR (choose any model you have access to) can be saved (*File* → *Save As…*) to the folder you created for the cloned workspace. Once you have saved a model you will see the file appear under the workspace's folder in the *PMR Workspaces* window. Note that the file appears under the workspace with a red patch on the logo indicating the the file is not yet flagged to upload. To upload the file to PMR, you need to choose *Synchronise Workspace With PMR…* from the right-click menu on the workspace folder. This will ask you to provide a description of the change you would like to submit to PMR, and display all the differences you will be synchronising.

When you now click the *OK* button, the changes will actually be submitted to PMR and you will see the file appear on the refreshed browser window. The file icon in the PMR Workspaces window will be shown without the red or green patch. Fig. 1.82 shows two CellML files that have been uploaded to PMR.



[Ca2+]i oscillations in sympathetic
A model for pacemaking in substar
A model for pacemaking in substar
A Primer on Modular Mass Action
A Quantitative Model of Human Je
A single compartment model of pa
A single compartment model of pa
A Tutorial on Creating a Systems M
Activation of the Liver Glycogen Ph
Adrian, Chandler, Hodgkin, 1970
Aguda, 1999

Test workspace
New - BG Fluids model 3.cellml
new - kidney.cellml

You are here: Home / Workspaces / Test workspace

Test workspace

Exposure Information

No simplified view available for this workspace as no related exposures

Workspace Summary

Description
Used for tutorial

Owner
Peter H <p.hunter@auckland.ac.nz>

URI for git clone/pull/push
https://teaching.physiomeproject.org/workspace/462

Files

Filename
New - BG Fluids model 3.cellml
new - kidney.cellml

© 2001-2017 - IUPS Physiome Project.

Fig. 1.82: Two CellML files (`New - BG Fluids model 3.cellml` and `new - kidney.cellml`) have been uploaded from OpenCOR to PMR and can be seen in the PMR workspace on the browser window on the right.

One of the main reasons to encode your mathematical models following the principles described in the previous sections is to make the model available for future scientists (again, including yourself!) to utilise the model in novel investigations. The Physiome Model Repository (PMR) has been developed to provide scientists with a suitable repository for sharing their work with the community. Full details are provided in the PMR documentation, but here we highlight some PMR capabilities.

- We have *previously introduced* the various ways to load models from PMR into OpenCOR. In *The Physiome Model Repository and the link to bioinformatics* we illustrate some of the extra information that is available via the PMR web interface.
- Choosing your favourite area of physiology, see if you can discover an interesting model in PMR and load it into OpenCOR to explore. Do you find this a more comprehensive way to understand the model than only reading the source literature?

The primary unit of information in PMR is the workspace. Each workspace is a version controlled repository which is permanently and persistently available, with each resource in the workspace assigned a unique identifier for each version of the resource submitted to the repository.

- In *Using PMR with OpenCOR* we present OpenCOR's capabilties for directly working with workspaces in PMR.
- Compare the method described in step 8 with the previous method scientists had to follow in order to get models

into PMR. Do you see why most scientists would prefer to email their models to Andre and let him add them to the repository?

- Using what you have just learnt, create a new workspace and add the models you created above to the workspace. Make sure that you can see the models in the web interface (you can easily access the workspace URL using the context menu in the *PMR Workspaces* window in OpenCOR).
- As we have mentioned *previously*, it is possible to export SED-ML from OpenCOR containing all your graph and simulation settings. Try exporting SED-ML for one of the models you created above and also saving it in th e PMR workspace you have created.
- Once you are happy with the content of your worspace and have everything synchronised with the repository, you can *submit your workspace for publication*. This will notify the tutors to check your workspace and see if they can reproduce your simulation experiments.

Extra for experts: if you look through the *PMR documentation* you will be able to find information on how to share your worksapces directly with collaborators and make *exposures* for specific revisions of the workspace.

1.3.3 A bond graph-based method for representing physiology

A new field of research that has emerged recently is the application of bond graph theory to the field of systems biology and physiology. This is providing the theoretical framework to guide the development of core modules that can be arbitrarily combined to capture a wide range of physiological phenomena. Combining this framework with CellML has started to provide an extremely powerful technology for both defining core modules spanning the required physics and the methods for integrating these modules into models of physiological systems.

Introduction

The bond graph approach to formulating models dealing with mass and energy transfer was developed by Henry Paynter in the 1960s to represent electro-mechanical control systems [Pay61]. It was later extended to include chemical processes by [Bre84], including concepts from the theory of network thermodynamics by Aharon Katchalsky and colleagues [OPK71]. Papers by Peter Gawthrop and Edmund Crampin have brought the approach into the bioengineering domain [GC14][GCC15][GSK+15][GC16].

The **first** key idea, based on recognising that energy and power are the only quantities that are common across different physical systems, is to separate energy transmission from storage and dissipation, and to provide the concept of *potential* (called “effort” in the engineering literature) with units of Joules per some_quantity as the common driving force behind the *flow* of that some_quantity per second. The product of potential and flow is then always power in units of Joules per second. The “some_quantity” has units of metres, meters^3, coulombs, candela, moles or entropy for, respectively, rigid body mechanics, continuum mechanics (including fluid flow), electrical, electromagnetic, chemical and heat transfer processes. As explained further below, the **second** key concept is that of a *0-junction*, where potential is defined and mass balance is applied, and a *1-junction*, where flow is defined and energy balance is applied. The extraordinary utility of these concepts is to recognise that Kirchhoff’s voltage law in electrical circuits, Newton’s force balance in a mechanical system, and stoichiometric balance in a biochemical system, are all just different manifestations of the same underlying principle of energy conservation and can therefore be represented by the same bond graph equation.

Units

Many physical systems can be described by a driving *potential* expressed as Joules per unit of some quantity, and a *flow* expressed as that quantity per second. The quantity could be coulomb, meters, moles, *etc.* in different physical systems. The power is always the product of the driving force and the flow expressed as Joules per second. The seven units of the SI system under the newly proposed definitions are now based on constants that are consistent with the use of Joules and seconds (together covering energy and power), metres, moles, entropy, coulombs and candela. Fig. 1.83 displays the bond graph concepts in different energy domains.

Formulation

In bond graph formulation, there are four basic variables. These variables for different physical systems are listed in . In electrical engineering these variables are given by: potential μ is energy density or *voltage* (J.C $^{-1}$), flow υ is *current* (C.s $^{-1}$), time integral of potential p is *momentum* (J.s.C $^{-1}$) and time integral of flow q is quantity or *coulombs* (C). Product $\mu.\upsilon$ is power (J.s $^{-1}$) which is a generalised coordinate to model the complete systems residing in several energy domains. A bond with covariables μ and υ is therefore used to represent *transmission of energy*. The bond represents a mechanism for the transmission of energy and power, and the arrow head indicates the assumed direction of power flow (see Fig. 1.84). The flow υ and potential μ must satisfy conservation laws.

Bond graph elements

Bond graph formulation is a graphical notation for the set of linear constraint equations (the conservation laws), but the constitutive relations can be nonlinear.

The constitutive equations of the bond graph elements are introduced via examples from the electrical and mechanical domains. The nature of the constitutive equations lay demands on the causality of the connected bonds. Bond graph elements are drawn as letter combinations (mnemonic codes) indicating the type of element. The bond graph elements are the following:

- *C*: static storage element, *e.g.* capacitor (stores charge), spring (stores displacement).
- *I*: dynamic storage element, *e.g.* inductor (stores flux linkage), mass (stores momentum).
- *R*: resistor dissipating free energy, *e.g.* electric resistor, mechanical friction.
- *SE* and *SF*: sources, *e.g.* electric mains (voltage source), gravity (force source), pump (flow source).
- *TF*: transformer, *e.g.* an electric transformer, toothed wheels, lever.
- *GY*: gyrator, *e.g.* electromotor, centrifugal pump.
- *0* and *1*: *0-junctions* and *1-junctions*, for ideal connecting two or more submodels.

Fig. 1.85 shows the relation of the state variables to the constitutive relations.

Storage elements

Storage elements store all kinds of free energy. As indicated above, there are two types of storage elements: *C-elements* and *I-elements*. In *C-elements*, like a capacitor or spring, the conserved quantity, *q*, is stored by accumulating the net flow, υ, to the storage element. This results in the differential equation:

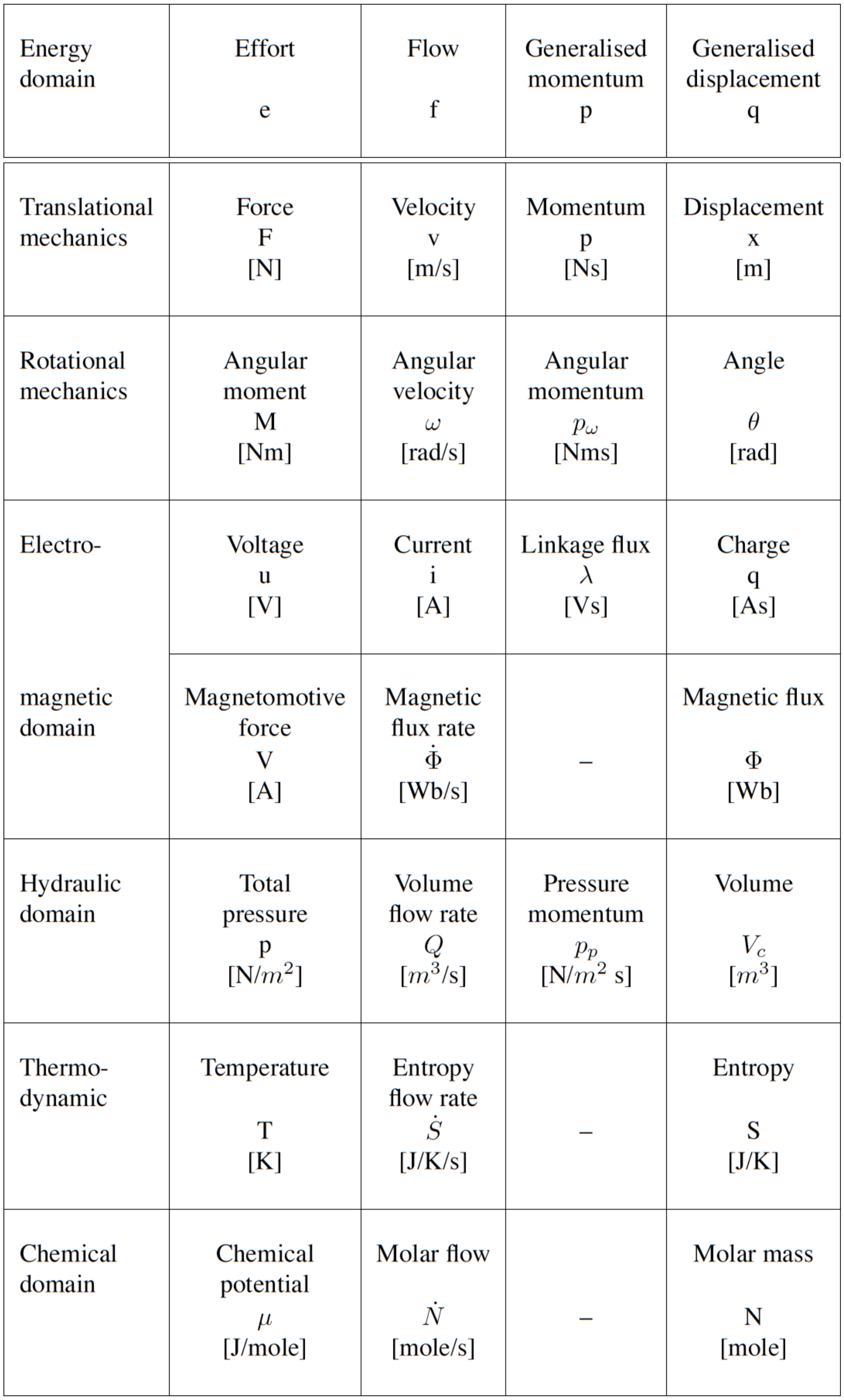
$$\dot{q} = \upsilon$$

which is called a balance equation, and forms a part of the constitutive equations of the storage element. In the other part of the constitutive equations, the state variable, *q*, is related to the potential:

$$\mu = \frac{q}{C}$$

In *I–elements*, like an inductor or mass, the conserved quantity, *p*, is stored by accumulating the net potential, μ, to the storage element. The resulting differential equation is:

$$\dot{p} = \mu$$



Energy domain	Effort e	Flow f	Generalised momentum p	Generalised displacement q
Translational mechanics	Force F [N]	Velocity v [m/s]	Momentum p [Ns]	Displacement x [m]
Rotational mechanics	Angular moment M [Nm]	Angular velocity ω [rad/s]	Angular momentum p_ω [Nms]	Angle θ [rad]
Electro-	Voltage u [V]	Current i [A]	Linkage flux λ [Vs]	Charge q [As]
magnetic domain	Magnetomotive force V [A]	Magnetic flux rate $\dot{\Phi}$ [Wb/s]	–	Magnetic flux Φ [Wb]
Hydraulic domain	Total pressure p [N/m^2]	Volume flow rate Q [m^3/s]	Pressure momentum p_p [N/m^2 s]	Volume V_c [m^3]
Thermo-dynamic	Temperature T [K]	Entropy flow rate $\dot{S}$ [J/K/s]	–	Entropy S [J/K]
Chemical domain	Chemical potential μ [J/mole]	Molar flow $\dot{N}$ [mole/s]	–	Molar mass N [mole]

Fig. 1.83: Power and energy variables in various energy domains.

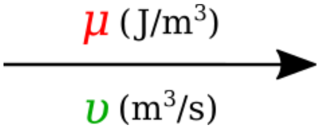


Fig. 1.84: Representation of energy bond.

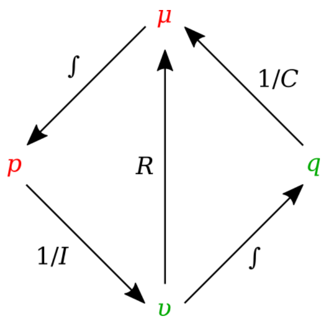


Fig. 1.85: State variables and constitutive relations in the bond graph approach.

which is the balance equation. The element-specific part of the constitutive equations is:

$$\upsilon = \frac{p}{I}$$

Resistors

Resistors, *R–elements*, dissipate free energy. Examples are dampers, frictions and electric resistors. The constitutive equation is an algebraic relation between the potential and flow.

$$\mu = \upsilon R$$

Sources

Sources represent the interaction of a system with its environment. Examples are external forces, voltage and current sources, ideal motors, *etc.* Depending on the type of the imposed variable, these elements are drawn as *SE* or *SF*.

Transformers

An ideal transformer is represented by *TF* and is power continuous (*i.e.* no power is stored or dissipated). The transformation can within the same domain (toothed wheel, lever) or between different domains (electromotor, winch). The equations are:

$$\mu_1 = n\mu_2$$

$$\upsilon_2 = n\upsilon_1$$

Potentials are transduced to potentials and flows to flows. The parameter *n* is the transformer ratio. Due to the power continuity, only one dimensionless parameter, *n*, is needed to describe both the potential transduction and the flow transduction. The parameter *n* is unambiguously defined as follows: μ_1 and υ_1 belong to the bond pointing towards the *TF*. If *n* is not constant, the transformer is a modulated transformer, a *MTF*. The transformer ratio now becomes an input signal to the *MTF*.

Gyrators

An ideal *gyrator* is represented by *GY*, and is also power continuous (*i.e.* no power is stored or is dissipated. Examples are an electromotor, a pump and a turbine. Real-life realisations of gyrators are mostly transducers representing a domain-transformation. The equations are:

$$\mu_1 = r\upsilon_2$$

$$\mu_2 = r\upsilon_1$$

The parameter *r* is the *gyrator ratio*, and due to the power continuity, only one parameter to describe both equations. No further definition is needed since the equations are symmetric (it does not matter which bond points inwards, only that one bond points towards and the other points form the gyrator). *r* has a physical dimension, since *r* is a relation between effort and flow (it has the same dimension as the parameter of the *R-element*). If *r* is not constant, the gyrator is a *modulated gyrator*, a *MGY*.

Junctions

Junctions couple two or more elements in a power continuous way: there is no energy storage or dissipation in a junction. Examples are a series connection or a parallel connection in an electrical network, a fixed coupling between parts of a mechanical system.

The *0-junction* represents a node at which all potentials of the connecting bonds are equal. An example is a parallel connection in an electrical circuit. Due to the power continuity, the sum of the flows of the connecting bonds is zero, considering the sign. The power direction (*i.e.* direction of the arrow) determines the sign of the flows: all inward pointing bonds get a plus and all outward pointing bonds get a minus. This summation is the Kirchhoff current law in electrical networks: all currents connecting to one node sum to zero, considering their signs: all inward currents are positive and all outward currents are negative. We can depict the *0-junction* as the representation of a potential variable, and often the *0-junction* will be interpreted as such. The *0-junction* is more than the (generalised) Kirchhoff current law, namely also the equality of the potentials (like electrical voltages being equal at a parallel connection).

The *1-junction* is the dual form of the *0-junction* (roles of potential and flow are exchanged). The *1-junction* represents a node at which all flows of the connecting bonds are equal. An example is a series connection in an electrical circuit. The potentials sum to zero, as a consequence of the power continuity. Again, the power direction (*i.e.* direction of the arrow) determines the sign of the potentials: all inward pointing bonds get a plus and all outward pointing bonds get a minus. This summation is the Kirchhoff voltage law in electrical networks: the sum of all voltage differences along one closed loop (a mesh) is zero. In the mechanical domain, the *1–junction* represents a force balance (also called the principle of d'Alembert), and is a generalisation of Newton's third law, action = – reaction). Just as with the *0–junction*, the *1–junction* is more than these summations, namely the equality of the flows. Therefore, we can depict the *1–junction* as the representation of a flow variable, and often the *1-junction* will be interpreted as such.

Causality

Causality establishes the cause and the effect relationship. It specifically implies that either the potential or flow variable on that bond is known. Causality is generally indicated by a causal stroke at the end to which the potential receiver is connected. Elements which store or dissipate energy do not impose causality on the system, but they have preferred causality for computational reasons. These elements with their preferred causality are shown in Fig. 1.86.

In the bond graph approach, junctions interconnect the corresponding elements and constrain the possible causalities of the element ports connected to it. A *0-junction* can only have one potential output. In a similar way, a *1-junction* can only have one flow output. Fig. 1.87 illustrates causality in four-port *0-junction* and *1-junction*.

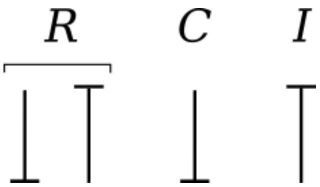


Fig. 1.86: Preferred causality for *R*, *C*, and *I* elements.

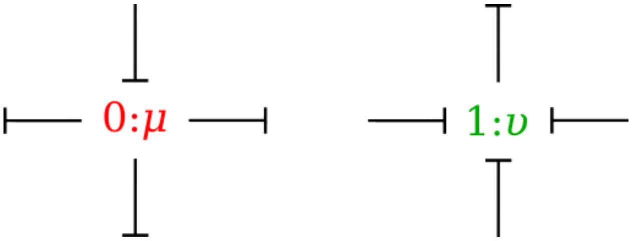


Fig. 1.87: Causality in four-port *0-junction* and *1-junction*.

Procedure

To generate a bond graph model starting from an ideal-physical model (IPM), a systematic method exist, which we will present here as a procedure. This procedure consists roughly of the identification of the domains and basic elements, the generation of the connection structure (called the junction structure), the placement of the elements, and possibly simplifying the graph. The procedure is different for the mechanical domain compared to the other domains. These differences are indicated between parenthesis. The reason is that elements need to be connected to difference variables or across variables. The potentials in the non-mechanical domains and the flows in the mechanical domains are the across variables we need.

Step 1 and 2 concern the identification of the domains and elements.

1. Determine which physical domains exist in the system and identify all basic elements like *C*, *I*, *R*, *SE*, *SF*, *TF* and *GY*. Give every element a unique name to distinguish them from each other.
2. Indicate in the ideal-physical model per domain a reference potential (reference flow with positive direction for the mechanical domains). Note that only the references in the mechanical domains have a direction.

Steps 3 through 6 describe the generation of the connection structure (called the junction structure).

3. Identify all other potentials (mechanical domains: flows) and give them unique names.
4. Draw these potentials (mechanical: flows), and not the references, graphically by *0-junctions* (mechanical: *1-junctions*). Keep if possible, the same layout as the IPM.
5. Identify all potential differences (mechanical: flow differences) needed to connect the ports of all elements enumerated in step 1 to the junction structure. Give these differences a unique name, preferably showing the difference nature. The difference between μ_1 and μ_2 can be indicated by μ $_{12}$.
6. Construct the potential differences using a *1-junction* (mechanical: flow differences with a *0-junction*) and draw them as such in the graph. The junction structure is now ready and the elements can be connected.
7. Connect the port of all elements found at step 1 with the *0-junctions* of the corresponding potential or potential differences (mechanical: *1-junctions* of the corresponding flows or flow differences).
8. Simplify the resulting graph by applying the following simplification rules:
   - A junction between two bonds can be left out, if the bonds have a 'through' power direction (one bond incoming, the other outgoing).

- A bond between two the same junctions can be left out, and the junctions can join into one junction.
- Two separately constructed identical potential or flow differences can join into one potential or flow difference.

We will illustrate these steps with a concrete example in the next section.

As this is a new and rapidly evolving field of research, the documentation is still being developed. This section of the DTP Computational Physiology module will be largely be taught using powerpoint slides that are available `here`.

Some references that are relevant to this work are given here for convenience.

- Paynter H. Analysis and Design of Engineering Systems (MIT, Cambridge, Mass., 1961).
- Oster G, Perelson A, and Katchalsky A. 1971. Network thermodynamics. *Nature* (Lond.). 234:393 (link).
- Gawthrop PJ and Crampin EJ. Energy based analysis of biochemical cycles using bond graphs. *Proc. R. Soc. A* 470:20140459, 2014 (link).
- Gawthrop PJ and Crampin EJ. Modular bond-graph modelling and analysis of biomolecularsystems. *IET Systems Biology*, 2015 (link).
- Gawthrop PJ, Cursons J and Crampin EJ. Hierarchical bond graph modelling of biochemical networks. *Proc. R. Soc A*, 471(2184), 2015 (link).
- Gawthrop PJ, Siekmann I, Kameneva T, Saha S, Ibbotson MR and Crampin EJ. The energetic cost of the action potential: bond graph modelling of electrochemical energy transduction in excitable membranes. arXiv:1512.00956 (link).
- Broenink, JF. 1999. Introduction to physical systems modelling with bond graphs. SiE Whitebook on Simulation Methodologies, 31.

Example models

A collection of models is available for this section of the module. The following steps can be used to create your own copy of these models for use in the tutorial.

1. Go to http://teaching.physiomeproject.org/workspace/2cd in your web browser.
2. Make sure you are logged in to the teaching instance of PMR.
3. Click on *Fork* in the workspace menu, followed by the *Fork* button. This will create your own private copy of the workspace containing these example models.
4. In OpenCOR, make sure you are *logged into the teaching instance of PMR*.
5. You might need to click the *Reload* button (green arrow) to update the window to show your copy of the *Example bond graph models* workspace.
6. Right-click on the example bond graph models workspace folder and *Make Local Workspace Copy…* the workspace (as per *step 5*).

You will now have a copy of the example models on your computer.

1.4 BondGraphTools

Contents:

In the following section a group of different examples in diverse domains of physics are modeled by using a Python library called **BondGraphTools**, designed and developed by Peter Cudmore.

1.4.1 Introduction

BondGraphTools is a python library for building and manipulating symbolic models of complex physical systems, built upon the standard scientific python libraries[1]. Here a number of examples from electrical circuits, fluid mechanics, and biochemical reactions are illustrated and modeled by the BondGraphTools (BGT). BondGraphTools dynamically generates Julia code, uses the DifferentialEqations.jl interface to solve the DAE, and passes the results back to python for analysis and plotting[1]. The codes can be executed by entering them into an IPython session or a Jupyter notebook[1]. Here we have opted for the Jupyter notebook.

1.4.2 Examples

Electrical Circuits

Electrical Circuit #1

The circuits are adopted from[2]

Fig. 1.88: Schematic of Circuit #1 (R: resistor, C: capacitor, u: potential, v: flow)

In the first step, the BondgraphTools library must be imported:

[1] Cudmore, P., & Gawthrop, P. J., & Pan, M., & Crampin, E. J. (2019). Computer-aided modelling of complex physical systems with BondGraphTools. Click for the article

[2] Hunter, P., & Safaei, S. (2017). Bond Graphs, CellML, ApiNATOMY & OpenCOR. Retrieved from here

```
import BondGraphTools as bgt
```

To create a new model using BondGraphTools (bgt), the command *bgt.new* is used:

```
model=bgt.new(name='circuit_1')
```

In the next step, all the parameters' values are defined:

```
C1_value=100e-6
C2_value=150e-6
R_value=100e+3
```

These values are then assigned to bgt components using the following commands:

```
C1=bgt.new("C", value=C1_value)
C2=bgt.new("C", value=C2_value)
R=bgt.new("R", value=R_value)
```

Also, a number of *"0-junctions & 1-junctions"* must be defined (based on our model) as follows:

```
zero_junc_1=bgt.new("0")
zero_junc_2=bgt.new("0")
one_junc=bgt.new("1")
```

Now creating the model and its components has finished and all of them must be assembled using the *bgt.add* command:

```
bgt.add(model,C1,C2,R,zero_junc_1,zero_junc_2,one_junc)
```

According to our bond graph model, these components must be connected to the related junctions by *bgt.connect*. Note that the first element in parenthesis represents the *"tail"* of the arrow and the second element represents the *"head"*:

```
bgt.connect(C1,zero_junc_1)
bgt.connect(zero_junc_1,one_junc)
bgt.connect(one_junc,R)
bgt.connect(one_junc,zero_junc_2)
bgt.connect(zero_junc_2,C2)
```

By drawing the model, one can see if the components are connected properly to each other or not:

```
bgt.draw(model)
```

A sketch of the network will then be produced:

Now that the bond graph demonstration of the system is done, we can illustrate its behaviour during a specific time interval and with arbitrary initial conditions for the state variables. The constitutive relations of the model can be shown as well:

```
timespan=[0,50]
model.state_vars
Out[ ]: {'x_0': (C: C1, 'q_0'), 'x_1': (C: C2, 'q_0')}
```

==>

```
x0={"x_0":1, "x_1":0}
```

==>

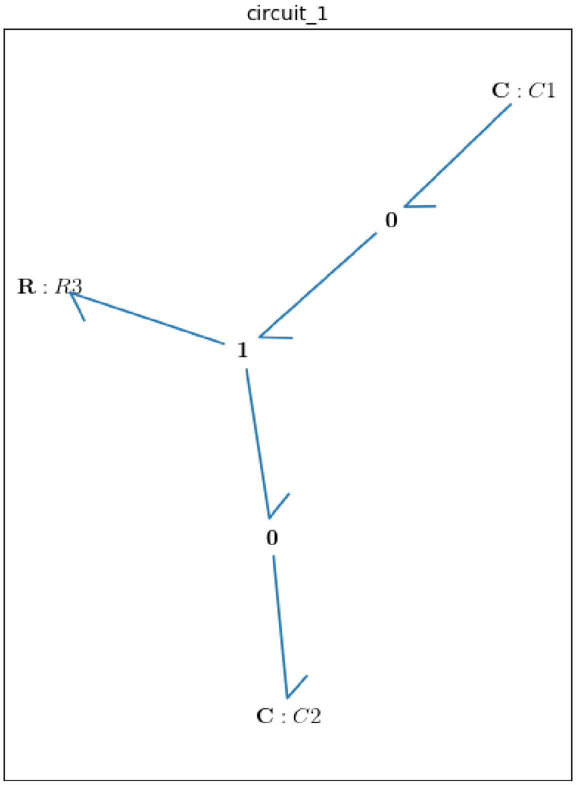


Fig. 1.89: Bondgraph diagram representing Electrical Circuit #1

```
model.constitutive_relations
Out[ ]: [dx_0 + x_0/10 - x_1/15, dx_1 - x_0/10 + x_1/15]
```

By using the command *“bgt.simulate”* and entering the model, time interval, and the initial conditions we prepare the requirements for plotting the system time behaviour:

```
t, x = bgt.simulate(model, timespan=timespan, x0=x0)
```

x vs t can be plotted by importing *“matplotlib.pyplot”* (q_{C1} & q_{C2} are state variables of the system which represent the amount of the electric charge accumulated in each capacitor):

```
import matplotlib.pyplot as plt
plt.plot(t,x[:,0], '-b', label='q_C1')
plt.plot(t,x[:,1], '-r', label='q_C2')
plt.xlabel("time (s)")
plt.ylabel("electric charge (Coulomb)")
plt.legend(loc='upper right')
plt.grid()
```

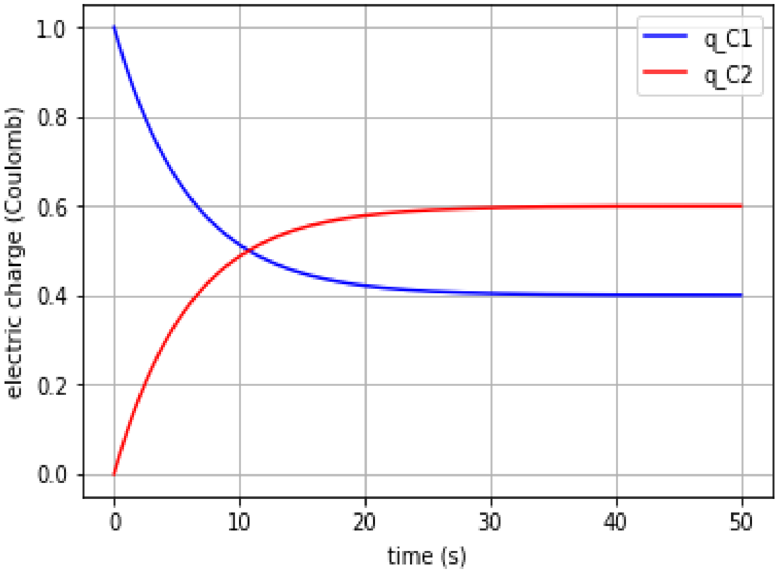


Fig. 1.90: Behaviour of the system in time (accumulated electric charge in each capacitor vs time)

Since the capacitor flow is the derivative of **q** with respect to **time**:

$$\frac{dq}{dt} = v$$

it can be plotted by converting the considered state variable (either x[:,0] or x[:,1]) to an array by importing the *numpy library* and then calculating its gradient with 0.1 steps:

```
# dq_C1/dt = v_C1 (flow in C1)
import numpy as np
f = np.array(x[:,0], dtype=float)
slope=np.gradient(f,0.1)
v_C1=slope
```

(continues on next page)

(continued from previous page)

```
# dq_C2/dt = v_C2 (flow in C2)
import numpy as np
f = np.array(x[:,1], dtype=float)
slope=np.gradient(f,0.1)
v_C2=slope
```

Plotting the flows in the two capacitors:

```
plt.plot(t,v_C1, '-b', label='V_C1')
plt.plot(t,v_C2, '-r', label='V_C2')
plt.xlabel("time (s)")
plt.ylabel("Flow (Coulomb/s)")
plt.legend(loc='upper right')
plt.grid()
```

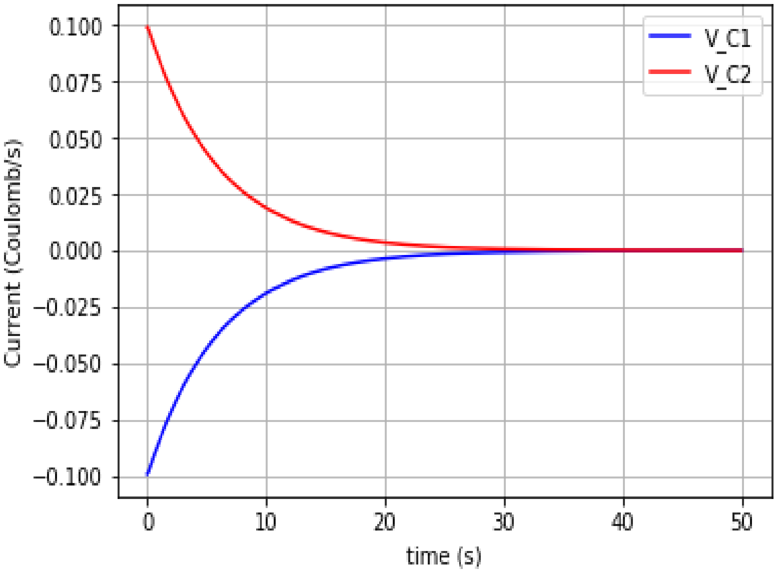


Fig. 1.91: Flow in C1 and C2 vs time

Moreover, the potential in each element can be calculated based on their constitutive equations:

Resistor: $u = R.v$

Capacitor: $u = q/C$

Thus:

```
u_R=R._params['r']*v_C2
u_C1=x[:,0]/C1._params['C']
u_C2=x[:,1]/C2._params['C']
```

The time variation of the corresponding potential for each component can be plotted *all-in-one* in a figure using the *for* command:

```
for u, c, label in [(u_C1,'-r','u_C1'), (u_C2,'-b','u_C2'), (u_R,'-g','u_R')]:
    fig=plt.plot(t,u,c,label=label)
    plt.legend(loc='upper right')
```

(continues on next page)

(continued from previous page)

```
plt.grid()
plt.xlabel("time (s)")
plt.ylabel("Potential (J/Coulomb)")
```

which results in:

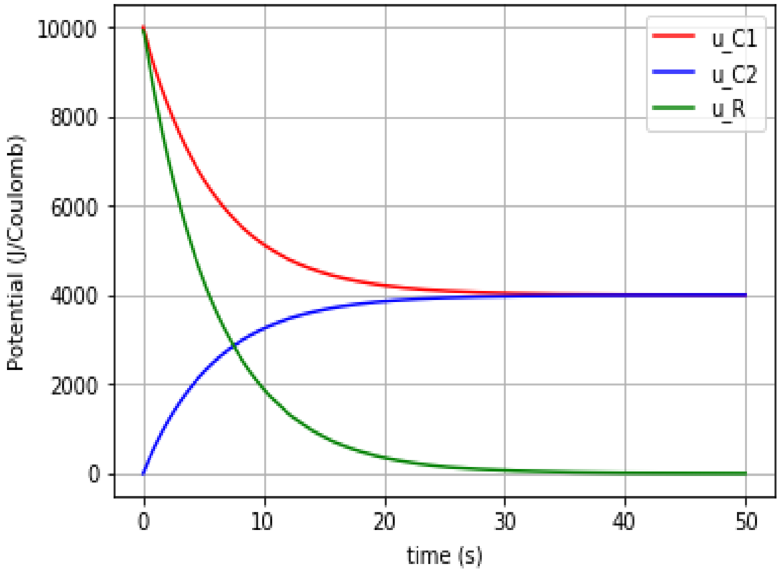


Fig. 1.92: Potential change in each component (R, C1, C2) vs time

Click to read Circuit #1 codes

Electrical Circuit #2

The circuits are adopted from[2]

Fig. 1.93: Schematic of Circuit #2 (R: resistor, C: capacitor, u: potential, v: flow)

The rationale behind the following set of commands are described in the Electrical Circuit #1 documentation.

```
import BondGraphTools as bgt
model=bgt.new(name='circuit_2')
# Parameters' values
C1_value=150e-6      #(150 uF)
C2_value=100e-6      #(100 uF)
C3_value=220e-6      #(220 uF)

R1_value=100e+3       #(100 k)
R2_value=10e+3        #(10 k)


C1=bgt.new("C", value=C1_value)
C2=bgt.new("C", value=C2_value)
```

(continues on next page)

(continued from previous page)

```
C3=bgt.new("C", value=C3_value)
R1=bgt.new("R", value=R1_value)
R2=bgt.new("R", value=R2_value)

zero_junc=bgt.new("0")
one_junc1=bgt.new("1")
one_junc2=bgt.new("1")

bgt.add(model,C1,C2,C3,R1,R2,zero_junc,one_junc1,one_junc2)

bgt.connect(C1,one_junc1)
bgt.connect(one_junc1,R1)
bgt.connect(one_junc1,zero_junc)
bgt.connect(zero_junc,C2)
bgt.connect(zero_junc,one_junc2)
bgt.connect(one_junc2,R2)
bgt.connect(one_junc2,C3)

bgt.draw(model)
```

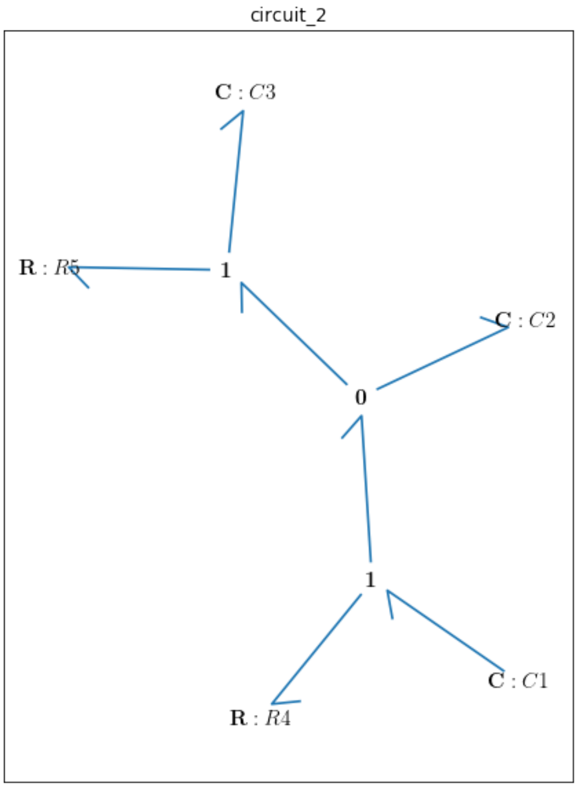


Fig. 1.94: Bondgraph diagram representing Electrical Circuit #2

Time interval and initial conditions for the state variables are defined as follows:

```
timespan=[0,50]
model.state_vars
Out[ ]:{'x_0': (C: C1, 'q_0'), 'x_1': (C: C2, 'q_0'), 'x_2': (C: C3, 'q_0')}
```

There are 3 state variables in this circuit: (q_{C1}, q_{C2}, q_{C3}) which are the electric charges corresponding to the 3 capacitors: {C1, C2, C3}

```
x0={"x_0":1, "x_1":0, "x_2":0}
```

The constitutive relations of the model are given as:

```
model.constitutive_relations
Out[ ]:
[dx_0 + x_0/15 - x_1/10,
 dx_1 - x_0/15 + 11*x_1/10 - 5*x_2/11,
 dx_2 - x_1 + 5*x_2/11]
```

Plotting the system time behaviour by entering the model, time interval, and the initial conditions:

```
t, x = bgt.simulate(model, timespan=timespan, x0=x0)
import matplotlib.pyplot as plt
plt.plot(t,x[:,0], '-b', label='q_C1')
plt.plot(t,x[:,1], '-r', label='q_C2')
plt.plot(t,x[:,2], '-g', label='q_C3')
plt.xlabel("time (s)")
plt.ylabel("electric charge (Coulomb)")
plt.legend(loc='upper right')
plt.grid()
```

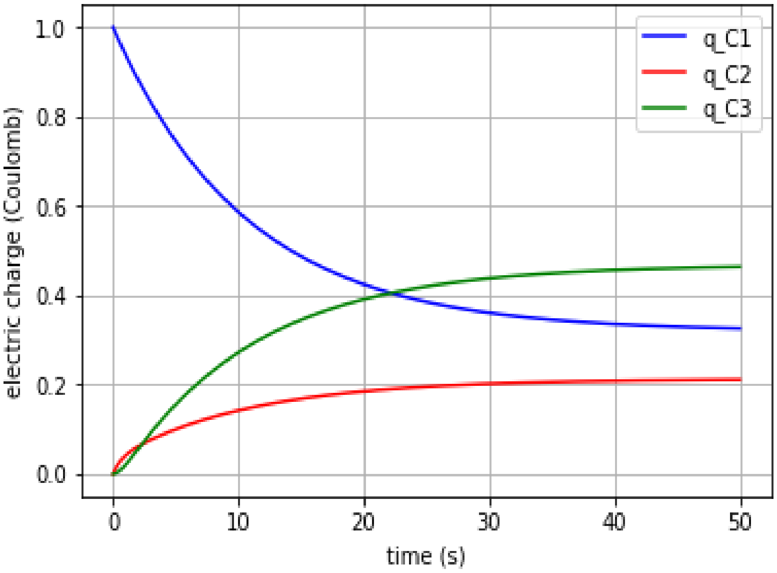


Fig. 1.95: Time behaviour of the system (accumulated electric charge in each capacitor vs time)

Since the capacitor flow is the derivative of **q** with respect to **time**:

$$\frac{dq}{dt} = v$$

it can be plotted by converting the considered state variable (either x[:,0], x[:,1] or x[:,2]) to an array by importing the *numpy library* and then calculating its gradient with 0.1 steps:

```
# - dq_C1/dt = v_C1 (flow)
import numpy as np
f = np.array(x[:,0], dtype=float)
slope=np.gradient(f,0.1)
v_C1=-slope
# dq_C2/dt = v_C2 (flow)
f = np.array(x[:,1], dtype=float)
slope=np.gradient(f,0.1)
v_C2=slope
# dq_C3/dt = v_C3 (flow)
f = np.array(x[:,2], dtype=float)
slope=np.gradient(f,0.1)
v_C3=slope
```

Plotting the flows in the three capacitors:

```
plt.plot(t,v_C1, '-b', label='v_C1')
plt.plot(t,v_C2, '-r', label='v_C2')
plt.plot(t,v_C3, '-g', label='v_C3')
plt.xlabel("time (s)")
plt.ylabel("flow (Coulomb/s)")
plt.legend(loc='upper right')
plt.grid()
```

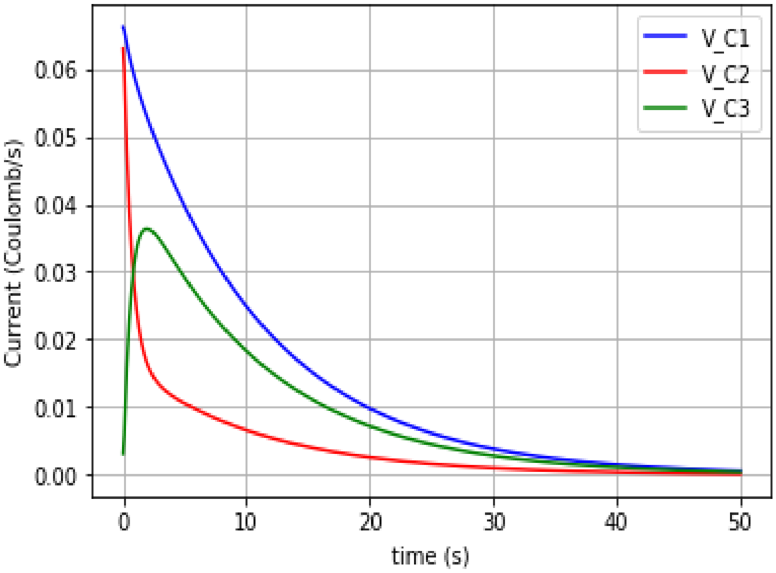


Fig. 1.96: Flows in C1, C2 & C3 vs time

Moreover, the potential in each element can be calculated based on their constitutive equations:

Resistor: $u = R.v$

Capacitor: $u = q/C$

Thus:

```
u_R1=R1._params['r']*v_C1
u_R2=R2._params['r']*v_C3
```

(continues on next page)

(continued from previous page)

```
u_C1=x[:,0]/C1._params['C']
u_C2=x[:,1]/C2._params['C']
u_C3=x[:,2]/C3._params['C']
```

The time variation of the corresponding potential for each capacitor is plotted in a figure using the *for* command:

```
for u, c, label in [(u_C1,'-b','u_C1'), (u_C2,'-r','u_C2'), (u_C3,'-g','u_C3')]:
    fig=plt.plot(t,u,c,label=label)
    plt.legend(loc='upper right')

plt.grid()
plt.xlabel("time (s)")
plt.ylabel("Potential (J/Coulomb)")
```

which results in:

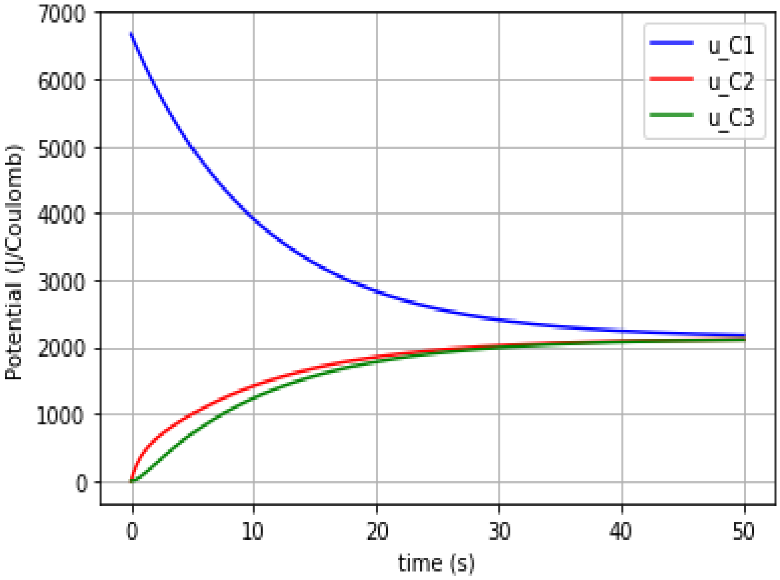


Fig. 1.97: Potential change in each capacitor (C1, C2, C3) vs time

Click to read Circuit #2 codes

Electrical Circuit #3

The circuits are adopted from[2] .

Fig. 1.98: Schematic of Circuit #3 (R: resistor, C: capacitor, u: potential, v: flow)

The rationale behind the following set of commands are described in the Electrical Circuit #1 documents.

```
import BondGraphTools as bgt
model=bgt.new(name='circuit_3')
```

(continues on next page)

(continued from previous page)

```
# Parameters' values
C1_value=1000e-6    #(1000 uF)
C2_value=470e-6     #(470 uF)
L1_value=100e-6     #(100 uH)
R1_value=10e3       #(10 k)
R2_value=10e3       #(10 k)
R3_value=1e3        #(1 k)
R4_value=1e3        #(1 k)

C1=bgt.new("C", value=C1_value)
C2=bgt.new("C", value=C2_value)
L1=bgt.new("I", value=L1_value)
R1=bgt.new("R", value=R1_value)
R2=bgt.new("R", value=R2_value)
R3=bgt.new("R", value=R3_value)
R4=bgt.new("R", value=R4_value)

zero_junc=bgt.new("0")
one_junc1=bgt.new("1")
one_junc2=bgt.new("1")

bgt.add(model,C1,C2,L1,R1,R2,R3,R4,zero_junc,one_junc1,one_junc2)

bgt.connect(C1,one_junc1)
bgt.connect(one_junc1,R1)
bgt.connect(one_junc1,R4)
bgt.connect(one_junc1,zero_junc)
bgt.connect(zero_junc,C2)
bgt.connect(zero_junc,one_junc2)
bgt.connect(one_junc2,R2)
bgt.connect(one_junc2,R3)
bgt.connect(one_junc2,L1)

bgt.draw(model)
```

Time interval and initial conditions for the state variables are defined as follows:

```
timespan=[0,100]
model.state_vars
Out[ ]:{'x_0': (C: C1, 'q_0'), 'x_1': (C: C2, 'q_0'), 'x_2': (I: I3, 'p_0')}
```

There are 3 state variables in this circuit: q_{C1}, q_{C2} corresponding to the 2 capacitors: {C1, C2} and p_{L1} corresponding to the only inductor {L1}. Here the initial conditions for the 3 state variables and the constitutive relations of the model are given as:

```
x0={"x_0":1, "x_1":0, "x_2":0}
model.constitutive_relations
Out[ ]:
[dx_0 + x_0/11 - 193423597678917*x_1/1000000000000000,
 dx_1 - x_0/11 + 193423597678917*x_1/1000000000000000 + 10000*x_2,
 dx_2 - 212765957446809*x_1/100000000000 + 110000000*x_2]
```

Plotting the system time behaviour by entering the model, time interval, and the initial conditions:

```
t, x = bgt.simulate(model, timespan=timespan, x0=x0)
import matplotlib.pyplot as plt
```

(continues on next page)

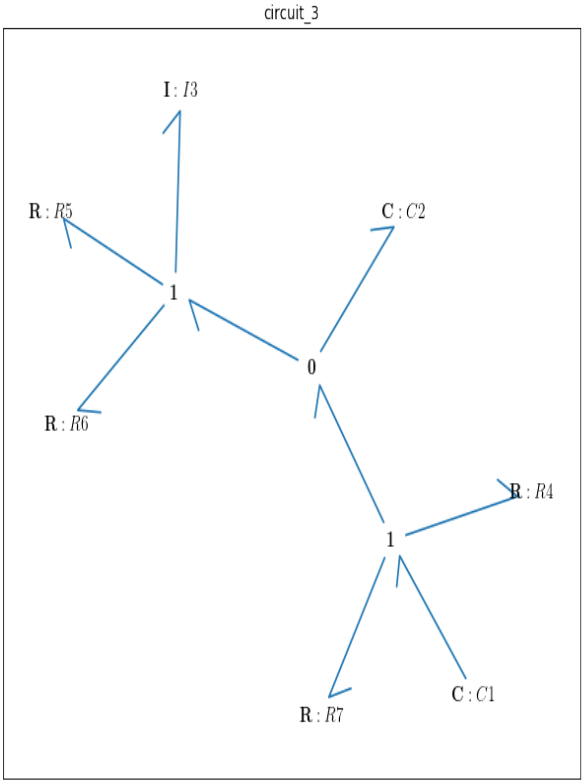


Fig. 1.99: Bondgraph diagram representing Electrical Circuit #3

(continued from previous page)

```
plt.plot(t,x[:,0], '-b', label='q_C1')
plt.plot(t,x[:,1], '-r', label='q_C2')
plt.xlabel("time (s)")
plt.ylabel("electric charge (Coulomb)")
plt.legend(loc='upper right')
plt.grid()
```

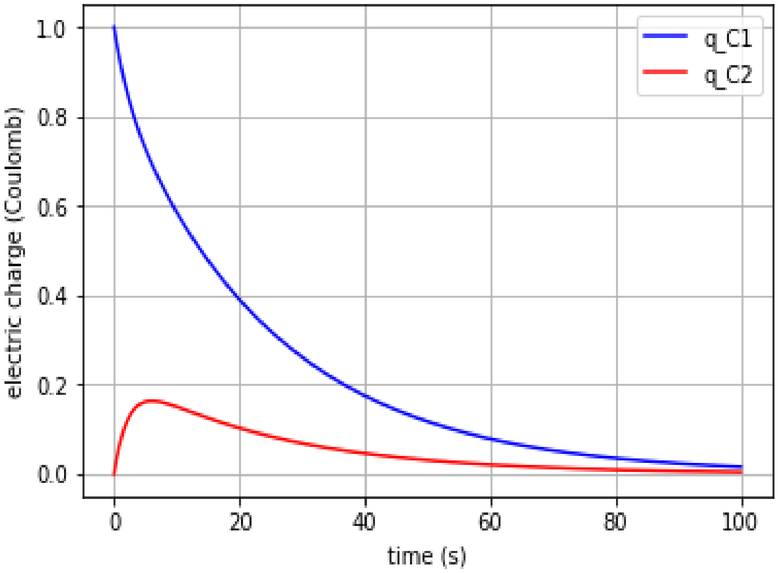


Fig. 1.100: Time behaviour of the system (accumulated electric charge in each capacitor vs time)

Plotting the flow in the inductor:

```
v_L1=x[:,2]/L1._params['L']
plt.plot(t,v_L1, '-g', label='V_L1')
plt.xlabel("time (s)")
plt.ylabel("flow (Coulomb/s)")
plt.legend(loc='upper right')
plt.grid()
```

Since the capacitor flow is the time derivative of **q** and the derivative of the inductor flow is the fraction of u to L :

$$\frac{dq}{dt} = v$$
$$\frac{dv}{dt} = \frac{u}{L}$$

the capacitors flows (v_{C1} & v_{C2}) can be plotted by converting the considered state variable (either x[:,0] or x[:,1]) to an array by importing the *numpy library* and then calculating its gradient with 0.1 steps. Note that the inductor flow can also be gained by deducting v_{C2} from v_{C1} :

```
# dq_C1/dt = v_C1 (flow in C1)
import numpy as np
f = np.array(x[:,0], dtype=float)
slope=np.gradient(f,0.1)
v_C1=-slope
```

(continues on next page)

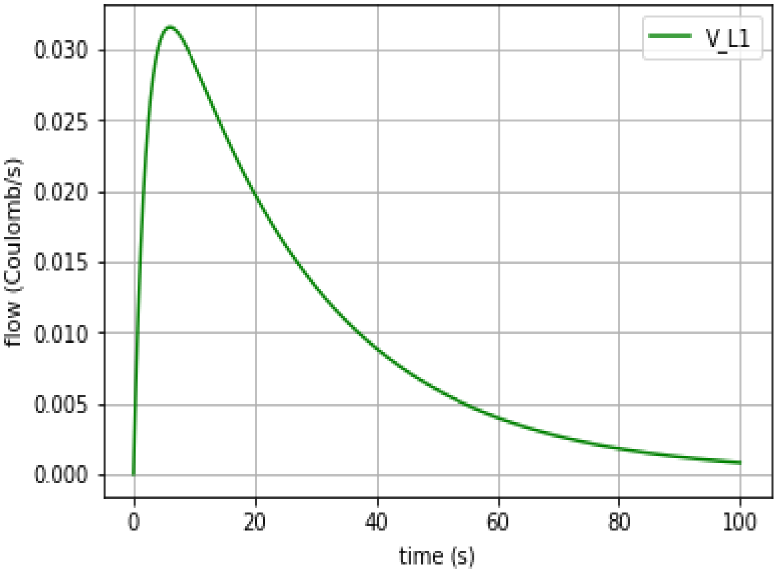


Fig. 1.101: Time behaviour of the system (flow of the inductor L1 vs time)

(continued from previous page)

```
# dq_C2/dt = v_C2 (flow in C2)
f = np.array(x[:,1], dtype=float)
slope=np.gradient(f,0.1)
v_C2=slope

# dV_L1/dt = a_L1
# v_L1=v_C1-v_C2
```

The time derivative of the inductor flow is:

$$a = dv/dt$$

which can be calculated by:

```
a_L1=np.gradient(v_L1,0.1)
```

The 3 flows in the 3 branches of the circuit are plotted:

```
plt.plot(t,v_C1, '-b', label='V_C1')
plt.plot(t,v_C2, '-r', label='V_C2')
plt.plot(t,v_L1, '-g', label='V_L1')
plt.xlabel("time (s)")
plt.ylabel("Flow (Coulomb/s)")
plt.legend(loc='upper right')
plt.grid()
```

Furthermore, the potential in each element can be calculated based on its constitutive equation:

Resistor: $u = R.v$

Capacitor: $u = q/C$

Inductor: $u = L.dv/dt$

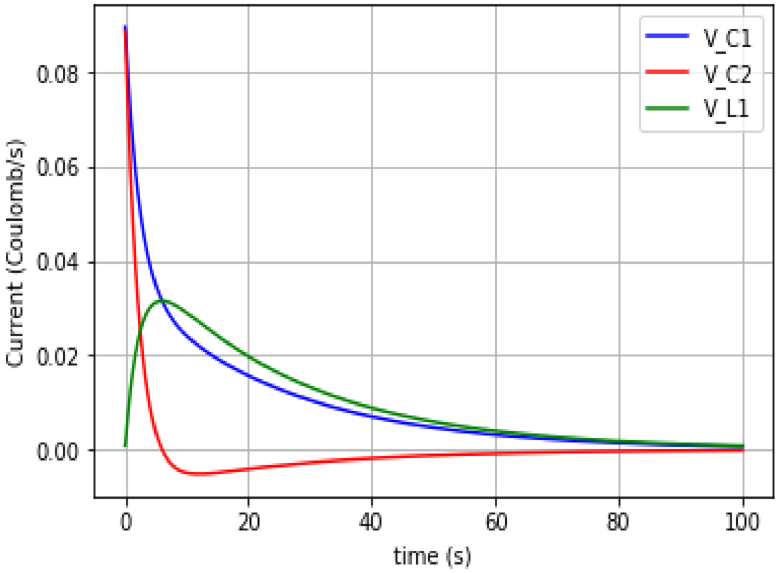


Fig. 1.102: Flows in C1, C2 & L1 (v_{C1}, v_{C2}, v_{L1} vs time)

Thus:

```
u_C1=x[:,0]/C1._params['C']
u_C2=x[:,1]/C2._params['C']

u_L1=L1._params['L']*a_L1

u_R1=R1._params['r']*v_C1
u_R2=R2._params['r']*v_L1
u_R3=R3._params['r']*v_L1
u_R4=R4._params['r']*v_C1
```

Then the potentials of the three elements (u_{C1}, u_{C2} & u_{L1}) are plotted:

```
for u, c, label in [(u_C1,'-b','u_C1'), (u_C2,'-r','u_C2'), (u_L1,'-g','u_L1')]:
    fig=plt.plot(t,u,c,label=label)
    plt.legend(loc='upper right')

plt.grid()
plt.xlabel("time (s)")
plt.ylabel("Potential (J/Coulomb)")
```

which results in:

Due to the scale difference in potential levels, the potential of the inductor (u_{L1}) is also plotted separately:

```
fig=plt.plot(t,u_L1,'-g', label='u_L1')
plt.grid()
plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("Potential (J/Coulomb)")
```

Click to read Circuit #3 codes

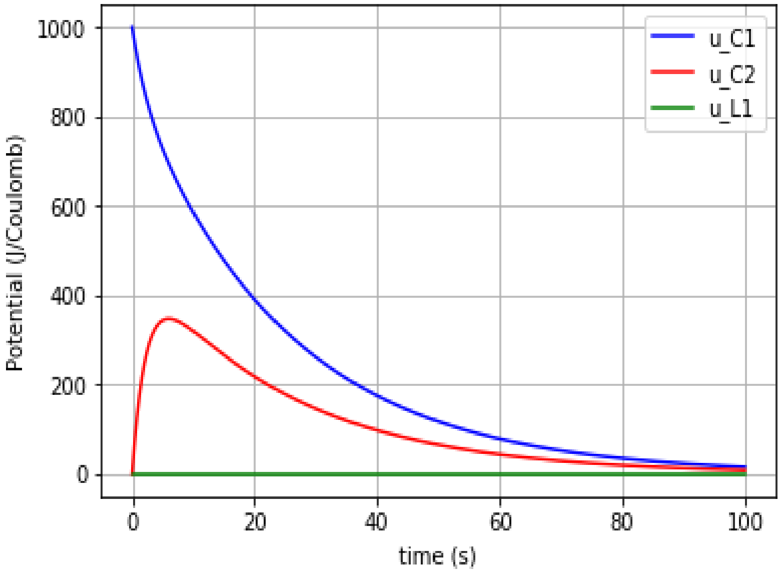


Fig. 1.103: Potential change in the capacitors & inductor (C1, C2 & L1) vs time

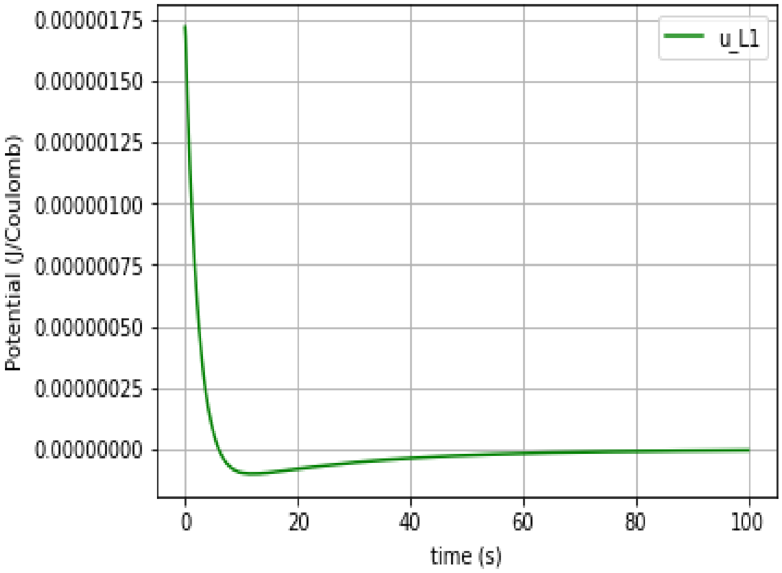


Fig. 1.104: Potential change in the inductor L1 vs time

Fluid Mechanics

Straight tube

The figures (1) & (2) are adopted from[2] and[3], respectively. The sample values of the parameters are also taken from the latter for the simulation to be more close to reality.

Fig. 1.105: Schematic of Straight vessel (u: potential)

Fig. 1.106: Bondgraph diagram representing Straight Vessel (R: viscous resistance, C: vessel wall compliance, I: mass inertial effect, u: potential, v: flow)

In the first step, the BondgraphTools library must be imported:

```
import BondGraphTools as bgt
```

To create a new model using BondGraphTools (bgt), the command *bgt.new* is used:

```
model=bgt.new(name='straight tube')
```

* Note that since we are working with the fluid mechanics components, the measures are different from electrical circuits but still the bond graph elements are the same.

In the next step, all the parameters' values are defined and directly assigned to their corresponding bgt components using the following commands:

```
Se1=bgt.new("Se",value=11.997e6)      #(J/m6)
Se2=bgt.new("Se",value=10.664e6)      #(J/m6)

C=bgt.new("C", value=0.60015e-6)      #(m6/J)

# The amounts R-elements are assumed to be equal in a straight tube
R1=bgt.new("R", value=10.664e-6)      #(J.s/m6)
R2=bgt.new("R", value=10.664e-6)      #(J.s/m6)

# The amounts of the I-elements are assumed to be equal in a straight tube
L1=bgt.new("I", value=0.06665e6)      #(J.s2/m6)
L2=bgt.new("I", value=0.06665e6)      #(J.s2/m6)
```

* Note that to create a pressure difference in the vessel we need to insert two potential sources (Se1, Se2).

Also, a number of *"0-junctions & 1-junctions"* must be defined (based on our model) as follows:

```
zero_junc=bgt.new("0")
one_junc_1=bgt.new("1")
one_junc_2=bgt.new("1")
```

Now creating the model and its components has finished and all of them must be assembled using the *bgt.add* command:

```
bgt.add(model,Se1,Se2,C,R1,R2,L1,L2,zero_junc,one_junc_1,one_junc_2)
```

[3] Safaei S, Blanco PJ, Muller LO, Hellevik LR and Hunter PJ (2018) Bond Graph Model of Cerebral Circulation: Toward Clinically Feasible Systemic Blood Flow Simulations. Front. Physiol. 9:148. doi: 10.3389/fphys.2018.00148 Click for the article

According to our bond graph model, these components must be connected to the related junctions by *bgt.connect*. Note that the first element in parenthesis represents the *“tail”* of the arrow and the second element represents the *“head”*:

```
bgt.connect(Se1,one_junc_1)
bgt.connect(one_junc_1,R1)
bgt.connect(one_junc_1,L1)
bgt.connect(one_junc_1,zero_junc)
bgt.connect(zero_junc,one_junc_2)
bgt.connect(zero_junc,C)
bgt.connect(one_junc_2,R2)
bgt.connect(one_junc_2,L2)
bgt.connect(Se2,one_junc_2)
```

By drawing the model, one can see if the components are connected properly to each other or not:

```
bgt.draw(model)
```

A sketch of the network will then be produced:

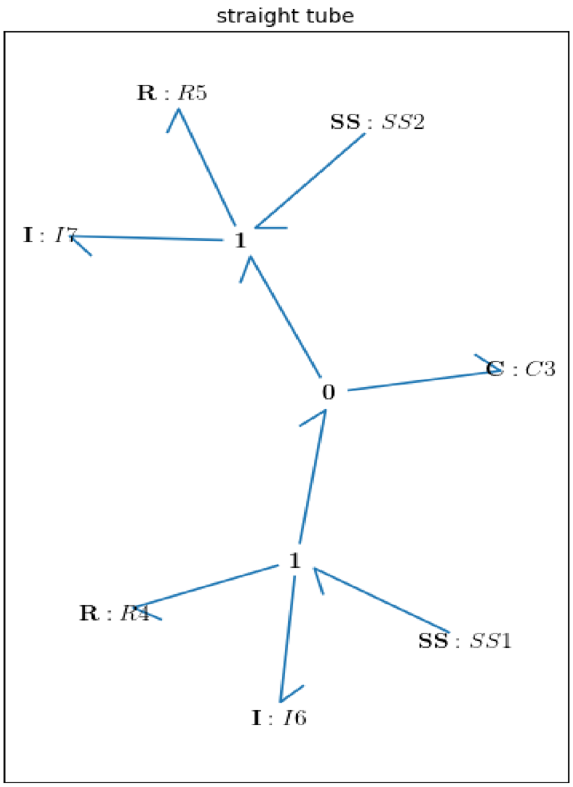


Fig. 1.107: Straight tube bond graph topology

Now that the bond graph demonstration of the system is done, we can illustrate its behaviour during a specific time interval and with arbitrary initial conditions for the state variables. The constitutive relations of the model can be shown as well:

```
timespan=[0,5]
model.state_vars
```

(continues on next page)

(continued from previous page)

```
Out[ ]:
{'x_0': (C: C3, 'q_0'), 'x_1': (I: I6, 'p_0'), 'x_2': (I: I7, 'p_0')}
```

Initial conditions:

```
x0={"x_0":5*1e-6, "x_1":0, "x_2":0}
```

Constitutive relations:

```
model.constitutive_relations
Out[ ]:
[dx_0 - 2344336084021*x_1/156250000000000000 + 2344336084021*x_2/156250000000000000,
 dx_1 + 166625010414063*x_0/100000000 + x_1/6250000000 - 11997000,
 dx_2 - 166625010414063*x_0/100000000 + x_2/6250000000 - 10664000]
```

By using the command *"bgt.simulate"* and entering the model, time interval, and the initial conditions we simulate the system over the given time period:

```
t, x = bgt.simulate(model, timespan=timespan, x0=x0)
```

x vs t can be plotted by importing *"matplotlib.pyplot"* (q_C, p_{L1} & p_{L2} are state variables of the system which represent the amount of volume accumulated in the C-element and the momentum in 2 identical I-elements, respectively). Here, the flow of the I-element is plotted first which is the fraction of momentum (x[:,1]) to L1 value:

I-element: $v = p/L$

Plotting the flow in the I-element:

```
v_L1=x[:,1]/L1._params['L']
import matplotlib.pyplot as plt

plt.plot(t,v_L1, '-r', label='v_L1 & v_L2')
plt.xlabel("time (s)")
plt.ylabel("flow (m3/s)")
plt.legend(loc='upper right')
plt.grid()
```

It can be anticipated that the scale of the stored volume (q_C) is smaller than of the momentum in the I-elements. Hence, the q_C is plotted separately:

```
plt.plot(t,x[:,0], '-b', label='q_C')
plt.xlabel("time (s)")
plt.ylabel("volume (m3)")  #metre3
plt.legend(loc='upper right')
plt.grid()
```

Since the flow in the C-element is the time derivative of **q** and the derivative of the I-element flow is the fraction of u to L,

$$\frac{dq}{dt} = v$$
$$\frac{dv}{dt} = \frac{u}{L}$$

by importing *numpy* and converting the first state variable q_C to an array and taking the gradient of it with 0.1 steps, one can obtain the flow passed through the C-element. The flows corresponding to the I-elements are merely the second and third state variables (x[:,1] & x[:,2]):

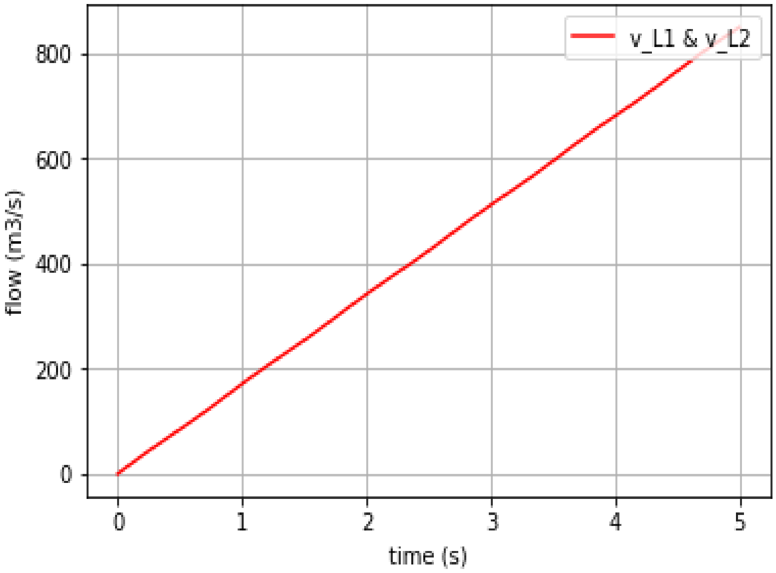


Fig. 1.108: Flow of the I-elements vs time

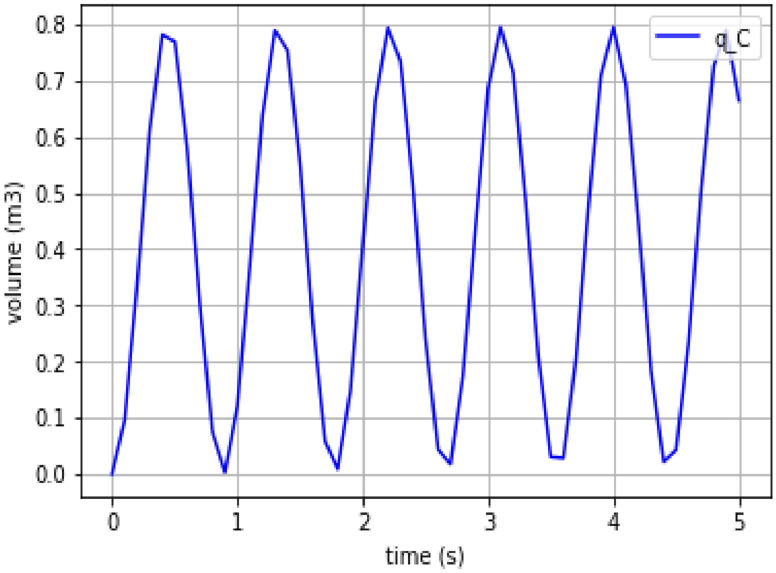


Fig. 1.109: Accumulated volume in the C-elements vs time

```
#  dq_C/dt = v_C (flow in the C-element)

import numpy as np
f = np.array(x[:,0], dtype=float)
slope=np.gradient(f,0.1)
v_C=slope
```

Plotting the flow of the C-element (v_C):

```
import matplotlib.pyplot as plt
plt.plot(t,v_C, '-g', label='v_C')
plt.xlabel("time (s)")
plt.ylabel("flow (m3/s)")
plt.legend(loc='upper right')
plt.grid()
```

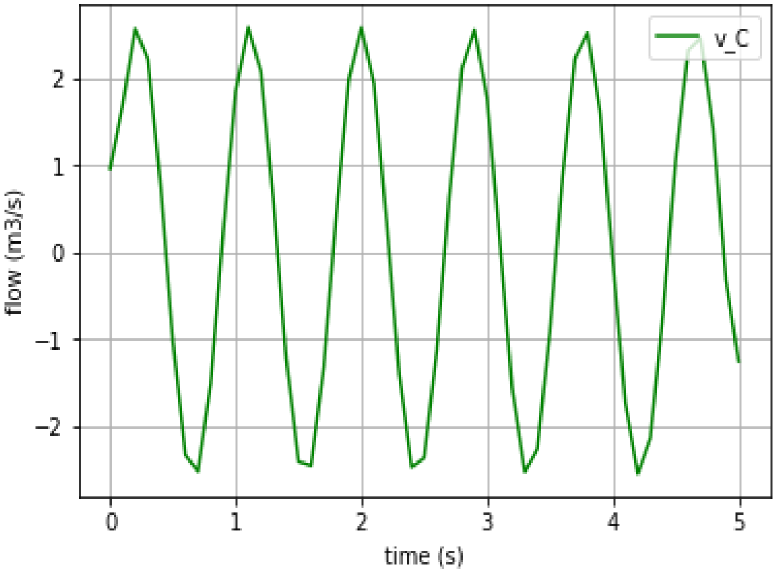


Fig. 1.110: Flow in the C-element vs time

In order to calculate the potential of the I-elements, we need to take the time derivative of their flows and multiply each by the mass inertial value (L). Also, to calculate the potential of the C-element we just need to multiply the compliance value (C) by the stored volume (q_C):

```
# u_L1=L1*a_L1 (potential of the identical I-elements)==> u_L1=u_L2
f = np.array(v_L1, dtype=float)
dv_L1=np.gradient(f,0.1)
a_L1=dv_L1
u_L1=L1._params['L']*a_L1

# u_C=C*v_C (potential of the C-element)
u_C=C._params['C']*(x[:,0])
```

To plot the potential of L1:

```
fig=plt.plot(t,u_L1,'-m', label='u_L1 & u_L2')
plt.grid()
```

(continues on next page)

(continued from previous page)

```
plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("Potential (J/m3)")
```

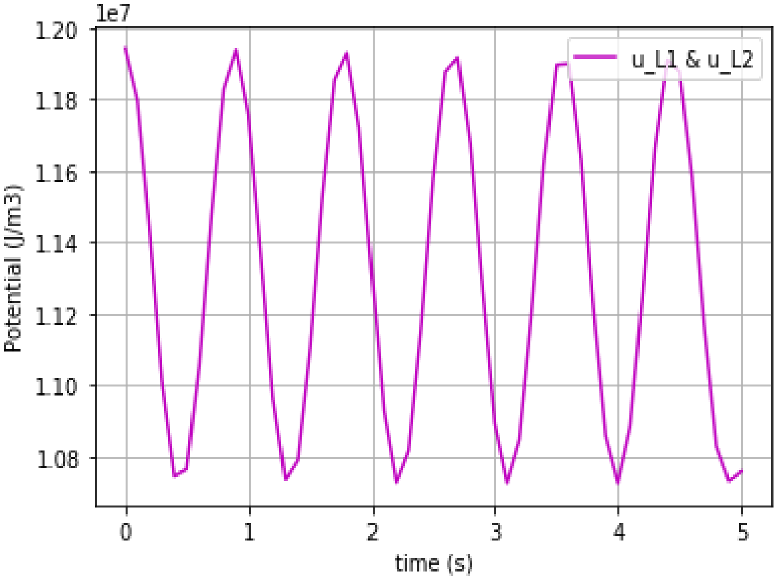


Fig. 1.111: Potential in the I-element vs time

To plot the potential of C-element:

```
fig=plt.plot(t,u_C,'-y', label='u_C')
plt.grid()
plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("Potential (J/m3)")
```

Click to read Straight tube codes

Branching vessel

The figures (1) & (2) are adopted from[2] and[3], respectively. The sample values of the parameters are also taken from the latter for the simulation to be more close to reality.

The following set of commands are described in the Straight tube documents.

```
import BondGraphTools as bgt
model=bgt.new(name='branching vessel')

Se=bgt.new("Se",value=9.331e6)
Sf1=bgt.new("Sf",value=7.998e6)
Sf2=bgt.new("Sf",value=7.998e6)

C=bgt.new("C", value=0.60015e-6)
C1=bgt.new("C", value=0.125281e-6)
C2=bgt.new("C", value=0.1125281e-6)
```

(continues on next page)

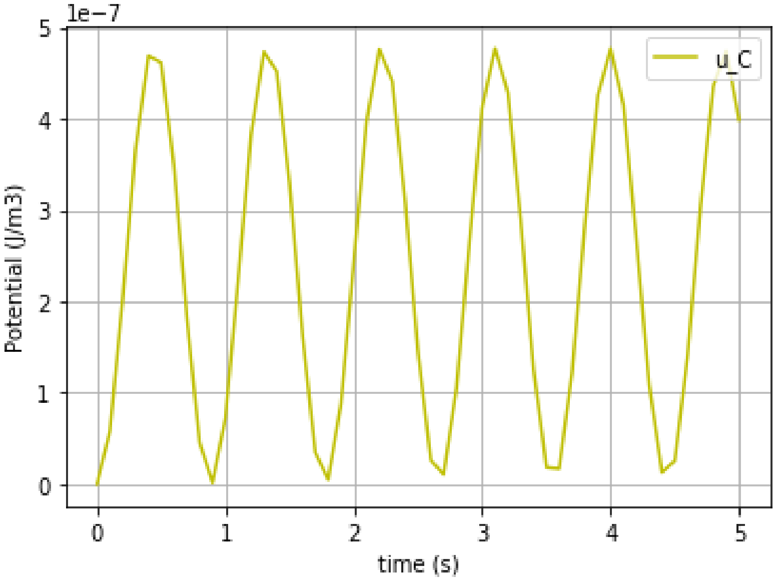


Fig. 1.112: Potential in the C-element vs time

Fig. 1.113: Schematic of Branching Vessel (u: potential, v: flow)

Fig. 1.114: Bondgraph diagram representing Branching Vessel (R: viscous resistance, C: vessel wall compliance, I: mass inertial effect, u: potential, v: flow)

(continued from previous page)

```

R=bgt.new("R", value=1.333e6)
R1=bgt.new("R", value=10.564e6)
R2=bgt.new("R", value=10.664e6)

L=bgt.new("I", value=0.123e6)
L1=bgt.new("I", value=0.08665e6)
L2=bgt.new("I", value=0.06665e6)
```

* Note that in order to represent the pressure and volume difference in the vessel, one must use the potential & flow sources (Se & Sf). These are illustrated in the latter figure by u_{in} and v_{out}.

“0-junctions & 1-junctions” :

```
zero_junc_1=bgt.new("0")
zero_junc_2=bgt.new("0")
zero_junc_3=bgt.new("0")

one_junc_1=bgt.new("1")
one_junc_2=bgt.new("1")
one_junc_3=bgt.new("1")
```

Assembling the model components:

```
bgt.add(model,Se,Sf1,Sf2,C,C1,C2,R,R1,R2,L,L1,L2,zero_junc_1,zero_junc_2,zero_junc_3,
↪one_junc_1,one_junc_2,one_junc_3)
```

Connecting the junctions and the components:

```
bgt.connect(Se,one_junc_1)
bgt.connect(one_junc_1,R)
bgt.connect(one_junc_1,L)
bgt.connect(one_junc_1,zero_junc_1)
bgt.connect(zero_junc_1,C)
bgt.connect(zero_junc_1,one_junc_2)
bgt.connect(zero_junc_1,one_junc_3)
bgt.connect(one_junc_2,R1)
bgt.connect(one_junc_2,L1)
bgt.connect(one_junc_2,zero_junc_2)
bgt.connect(zero_junc_2,C1)
bgt.connect(zero_junc_2,Sf1)
bgt.connect(one_junc_3,L2)
bgt.connect(one_junc_3,R2)
bgt.connect(one_junc_3,zero_junc_3)
bgt.connect(zero_junc_3,C2)
bgt.connect(zero_junc_3,Sf2)
```

Drawing the bong graph representation of the model:

```
bgt.draw(model)
```

Defining the time span:

```
timespan=[0,12.5]
```

Depicting the state variables of the model (6 state variables):

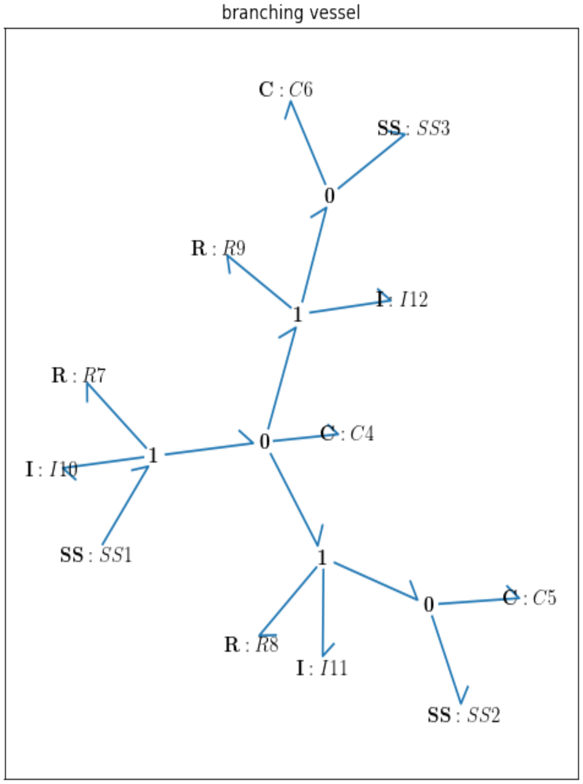


Fig. 1.115: Bondgraph diagram for Branching Vessel

```
model.state_vars
Out[ ]:
{'x_0': (C: C4, 'q_0'),
 'x_1': (C: C5, 'q_0'),
 'x_2': (C: C6, 'q_0'),
 'x_3': (I: I10, 'p_0'),
 'x_4': (I: I11, 'p_0'),
 'x_5': (I: I12, 'p_0')}
```

Setting the initial conditions of the state variables:

```
x0={"x_0":10*1e-6, "x_1":4*1e-6, "x_2":4*1e-6, "x_3":0, "x_4":0, "x_5":0}
```

Constitutive relations of the model:

```
model.constitutive_relations
Out[ ]:
{dx_0 - 813008130081301*x_3/100000000000000000000 + 115406809001731*x_4/
↪10000000000000000000 + 2344336084021*x_5/156250000000000000,
 dx_1 - 115406809001731*x_4/10000000000000000000 - 7998000,
 dx_2 - 2344336084021*x_5/156250000000000000 - 7998000,
 dx_3 + 166625010414063*x_0/100000000 + 108373983739837*x_3/10000000000000 - 9331000,
 dx_4 - 166625010414063*x_0/100000000 + 798205633735363*x_1/100000000 +␣
↪121915753029429*x_4/1000000000000,
 dx_5 - 166625010414063*x_0/100000000 + 888666919640517*x_2/100000000 + 160*x_5}
```

Simulating th model over the given time period and with the initial conditions:

```
t, x = bgt.simulate(model, timespan=timespan, x0=x0)
```

Plotting the first 3 state variables vs time (q_C, q_{C1} & q_{C2} which are the stored volume in the three C-elements):

```
# plotting state variables (q_C, q_C1 & q_C2) in 3 C-elements (C, C1 & C2)
import matplotlib.pyplot as plt
for q, c, label in [(x[:,0],'r', 'q_C'), (x[:,1],'b', 'q_C1'), (x[:,2],'g', 'q_C2')]:
    fig=plt.plot(t,q,c, label=label)
    plt.xlabel("time (s)")
    plt.ylabel("volume (m3)") #metre3
    plt.legend(loc='upper right')
    plt.grid()
```

In the same way the momentum in the 3 I-elements (p_L, p_{L1} & p_{L2}) are shown:

```
# plotting state variables (p_L, p_L1 & p_L2) in 3 I-elements (L, L1 & L2)
import matplotlib.pyplot as plt
for l, c, label in [(x[:,3],'r', 'p_L'), (x[:,4],'b', 'p_L1'), (x[:,5],'g', 'p_L2')]:
    fig=plt.plot(t,l,c, label=label)
    plt.xlabel("time (s)")
    plt.ylabel("momentum (J.s/m3)")
    plt.legend(loc='upper right')
    plt.grid()
```

Since the flow in the C-element is the time derivative of **q** and the time derivative of the I-element flow is the fraction

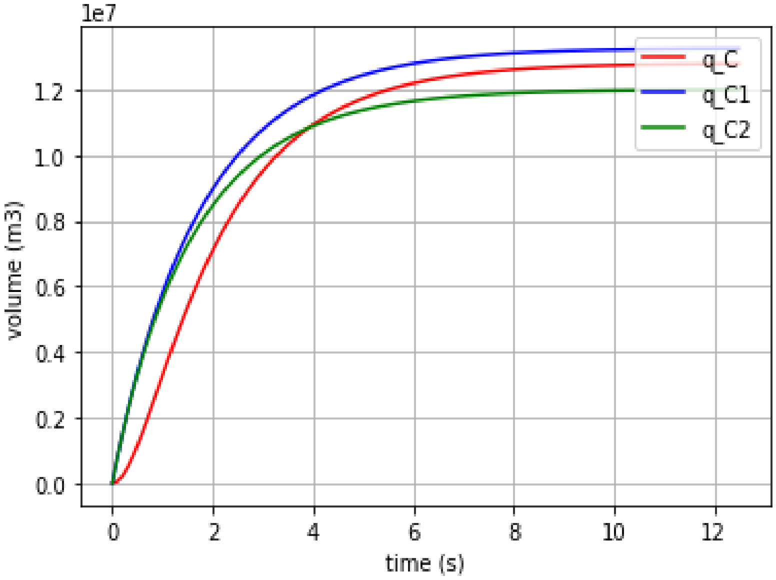


Fig. 1.116: Accumulated volume in the C-elements vs time

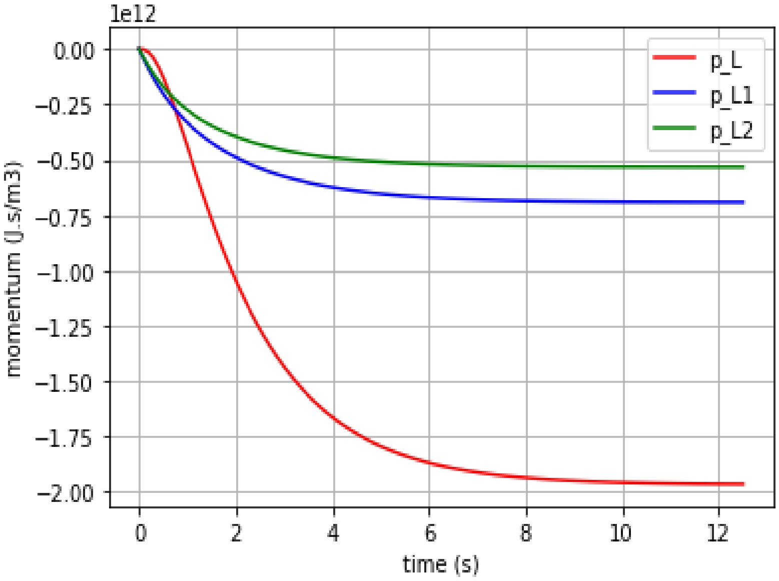


Fig. 1.117: Momentum in the I-elements vs time

of u to L,

$$\frac{dq}{dt} = v$$
$$\frac{dv}{dt} = \frac{u}{L}$$

by importing *numpy* and converting the first three state variables q_C, q_{C1} & q_{C2} to arrays and taking the gradient of them with 0.1 steps, one can obtain the flow passed through the C-elements. The flows corresponding to the I-elements are merely the fraction of the second 3 state variables (x[:,3], x[:,4] & x[:,5]) to their L values.

Calculating the flow & potential in C:

```
import numpy as np
f = np.array(x[:,0], dtype=float)
slope=np.gradient(f,0.1)
v_C=slope

u_C=(1/C._params['C'])*x[:,0]]
```

Calculating the flow & potential in C1:

```
import numpy as np
f = np.array(x[:,1], dtype=float)
slope=np.gradient(f,0.1)
v_C1=slope

u_C1=(1/C1._params['C'])*x[:,1]
```

Calculating the flow & potential in C2:

```
import numpy as np
f = np.array(x[:,2], dtype=float)
slope=np.gradient(f,0.1)
v_C2=slope

u_C2=(1/C2._params['C'])*x[:,2]
```

Calculating the flow & potential in L:

```
v_L=x[:,3]/L._params['L']
import numpy as np
f = np.array(v_L, dtype=float)
slope=np.gradient(f,0.1)
dv_L=slope
u_L=L._params['L']*dv_L
```

Calculating the flow & potential in L1:

```
v_L1=x[:,4]/L1._params['L']
import numpy as np
f = np.array(v_L1, dtype=float)
slope=np.gradient(f,0.1)
dv_L1=slope
u_L1=L1._params['L']*dv_L1
```

Calculating the flow & potential in L2:

```
v_L2=x[:,5]/L2._params['L']
import numpy as np
f = np.array(v_L2, dtype=float)
slope=np.gradient(f,0.1)
dv_L2=slope
u_L2=L2._params['L']*dv_L2
```

Now by using the *for* command the potentials of the 3 C-elements can be plotted in one figure for comparison:

```
for u, c, label in [(u_C,'r', 'u_C'), (u_C1,'b', 'u_C1'), (u_C2,'g', 'u_C2')]:
    fig=plt.plot(t,u,c,label=label)
    plt.xlabel("time (s)")
    plt.ylabel("potential (J/m3)")
    plt.legend(loc='upper right')
    plt.grid()
```

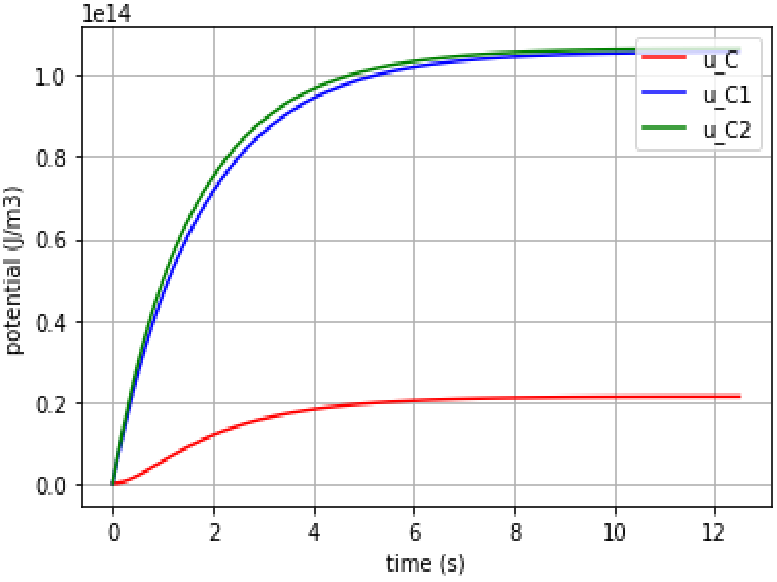


Fig. 1.118: Potential in the C-elements vs time

Plotting the potentials in the three I-elements:

```
for u, c, label in [(u_L,'r','u_L'), (u_L1,'b','u_L1'), (u_L2,'g','u_L2')]:
    fig=plt.plot(t,u,c,label=label)
    plt.xlabel("time (s)")
    plt.ylabel("potential (J/m3)")
    plt.legend(loc='upper right')
    plt.grid()
```

Plotting the flows in the three C-elements:

```
for v, c, label in [(v_C,'r','v_C'), (v_C1,'b','v_C1'), (v_C2,'g','v_C2')]:
    fig=plt.plot(t,v,c, label=label)
    plt.xlabel("time (s)")
    plt.ylabel("flow (m3/s)")
    plt.legend(loc='upper right')
    plt.grid()
```

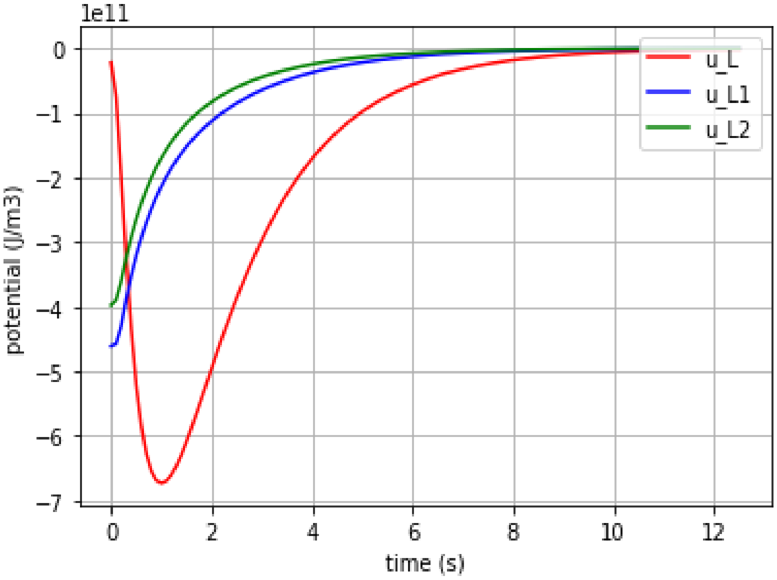


Fig. 1.119: Potential in the I-elements vs time

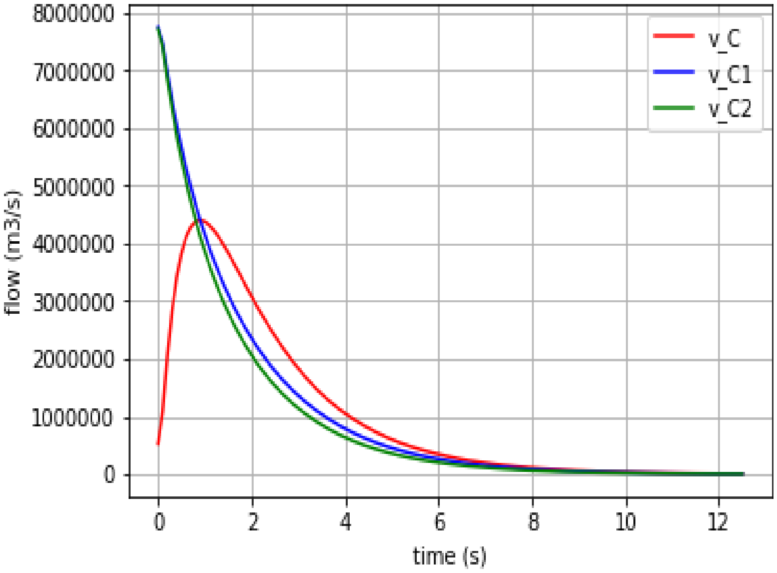


Fig. 1.120: Flow in the C-elements vs time

Plotting the flows in the three I-elements:

```
for v, c, label in [(v_L,'r','v_L'), (v_L1,'b','v_L1'), (v_L2,'g','v_L2')]:
    fig=plt.plot(t,v,c,label=label)
    plt.xlabel("time (s)")
    plt.ylabel("flow (m3/s)")
    plt.legend(loc='upper right')
    plt.grid()
```

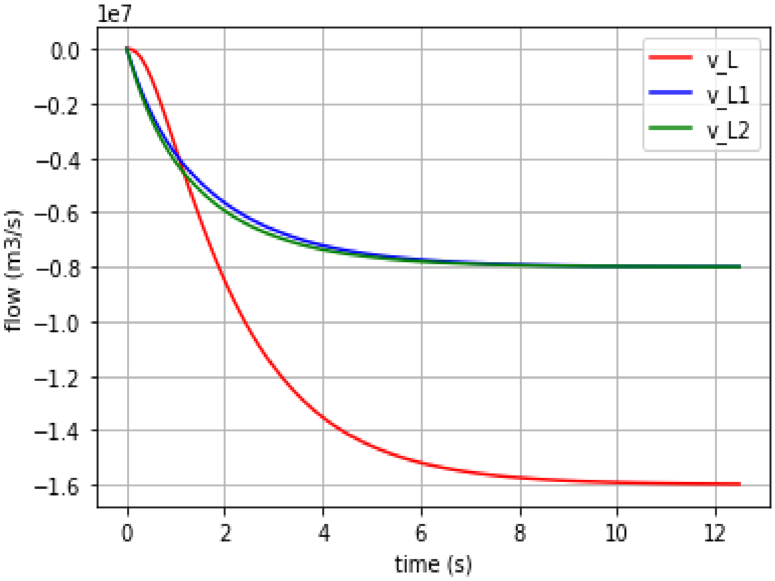


Fig. 1.121: Flow in the I-elements vs time

Click to read the codes for Branching vessel

Biochemical Reactions

Diffusion

The definitions in this document are adopted from[2].

Fig. 1.122: Schematic of Diffusion

In the first step, the BondgraphTools library must be imported:

```
import BondGraphTools as bgt
```

To create a new model using BondGraphTools (bgt), the command *bgt.new* is used:

```
model=bgt.new(name='Diffusion')
```

In the next step, the parameters' values are defined:

```
K_A=5263.6085
K_B=3803.6518
R=8.314
T=300
```

where *K_A* & *K_B* are species thermodynamic constants $[mol^{-1}]$, *R* is the ideal gas constant $[J/K/mol]$ and *T* is the absolute temperature $[K]$[4]. These values are then assigned to bgt components using the following commands:

```
Ce_A = bgt.new("Ce", name="A", library="BioChem", value={'k':K_A, 'R':R, 'T':T})
Ce_B = bgt.new("Ce", name="B", library="BioChem", value={'k':K_B, 'R':R, 'T':T})
reaction = bgt.new("Re", library="BioChem", value={'r':None, 'R':R, 'T':T})
```

The reaction rate 'r' is set to *"None"* in order to change it inside a *for* loop.

Also, two *"0-junctions"* must be defined (based on the diffusion model) as follows:

```
A_junction = bgt.new("0")
B_junction = bgt.new("0")
```

Now creating the model and its components has finished and all of them must be assembled using the *bgt.add* command:

```
bgt.add(model, Ce_A, Ce_B,A_junction, B_junction, reaction)
```

According to our bond graph model, these components must be connected to the related junctions by *bgt.connect*. Note that the first element in parenthesis represents the *"tail"* of the arrow and the second element represents the *"head"*:

```
bgt.connect(Ce_A, A_junction)
bgt.connect(A_junction, reaction)
bgt.connect(reaction, B_junction)
bgt.connect(B_junction, Ce_B)
```

By drawing the model, one can see if the components are connected properly to each other or not:

```
bgt.draw(model)
```

A sketch of the network will then be produced:

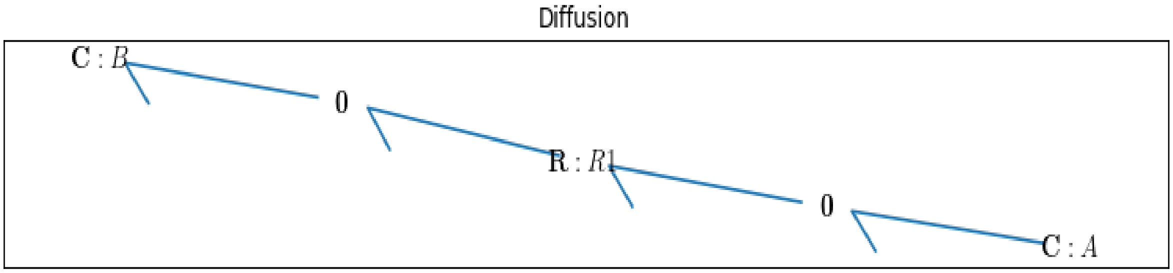


Fig. 1.123: Bondgraph diagram representing Diffusion

Now that the bond graph demonstration of the system is done, we can illustrate its behaviour during a specific time interval and with arbitrary initial conditions for the state variables. The constitutive relations of the model can be shown as well:

[4] Pan, M., & Gawthrop, P. J., & Tran, K., & Cursons, J. (2018). A thermodynamic framework for modelling membrane transporters. Click for the article

```
model.state_vars
Out[ ]: {'x_0': (C: A, 'q_0'), 'x_1': (C: B, 'q_0')}
```

==>

```
model.constitutive_relations
Out[ ]: [dx_0 + 10527217*u_0*x_0/2000 - 19018259*u_0*x_1/5000,
dx_1 - 10527217*u_0*x_0/2000 + 19018259*u_0*x_1/5000]
```

By using the command *"bgt.simulate"* and entering the model, time interval, and the initial conditions one can plot the time behaviour of the system. x vs t can be plotted by importing *"matplotlib.pyplot"* (q_{Ce_A} & q_{Ce_B} are state variables of the system which represent the molar amount in each solute):

```
import matplotlib.pyplot as plt
x0 = {"x_0":1, "x_1":0}
t_span = [0,3]

for c, kappa, label in [('r', 0.00019926, 'kappa=0.00019926'), ('b', 0.00053004,
↪'kappa=0.00053004'), ('g', 0.001,'kappa=0.001')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    plt.plot(t,x[:,0], c, label=label)
    plt.title('"Solute A"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
```

Three different amounts for the control variable (*kappa*) are considered to show its impact on the molar amount of each solute during the diffusion.

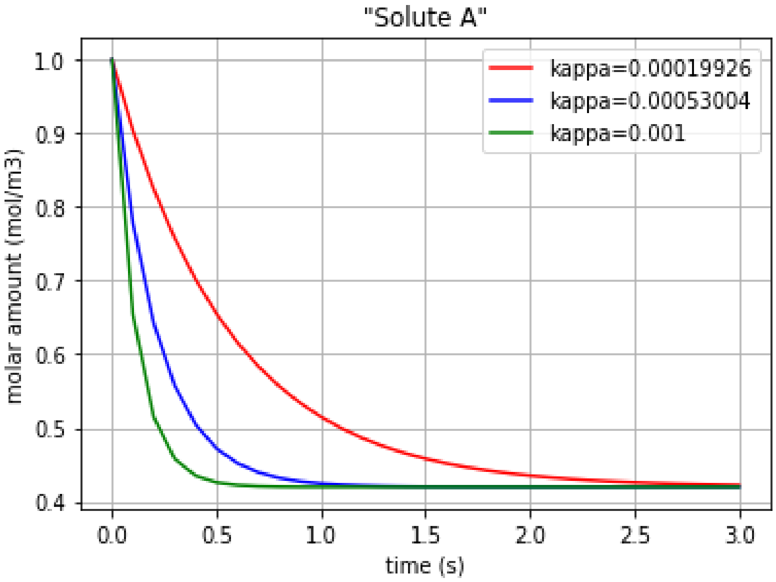


Fig. 1.124: Molar amount of solute A during diffusion with 3 different amounts for 'kappa'

The same method can be manipulated to plot the molar amount of solute B vs time:

```
for c, kappa, label in [('r', 0.00019926, 'kappa=0.00019926'), ('b', 0.00053004,
↪'kappa=0.00053004'), ('g', 0.001,'kappa=0.001')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    plt.plot(t,x[:,1], c+':', label=label)
    plt.title('"Solute B"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='lower right')
    plt.grid()
```

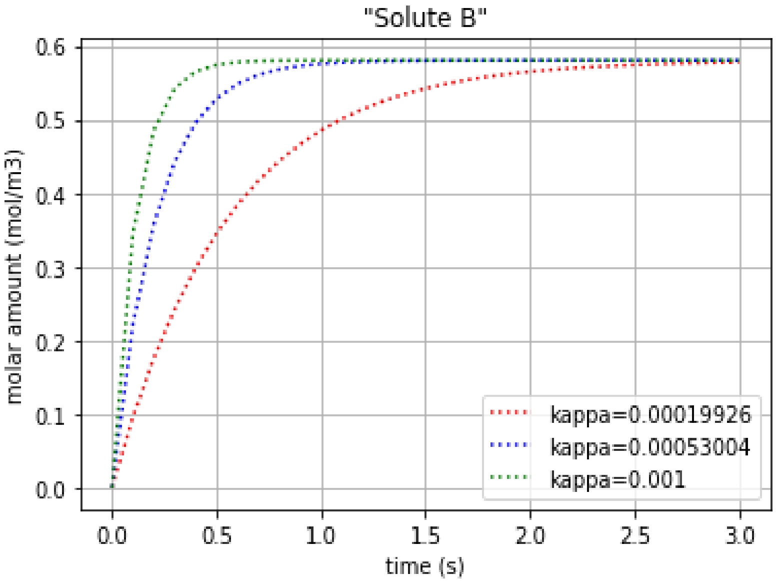


Fig. 1.125: Molar amount of solute B during diffusion with 3 different amounts for 'kappa'

Since the molar concentration flow rate (v) of a solute is the time derivative of **q**:

$$\frac{dq}{dt} = v$$

it can be plotted by converting the considered state variable (either x[:,0] or x[:,1]) to an array by importing the *numpy library* and then calculating its gradient with 0.1 steps:

```
# Calculating the molar concentration flow rate of both the solutes
#  dq_Ce_A/dt = v_Ce_A (flow in the Ce_A)
#  dq_Ce_B/dt = v_Ce_B (flow in the Ce_B)

import matplotlib.pyplot as plt
import numpy as np
for c, kappa, title in [('r', 0.00019926, 'kappa=0.00019926'), ('b', 0.00053004,
↪'kappa=0.00053004'), ('g', 0.001,'kappa=0.001')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    f = np.array(x[:,0], dtype=float)
    v_Ce_A=np.gradient(f,0.1)

    f = np.array(x[:,1], dtype=float)
    slope=np.gradient(f,0.1)
    v_Ce_B=slope
```

Plotting the molar concentration flow rates of the two solutes:

```
plt.plot(t,v_Ce_A, c, label='v_Ce_A')
plt.plot(t,v_Ce_B, c+':', label='v_Ce_B')
leg1=plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("molar concentration flow rate (mol/m3/s)")
plt.title(title)
plt.grid()
plt.show()
```

Each of the three following figures are plotted with a different amount for the variable ‘kappa’. The molar concentration flow rate of the both solutes (A & B) is depicted in each figure:

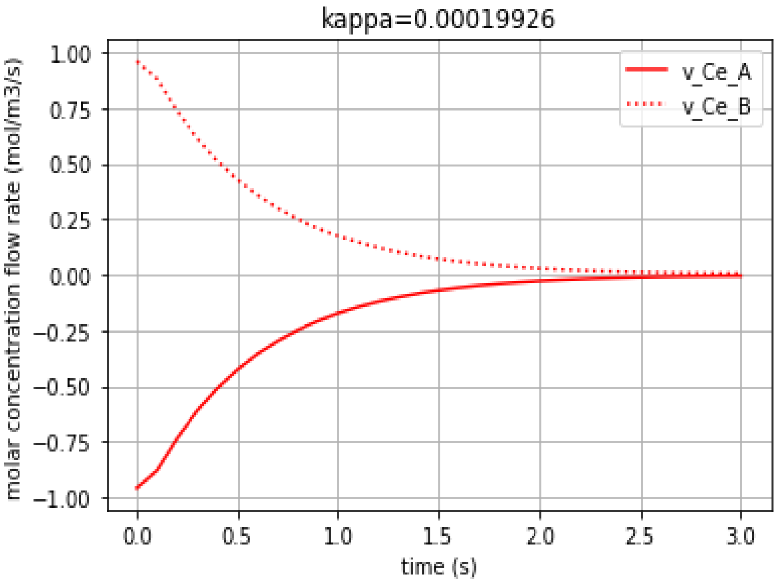


Fig. 1.126: Molar concentration flow rate of solute A & B with kappa=0.00019926.

Moreover, the chemical potential in each solute can be calculated based on its constitutive equations:

Solute: $u = R.T.ln(K_s.q)$

where u is the chemical potential of the solute, K_s is the species thermodynamic constant, and q is the molar concentration.

Thus, by choosing kappa=0.00019926 as a sample constant for the simulation:

```
# Calculating & plotting the solutes chemical potential (u_Ce_A & u_Ce_B)
# u=R.T.ln(K.q)
# for kappa=0.00019926

kappa=0.00019926
import math

t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})

q_Ce_A = np.array(x[:,0], dtype=float)
u_Ce_A=R*T*np.log(K_A*q_Ce_A)
```

(continues on next page)

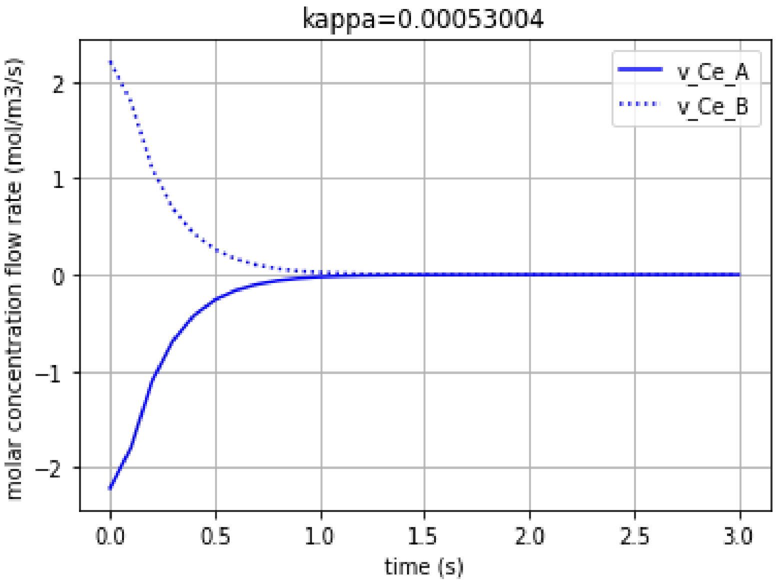


Fig. 1.127: Molar concentration flow rate of solute A & B with kappa=0.00053004.

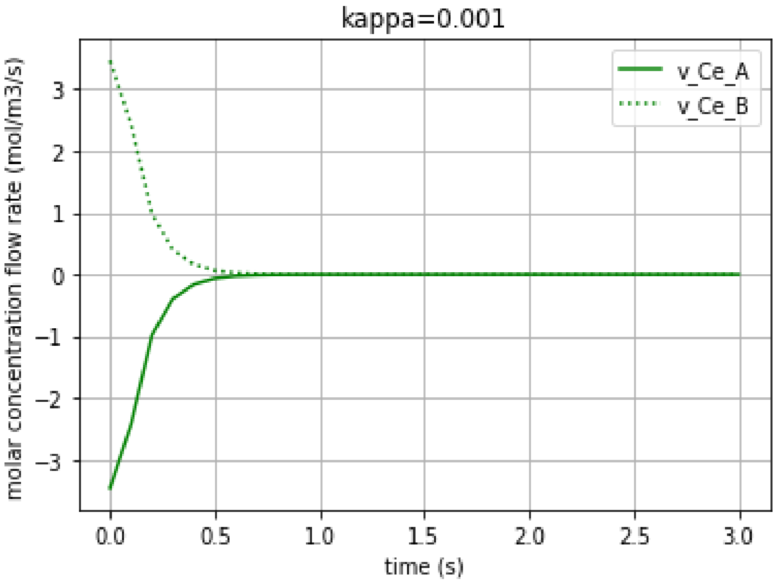


Fig. 1.128: Molar concentration flow rate of solute A & B with kappa=0.001.

(continued from previous page)

```
q_Ce_B = np.array(x[:,1], dtype=float)
u_Ce_B=R*T*np.log(K_B*q_Ce_B)
```

The time variation of the corresponding chemical potential for each solute (u_{Ce_A} & u_{Ce_B}) can be both plotted in a figure:

```
plt.plot(t,u_Ce_A, 'm', label='u_Ce_A')
plt.plot(t,u_Ce_B, 'c', label='u_Ce_B')
plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("Chemical potential (J/mol)")
plt.title('Chemical potential of the solutes')
plt.grid()
```

which results in:

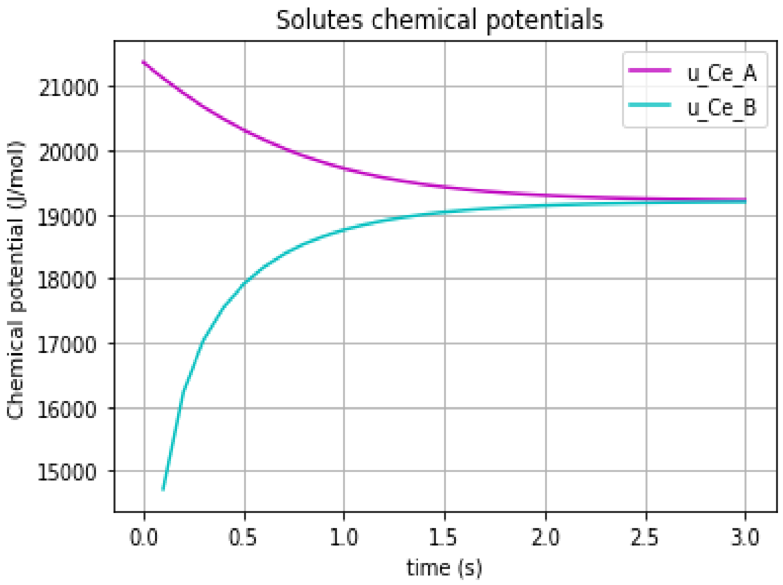


Fig. 1.129: Chemical potential of each solute (A & B) vs time

Click to read diffusion codes

Simple Reaction

The definitions in this document are adopted from[2]

Fig. 1.130: Schematic of Simple Reaction

The rationale behind the following codes are explained thoroughly in *"Diffusion"* documentation.

```
import BondGraphTools as bgt
model = bgt.new(name="Simple Reaction")
K_A=50
K_B=20
```

(continues on next page)

(continued from previous page)

```
K_C=10
K_D=5

Ce_A = bgt.new("Ce", name="A", library="BioChem", value={'k':K_A , 'R':8.314, 'T':300}
↪)
Ce_B= bgt.new("Ce", name="B", library="BioChem", value={'k':K_B, 'R':8.314, 'T':300})
Ce_C= bgt.new("Ce", name="C", library="BioChem", value={'k':K_C, 'R':8.314, 'T':300})
Ce_D= bgt.new("Ce", name="D", library="BioChem", value={'k':K_D, 'R':8.314, 'T':300})
reaction = bgt.new("Re", library="BioChem", value={'r':None, 'R':8.314, 'T':300})

A_junction = bgt.new("0")
B_junction = bgt.new("0")
C_junction = bgt.new("0")
D_junction = bgt.new("0")

one_junction_1 = bgt.new("1")
one_junction_2 = bgt.new("1")

bgt.add(model, Ce_A, Ce_B, Ce_C, Ce_D, A_junction, B_junction, C_junction, D_junction,
↪ one_junction_1, one_junction_2, reaction)
bgt.connect(Ce_A, A_junction)
bgt.connect(A_junction, one_junction_1)
bgt.connect(Ce_B, B_junction)
bgt.connect(B_junction, one_junction_1)
bgt.connect(one_junction_1, reaction)
bgt.connect(reaction, one_junction_2)
bgt.connect(one_junction_2, C_junction)
bgt.connect(C_junction, Ce_C)
bgt.connect(one_junction_2, D_junction)
bgt.connect(D_junction, Ce_D)
```

The reaction rate ‘r’ is set to *“None”* in order to change it inside a *for* loop.

By drawing the model, one can see if the components are connected properly to each other or not:

```
bgt.draw(model)
```

A sketch of the network will then be produced:

Now that the bond graph demonstration of the system is done, we can illustrate its behaviour during a specific time interval and with arbitrary initial conditions for the state variables. The constitutive relations of the model can be shown as well:

```
model.state_vars
Out[ ]: {'x_0': (C: A, 'q_0'),
'x_1': (C: B, 'q_0'),
'x_2': (C: C, 'q_0'),
'x_3': (C: D, 'q_0')}
```

==>

```
model.constitutive_relations
Out[ ]: [dx_0 + 1000*u_0*x_0*x_1 - 50*u_0*x_2*x_3,
dx_1 + 1000*u_0*x_0*x_1 - 50*u_0*x_2*x_3,
dx_2 - 1000*u_0*x_0*x_1 + 50*u_0*x_2*x_3,
dx_3 - 1000*u_0*x_0*x_1 + 50*u_0*x_2*x_3]
```

By using the command *“bgt.simulate”* and entering the model, time interval, and the initial conditions one can plot

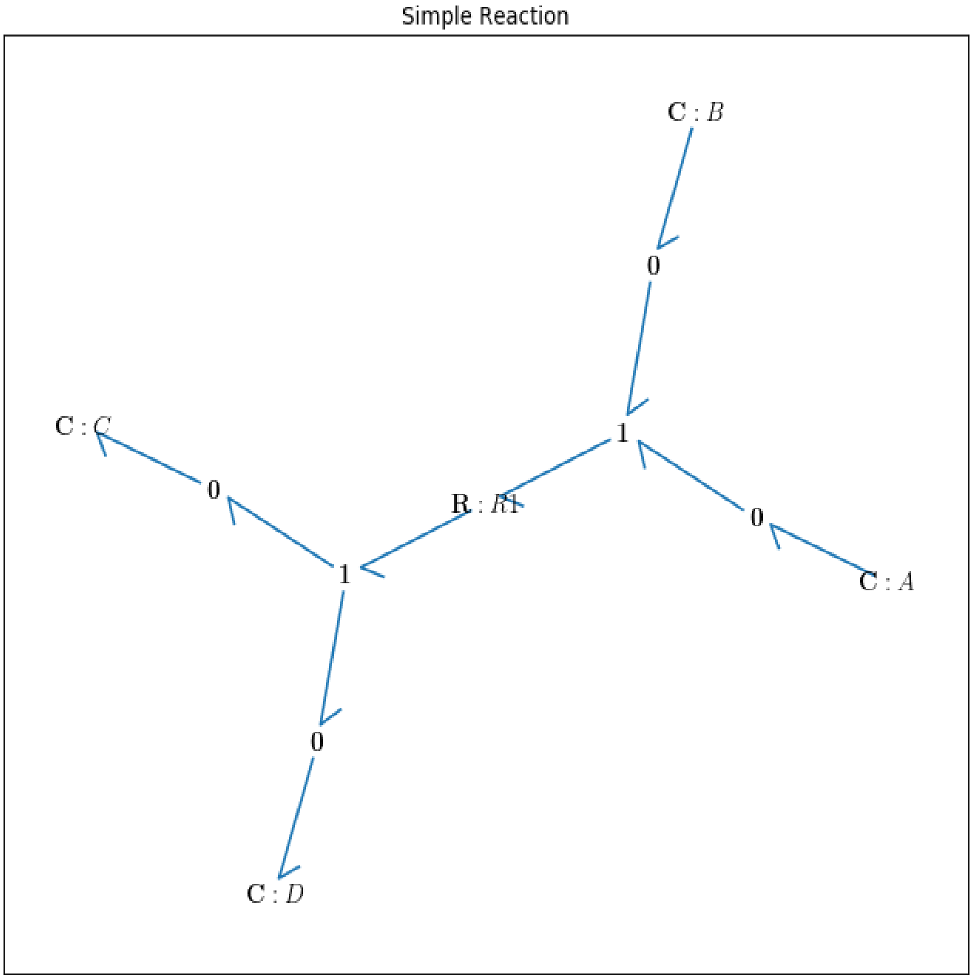


Fig. 1.131: Bondgraph diagram representing Simple Reaction

the time behaviour of the system. x vs t can be plotted by importing *"matplotlib.pyplot"* (q_{Ce_A} & q_{Ce_B} & q_{Ce_C} & q_{Ce_D} are state variables of the system which represent the molar amount in each solute):

```
import matplotlib.pyplot as plt
x0 = {"x_0":1, "x_1":1, "x_2":0, "x_3":0}
t_span = [0,6]

for c, kappa, label in [('r', 0.0005, 'kappa=0.0005'), ('b', 0.001, 'kappa=0.001'), (
↪'g', 0.01,'kappa=0.01')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    plt.plot(t,x[:,0], c, label=label)
    plt.title('"Solute A & B"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
plt.show()


for c, kappa, label in [('r', 0.0005, 'kappa=0.0005'), ('b', 0.001, 'kappa=0.001'), (
↪'g', 0.01,'kappa=0.01')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    plt.plot(t,x[:,2], c, label=label)
    plt.title('"Solute C & D"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
plt.show()
```

Three different amounts for the control variable (*kappa*) are considered to show its impact on the molar amount of each solute during the reaction period. Also, since the pairs $\{q_{Ce_A}$ & $q_{Ce_B}\}$ and $\{q_{Ce_C}$ & $q_{Ce_D}\}$ are identical in amounts, just two figures are plotted.

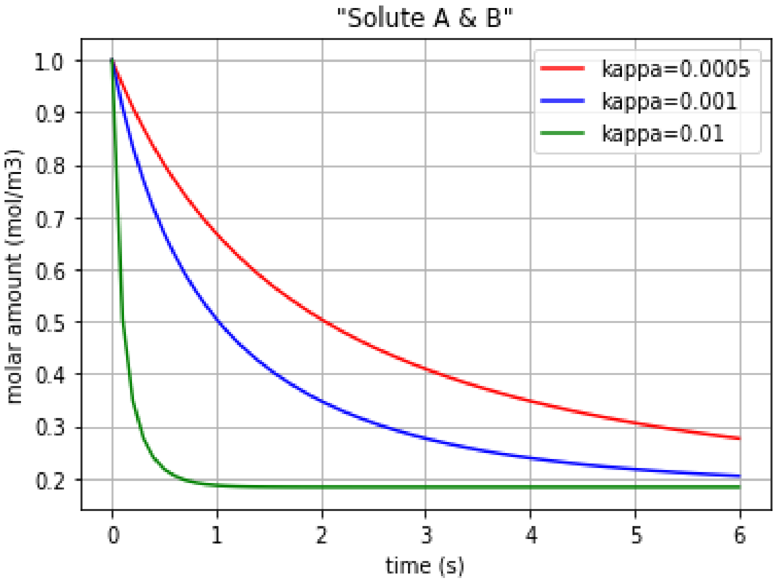


Fig. 1.132: Molar amount of solutes A & B during a simple reaction with 3 different amounts for 'kappa'

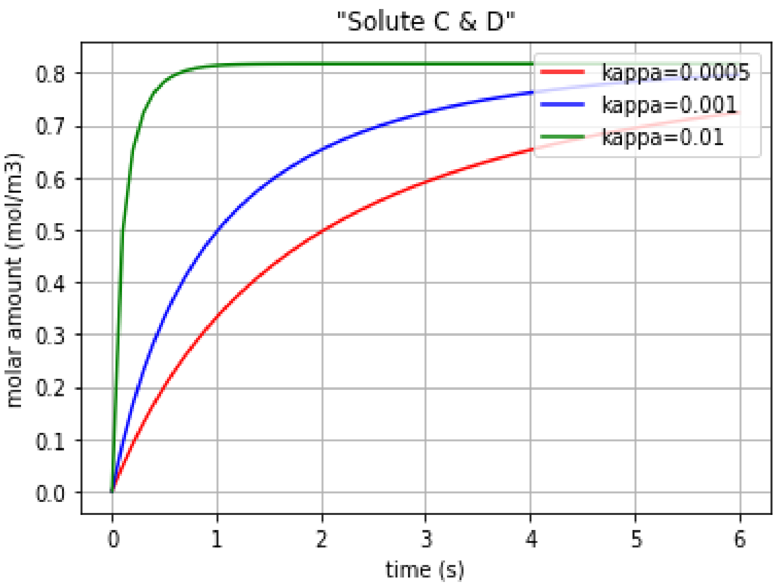


Fig. 1.133: Molar amount of solutes C & D during a simple reaction with 3 different amounts for 'kappa'

Since the molar concentration flow rate (v) of a solute is the time derivative of **q**:

$$\frac{dq}{dt} = v$$

it can be plotted by converting the considered state variable (either x[:,0], x[:,1], x[:,2] or x[:,3]) to an array by importing the *numpy library* and then calculating its gradient with 0.1 steps:

```
# Calculating the molar concentration flow rate of the solutes
#  dq_Ce_A/dt = v_Ce_A (flow in the Ce_A)
#  dq_Ce_B/dt = v_Ce_B (flow in the Ce_B)
#  dq_Ce_C/dt = v_Ce_C (flow in the Ce_C)
#  dq_Ce_D/dt = v_Ce_D (flow in the Ce_D)

import matplotlib.pyplot as plt
import numpy as np
for c, kappa, title in [('r', 0.0005, 'kappa=0.0005'), ('b', 0.001, 'kappa=0.001'), (
↪'g', 0.01,'kappa=0.01')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    f = np.array(x[:,0], dtype=float)
    v_Ce_A=np.gradient(f,0.1)

    f = np.array(x[:,1], dtype=float)
    slope=np.gradient(f,0.1)
    v_Ce_B=slope

    f = np.array(x[:,2], dtype=float)
    slope=np.gradient(f,0.1)
    v_Ce_C=slope

    f = np.array(x[:,3], dtype=float)
    slope=np.gradient(f,0.1)
    v_Ce_D=slope
```

Plotting the molar concentration flow rates of the solutes for three different amounts of *kappa*:

```
plt.plot(t,v_Ce_A, c, label='v_Ce_A & v_Ce_B')          # v_Ce_A & v_Ce_B have the same␣
↪amounts
plt.plot(t,v_Ce_C, c+'*', label='v_Ce_C & v_Ce_D')      # v_Ce_C & v_Ce_D have the same␣
↪amounts

leg1=plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("molar concentration flow rate (mol/m3/s)")
plt.title(title)
plt.grid()
plt.show()
```

Again note that since the amounts for the pairs $\{v_{Ce_A}$ & $v_{Ce_B}\}$ and $\{v_{Ce_C}$ & $v_{Ce_D}\}$ are equal, we have just demonstrated one figure representing each pair. Each of the three following figures are plotted with a different amount for the variable 'kappa'. The molar concentration flow rate of each pair of solutes is depicted in each figure:

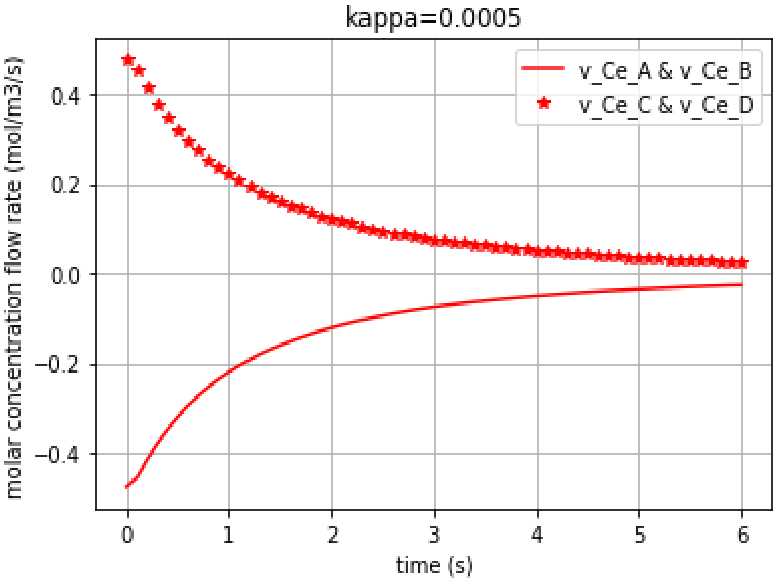


Fig. 1.134: Molar concentration flow rate of solute A & B with kappa=0.0005.

Moreover, the chemical potential in each solute can be calculated based on its constitutive equations:

Solute: $u = R.T.ln(K_s.q)$

where u is the chemical potential of the solute $[J/mol]$, K_s is the species thermodynamic constant $[mol^{-1}]$, *R* is the ideal gas constant $[J/K/mol]$, *T* is the absolute temperature $[K]$ and q is the molar concentration[4].

Thus, the u in each solute is calculated for a sample reaction rate of *kappa=0.001*:

```
# Calculating & plotting the solutes chemical potentials (u_Ce_A & u_Ce_B & u_Ce_C &␣
↪u_Ce_D)
# for kappa=0.001

kappa=0.001
t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})

q_Ce_A = np.array(x[:,0], dtype=float)
u_Ce_A=R*T*np.log(K_A*q_Ce_A)
```

(continues on next page)

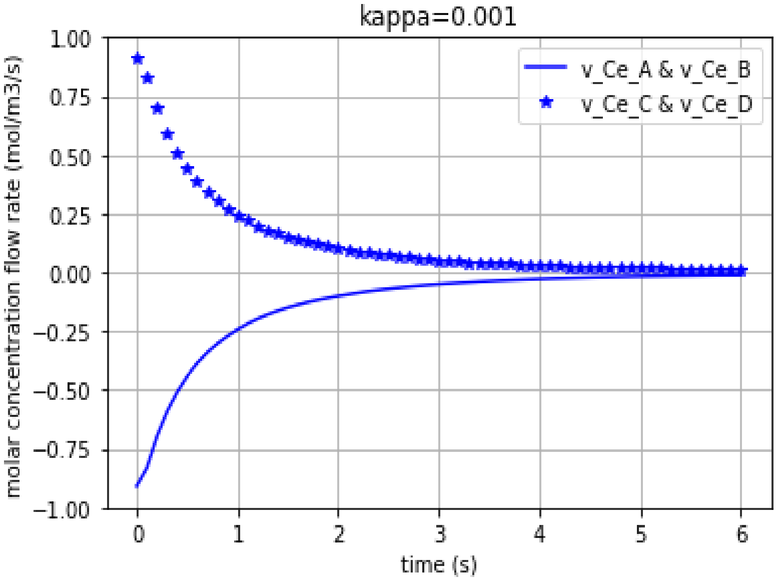


Fig. 1.135: Molar concentration flow rate of solute A & B with kappa=0.001.

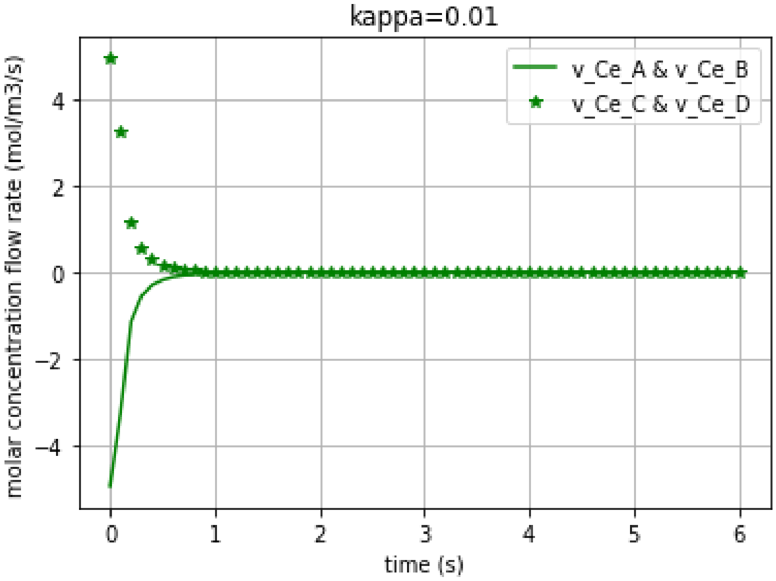


Fig. 1.136: Molar concentration flow rate of solute A & B with kappa=0.01.

(continued from previous page)

```
q_Ce_B = np.array(x[:,1], dtype=float)
u_Ce_B=R*T*np.log(K_B*q_Ce_B)

q_Ce_C = np.array(x[:,2], dtype=float)
u_Ce_C=R*T*np.log(K_C*q_Ce_C)

q_Ce_D = np.array(x[:,3], dtype=float)
u_Ce_D=R*T*np.log(K_D*q_Ce_D)
```

The time variation of the corresponding chemical potential for each solute (u_{Ce_A} & u_{Ce_B} & u_{Ce_C} & u_{Ce_D}) can be both plotted in a figure:

```
plt.plot(t,u_Ce_A, 'm', label='u_Ce_A')
plt.plot(t,u_Ce_B, 'c', label='u_Ce_B')
plt.plot(t,u_Ce_C, 'y', label='u_Ce_C')
plt.plot(t,u_Ce_D, 'k', label='u_Ce_D')
plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("Chemical potential (J/mol)")
plt.title('Chemical potential of the solutes')
plt.grid()
```

which results in:

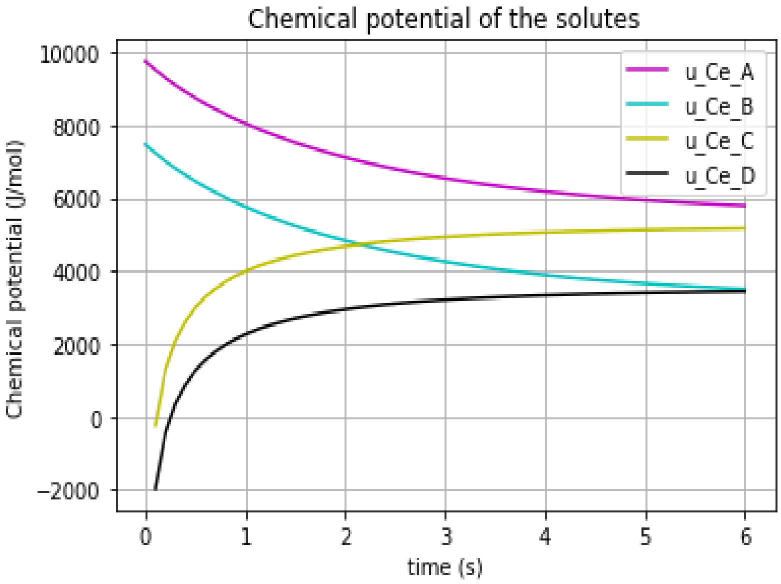


Fig. 1.137: Chemical potential of each solute (A,B,C,D) vs time

* Note the differences which the “K”s make in the chemical potential of the solutes (K_A, K_B, K_C, K_D).

Click to read Simple reaction codes

Enzyme Catalysed Reaction #1

The definitions in this document are adopted from[2] and[5] .

Fig. 1.138: Schematic of Enzyme Catalysed Reaction #1

The rationale behind the following codes are explained thoroughly in *"Diffusion"* and *"Simple reaction"* documentation.

```
import BondGraphTools as bgt
model = bgt.new(name="Enzyme catalysed Reaction")
K_A=50
K_E=1
K_B=5

R=8.314
T=300

Ce_A = bgt.new("Ce", name="A", library="BioChem", value={'k':K_A , 'R':R, 'T':T})
Ce_B= bgt.new("Ce", name="B", library="BioChem", value={'k':K_B, 'R':R, 'T':T})
Ce_E= bgt.new("Ce", name="E", library="BioChem", value={'k':K_E, 'R':R, 'T':T})

reaction = bgt.new("Re", library="BioChem", value={'r':None, 'R':R, 'T':T})

zero_junction = bgt.new("0")
one_junction_1 = bgt.new("1")
one_junction_2 = bgt.new("1")

bgt.add(model, Ce_A, Ce_B, Ce_E, zero_junction, one_junction_1, one_junction_2,␣
↪reaction)

bgt.connect(Ce_A,one_junction_1)
bgt.connect(one_junction_1,reaction)
bgt.connect(reaction,one_junction_2)
bgt.connect(one_junction_2,Ce_B)
bgt.connect(zero_junction,one_junction_1)
bgt.connect(zero_junction,Ce_E)
bgt.connect(one_junction_2,zero_junction)
```

The reaction rate 'r' is set to *"None"* in order to change it inside a *for* loop.

By drawing the model, one can see if the components are connected properly to each other or not:

```
bgt.draw(model)
```

A sketch of the network will then be produced:

Now that the bond graph demonstration of the system is done, we can illustrate its behaviour during a specific time interval and with arbitrary initial conditions for the state variables. The constitutive relations of the model can be shown as well:

```
model.state_vars
Out[ ]: {'x_0': (C: A, 'q_0'),
'x_1': (C: B, 'q_0'),
'x_2': (C: E, 'q_0')}
```

[5] *BondGraphTools Tutorial* (2018). Retrieved August 13, 2019, from here

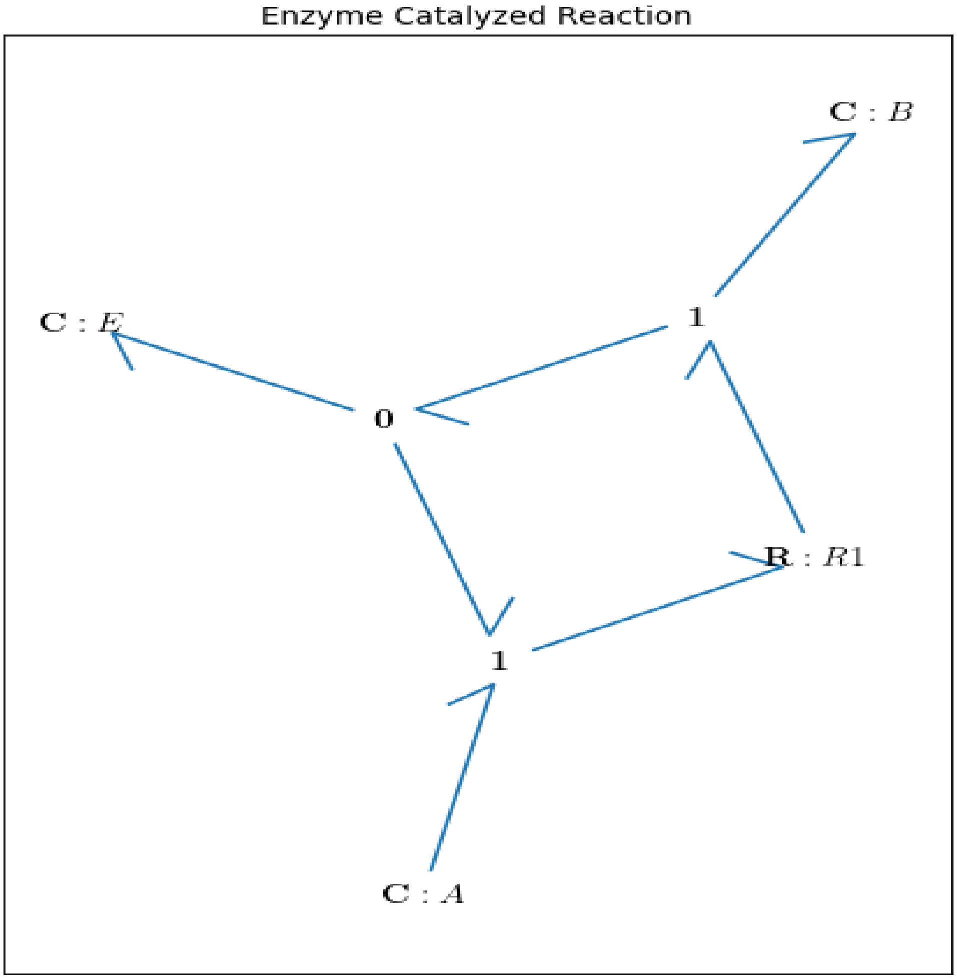


Fig. 1.139: Bondgraph diagram representing Enzyme Catalysed Reaction #1

==>

```
model.constitutive_relations
Out[ ]: [dx_0 + 50*u_0*x_0*x_2 - 5*u_0*x_1*x_2,
dx_1 - 50*u_0*x_0*x_2 + 5*u_0*x_1*x_2,
dx_2]
```

By using the command *"bgt.simulate"* and entering the model, time interval, and the initial conditions one can plot the time behaviour of the system. x vs t can be plotted by importing *"matplotlib.pyplot"* (q_{Ce_A} & q_{Ce_B} & q_{Ce_E} are state variables of the system which represent the molar amount in each solute/enzyme):

```
import matplotlib.pyplot as plt
x0 = {"x_0":1, "x_1":0.001, "x_2":1}
t_span = [0,10]

for c, kappa, label in [('r', 0.005, 'kappa=0.005'), ('b', 0.01, 'kappa=0.01'), ('g',␣
↪0.1,'kappa=0.1')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    plt.plot(t,x[:,0], c, label=label)
    plt.title('"Solute A"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
plt.show()


for c, kappa, label in [('r', 0.005, 'kappa=0.005'), ('b', 0.01, 'kappa=0.01'), ('g',␣
↪0.1,'kappa=0.1')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    plt.plot(t,x[:,1], c, label=label)
    plt.title('"Solute B"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
plt.show()

for c, kappa, label in [('r', 0.005, 'kappa=0.005'), ('b', 0.01, 'kappa=0.01'), ('g',␣
↪0.1,'kappa=0.1')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    plt.plot(t,x[:,2], c, label=label)
    plt.title('"Enzyme"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
plt.show()
```

Three different amounts for the control variable (*kappa*) are considered to show its impact on the molar amount of each solute/enzyme during the reaction period.

* Note that since the enzyme E is neither produced nor consumed, its molar amount is remained constant.

The molar concentration flow rate (v) of a solute is the time derivative of **q**:

$$\frac{dq}{dt} = v$$

and it can be plotted by converting the considered state variable (either x[:,0], x[:,1] or x[:,2]) to an array by importing

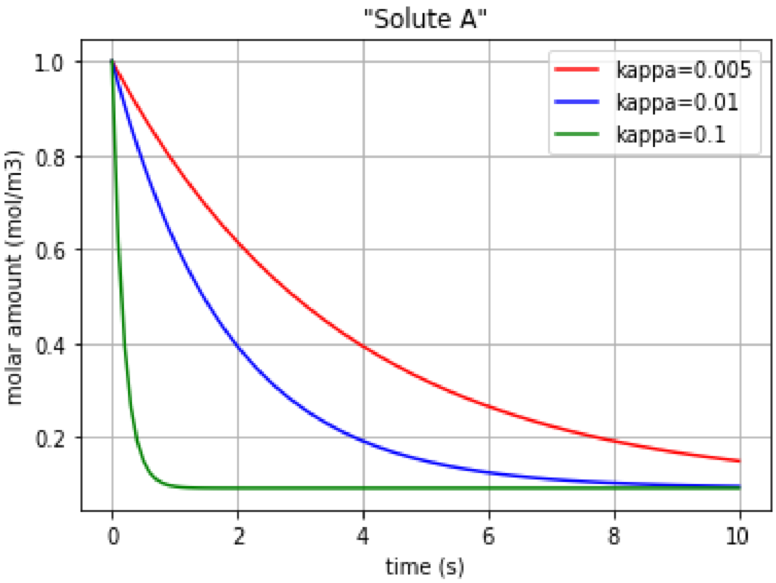


Fig. 1.140: Molar amount of solute A during an enzyme catalysed reaction with 3 different amounts for ‘kappa’

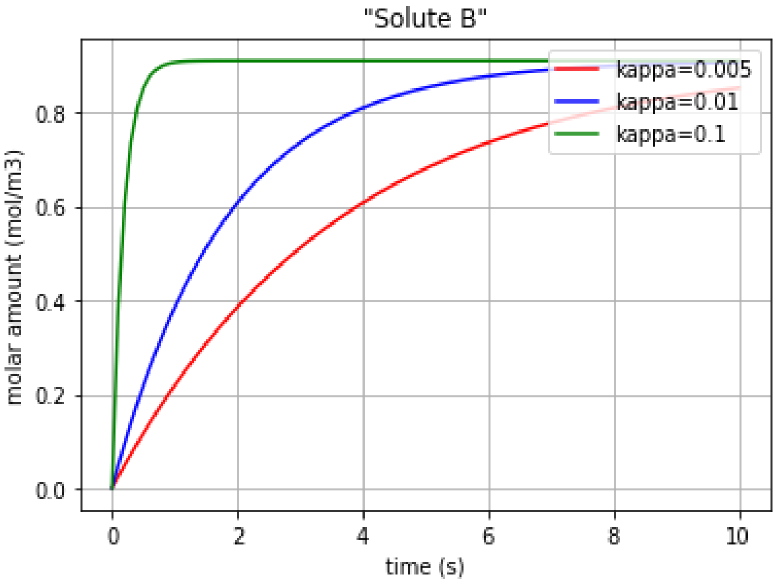


Fig. 1.141: Molar amount of solute B during an enzyme catalysed reaction with 3 different amounts for ‘kappa’

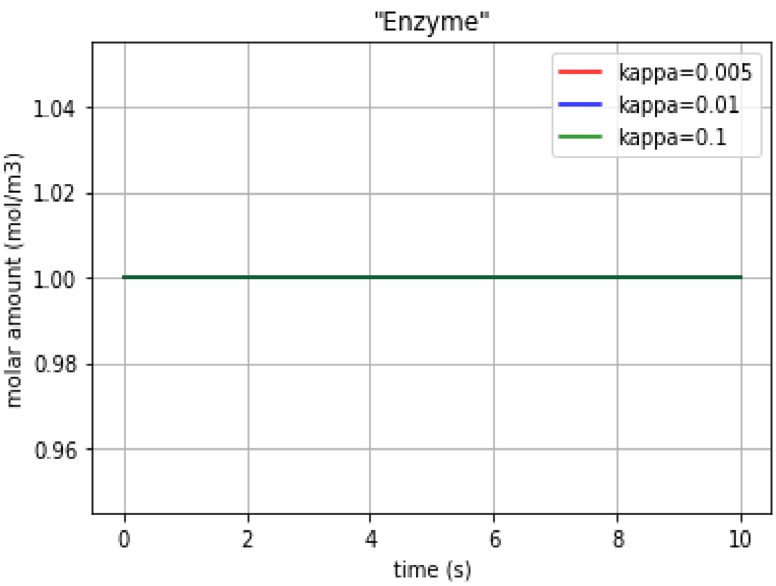


Fig. 1.142: Molar amount of the enzyme E during an enzyme catalysed reaction with 3 different amounts for 'kappa'

the *numpy library* and then calculating its gradient with 0.1 steps:

```
import matplotlib.pyplot as plt
import numpy as np
for c, kappa, title in [('r', 0.005, 'kappa=0.005'), ('b', 0.01, 'kappa=0.01'), ('g',␣
↪0.1,'kappa=0.1')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})
    f = np.array(x[:,0], dtype=float)
    v_Ce_A=np.gradient(f,0.1)

    f = np.array(x[:,1], dtype=float)
    slope=np.gradient(f,0.1)
    v_Ce_B=slope

    f = np.array(x[:,2], dtype=float)
    slope=np.gradient(f,0.1)
    v_Ce_E=slope
```

Plotting the molar concentration flow rates of the solutes for three different amounts of *kappa*:

```
plt.plot(t,v_Ce_A, c, label='v_Ce_A')
plt.plot(t,v_Ce_B, c+':', label='v_Ce_B')
plt.plot(t,v_Ce_E, c+'*', label='v_Ce_E')

leg1=plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("molar concentration flow rate (mol/m3/s)")
plt.title(title)
plt.grid()
plt.show()
```

Each of the three following figures are plotted with a different amount for the variable 'kappa'. The molar concentration flow rate of each solute/enzyme is depicted in each figure:

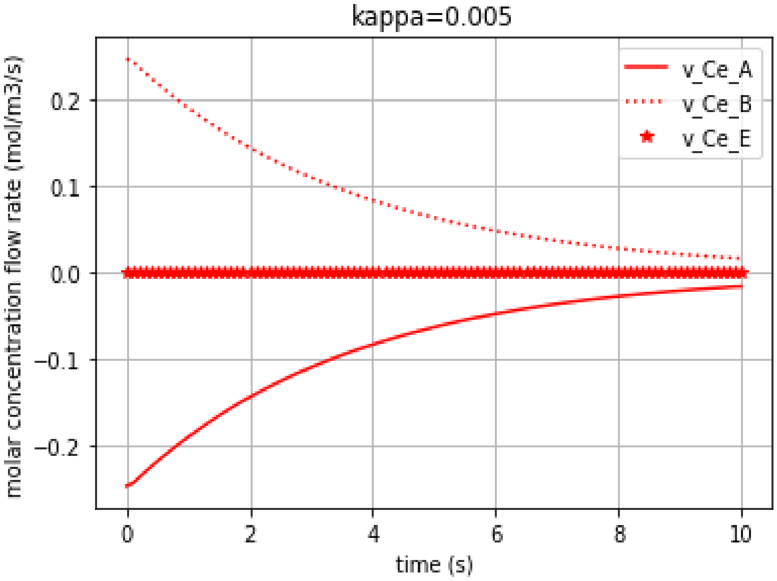


Fig. 1.143: Molar concentration flow rate of solutes A & B and the enzyme E with kappa=0.005.

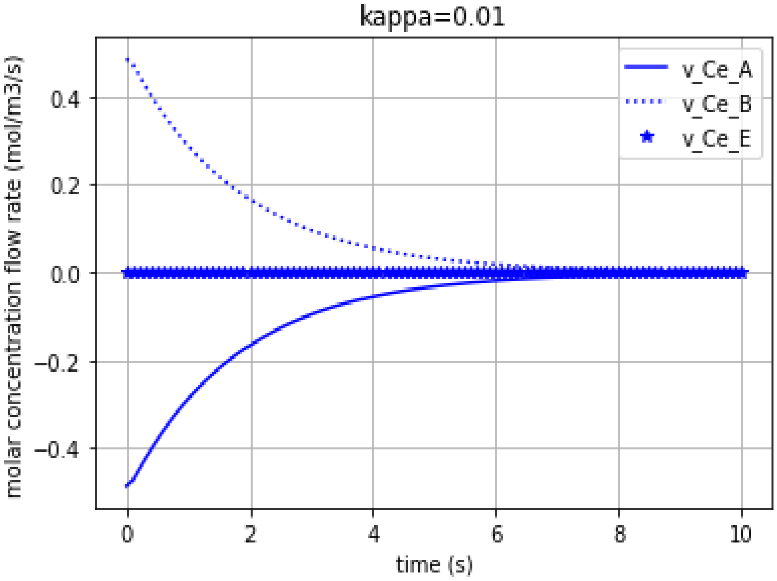


Fig. 1.144: Molar concentration flow rate of solutes A & B and the enzyme E with kappa=0.01.

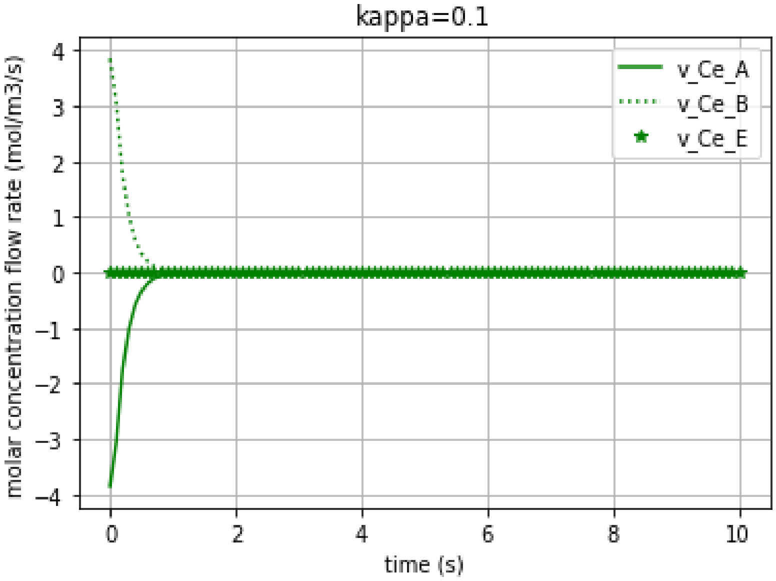


Fig. 1.145: Molar concentration flow rate of solutes A & B and the enzyme E with kappa=0.1.

Moreover, the chemical potential in each solute/enzyme can be calculated based on its constitutive equations:

Solute: $u = R.T.ln(K_s.q)$

where u is the chemical potential of the solute $[J/mol]$, K_s is the species thermodynamic constant $[mol^{-1}]$, *R* is the ideal gas constant $[J/K/mol]$, *T* is the absolute temperature $[K]$ and q is the molar concentration[4].

Thus, the u in each solute is calculated for a sample reaction rate of *kappa=0.01*:

```
kappa=0.01
t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})

q_Ce_A = np.array(x[:,0], dtype=float)
u_Ce_A=R*T*np.log(K_A*q_Ce_A)

q_Ce_B = np.array(x[:,1], dtype=float)
u_Ce_B=R*T*np.log(K_B*q_Ce_B)

q_Ce_E = np.array(x[:,2], dtype=float)
u_Ce_E=R*T*np.log(K_E*q_Ce_E)
```

The time variation of the corresponding chemical potential for each solute/enzyme (u_{Ce_A} & u_{Ce_B} & u_{Ce_E}) can be all plotted in a figure:

```
plt.plot(t,u_Ce_A, 'm', label='u_Ce_A')
plt.plot(t,u_Ce_B, 'c', label='u_Ce_B')
plt.plot(t,u_Ce_E, 'k', label='u_Ce_E')
plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("Chemical potential (J/mol)")
plt.title('Chemical potential of the solutes')
plt.grid()
```

which results in:

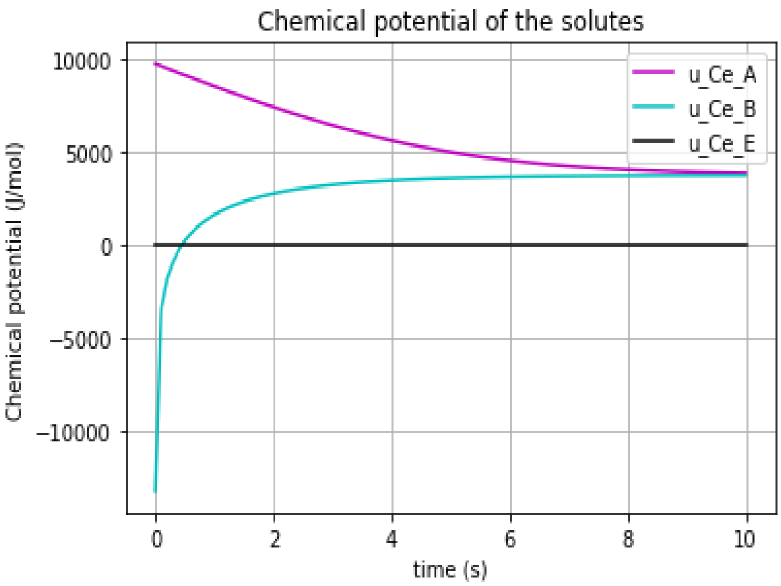


Fig. 1.146: Chemical potential of each solute/enzyme (A,B,E) vs time

At this stage we would like to demonstrate the relation between the molar amount of the substrate "A" and the molar flow rate of producing the product "B". * Note that by the declining of the molar amount of the substrate "A", the production rate of "B" reduces:

```
plt.plot(q_Ce_A,v_Ce_B,'*-r')
plt.xlabel("molar amount of solute A (mol/m3)")
plt.ylabel("molar concentration flow rate of solute B (mol/m3/s)")
plt.grid()
```

Click to read Enzyme Catalysed Reaction #1 codes

Enzyme Catalysed Reaction #2

The definitions in this document are adopted from[2] and[5] .

The rationale behind the following codes are explained thoroughly in *"Diffusion"* and *"Simple reaction"* documentation.

```
import BondGraphTools as bgt
model = bgt.new(name="Enzyme catalysed Reaction 2")
K_A=20
K_B=20
K_E=1
K_C=20

R=8.314
T=300

Ce_A = bgt.new("Ce", name="A", library="BioChem", value={'k':K_A , 'R':R, 'T':T})
Ce_B= bgt.new("Ce", name="B", library="BioChem", value={'k':K_B, 'R':R, 'T':T})
Ce_E= bgt.new("Ce", name="E", library="BioChem", value={'k':K_E, 'R':R, 'T':T})
Ce_C= bgt.new("Ce", name="C", library="BioChem", value={'k':K_C, 'R':R, 'T':T})
```

(continues on next page)

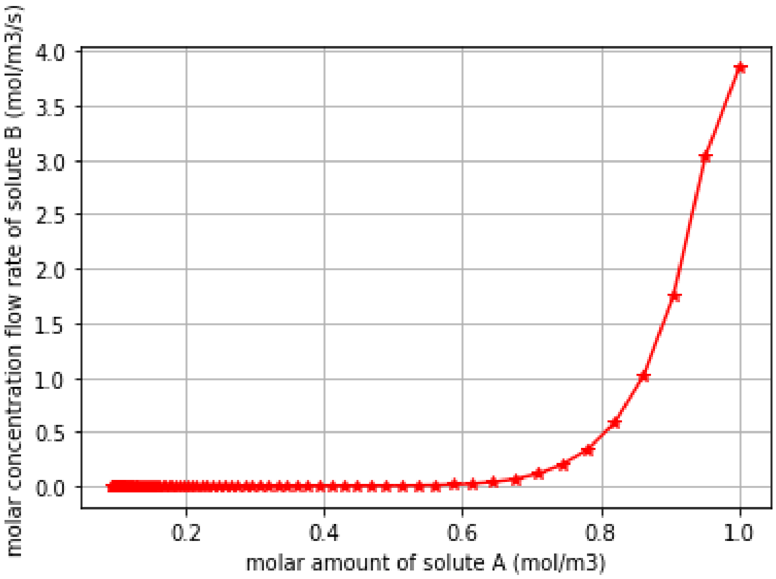


Fig. 1.147: Molar concentration flow rate of solute B vs the molar amount of solute A

Fig. 1.148: Schematic of Enzyme Catalysed Reaction #2

(continued from previous page)

```
reaction_1 = bgt.new("Re", library="BioChem", value={'r':None, 'R':R, 'T':T})
reaction_2 = bgt.new("Re", library="BioChem", value={'r':None, 'R':R, 'T':T})


zero_junction_1 = bgt.new("0")
zero_junction_2 = bgt.new("0")

one_junction_1 = bgt.new("1")
one_junction_2 = bgt.new("1")

bgt.add(model, Ce_A, Ce_B, Ce_E,Ce_C, zero_junction_1, zero_junction_2,
   one_junction_1, one_junction_2, reaction_1, reaction_2)

bgt.connect(Ce_A,one_junction_1)
bgt.connect(one_junction_1,reaction_1)
bgt.connect(reaction_1,zero_junction_1)
bgt.connect(zero_junction_1,Ce_B)
bgt.connect(zero_junction_1,reaction_2)
bgt.connect(reaction_2,one_junction_2)
bgt.connect(one_junction_2,Ce_C)
bgt.connect(one_junction_2,zero_junction_2)
bgt.connect(zero_junction_2,Ce_E)
bgt.connect(zero_junction_2,one_junction_1)
```

The reactions rate ‘r’ is set to *“None”* in order to change it inside a *for* loop.

By drawing the model, one can see if the components are connected properly to each other or not:

```
bgt.draw(model)
```

A sketch of the network will then be produced:

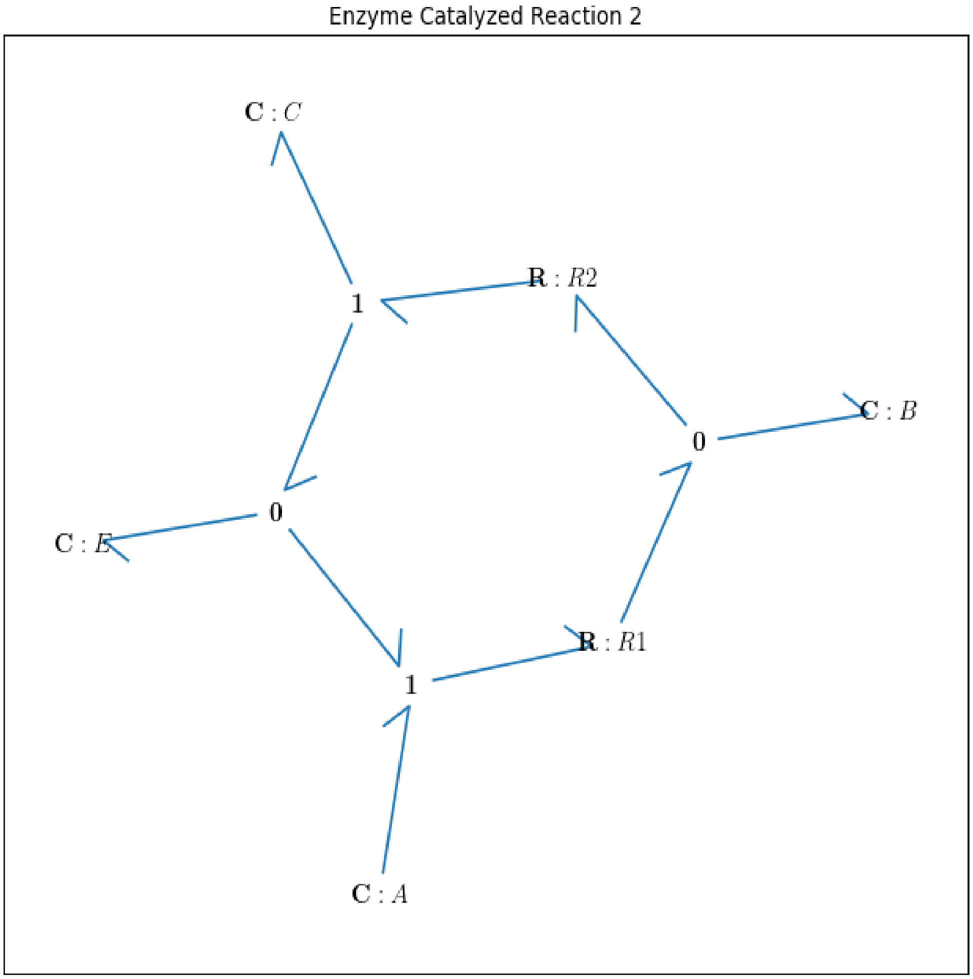


Fig. 1.149: Bondgraph diagram representing Enzyme Catalysed Reaction #2

Now that the bond graph demonstration of the system is done, we can illustrate its behaviour during a specific time interval and with arbitrary initial conditions for the state variables. The constitutive relations of the model can be shown as well:

```
model.state_vars
Out[ ]: {'x_0': (C: A, 'q_0'),
'x_1': (C: B, 'q_0'),
'x_2': (C: E, 'q_0'),
'x_3': (C: C, 'q_0')}
```

==>

```
model.constitutive_relations
Out[ ]: [dx_0 + 20*u_0*x_0*x_2 - 20*u_0*x_1,
dx_1 - 20*u_0*x_0*x_2 + 20*u_0*x_1 + 20*u_1*x_1 - 20*u_1*x_2*x_3,
```

(continues on next page)

(continued from previous page)

```
dx_2 + 20*u_0*x_0*x_2 - 20*u_0*x_1 - 20*u_1*x_1 + 20*u_1*x_2*x_3,
dx_3 - 20*u_1*x_1 + 20*u_1*x_2*x_3]
```

By using the command *"bgt.simulate"* and entering the model, time interval, and the initial conditions one can plot the time behaviour of the system. x vs t can be plotted by importing *"matplotlib.pyplot"* (q_{Ce_A} & q_{Ce_B} & q_{Ce_C} & q_{Ce_E} are state variables of the system which represent the molar amount in each solute/enzyme):

```
import matplotlib.pyplot as plt
x0 = {"x_0":1, "x_1":1, "x_2":1, "x_3":0.001}
t_span = [0,10]

for c, kappa, label in [('r', 0.005, 'kappa=0.005'), ('b', 0.01, 'kappa=0.01'), ('g',␣
↪0.1,'kappa=0.1')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa, "u_1
↪":kappa})
    plt.plot(t,x[:,0], c, label=label)
    plt.title('"Solute A"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
plt.show()


for c, kappa, label in [('r', 0.005, 'kappa=0.005'), ('b', 0.01, 'kappa=0.01'), ('g',␣
↪0.1,'kappa=0.1')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa, "u_1
↪":kappa})
    plt.plot(t,x[:,1], c, label=label)
    plt.title('"Solute B"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
plt.show()

for c, kappa, label in [('r', 0.005, 'kappa=0.005'), ('b', 0.01, 'kappa=0.01'), ('g',␣
↪0.1,'kappa=0.1')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa, "u_1
↪":kappa})
    plt.plot(t,x[:,2], c, label=label)
    plt.title('"Enzyme"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
    plt.legend(loc='upper right')
    plt.grid()
plt.show()

for c, kappa, label in [('r', 0.005, 'kappa=0.005'), ('b', 0.01, 'kappa=0.01'), ('g',␣
↪0.1,'kappa=0.1')]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa, "u_1
↪":kappa})
    plt.plot(t,x[:,3], c, label=label)
    plt.title('"Solute C"')
    plt.xlabel("time (s)")
    plt.ylabel("molar amount (mol/m3)")
```

(continues on next page)

(continued from previous page)

```
    plt.legend(loc='upper right')
    plt.grid()
plt.show()
```

Three different amounts for the control variable (*kappa*) are considered to show its impact on the molar amount of each solute/enzyme during the reaction period.

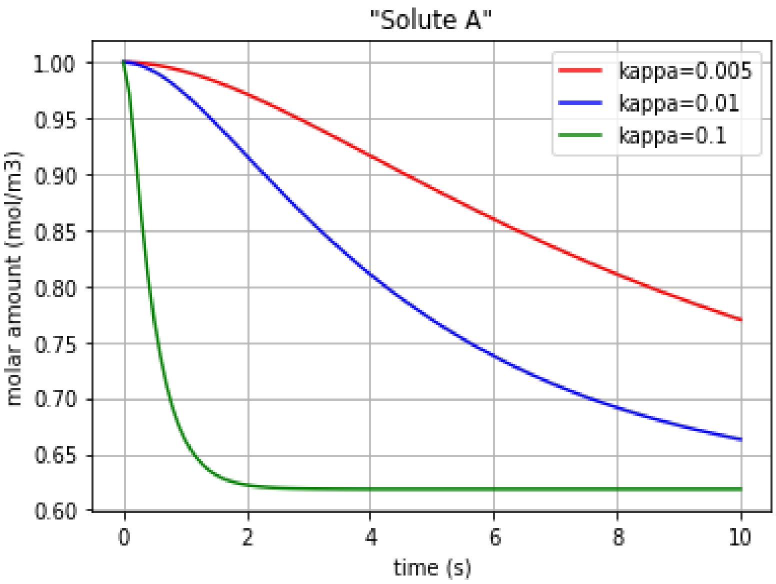


Fig. 1.150: Molar amount of solute A during an enzyme catalysed reaction with 3 different amounts for ‘kappa’

* Note that since the enzyme E is now both produced and consumed.

The molar concentration flow rate (v) of a solute is the time derivative of **q**:

$$\frac{dq}{dt} = v$$

and it can be plotted by converting the considered state variable (either x[:,0], x[:,1], x[:,2] or x[:,3]) to an array by importing the *numpy library* and then calculating its gradient with 0.1 steps. The *kappa=0.02* is chosen for a better distinction:

```
import matplotlib.pyplot as plt
import numpy as np
kappa=0.02
t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa, "u_1
↪":kappa})
f = np.array(x[:,0], dtype=float)
v_Ce_A=np.gradient(f,0.1)

f = np.array(x[:,1], dtype=float)
slope=np.gradient(f,0.1)
v_Ce_B=slope

f = np.array(x[:,2], dtype=float)
slope=np.gradient(f,0.1)
v_Ce_E=slope
```

(continues on next page)

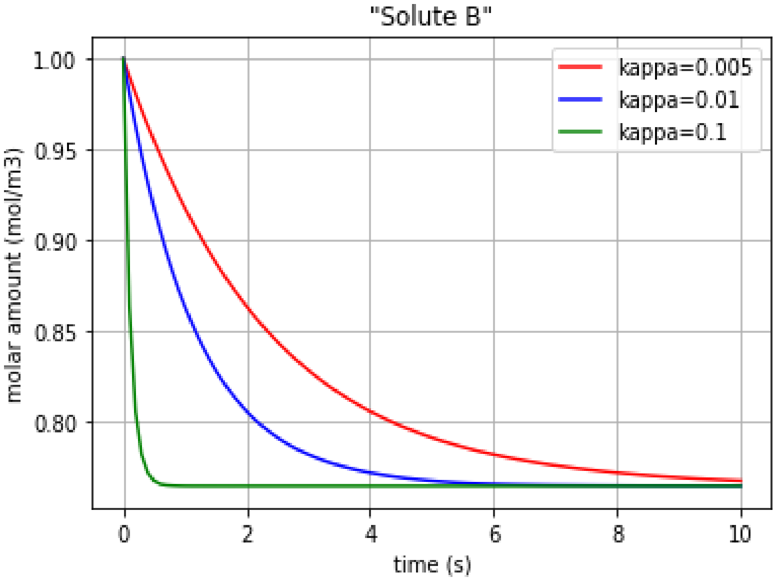


Fig. 1.151: Molar amount of solute B during an enzyme catalysed reaction with 3 different amounts for 'kappa'

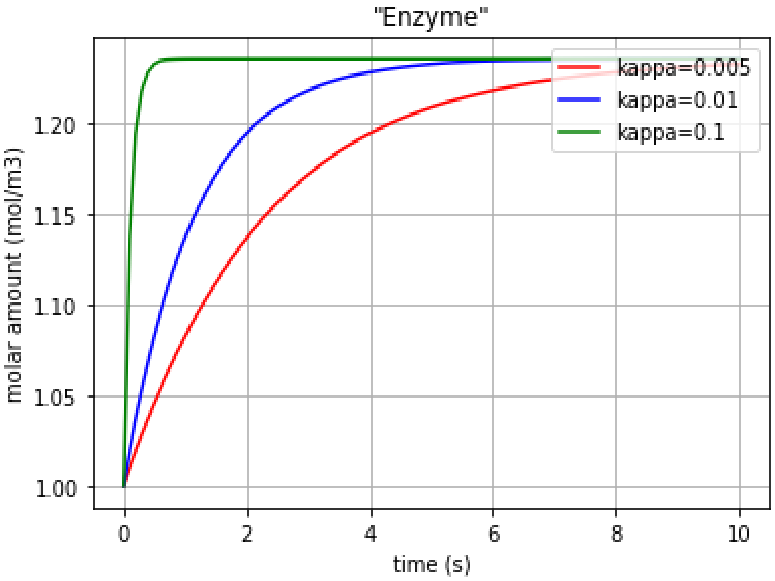


Fig. 1.152: Molar amount of the enzyme E during an enzyme catalysed reaction with 3 different amounts for 'kappa'

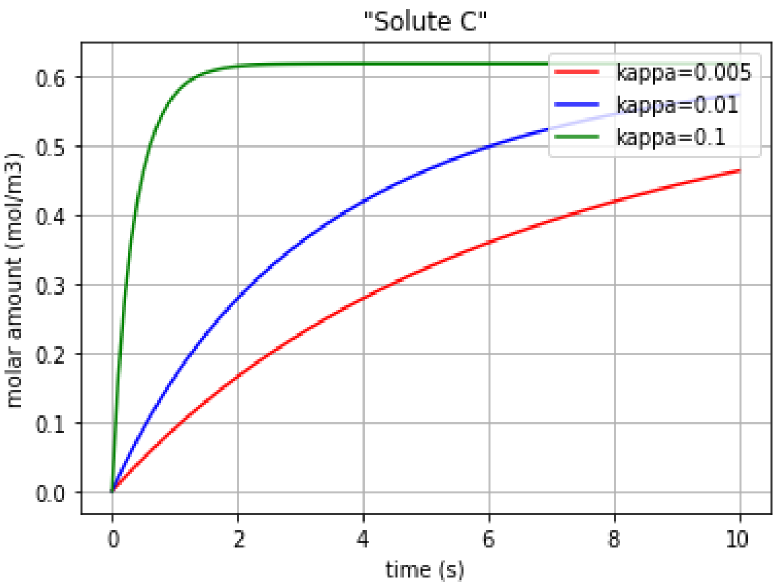


Fig. 1.153: Molar amount of solute C during an enzyme catalysed reaction with 3 different amounts for ‘kappa’

(continued from previous page)

```
f = np.array(x[:,3], dtype=float)
slope=np.gradient(f,0.1)
v_Ce_C=slope
```

Plotting the molar concentration flow rates of the solutes/enzyme for *kappa=0.02*:

```
plt.plot(t,v_Ce_A, 'r', label='v_Ce_A')
plt.plot(t,v_Ce_B, 'b', label='v_Ce_B')
plt.plot(t,v_Ce_C, 'c', label='v_Ce_C')
plt.plot(t,v_Ce_E, 'k', label='v_Ce_E')

leg1=plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("molar concentration flow rate (mol/m3/s)")
plt.title('kappa=0.02')
plt.grid()
plt.show()
```

The molar concentration flow rate of each solute/enzyme is depicted in the following figure:

Moreover, the chemical potential in each solute/enzyme can be calculated based on its constitutive equations:

Solute: $u = R.T.ln(K_s.q)$

where u is the chemical potential of the solute $[J/mol]$, K_s is the species thermodynamic constant $[mol^{-1}]$, *R* is the ideal gas constant $[J/K/mol]$, *T* is the absolute temperature $[K]$ and q is the molar concentration[4].

Thus, the u in each solute/enzyme is calculated for a sample reaction rate of *kappa=0.02*:

```
kappa=0.02
t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa, "u_1
↪":kappa})
```

(continues on next page)

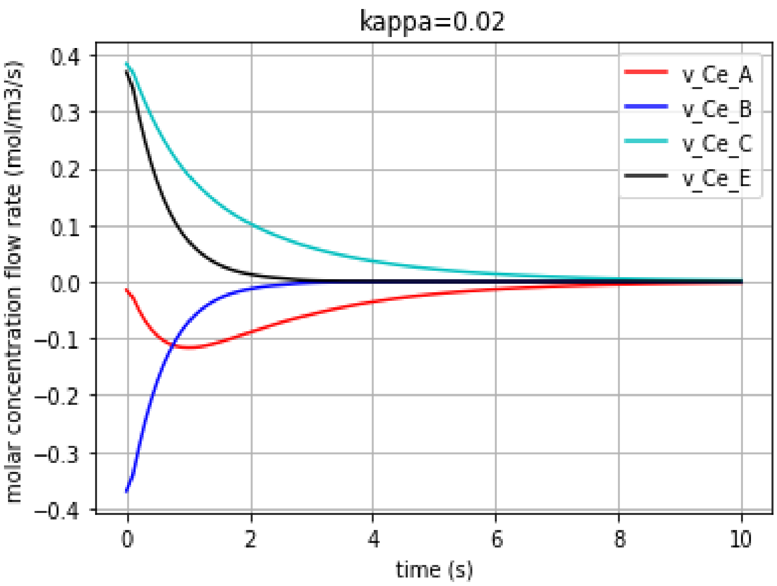


Fig. 1.154: Molar concentration flow rate of solutes A & B & C and the enzyme E with kappa=0.02.

(continued from previous page)

```

q_Ce_A = np.array(x[:,0], dtype=float)
u_Ce_A=R*T*np.log(K_A*q_Ce_A)

q_Ce_B = np.array(x[:,1], dtype=float)
u_Ce_B=R*T*np.log(K_B*q_Ce_B)

q_Ce_E = np.array(x[:,2], dtype=float)
u_Ce_E=R*T*np.log(K_E*q_Ce_E)

q_Ce_C = np.array(x[:,3], dtype=float)
u_Ce_C=R*T*np.log(K_C*q_Ce_C)
```

The time variation of the corresponding chemical potential for each solute/enzyme (u_{Ce_A} & u_{Ce_B} & u_{Ce_E} & u_{Ce_C}) can be all plotted in a figure:

```
plt.plot(t,u_Ce_A, 'm', label='u_Ce_A')
plt.plot(t,u_Ce_B, 'c', label='u_Ce_B')
plt.plot(t,u_Ce_C, 'y', label='u_Ce_C')
plt.plot(t,u_Ce_E, 'k', label='u_Ce_E')
plt.legend(loc='lower right')
plt.xlabel("time (s)")
plt.ylabel("Chemical potential (J/mol)")
plt.title('Chemical potential of the solutes')
plt.grid()
```

which results in:

At this stage we would like to demonstrate the relation between the molar amount of the substrate “A” and the molar flow rate of producing the product “C”. * Note that by the declining of the molar amount of the substrate “A”, the production rate of “C” reduces:

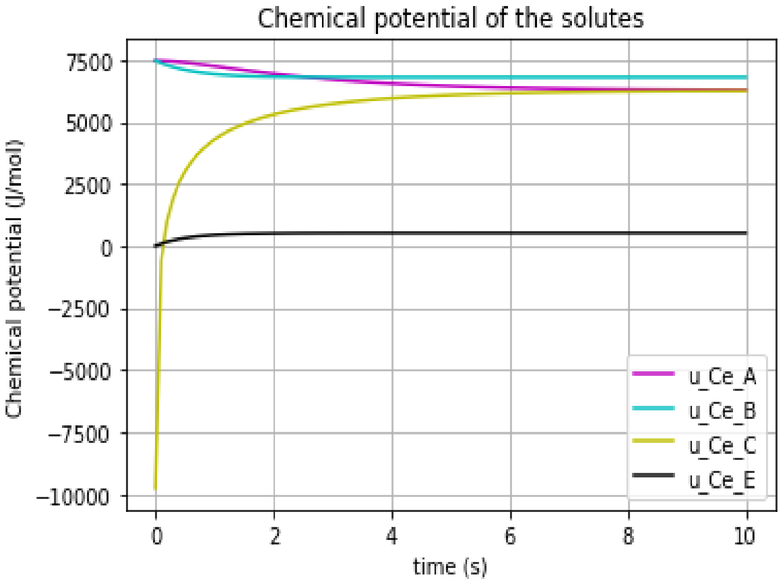


Fig. 1.155: Chemical potential of each solute/enzyme (A,B,C,E) vs time

```
plt.plot(q_Ce_A,v_Ce_C, '*-g')
plt.xlabel("molar amount of solute A (mol/m3)")
plt.ylabel("molar concentration flow rate of solute C (mol/m3/s)")
plt.grid()
```

Click to read Enzyme Catalysed Reaction #2 codes

Reaction with Mixed Stoichiometry

The definitions in this document are adopted from[2] and[5] .

The rationale behind the following codes are explained thoroughly in *"Diffusion"* and *"Simple reaction"* documentation.

```
import BondGraphTools as bgt
model = bgt.new(name="Reaction with mixed stoichiometry")
K_A=20
K_B=20
K_C=20
K_D=20

R=8.314
T=300

Ce_A = bgt.new("Ce", name="A", library="BioChem", value={'k':K_A , 'R':R, 'T':T})
Ce_B= bgt.new("Ce", name="B", library="BioChem", value={'k':K_B, 'R':R, 'T':T})
Ce_C= bgt.new("Ce", name="C", library="BioChem", value={'k':K_C, 'R':R, 'T':T})
Ce_D= bgt.new("Ce", name="D", library="BioChem", value={'k':K_D, 'R':R, 'T':T})

TF_1_ratio=0.5
TF_1=bgt.new("TF", value=TF_1_ratio)
```

(continues on next page)

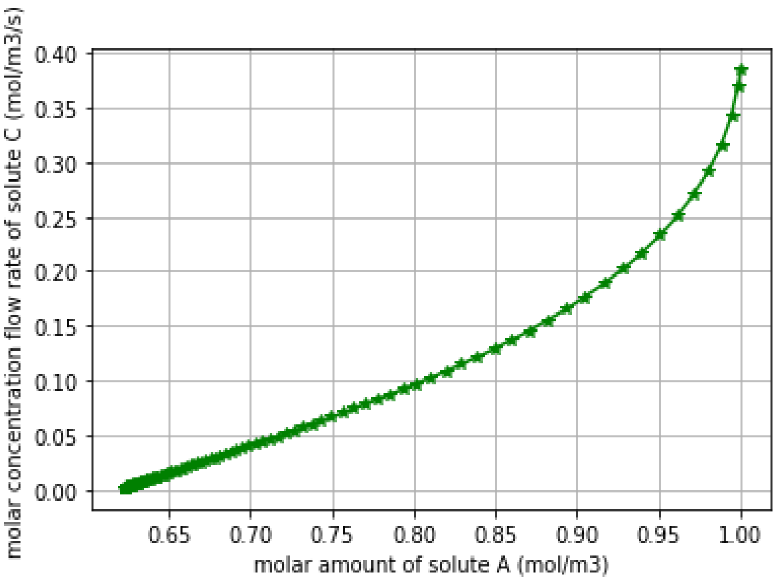


Fig. 1.156: Molar concentration flow rate of solute C vs the molar amount of solute A

Fig. 1.157: Schematic of Reaction with Mixed Stoichiometry.

(continued from previous page)

```
TF_2_ratio=0.5
TF_2=bgt.new("TF", value=TF_2_ratio)


reaction = bgt.new("Re", library="BioChem", value={'r':None, 'R':R, 'T':T})

zero_junction_1 = bgt.new("0")
zero_junction_2 = bgt.new("0")
zero_junction_3 = bgt.new("0")
zero_junction_4 = bgt.new("0")

one_junction_1 = bgt.new("1")
one_junction_2 = bgt.new("1")

bgt.add(model, Ce_A, Ce_B, Ce_C, Ce_D, TF_1, TF_2, zero_junction_1, zero_junction_2,␣
↪zero_junction_3, zero_junction_4, one_junction_1, one_junction_2, reaction)

bgt.connect(Ce_A, zero_junction_1)
bgt.connect(zero_junction_1,one_junction_1)
bgt.connect(Ce_B,zero_junction_2)
bgt.connect(zero_junction_2,(TF_1,0))
bgt.connect((TF_1,1),one_junction_1)
bgt.connect(one_junction_1,reaction)
bgt.connect(reaction,one_junction_2)
bgt.connect(one_junction_2,zero_junction_3)
bgt.connect(zero_junction_3,Ce_C)
bgt.connect(one_junction_2,(TF_2,0))
bgt.connect((TF_2,1),zero_junction_4)
bgt.connect(zero_junction_4,Ce_D)
```

The reaction rate 'r' is set to *"None"* in order to change it inside a *for* loop. * Note that when dealing with mixed stoichiometry in bond graphs, one must take the advantages of using the **Transformer** component (TF), while the **TF** is a power conserving transformation[4].

By drawing the model, one can see if the components are connected properly to each other or not:

```
bgt.draw(model)
```

A sketch of the network will then be produced:

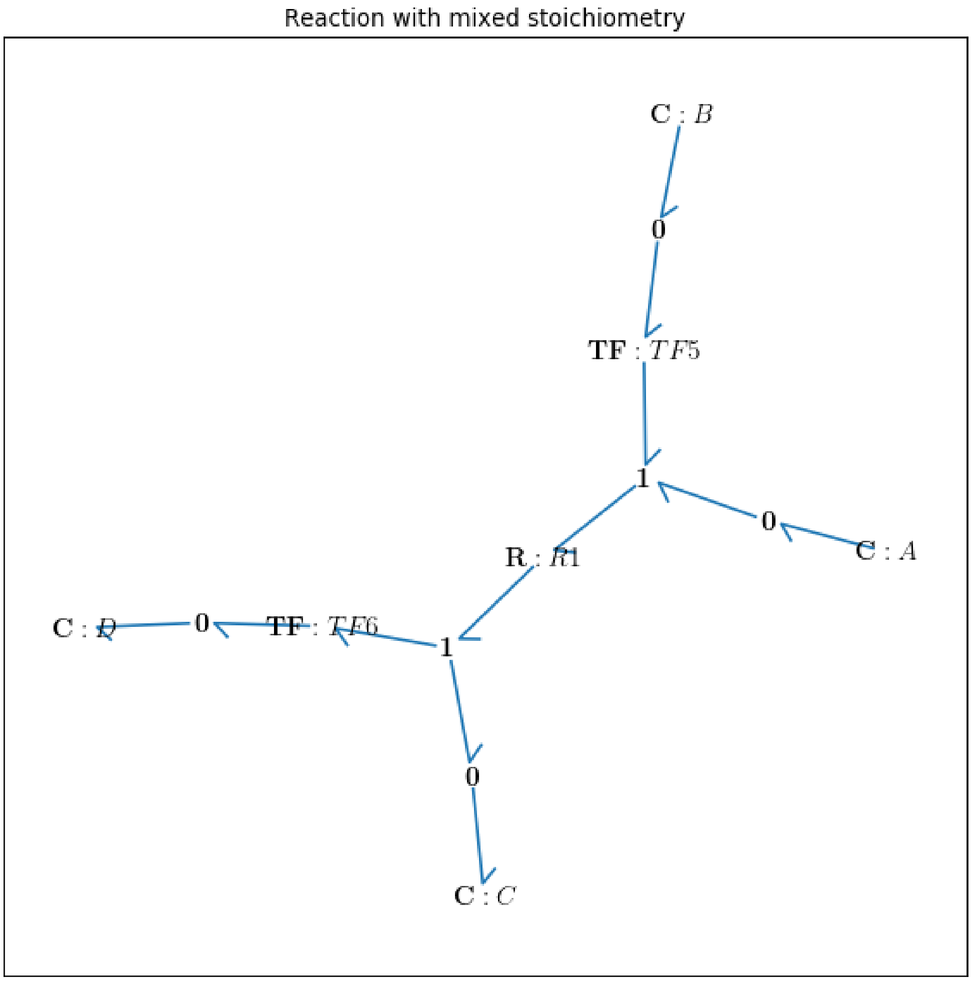


Fig. 1.158: Bondgraph diagram representing Reaction with Mixed Stoichiometry

Now that the bond graph demonstration of the system is done, we can illustrate its behaviour during a specific time interval and with arbitrary initial conditions for the state variables. The constitutive relations of the model can be shown as well:

```
model.state_vars
Out[ ]: {'x_0': (C: A, 'q_0'),
'x_1': (C: B, 'q_0'),
'x_2': (C: C, 'q_0'),
'x_3': (C: D, 'q_0')}
```

==>

```
model.constitutive_relations
Out[ ]: [dx_0 + 223606797749979*u_0*x_0*sqrt(x_1)/2500000000000 - 8000*u_0*x_2*x_3**2,
dx_1 + 223606797749979*u_0*x_0*sqrt(x_1)/5000000000000 - 4000*u_0*x_2*x_3**2,
dx_2 - 223606797749979*u_0*x_0*sqrt(x_1)/2500000000000 + 8000*u_0*x_2*x_3**2,
dx_3 - 178885438199983*u_0*x_0*sqrt(x_1)/1000000000000 + 16000*u_0*x_2*x_3**2]
```

By using the command *"bgt.simulate"* and entering the model, time interval, and the initial conditions one can plot the time behaviour of the system. x vs t can be plotted by importing *"matplotlib.pyplot"* (q_{Ce_A} & q_{Ce_B} & q_{Ce_C} & q_{Ce_D} are state variables of the system which represent the molar amount in each solute). Here *kappa=0.001* is chosen for a better demonstration:

```
import matplotlib.pyplot as plt
x0 = {"x_0":1, "x_1":1, "x_2":0.001, "x_3":0.001}
t_span = [0,6]
kappa=0.001
t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={"u_0":kappa})

for c, i, label in [('r', 0, 'Solute A'), ('g', 1, 'Solute B'), ('b', 2, 'Solute C'),␣
↪('y', 3, 'Solute D')]:
    plt.plot(t,x[:,i], c, label=label)
plt.xlabel("time (s)")
plt.ylabel("molar amount (mol/m3)")
plt.legend(loc='center right')
plt.grid()
```

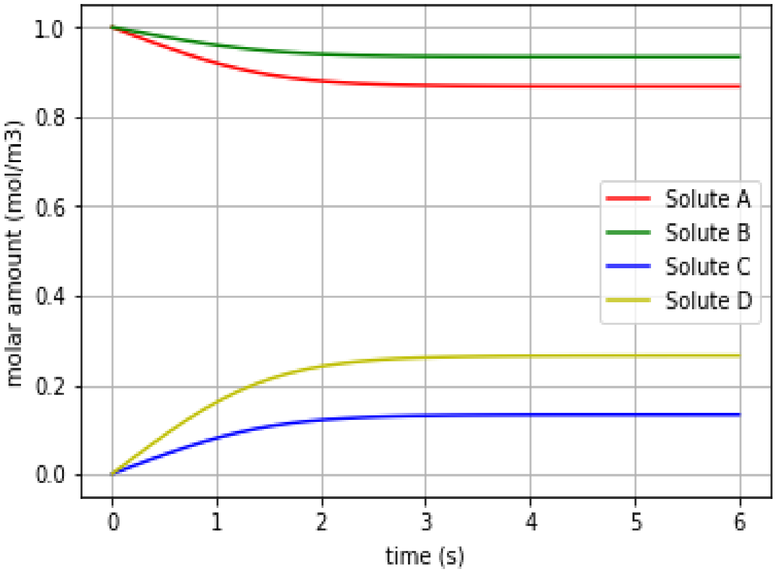


Fig. 1.159: Molar amount of the solutes A, B, C & D during a reaction with mixed stoichiometry for 'kappa=0.001'

* Note that the final molar amounts of the solutes B & D are twice as much as the final molar amounts of the solutes A & C, respectively.

The molar concentration flow rate (v) of a solute is the time derivative of **q**:

$$\frac{dq}{dt} = v$$

and it can be plotted by converting the considered state variable (either x[:,0], x[:,1], x[:,2] or x[:,3]) to an array by importing the *numpy library* and then calculating its gradient with 0.1 steps:

```
import matplotlib.pyplot as plt
import numpy as np

for c, i, label in [('r', 0, 'v_Ce_A'), ('g', 1, 'v_Ce_B'), ('b', 2, 'v_Ce_C'), ('y',␣
↪3, 'v_Ce_D')]:
    f = np.array(x[:,i], dtype=float)
    slope=np.gradient(f,0.1)

    plt.plot(t,slope, c, label=label)

plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("molar concentration flow rate (mol/m3/s)")
plt.grid()
plt.show()
```

The molar concentration flow rate of each solute is depicted in the following figure:

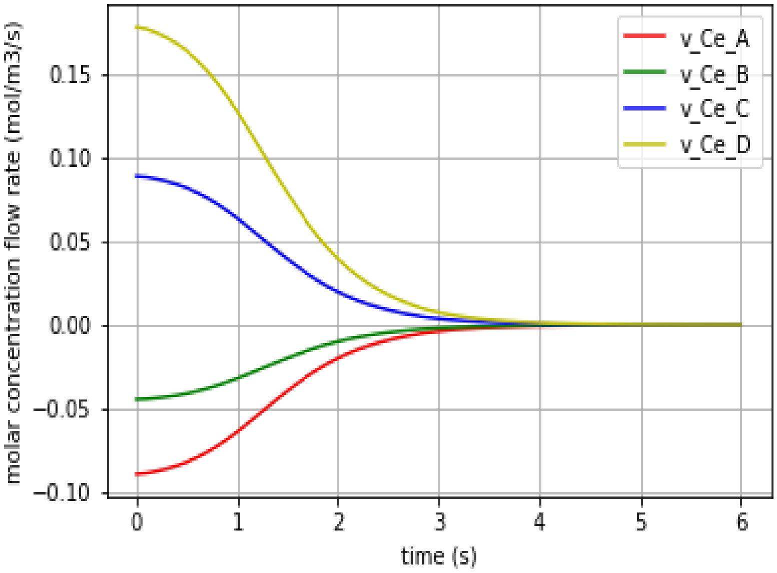


Fig. 1.160: Molar concentration flow rate of solutes A & B & C and D with kappa=0.001.

Moreover, the chemical potential in each solute can be calculated based on its constitutive equations:

Solute: $u = R.T.ln(K_s.q)$

where u is the chemical potential of the solute $[J/mol]$, K_s is the species thermodynamic constant $[mol^{-1}]$, *R* is the ideal gas constant $[J/K/mol]$, *T* is the absolute temperature $[K]$ and q is the molar concentration[4].

Thus, the u in each solute is calculated for a sample reaction rate of *kappa=0.001*. The time variation of the corresponding chemical potential for each solute (u_{Ce_A} & u_{Ce_B} & u_{Ce_C} & u_{Ce_D}) can be all plotted in a figure:

```
for c, i, k, label in [('r',0,K_A,'u_Ce_A'), ('g',1,K_B,'u_Ce_B'), ('b',2,K_C,'u_Ce_C
↪'), ('y',3,K_D,'u_Ce_D')]:
    q= np.array(x[:,i], dtype=float)
    u=R*T*np.log(k*q)
    plt.plot(t,u, c, label=label)

```

(continues on next page)

(continued from previous page)

```
plt.legend(loc='lower right')
plt.xlabel("time (s)")
plt.ylabel("Chemical potential (J/mol)")
plt.title('Chemical potential of the solutes')
plt.grid()
```

which results in:

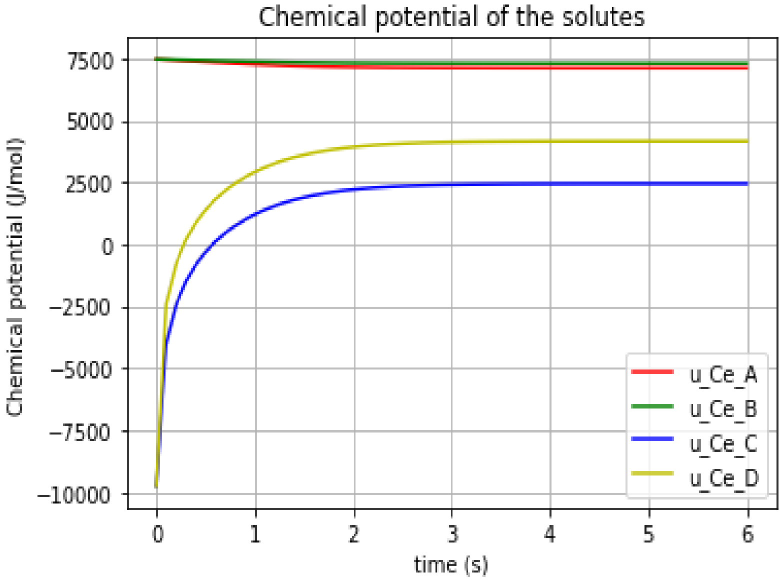


Fig. 1.161: Chemical potential of each solute (A,B,C,D) vs time

For scaling purpose, each pair of chemical potentials of the solutes [(A,B) , (C,D)] are plotted again the separate figures:

```
for c, i, k, label in [('r',0,K_A,'u_Ce_A'), ('g',1,K_B,'u_Ce_B')]:
    q= np.array(x[:,i], dtype=float)
    u=R*T*np.log(k*q)
    plt.plot(t,u, c, label=label)
plt.legend(loc='upper right')
plt.xlabel("time (s)")
plt.ylabel("Chemical potential (J/mol)")
plt.title('Chemical potential of the solutes')
plt.grid()
plt.show()

for c, i, k, label in [('b',2,K_C,'u_Ce_C'), ('y',3,K_D,'u_Ce_D')]:
    q= np.array(x[:,i], dtype=float)
    u=R*T*np.log(k*q)
    plt.plot(t,u, c, label=label)
plt.legend(loc='lower right')
plt.xlabel("time (s)")
plt.ylabel("Chemical potential (J/mol)")
plt.title('Chemical potential of the solutes')
plt.grid()
plt.show()
```

Then the comparison would be more straightforward:

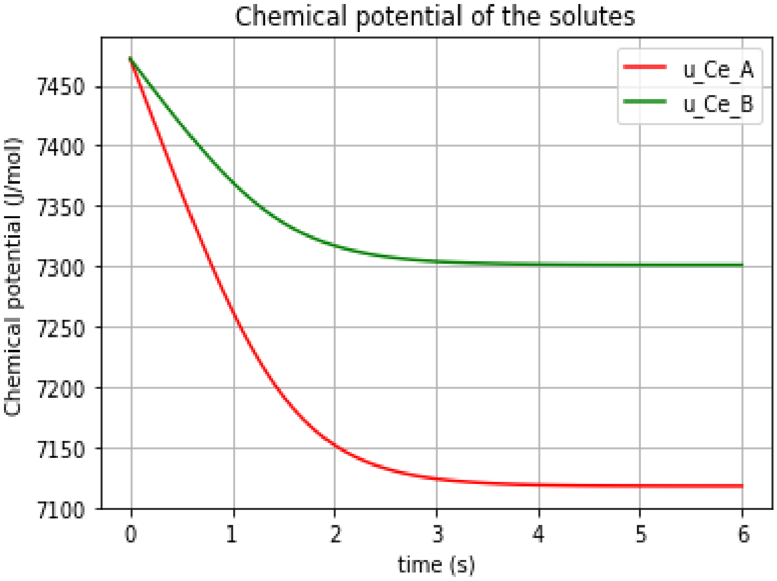


Fig. 1.162: Chemical potential of the solutes A & B vs time

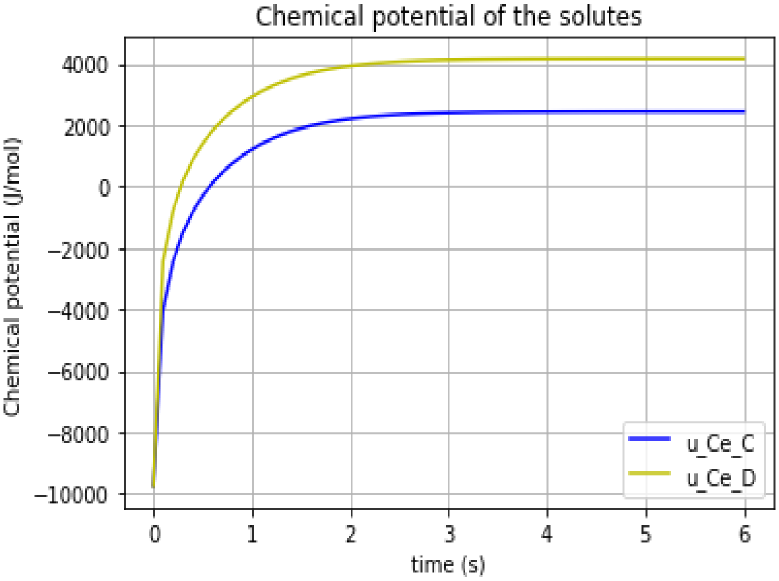


Fig. 1.163: Chemical potential of the solutes C & D vs time

Click to read Reaction with Mixed Stoichiometry codes

1.5 Projects

In these projects we attempt to use "real world" scenarios to demonstrate the process of creating your own workflows, making use of your experience working through the *Clinical Workflows* tasks.

1.5.1 Project: Parameter Estimation

This project was created as part of the Computational Physiology module in the MedTech CoRE Doctoral Training Programme.

This project requires you to put together what you have learned in the tutorials to define a complete workflow which we will use to manually optimise the material properties of a tissue model for use in mechanical simulations.

Outline

In this project we recreate a typical workflow that is often performed by scientists when creating a model. The standard cardiac workflow used in the DTP module is shown in Fig. 1.164, which can be abstracted into the generic workflow shown in Fig. 1.165.

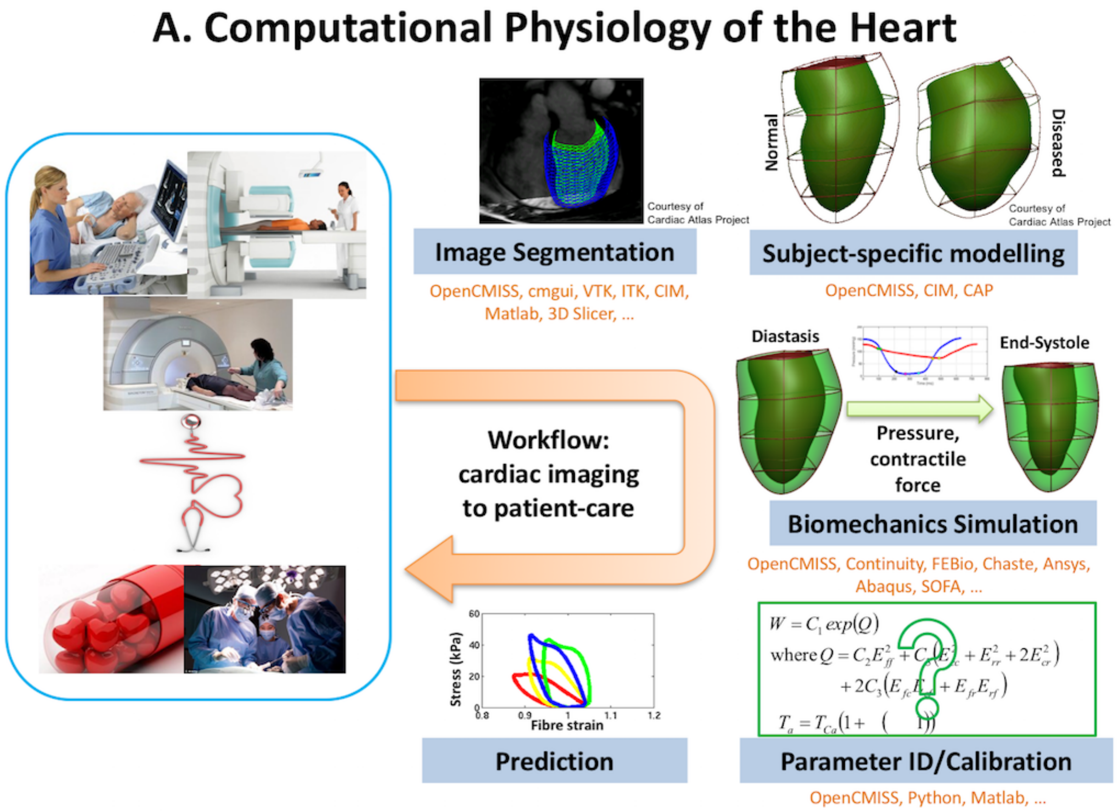
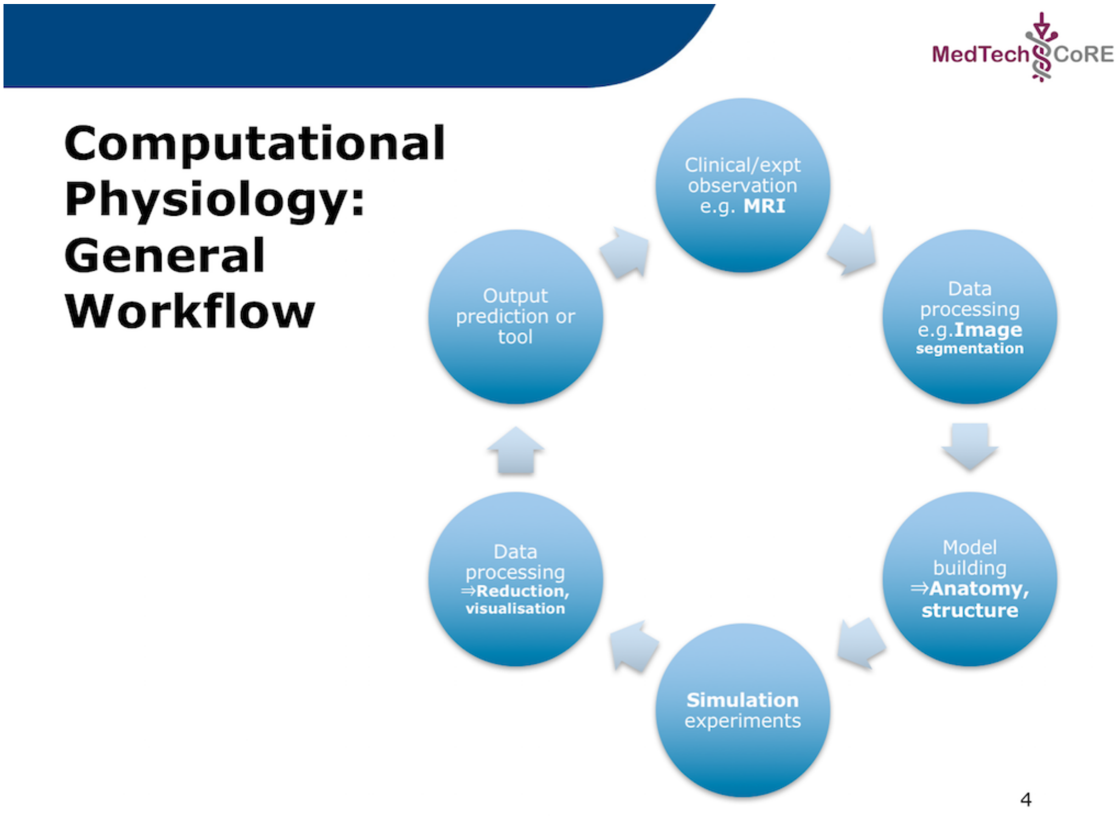

A. Computational Physiology of the Heart
Courtesy of Cardiac Atlas Project
Normal
Diseased
Courtesy of Cardiac Atlas Project
Image Segmentation
OpenCMISS, cmgui, VTK, ITK, CIM, Matlab, 3D Slicer, ...
Subject-specific modelling
OpenCMISS, CIM, CAP
Diastasis
End-Systole
Pressure, contractile force
Workflow: cardiac imaging to patient-care
Biomechanics Simulation
OpenCMISS, Continuity, FEBio, Chaste, Ansys, Abaqus, SOFA, ...
$W = C_1 exp(Q)$
where $Q = C_2E_{ff}^2 + C_3(E_{cc}^2 + E_{rr}^2 + 2E_{cr}^2)$
$+2C_3(E_{fc}E_{cf} + E_{fr}E_{rf})$
$T_a = T_{Ca}(1+ \;\;\;))$
?
60
40
20
0
0.8
0.9
1
1.1
1.2
Stress (kPa)
Fibre strain
Prediction
Parameter ID/Calibration
OpenCMISS, Python, Matlab, ...

Fig. 1.164: The standard example cardiac workflow used in the DTP Computational Physiology module.



MedTech CoRE

Computational Physiology: General Workflow

Clinical/expt observation e.g. **MRI**

Data processing e.g.**Image segmentation**

Model building ⇒**Anatomy, structure**

Simulation experiments

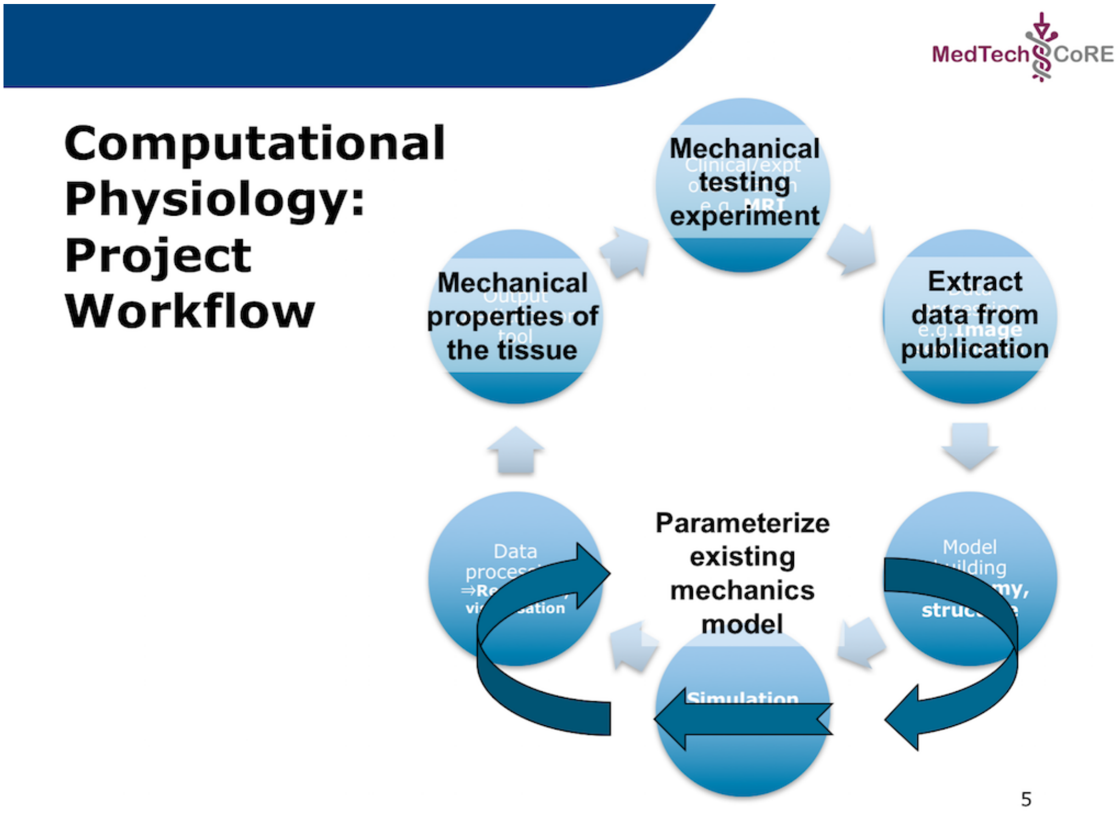
Data processing ⇒**Reduction, visualisation**

Output prediction or tool

4

Fig. 1.165: The generic DTP Computational Physiology workflow.

In this project, we adapt the generic workflow shown in Fig. 1.165 to the specific scenario we are recreating. The specific workflow for this project is described in Fig. 1.166.



Computational Physiology: Project Workflow

MedTech CoRE
Mechanical testing experiment
Extract data from publication
Mechanical properties of the tissue
Parameterize existing mechanics model
Simulation
Data
Model
5

Fig. 1.166: The specific workflow for this project. The output for this workflow is to predict the passive mechanical material properties of a piece of cardiac tissue given the results from a mechanical testing experiment. As is commonly done, the actual experimental data will be extracted from a published paper where the actual data is only available as a printed figure. The extracted data will be used to predict the material properties of an existing cardiac tissue model.

Geometric model

As you may know, caridac tissue consists of cells aligned in fibres, as shown in Fig. 1.167. Mechanically, the tissue is much stiffer in the fibre direction than in the cross-fibre direction, and so it is very important for any model of cardiac tissue to take this into account.

In this project, we use the simplified geometric model shown in Fig. 1.168. While this is a relatively trivial model, it is a reasonable approximation to an often used experimental preparation - the cardiac trabecula.

Fig. 1.169 shows some simulation results when performing some passive stretch experiments using the tissue model from Fig. 1.168. These results illustrate the difference in material properties when stretching the tissue in the fibre vs cross-fibre direction.

Data collection

The first step in this project is to collect the experimental data that will be used in estimating the material properties of this tissue. In this project we are using simulated experimental data so that we have some hope that this will be an

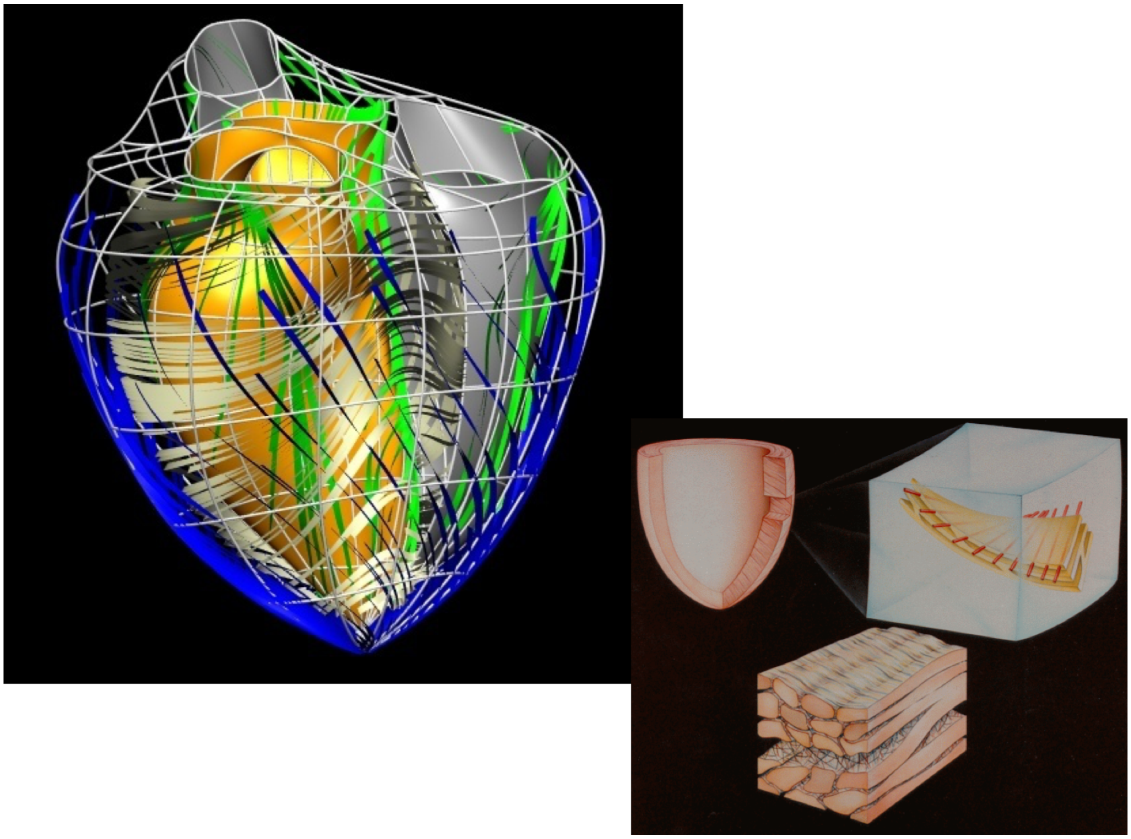


Fig. 1.167: Illustrations of the fibrous nature of cardiac tissue.

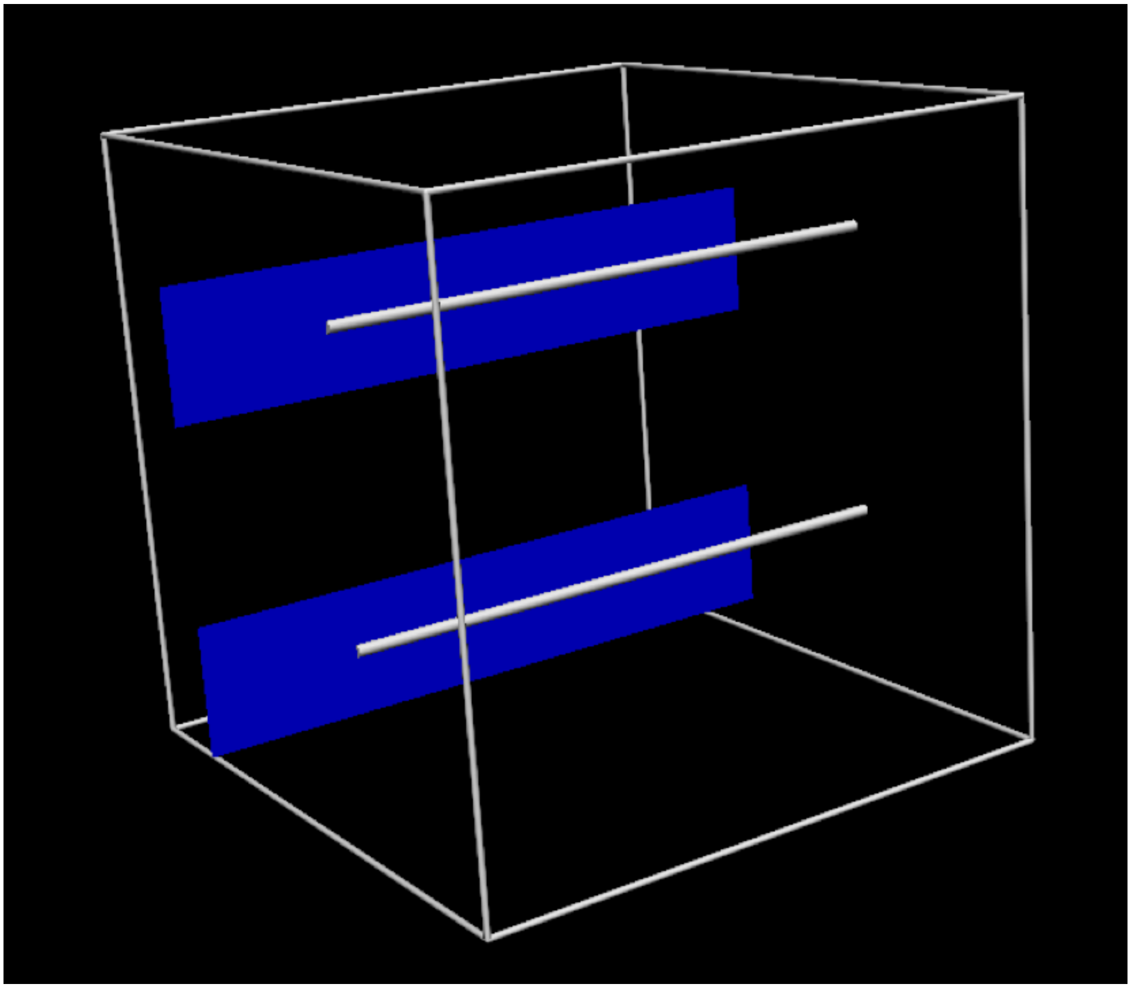


Fig. 1.168: The specific geometric model used in this project. In this tissue model the muscle fibres are aligned in the same direction, indicated by the silver line. The blue plane indicates the orthogonal direction.

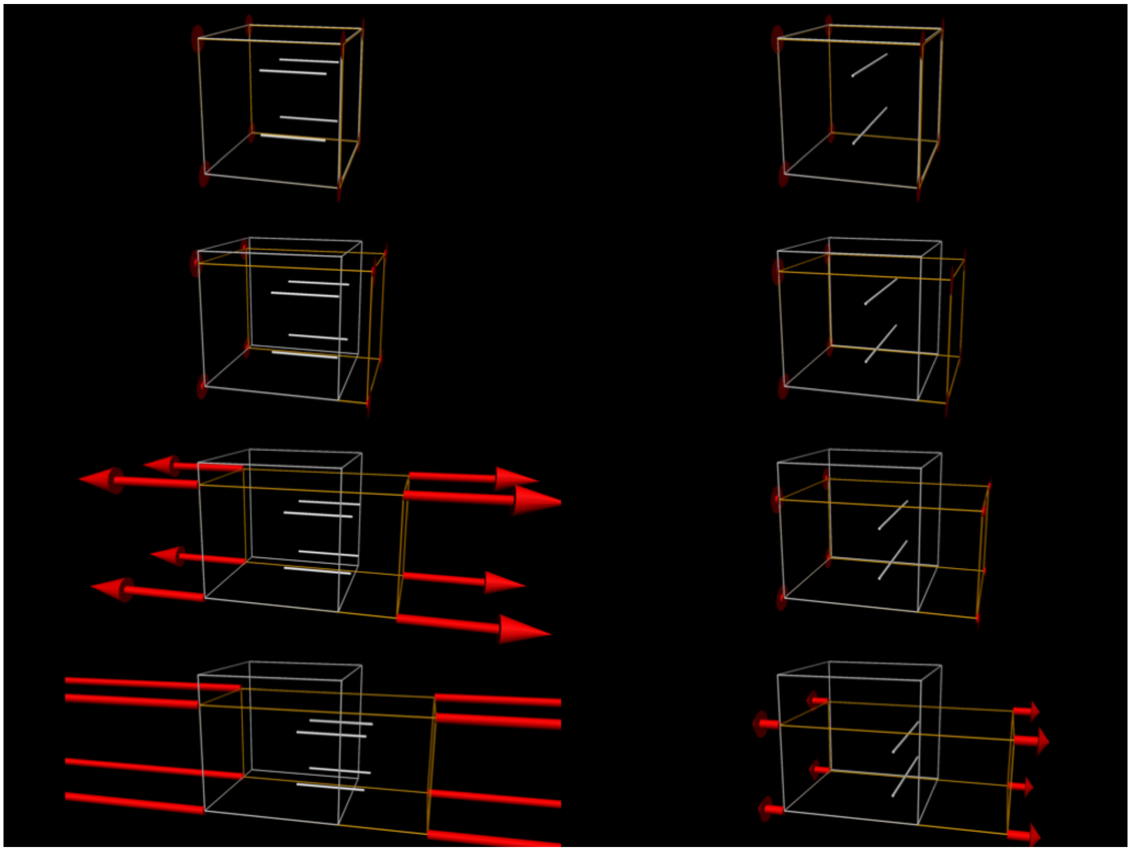


Fig. 1.169: Simulation results from performing passive stretch experiments with our cardiac tissue model. The results on the left show the resulting deformation and reaction forces when stretching the tissue in the direction of the tissue fibres, while those on the right show the results when stretching in the cross-fibre direction. Each pair of results is the same applied stretch. In particular, notice the much larger reaction forces in the fibre stretches - i.e., much more energy needs to be applied in order to stretch the tissue along the fibres compared to across the fibres for the same magnitude stretch.

achievable task. You can see typical experimental data from a real cardiac trabecula that would be used in a lab here: https://youtu.be/_VHZyPEpxsc.

Since our model is homogeneous and transversely isotropic, we can reduce the data required to parameterise the model to two stress-strain relationships - one for the fibre direction and one of the orthogonal cross-fibre direction (see Fig. 1.169). Example data similar to what we will use is shown in Fig. 1.170 and a potential segmentation method used to extract the numerical values of the data is shown in Fig. 1.171.

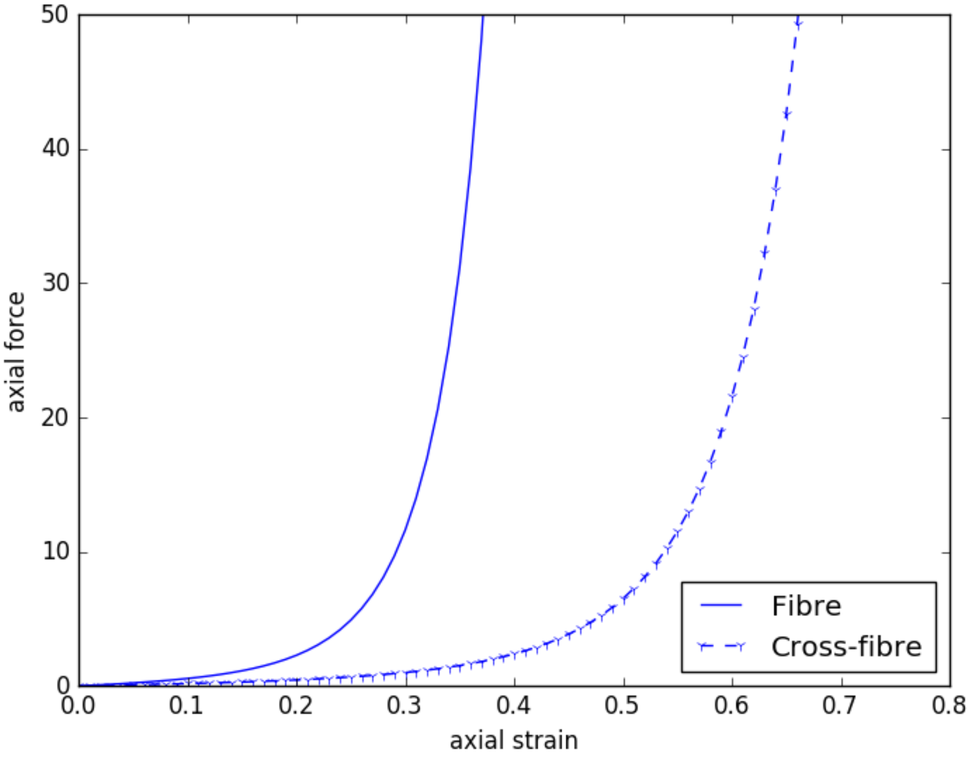


Fig. 1.170: Example "experimental" data that could be used to estimate the material properties of our cardiac tissue model. This is not the data to be used in this project.

In this project, you will need to segment the stress-strain data available in this high-impact scientific paper: https://doi.org/10.17608/k6.auckland.9810233.v1. You may use any method you like to segment the data. When extracting the numerical values, you will need to collect **5 data points** for each of the fibre and cross-fibre relationships.

Constitutive model

For this project, we want to predict the passive material properties of the cardiac tissue given the observed experimental data in Fig. 1.170. So in addition to the geometric model in Fig. 1.168 we need a material constitutive model that captures the cardiac tissue properties that we are interested in. For this project we will use the Guccione model. This model, and many other potential models to use, is available in the Physiome Repository - https://models.physiomeproject.org/e/26d/guccione.cellml/view. **You will need to download the CellML file for this model from the repository, make sure you save it somewhere convenient.**

The equations for the Guccione model are shown in Fig. 1.172 (you can see the exact equations used in the model repository - https://models.physiomeproject.org/e/26d/guccione.cellml/@@cellml_math).

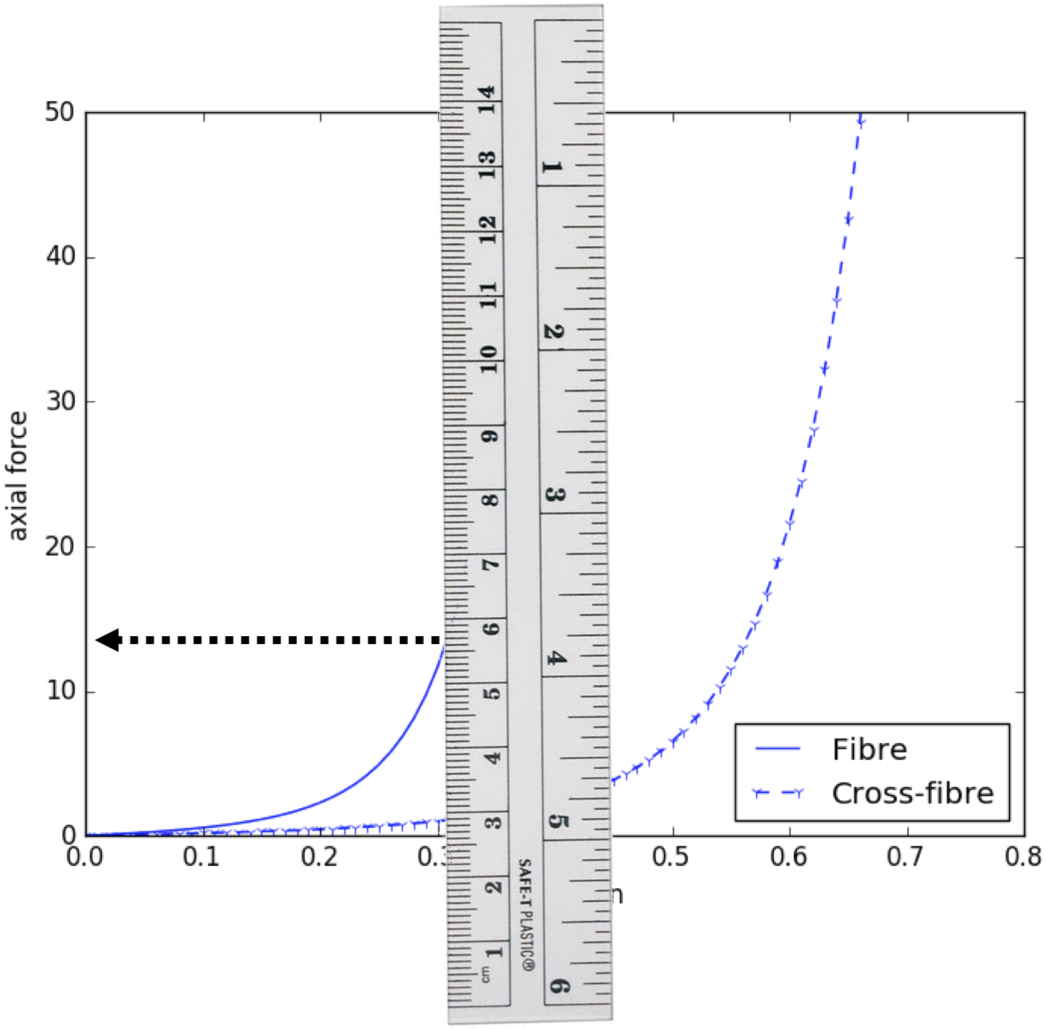


Fig. 1.171: Illustrating the “manual segmentation” method that could be used to obtain the actual experimental data.

$$T_{11} = c_1 \cdot e^Q \cdot c_2 \cdot E_{11} \qquad T_{12} = c_1 \cdot e^Q \cdot c_4 \cdot E_{12}$$

$$T_{22} = c_1 \cdot e^Q \cdot c_3 \cdot E_{22} \qquad T_{13} = c_1 \cdot e^Q \cdot c_4 \cdot E_{13}$$

$$T_{33} = c_1 \cdot e^Q \cdot c_3 \cdot E_{33} \qquad T_{23} = c_1 \cdot e^Q \cdot c_3 \cdot E_{23}$$

$$Q = c_2 \cdot E_{11}^2 + c_3 \cdot (E_{33}^2 + E_{22}^2 + E_{23}^2 \cdot 2) + 2 \cdot c_4 \cdot (E_{13}^2 + E_{12}^2)$$

9

Fig. 1.172: The equations of Guccione model defining the 6 components of the stress tensor, *T*, given the 6 components of the strain tensor, *E*. The four material parameters we want to estimate in this project are highlighted ($c_1 \ldots c_4$). The transverse isotropic nature of this model can be seen by comparing the equation for the fibre stress (T_{11}) to that for both of the the cross-fibre stresses (T_{22} and T_{33}). Understanding which material parameters are important for each of the stress-strain relationships will help guide your parameter estimation.

Workflow construction

Following the "manual segmentation" method, we will be using a manual parameter estimation method to predict the passive material properties of the cardiac tissue described in Fig. 1.170. Therefore, the second step in this project is to construct a workflow in MAP Client that will let you perform some simulations using the cardiac tissue model with your estimated material parameters, and then compare your simulation results to the data you measured.

You will need to start MAP Client and create a new workflow via the menu item *File* → *New* → *Workflow*. This just requires you to select a folder: create a new, empty folder, for example `estimationproject` on the Desktop, and select it.

For this project you will need to make use of the following MAP Client steps.

- *Iron Simulation* – to perform the actual simulation.
- *Simulation Review* – to compare simulated data to measured data.
- *Graph Segmentation* – to enter the numeric values extracted from the experimental data.
- *File Chooser* – to provide the CellML file for the Guccione model.
- *Directory Chooser* – to define an *output* folder to store the simulation results.
- *Parameter Setting* – to define your current estimate of the passive material properties.

We suggest starting with the *Iron Simulation* step and using the port definitions to determine the workflow connectivity. In the configuration of the workflow, you should be able to enter your current estimates for the four material parameters and the numeric values measured from the experimental data. The *Graph Segmentation* step expects 10 data points to be given – the first 5 should be the fibre direction and the second 5 the cross-fibre direction.

On execution, your workflow should perform the required simulations and present you with a comparison of your simulation results with the measured experimental data.

Parameter estimation

As mentioned, we will be using a manual estimation process in this project. This requires you to use your workflow to estimate parameter values and compare the predicted tissue behaviour to that you have measured from the experimental data. And then repeat until you are happy with the comparison.

Some tips that might help:

- the parameter values are whole numbers
- what is the main effect of each of the parameters?
- how accurate are your measured values?
- how accurate is the printed graph?

1.5.2 Project: Femur Fitting

This project was created as part of the Computational Physiology module in the MedTech CoRE Doctoral Training Programme.

This project requires you to put together what you have learned in the tutorials to define a complete workflow which will create a customised model of the distal (lower) end of a femur bone by fitting a generated mesh to points digitised off medical images, perform some computational simulation with it and visualise the result.

Outline

You will need to start MAP Client and create a new workflow via the menu item *File* → *New* → *Workflow*. This just requires you to select a folder: create a new, empty folder, for example “femurproject” on the Desktop, and select it.

At this point the MAP Client interface will show a blank project and a list of plugins you can use, as shown in Fig. 1.173:

For this project you will use the following plugins for steps in the workflow:

- **Image Source** - for selecting a folder of images to digitise
- **Segmentation (Manual)** - for digitising points in the image stack for segmenting the femur surface
- **Mesh Generator** - for generating one or more meshes to fit to the femur data
- **Mesh Merger** - for merging multiple meshes, if you choose to do this
- **smoothfit** - for fitting the input model to the digitised points
- **Load Femur** - for performing some heavy math on the fitted model
- **simpleviz** - for visualising the results

These are added to your workflow by dragging them into the working area (with the blue grid, like graph paper). Each step must be configured by clicking on its gear-shaped settings icon, which is initially red but changes to green when correctly configured. At a minimum you must give each step a unique name in this workflow (intermediate data is saved under this name), and set any other options for sources to the workflow. You will need to connect the input and output ports for each step.

Tips for completing the project

- For initial input to the Segmentation tool you will want to select folder `DTP-data/complete_workflow/knee_mri` (the location of the `DTP-data` folder will be given to you).
- When digitising points on the medical images, make sure you create adequate points to define the curved surfaces: it’s easy to underspecify the shape at the end of rounded body. You may wish to rotate the plane you’re digitising on to suit capturing the femur shape. It takes a while to digitise so routinely save the data, and a final time before clicking ‘done’. Next time you run the workflow you can click ‘load’ to reload it.

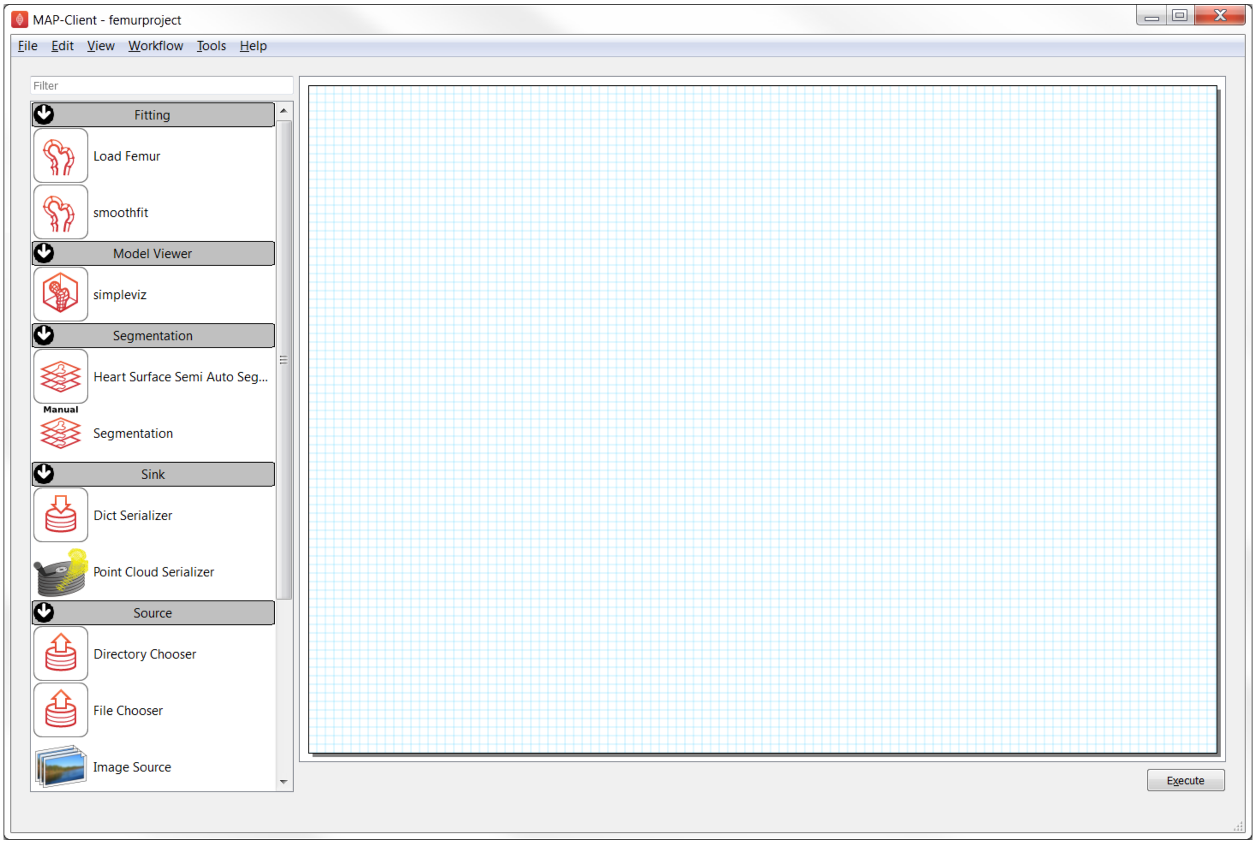
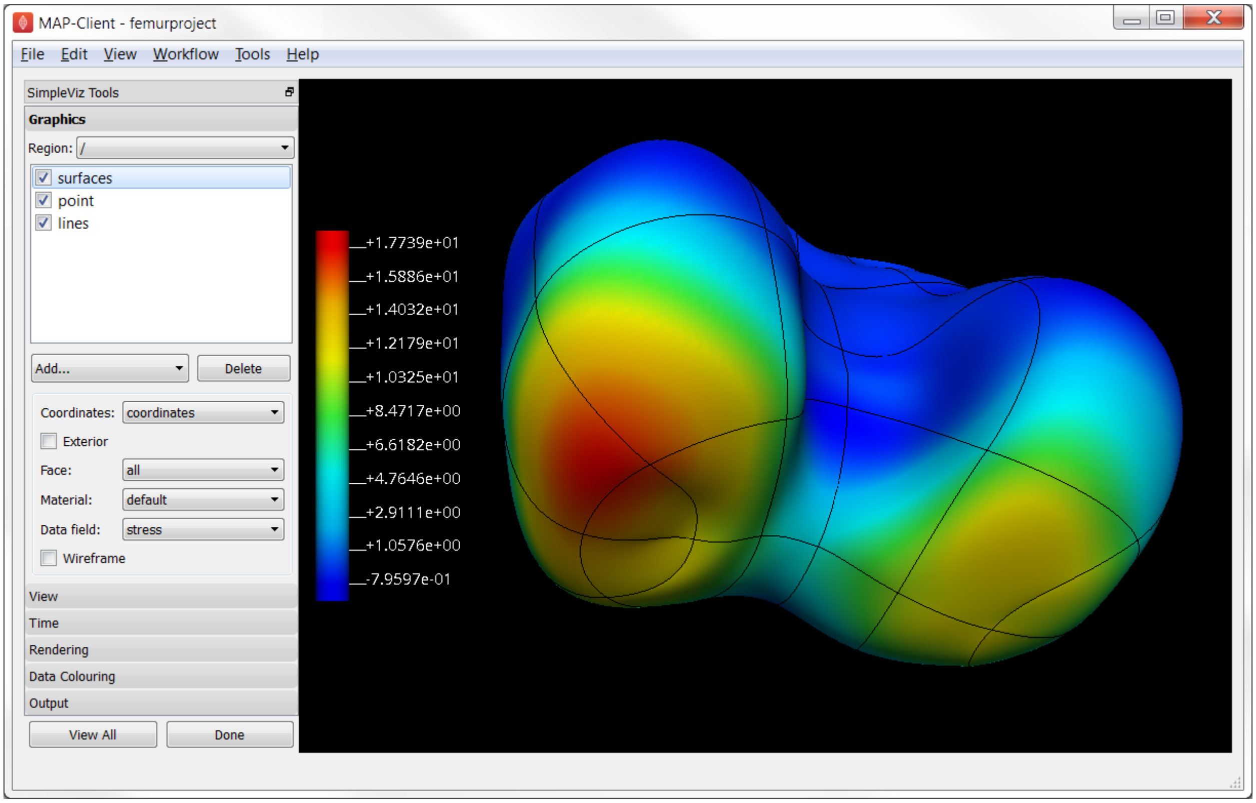


Fig. 1.173: The MAP Client interface showing some of the plugins to use in this project. (Note that your version should have additional plugins.)

- Your generated mesh must be 2-D and must have the bottom of a 2-D sphere at the base of the femur. It is suggested that 8 to 12 elements are used around the sphere, to capture the shape while keeping the model size down. Remember you can scale the generated mesh differently in each direction, which may help. With more elements you'll need plenty of high quality digitised points, and careful use of the strain penalty when fitting!
- The smoothfit step has 3 inputs: the top one is the initial model, the bottom one is a cloud of data points (as output by the Segmentation tool). The middle one is unused here as it used as an alternative data point cloud, read from an EX format file. Be sure to save your final transformation so you can re-load it next time the workflow is run.
- Note it takes a few seconds to perform each fit iteration, and a similar amount of time to perform the Load Femur maths. Be patient; the interface for the next step will display when ready.
- When initially visualising you will see nothing! Create surfaces and try to view the stress on it. Add a colour bar and try creating some other graphics. Save the resulting image.

Your final visualisation will look something like Fig. 1.174:



Region: /

☑ surfaces
☑ point
☑ lines

Coordinates: coordinates
☐ Exterior
Face: all
Material: default
Data field: stress
☐ Wireframe

Fig. 1.174: Visualisation of the final result of the femur project. Your result may use different graphics, and will differ because your digitisation and fitting will always make a unique result.

Scoring

The tutors will give a score out of 10 based on both how realistic the final femur shape is, and how attractive the output image is (tip: choose visualisation settings maximise quality, and think about balance of the image). You may re-run the workflow to add or fix the digitised points (be sure to load the current ones first!), improve fitting etc. to try to increase your score.

1.5.3 Project: Bond Graph

This project was created as part of the Computational Physiology module in the MedTech CoRE Doctoral Training Programme.

Project outline

This project requires you to put together what you have learned in the tutorials to define a complete workflow which will create a hoisting device model using the bond graph technique. The hoisting device consists of an electromotor fed by electric mains, a cable drum and a load (Fig. 1.175).

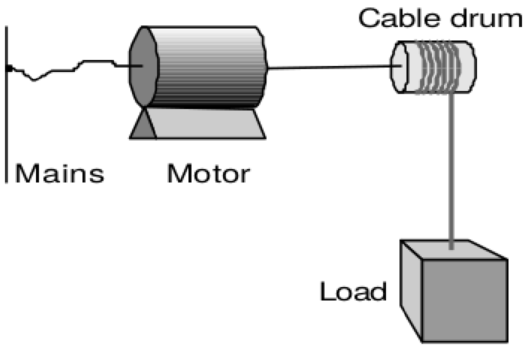


Fig. 1.175: Sketch of the hoisting device.

The mains is modelled as an ideal voltage source. At the electromotor, the inductance, electric resistance of the coils, bearing friction and rotary inertia are taken into account. The cable drum is the transformation from rotation to translation, which we consider as ideal. The load consists of a mass and the gravity force. Starting from the schematic in Fig. 1.176, you would be able to construct a bond graph model using the steps mentioned before.

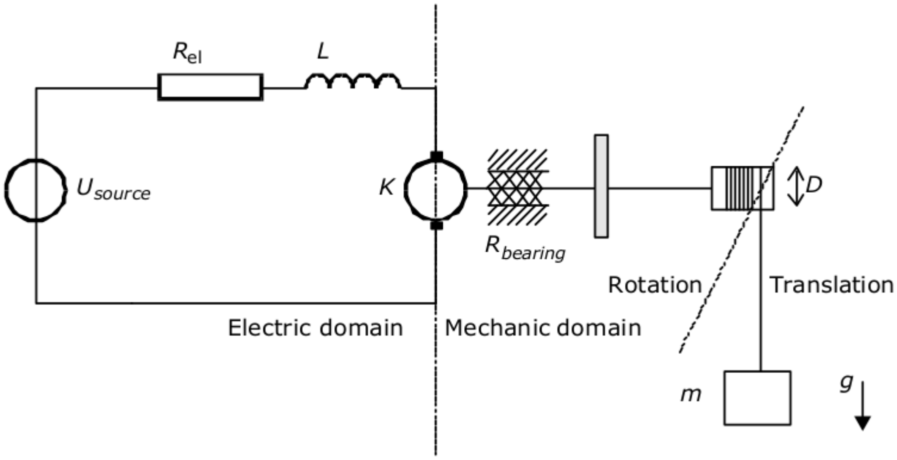


Fig. 1.176: Possible ideal-physical model augmented with the domain information.

Additional info

- Voltage source is constant with given value of 20 J.C^{-1}.

- Load and rotary inertia are 1 J.s^2 .m^{-2} and 2 J.s^2 .rad^{-2}, respectively.
- Electric resistance is 10000 J.s.C^{-2} and bearing friction is 10 J.s.rad^{-2}.
- Gyrator ratio and transformer ratio are 3 J.s.C^{-1} .m^{-1} and 5, respectively.

Simulation

Using the bond graph model, now we can derive the equations and implement them in OpenCOR. The equations that we are looking for are: conservation of flow for *0-junctions*, conservation of energy for *1-junctions*, and constitutive relations for the elements. The input boundary condition for solving this system of ODEs is the voltage source or *SE* in the bond graph model. By running the simulation, you would be able to plot the potential and flow for all the elements in time.

1.6 DTP Computational Physiology to do list

Contents:

1.6.1 General

Todo:

- any general todo's go here.
- make sure the todo's are turned off in the config?

1.6.2 Within sections

Todo: This section needs more work. Need to add documentation on how to do the examples using a GUI client like TortoiseGit.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dtp-compphys/checkouts/latest/source/PMR2/embeddedworkspaces.rst, line 12.)

Todo:

- Update all 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/dtp-compphys/checkouts/latest/source/PMR2/index.rst, line 38.)

Todo:

- Colour background of *CellML Text*
- Annotate screen shots with svg for same look and feel
- *CellML Text* code is not highlighted for all display situations, currently only in environments that are using an adapted version of pygments
- Tidy up citations and BiBTeX source (possibly use Zotero to manage?)
- Make horizontal line for footnotes only visible in html output
- Check external references markup
- Consider a more suitable theme (may require changes to an existing one to get a good result)
- Must check over output (and models) from screenshots to make sure that it matches the current release of OpenCOR, especially against running experiments for the first time.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dtp-compphys/checkouts/latest/source/opencor-tutorial/source/index.rst, line 77.)

Todo:

- any general todo's go here.
- make sure the todo's are turned off in the config?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dtp-compphys/checkouts/latest/source/todolist.rst, line 11.)

In addition to the material above, we provide here some further material that is relevant to this module. This includes the PMR user documentation and a tutorial introducing CellML, OpenCOR, and PMR.

1.7 Tutorial on CellML, OpenCOR & the Physiome Model Repository

Note: This tutorial originated from a translation from a single Word document in July 2015. Aspects of formatting and presentation may need further work. For reference, the original tutorial is available here: `OpenCOR-Tutorial-v17.pdf`.

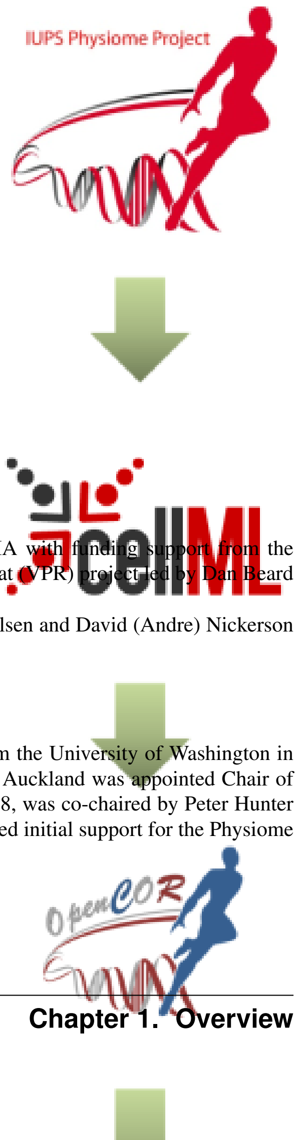
The current tutorial has now progressed well beyond the original version and we recommend using this online version or the PDF available via ReadTheDocs.

This tutorial shows you how to install and run the OpenCOR[1] software [APJ15], to author and edit CellML models[2] [DPPJ03] and to use the Physiome Model Repository (PMR)[3] [eal11]. We start by giving a brief background on the VPH-Physiome project. We then create a simple model, save it as a CellML file and run model simulations. We next try opening existing CellML models, both from a local directory and from the Physiome Model Repository. The various features of CellML[4] and OpenCOR are then explained in the context of increasingly complex biological models. A simple linear first order ODE model and a nonlinear third order model are introduced. Ion channel gating models are used to introduce the way that CellML handles units, components, encapsulation groups and connections. More complex potassium and sodium ion channel models are then developed and subsequently imported into the Hodgkin-Huxley 1952 squid axon neural model using the CellML model import facility. The Noble 1962 model of a cardiac cell action potential is used to illustrate importing of units and parameters. The tutorial finishes with sections on model annotation and the facilities available on the CellML website and the Physiome Model Repository to support model development, including the links to bioinformatic databases. There is a strong emphasis in the tutorial on establishing 'best practice' in the creation of CellML models and using the PMR resources, particularly in relation to modular approaches (model hierarchies) and model annotation.

Note: This tutorial relies on readers having some background in algebra and calculus, but tries to explain all mathematical concepts beyond this, along with the physical principles, as they are needed for the development of CellML models.[5]

1.7.1 Background to the VPH-Physiome project

To be of benefit to applications in healthcare, organ and whole organism physiology needs to be understood at both a systems level and in terms of subcellular function and tissue properties. Understanding a re-entrant arrhythmia in the heart, for example, depends on knowledge of not only numerous cellular ionic current mechanisms and signal transduction pathways, but also larger scale myocardial tissue structure and the spatial variation in protein expression. As reductionist biomedical science succeeds in elucidating ever more detail at the molecular level, it is increasingly difficult for physiologists to relate integrated whole organ function to underlying biophysically detailed mechanisms that exploit this molecular knowledge. Multi-scale computational modelling is used by engineers and physicists to design and analyse mechanical, electrical and chemical engineering systems. Similar approaches could benefit the understanding of physiological systems. To address these challenges and to take advantage of bioengineering approaches to modelling anatomy and physiology, the International Union of Physiological Sciences (IUPS) formed the Physiome Project in 1997 as an international collaboration to provide a computational framework for understanding human physiology[1].



[1] OpenCOR is an open source, freely available, C++ desktop application written by Alan Garny at INRIA with funding support from the Auckland Bioengineering Institute (http://www.abi.auckland.ac.nz) and the NIH-funded Virtual Physiological Rat (VPR) project led by Dan Beard at the University of Michigan (http://virtualrat.org).

[2] For an overview and the background of CellML see http://www.cellml.org. This project is led by Poul Nielsen and David (Andre) Nickerson at the Auckland (University) Bioengineering Institute (ABI).

[3] https://models.physiomeproject.org. The PMR project is led by Tommy Yu at the ABI.

[4] For details on the specifications of CellML1.0 see http://www.cellml.org/specifications/cellml_1.0.

[5] Please send any errors discovered or suggested improvements to p.hunter@auckland.ac.nz.

[1] www.iups.org. The IUPS President, Denis Noble from Oxford University, and Jim Bassingthwaighte from the University of Washington in Seattle have been two of the driving forces behind the Physiome Project. Peter Hunter from the University of Auckland was appointed Chair of the newly created Physiome Commission of the IUPS in 2000. The IUPS Physiome Committee, formed in 2008, was co-chaired by Peter Hunter and Sasha Popel (JHU) and is now chaired by Andrew McCulloch from UCSD. The UK Wellcome Trust provided initial support for the Physiome Project through the Heart Physiome grant awarded in 2004 to David Paterson, Denis Noble and Peter Hunter.

Primary Goals

One of the primary goals of the Physiome Project [PJ04] has been to promote the development of standards for the exchange of information between models. The first of these standards, dealing with time varying but spatially lumped processes, is CellML [VarYY]. The second (dealing with spatially and time varying processes) is FieldML [CPJ09][P13][2]. A further goal of the Physiome Project has been the development of open source tools for creating and visualizing standards-based models and running model simulations. OpenCOR is the latest in a series of software projects aimed at providing a modelling environment for CellML models. Similar tools exist for FieldML models.

Following the publication of the STEP[3] (*Strategy for a European Physiome*) Roadmap in 2006, the European Commission in 2007 initiated the Virtual Physiological Human (VPH) project [ea13]. A related US initiative by the Interagency Modeling and Analysis Group (IMAG) began in 2003[4]. These projects and similar initiatives are now coordinated and are collectively referred to here as the 'VPH-Physiome' project[5]. The VPH-Institute[6] was formed in 2012 as a virtual organisation to providing strategic leadership, initially in Europe but now globally, for the VPH-Physiome Project.

1.7.2 Create and run a simple CellML model: editing and simulation

In this example we create a simple CellML model and run it. The model is the Van der Pol oscillator[1] defined by the second order equation

$$\frac{d^2x}{dt^2} - \mu\left(1 - x^2\right)\frac{\mathrm{dx}}{\mathrm{dt}} + x = 0$$

with initial conditions $x = -2;\ \frac{\mathrm{dx}}{\mathrm{dt}} = 0$. The parameter μ controls the magnitude of the damping term. To create a CellML model we convert this to two first order equations[2] by defining the velocity $\frac{\mathrm{dx}}{\mathrm{dt}}$ as a new variable y:

$$\frac{\mathrm{dx}}{\mathrm{dt}} = y \tag{1.4}$$

$$\frac{\mathrm{dy}}{\mathrm{dt}} = \mu\left(1 - x^2\right) y - x \tag{1.5}$$

The initial conditions are now $x = -2; y = 0$.

With the central pane in *Editing* mode (e.g. *CellML Text* view), create a new CellML file: *File* → *New* → *CellML File* and then type in the following lines of code after deleting the three lines that indicate where the code should go:

[2] CellML began as a joint public-private initiative in 1998 with funding by the US company Physiome Sciences (CEO Jeremy Levin), before being launched under IUPS as a fully open source project in 1999.

[3] The STEP report, led by Marco Viceconte (University of Sheffield, UK), is available at www.europhysiome.org/roadmap.

[4] This coordinates various US Governmental funding agencies involved in multi-scale bioengineering modeling research including NIH, NSF, NASA, the Dept of Energy (DoE), the Dept of Defense (DoD), the US Dept of Agriculture and the Dept of Veteran Affairs. See www.nibib.nih.gov/Research/MultiScaleModeling/IMAG. Grace Peng of NHBIB leads the IMAG group.

[5] Other significant contributions to the VPH-Physiome project have come from Yoshi Kurachi in Japan (www.physiome.jp), Stig Omholt in Norway (www.ntnu) and Chae-Hun Leem in Korea (www.physiome.or.kr).

[6] www.vph-institute.org. Formed in 2012, the inaugural Director was Marco Viceconti. The current Director is Adriano Henney. The inaugural and current President of the VPH-Institute is Denis Noble.

[1] http://en.wikipedia.org/wiki/Van_der_Pol_oscillator

[2] Equations (1.4) and (1.5) are equations that are implemented directly in OpenCOR.

```
def model van_der_pol_model as
    def comp main as
        var t: dimensionless {init: 0};
        var x: dimensionless {init: -2};
        var y: dimensionless {init: 0};
        var mu: dimensionless {init: 1};
        // These are the ODEs
        ode(x,t)=y;
        ode(y,t)=mu*(1{dimensionless}-sqr(x))*y-x;
    enddef;
enddef;
```

Things to note[3] are:

i. the closing semicolon at the end of each line (apart from the first two *def* statements that are opening a CellML construct);

ii. the need to indicate dimensions for each variable and constant (all dimensionless in this example – but more on dimensions later);

iii. the use of *ode(x,t)* to indicate a first order[4] ODE in *x* and *t*

iv. the use of the squaring function *sqr(x)* for x^2, and

v. the use of '//' to indicate a comment.

A partial list of mathematical functions available for OpenCOR is:

x^2	sqr(x)	$\sqrt{x}$	sqrt(x)	$\ln x$	ln(x)	$\log_{10} x$	log(x)	e^x	exp(x)	x^a	pow(x,a)
$\sin x$	sin(x)	$\cos x$	cos(x)	$\tan x$	tan(x)	$\csc x$	csc(x)	$\sec x$	sec(x)	$\cot x$	cot(x)
$\sin^{-1} x$	asin(x)	$\cos^{-1} x$	acos(x)	$\tan^{-1} x$	atan(x)	$\csc^{-1} x$	acsc(x)	$\sec^{-1} x$	asec(x)	$\cot^{-1} x$	acot(x)
$\sinh x$	sinh(x)	$\cosh x$	cosh(x)	$\tanh x$	tanh(x)	$\operatorname{csch} x$	csch(x)	$\operatorname{sech} x$	sech(x)	$\coth x$	coth(x)
$\sinh^{-1} x$	asinh(x)	$\cosh^{-1} x$	acosh(x)	$\tanh^{-1} x$	atanh(x)	$\operatorname{csch}^{-1} x$	acsch(x)	$\operatorname{sech}^{-1} x$	asech(x)	$\coth^{-1} x$	acoth(x)

Table 1. Partial list of mathematical functions available for coding in OpenCOR.

Positioning the cursor over either of the ODEs renders the maths in standard form above the code as shown in Fig. 1.177.

Note that CellML is a declarative language[5] (unlike say C, Fortran or Matlab, which are procedural languages) and therefore the order of statements does not affect the solution. For example, the order of the ODEs could equally well be

```
ode(y,t)=mu*(1{dimensionless}-sqr(x))*y-x;
ode(x,t)=y;
```

The significance of this will become apparent later when we import several CellML models to create a composite model.

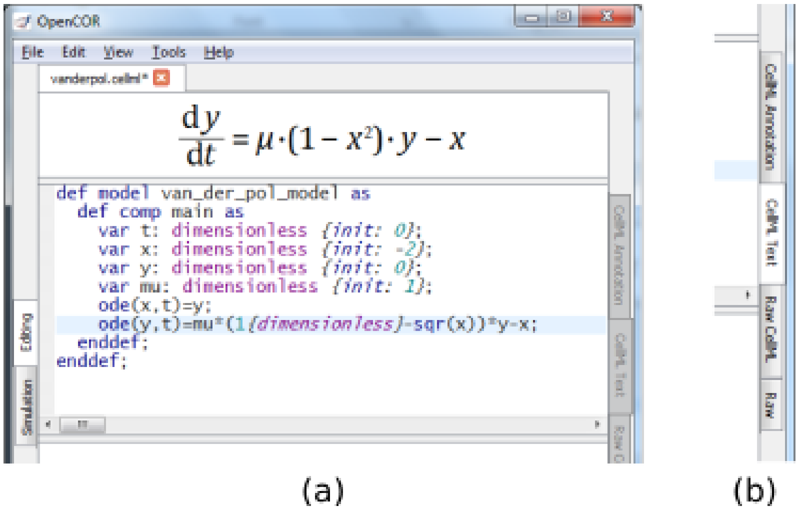
Now save the code to a local folder using *Save* under the *File* menu (*File* → *Save*) (or 'CTRL-S') and choosing *.cellml* as the file format[6]. With the CellML model saved various views, accessed via the tabs on the right hand edge of the window, become available. One is the *CellML Text* view (the view used to enter the code above); another is the *Raw CellML* view that displays the way the model is stored and is intentionally verbose to ensure that the meaning is always

[3] For more on the *CellML Text* view see http://opencor.ws/user/plugins/editing/CellMLTextView.html.

[4] Note that a more elaborated version of this is `ode(x, t, 1{dimensionless})` and a 2nd order ODE can be specified as `ode(x, t, 2{dimensionless})`. 1st order is assumed as the default.

[5] Note also that the mathematical expressions in CellML are based on MathML – see http://www.w3.org/Math/

[6] Note that `.cellml` is not strictly required but is best practice.



$$\frac{dy}{dt} = \mu \cdot (1 - x^2) \cdot y - x$$

```
def model van_der_pol_model as
  def comp main as
    var t: dimensionless {init: 0};
    var x: dimensionless {init: -2};
    var y: dimensionless {init: 0};
    var mu: dimensionless {init: 1};
    ode(x,t)=y;
    ode(y,t)=mu*(1{dimensionless}-sqr(x))*y-x;
  enddef;
enddef;
```

Fig. 1.177: (a) Positioning the cursor over an equation and clicking (shown by the highlighted line) renders the maths. (b) Once the model has been successfully saved, the *CellML Text* view tab becomes white rather than grey. The right hand tabs provide different views of the CellML code.

unambiguous (note that positioning the cursor over part of the code shows the maths in this view also); and another is the *Raw* view. Notice that 'CTRL-T' in the *Raw CellML* view performs validation tests on the CellML model. The *CellML Text* view provides a much more convenient format for entering and editing the CellML model.

With the equations and initial conditions defined, we are ready to run the model. To do this, click on the *Simulation* tab on the left hand edge of the window. You will see three main areas - at the left hand side of the window are the *Simulation*, *Solvers*, *Graphs* and *Parameters* panels, which are explained below. At the right hand side is the graphical output window, and running along the bottom of the window is a status area, where status messages are displayed.

Simulation Panel

This area is used to set up the simulation settings.

- Starting point - the value of the variable of integration (often time) at which the simulation will begin. Leave this at 0.
- Ending point - the point at which the simulation will end. Set to 100.
- Point interval - the interval between data points on the variable of integration. Set to 0.1.

Just above the *Simulation panel* are controls for running the simulation. These are:

Run (▶), *Pause* (■), *Reset parameters* (↻), *Clear simulation data* (🗑), *Interval delay* (▭), *Add*(✚)/*Subtract*(▬) *graphical output windows* and *Output solution to a CSV file* (🗎).

For this model, we suggest that you create three graphical output windows using the **+** button.

Solvers Panel

This area is used to configure the solver that will run the simulation.

- Name - this is used to set the solver algorithm. It will be set by default to be the most appropriate solver for the equations you are solving. OpenCOR allows you to change this to another solver appropriate to the type of equations you are solving if you choose to. For example, CVODE for ODE (ordinary differential equation) problems, IDA for DAE (differential algebraic equation) problems, KINSOL for NLA (non-linear algebraic) problems[7].

- Other parameters for the chosen solver – e.g. *Maximum step*, *Maximum number of steps*, and *Tolerance* settings for CVODE and IDA. For more information on the solver parameters, please refer to the documentation for the particular solver.

Note: these can all be left at their default values for our simple demo problem[8].

Graphs Panel

This shows what parameters are being plotted once these have been defined in the *Parameters panel*. These can be selected/deselected by clicking in the box next to a parameter.

Parameters Panel

This panel lists all the model parameters, and allows you to select one or more to plot against the variable of integration or another parameter in the graphical output windows. OpenCOR supports graphing of any parameter against any other. All variables from the model are listed here, arranged by the components in which they appear, and in alphabetical order. Parameters are displayed with their variable name, their value, and their units. The icons alongside them have the following meanings:

C Editable constant	S Editable state variable
C Computed constant	R Rate variable
V Variable of integration	A Algebraic quantity

Right clicking on a parameter provides the options for displaying that parameter in the currently selected graphical output window. With the cursor highlighting the top graphical output window (a blue line appears next to it), select *x* then *Plot Against Variable of Integration* – in this case *t* - in order to plot *x(t)*. Now move the cursor to the second graphical output window and select *y* then *t* to plot *y(t)*. Finally select the bottom graphical output window, select *y* and select *Plot Against* then *Main* then *x* to plot *y(x)*.

Now click on the *Run* control. You will see a progress bar running along the bottom of the status window. Status messages about the successful simulation, including the time taken, are displayed in the bottom panel. This can be hidden by dragging down on the bar just above the panel. Fig. 1.178 shows the results. Use the *interval delay* wheel to slow down the plotting if you want to watch the solution evolve. You can also pause the simulation at any time by clicking on the *Run* control and if you change a parameter during the pause, the simulation will continue (when you click the *Run* control button again) with the new parameter.

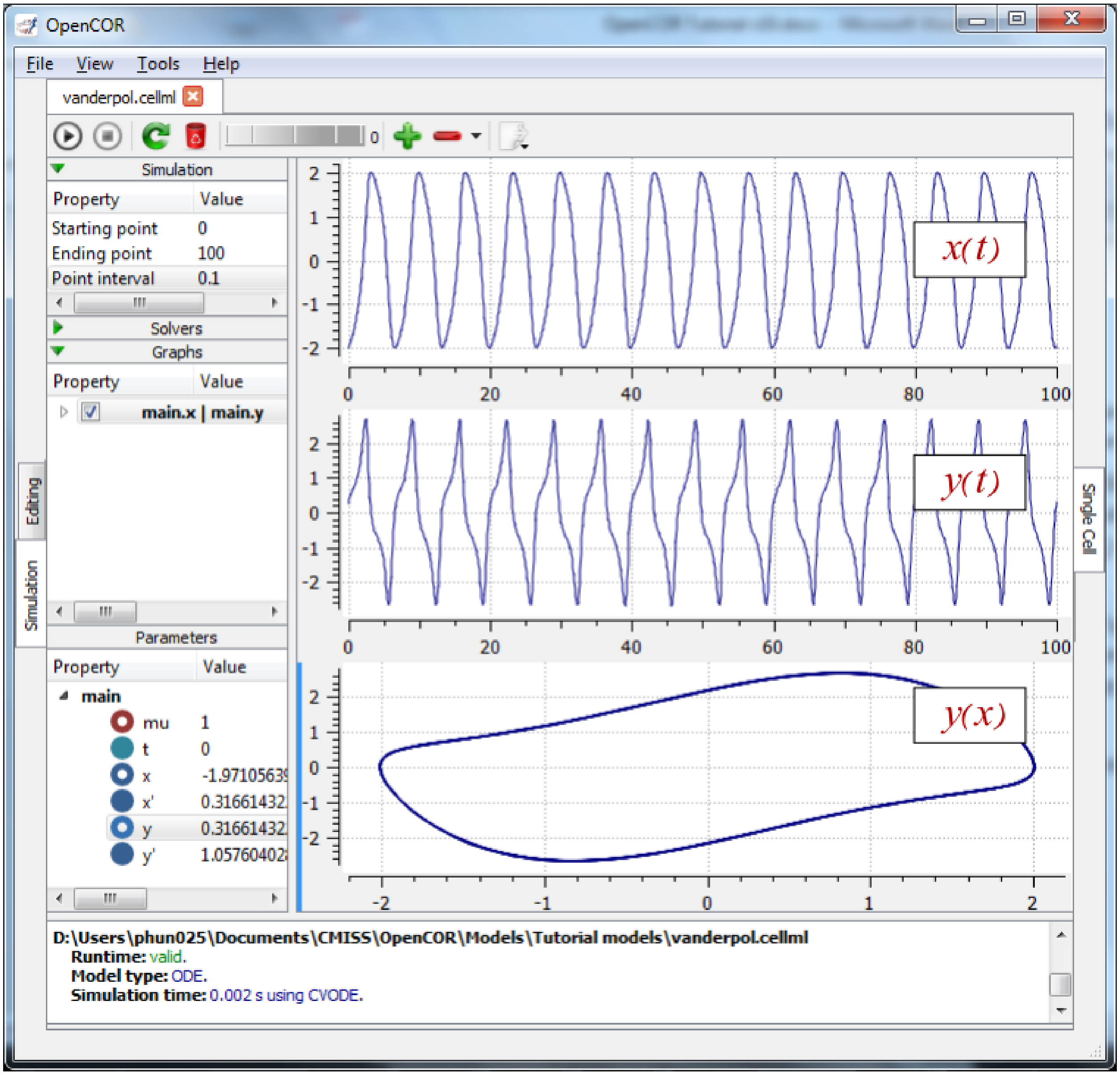
Note that the values shown for the various parameters are the values they have at the end of the solution run. To restore these to their initial values, use the *Reset parameters* (↻) button. To clear the graphical output traces, click on the *Clear simulation data* (🗑) button.

The top two graphical output panels are showing the time-dependent solution of the *x* and *y* variables. The bottom panel shows how *y* varies as a function of *x*. This is called the solution in state space and it is often useful to analyse the state space solution to capture the key characteristics of the equations being solved.

To obtain numerical values for all variables (i.e. *x(t)* and *y(t)*), click on the *CSV file* button (🗎). You will be asked to enter a filename and type (use .csv). Opening this file (e.g. with Microsoft Excel) provides access to the numerical values. Other output types (e.g. BiosignalML) will be available in future versions of OpenCOR.

[7] Other solvers include forward Euler, Heun and Runga-Kutta solvers (RK2 and RK4).

[8] Note that a model that requires a stimulus protocol should have the maximum step value of the CVODE solver set to the length of the stimulus.



Simulation

Property	Value
Starting point	0
Ending point	100
Point interval	0.1

Parameters

Property	Value
main	
mu	1
t	0
x	-1.9710563
x'	0.31661432
y	0.31661432
y'	1.05760402

D:\Users\phun025\Documents\CMISS\OpenCOR\Models\Tutorial models\vanderpol.cellml
Runtime: valid.
Model type: ODE.
Simulation time: 0.002 s using CVODE.

Fig. 1.178: Graphical output from OpenCOR. The top window is *x(t)*, the middle is *y(t)* and the bottom is *y(x)*. The *Graphs* panel shows that *y(x)* is being plotted on the graph output window highlighted by the LH blue line. The window at the very bottom provides runtime information on the type of equation being solved and the simulation time (2ms in this case). The computed variables shown in the left hand panel are at the values they have at the end of the simulation.

You can move the graphical output traces around with 'left click and drag' and you can change the horizontal or vertical scale with 'right click and drag'. Holding the SHIFT key down while clicking on a graphical output panel allows you to interrogate the solution at any point. Right clicking on a panel provides zoom facilities.

Note: The simulation described above can also be loaded and run directly in OpenCOR using this link.

The various plugins used by OpenCOR can be viewed under the Tools menu. A French language version of OpenCOR is also available under the *Tools* menu. An option under the *File* menu allows a file to be locked (also 'CTRL-L'). To indicate that the file is locked, the background colour switches to pink in the *CellML Text* and *Raw CellML* views and a lock symbol appears on the filename tab. Note that OpenCOR text is case sensitive.

1.7.3 Open an existing CellML file from a local directory or the Physiome Model Repository

Go to the *File* menu and select *Open...* (*File* → *Open*). Browse to the folder that contains your existing models and select one. Note that this brings up a new tabbed window and you can have any number of CellML models open at the same time in order to quickly move between them. A model can be removed from this list by clicking on ☒ next to the CellML model name.

You can also access models from the left hand panel in Fig. 1.1(a). If this panel is not currently visible, use 'CTRL-spacebar' to make it reappear. Models can then be accessed from any one of the three subdivisions of this panel – *File Browser*, *Physiome Model Repository* or *File Organiser*. For a file under *File Browser* or *File Organiser*, either double-click it or 'drag&drop' it over the central workspace to open that model. Clicking on a model in the *Physiome Model Repository* (PMR) (e.g. Chen, Popel, 2007) opens a new browser window with that model (PMR is covered in more detail in Section 13). You can either load this model directly into OpenCOR or create an identical copy (clone) of the model in your local directory. Note that PMR contains *workspaces* and *exposures*. Workspaces are online environments for the collaborative development of models (e.g. by geographically dispersed groups) and can have password protected access. Exposures are workspaces that are exposed for public view and mostly contain models from peer-reviewed journal publications. There are about 600 exposures based on journal papers and covering many areas of cell processes and other ODE/algebraic models, but these are currently being supplemented with reusable protein-based models – see discussion in a Section 13.

To load a model directly into OpenCOR, click on the right-most of the two buttons in Fig. 1.179 - this lists the CellML models in that exposure - and then click on the model you want. Clicking on the left hand button copies the PMR workspace to a local directory that you specify. This is useful if you want to use that model as a template for a new one you are creating.

In the PMR window (Fig. 1.179) the buttons on the right-hand side [**1**] lists all the CellML files for this model. Clicking on one of those [**2**] uploads the model into OpenCOR. The left-hand buttons [**3**] copies the PMR workspace to a local directory.

1.7.4 A simple first order ODE

The simplest example of a first order ODE is

$$\frac{\mathrm{dy}}{\mathrm{dt}} = -ay + b$$

with the solution

$$y\left(t\right) = \frac{b}{a} + \left(y\left(0\right) - \frac{b}{a}\right).e^{-at},$$

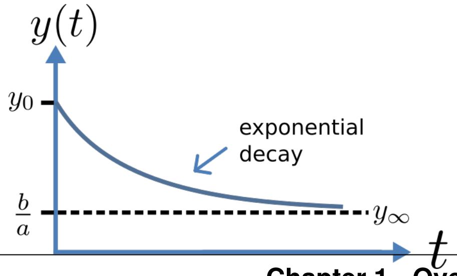
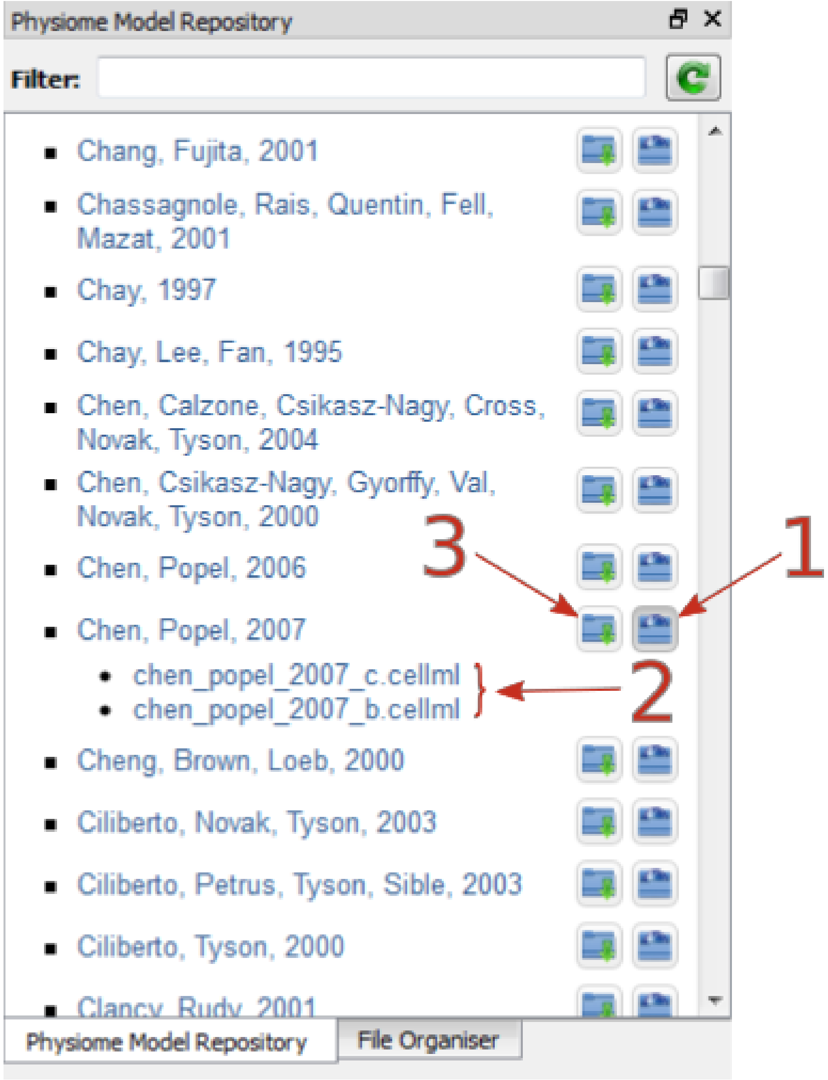


Fig. 1.180: Solution of 1st order equation.



- Chang, Fujita, 2001
- Chassagnole, Rais, Quentin, Fell, Mazat, 2001
- Chay, 1997
- Chay, Lee, Fan, 1995
- Chen, Calzone, Csikasz-Nagy, Cross, Novak, Tyson, 2004
- Chen, Csikasz-Nagy, Gyorffy, Val, Novak, Tyson, 2000
- Chen, Popel, 2006
- Chen, Popel, 2007
  - chen_popel_2007_c.cellml
  - chen_popel_2007_b.cellml
- Cheng, Brown, Loeb, 2000
- Ciliberto, Novak, Tyson, 2003
- Ciliberto, Petrus, Tyson, Sible, 2003
- Ciliberto, Tyson, 2000
- Clancy, Rudy, 2001

Fig. 1.179: The Physiome Model Repository (PMR) window listing all PMR models. These can be opened from within OpenCOR using the two buttons to the right of a model, as explained below.

where $y(0)$ or y_0, the value of $y(t)$ at $t = 0$, is the *initial condition*. The final steady state solution as $t \to \infty$ is $y(\left. t\right|_{\infty}) = y_{\infty} = \frac{b}{a}$ (see Figure 6). Note that $t = \tau = \frac{1}{a}$ is called the *time constant* of the exponential decay, and that

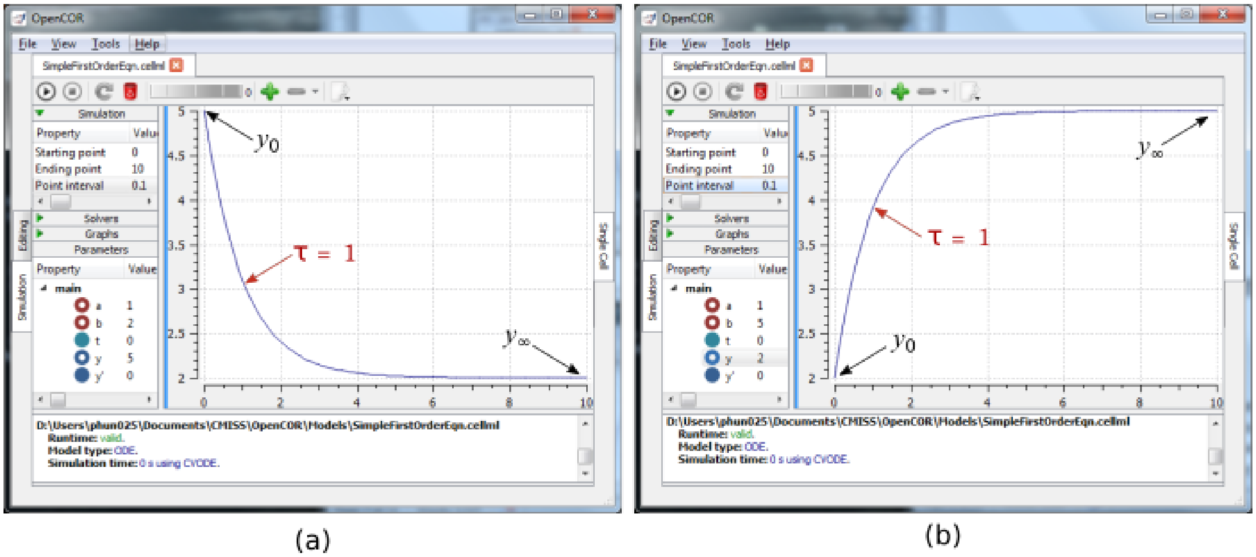
$$y(\tau) = \frac{b}{a} + \left(y(0) - \frac{b}{a} \right).e^{-1}.$$

At $t = \tau$, $y(t)$ has therefore fallen to $\frac{1}{e}$ (or about 37%) of the difference between the initial ($y(0)$) and final steady state ($y(\infty)$) values[1].

Choosing parameters $a = \tau = 1; b = 2$ and $y(0) = 5$, the *CellML Text* for this model is

```
def model first_order_model as
    def comp main as
        var t: dimensionless {init: 0};
        var y: dimensionless {init: 5};
        var a: dimensionless {init: 1};
        var b: dimensionless {init: 2};
        ode(y,t)=-a*y+b;
    enddef;
enddef;
```

The solution by OpenCOR is shown in Fig. 1.181(a) for these parameters (a decaying exponential) and in Fig. 1.181(b) for parameters $a = 1; b = 5$ and $y(0) = 2$ (an inverted decaying exponential). Note the simulation panel with *Ending point*=10, *Point interval*=0.1. Try putting $a = -1$.



(a)

Property	Value
Starting point	0
Ending point	10
Point interval	0.1

Property	Value
main	
a	1
b	2
t	0
y	5
y'	0

```
D:\Users\phun025\Documents\CMISS\OpenCOR\Models\SimpleFirstOrderEqn.cellml
   Runtime: valid.
   Model type: ODE.
   Simulation time: 0 s using CVODE.
```

(b)

Property	Value
Starting point	0
Ending point	10
Point interval	0.1

Property	Value
main	
a	1
b	5
t	0
y	2
y'	0

```
D:\Users\phun025\Documents\CMISS\OpenCOR\Models\SimpleFirstOrderEqn.cellml
   Runtime: valid.
   Model type: ODE.
   Simulation time: 0 s using CVODE.
```

Fig. 1.181: OpenCOR output $y(t)$ for the simple ODE model with parameters (a) $a = 1; b = 2$ and $y(0) = 5$ (OpenCOR link), and (b) $a = 1; b = 5$ and $y(0) = 2$. The red arrow indicates the point at which the trace reaches the time constant τ (e^{-1} or $\approx 37\%$ of the difference between the initial and final solution values). The black arrows indicate the initial and final (steady state) solutions. Note that the parameters on the left have been reset to their initial values for this figure - normally they would be at their final solution values.

These two solutions have the same exponential time constant ($\tau = \frac{1}{a} = 1$) but different initial and final (steady state) values.

[1] It is often convenient to write a first order equation as $\tau \frac{\mathrm{dy}}{\mathrm{dt}} = -y + y_{\infty}$, so that its solution is expressed in terms of time constant τ, initial condition y_0 and steady state solution y_{∞} as: $y(t) = y_{\infty} + (y_0 - y_{\infty}).e^{-\frac{t}{\tau}}$.

The exponential decay curve shown on the left in Fig. 1.181 is a common feature of many models and in the case of radioactive decay (for example) is a statement that the **rate of decay** ($-\frac{\mathrm{dy}}{\mathrm{dt}}$) is proportional to the **current amount of substance** (y). This is illustrated on the NZ$100 note (should you be lucky enough to possess one), shown in Figure 8.

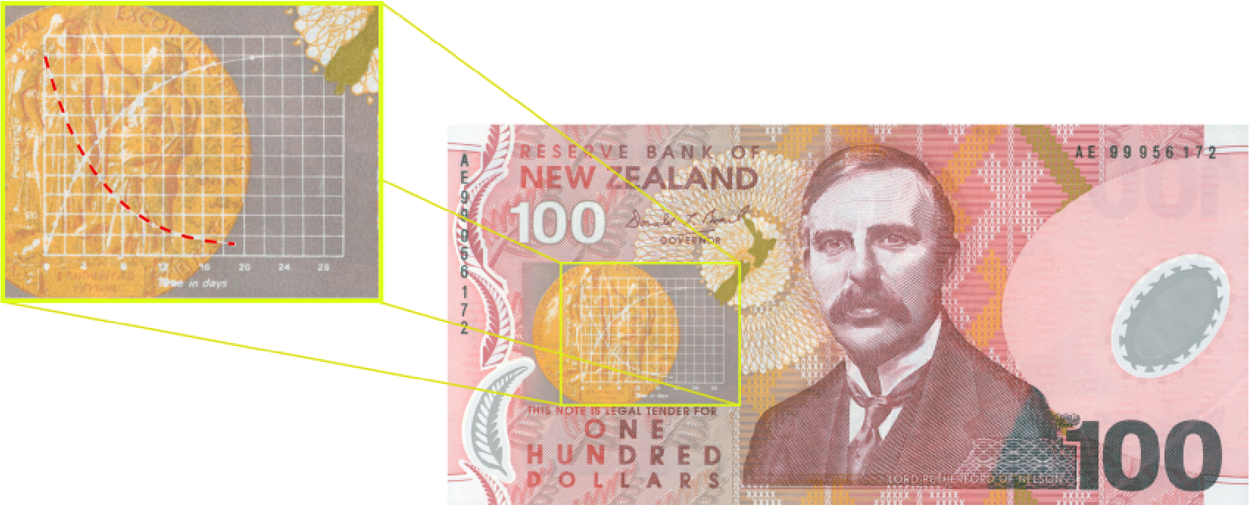


Fig. 1.182: The **exponential curve** representing the naturally occurring radioactive decay explained by the New Zealand Noble laureate Sir Ernest Rutherford - best known for 'splitting the atom'. This may be the only bank note depicting the mathematical solution of a first order ODE.

1.7.5 The Lorenz attractor

An example of a third order ODE system (i.e. three 1$^{\mathrm{st}}$ order equations) is the *Lorenz equations*[1].

This system has three equations:

$$\frac{\mathrm{dx}}{\mathrm{dt}} = \sigma\left(y - x\right)$$

$$\frac{\mathrm{dy}}{\mathrm{dt}} = x\left(\rho - z\right) - y$$

$$\frac{\mathrm{dz}}{\mathrm{dt}} = xy - \beta z$$

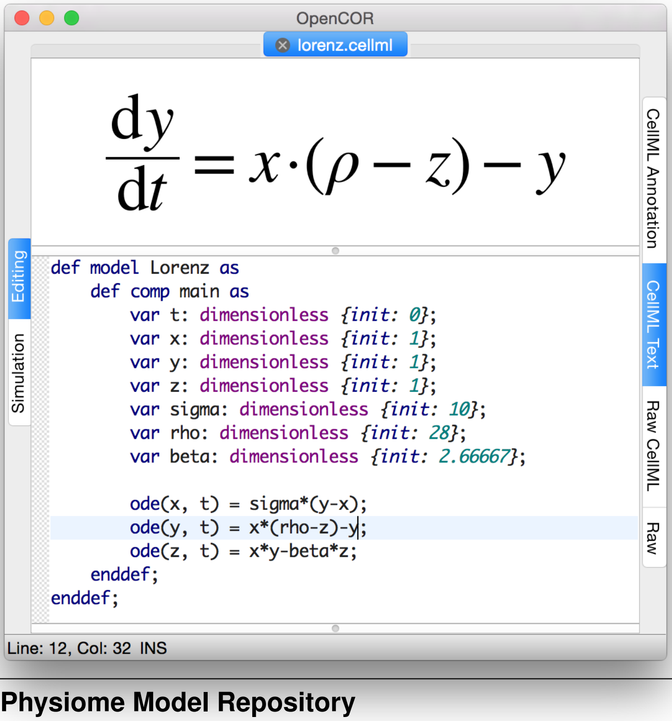
where σ, ρ and β are parameters.

The *CellML Text* code entered for these equations is shown in Fig. 1.183 with parameters

$\sigma = 10$, $\rho = 28$, $\beta = 8/3 = 2.66667$

and initial conditions

$x\left(0\right) = y\left(0\right) = z\left(0\right) = 1$.



$$\frac{dy}{dt} = x \cdot (\rho - z) - y$$

```
def model Lorenz as
    def comp main as
        var t: dimensionless {init: 0};
        var x: dimensionless {init: 1};
        var y: dimensionless {init: 1};
        var z: dimensionless {init: 1};
        var sigma: dimensionless {init: 10};
        var rho: dimensionless {init: 28};
        var beta: dimensionless {init: 2.66667};

        ode(x, t) = sigma*(y-x);
        ode(y, t) = x*(rho-z)-y;
        ode(z, t) = x*y-beta*z;
    enddef;
enddef;
```

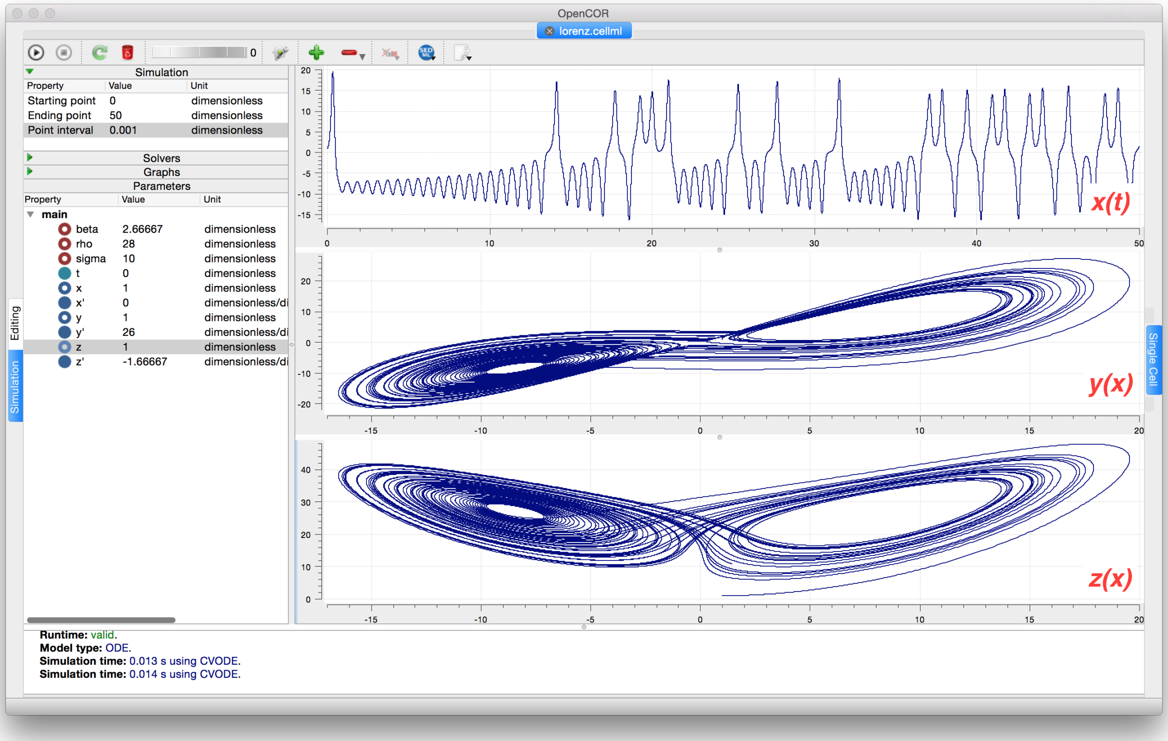
Line: 12, Col: 32 INS

Fig. 1.183: *CellML Text* code for the Lorenz equations.

[1] http://en.wikipedia.org/wiki/Lorenz_system

Solutions for $x\left(t\right)$, $y\left(x\right)$ and $z\left(x\right)$, corresponding to the time integration parameters shown on the LHS, are shown in Fig. 1.184. Note that this system exhibits 'chaotic dynamics' with small changes in the initial conditions leading to quite different solution paths.

This example illustrates the value of OpenCOR's ability to plot variables as they are computed. Use the *Simulation Delay* wheel to slow down the plotting by a factor of about 5-10,000 - in order to follow the solution as it spirals in ever widening trajectories around the left hand wing of the attractor before coming close to the origin that then sends it off to the right hand wing of the attractor.



Simulation

Property	Value	Unit
Starting point	0	dimensionless
Ending point	50	dimensionless
Point interval	0.001	dimensionless

Solvers

Graphs

Parameters

Property	Value	Unit
main		
beta	2.66667	dimensionless
rho	28	dimensionless
sigma	10	dimensionless
t	0	dimensionless
x	1	dimensionless
x'	0	dimensionless/di
y	1	dimensionless
y'	26	dimensionless/di
z	1	dimensionless
z'	-1.66667	dimensionless/di

Runtime: valid.
Model type: ODE.
Simulation time: 0.013 s using CVODE.
Simulation time: 0.014 s using CVODE.

Fig. 1.184: Solutions of the Lorenz equations. Note that the parameters on the left have been reset to their initial values for this figure – normally they would be at their final solution values.

Solutions to the Lorenz equations are organised by the 2D 'Lorenz manifold'. This surface has a very beautiful shape and has become an art form - even rendered in crochet![2] (See Fig. 1.185).

[2] http://www.math.auckland.ac.nz/~hinke/crochet/

Fig. 1.185: The crocheted Lorenz manifold made by Professors Hinke Osinga and Bernd Krauskopf of the Mathematics Department at the University of Auckland, New Zealand.

Note: The simulation presented in Fig. 1.184 can be loaded direction into OpenCOR using this link.

Exercise for the reader

Another example of intriguing and unpredictable behaviour from a simple deterministic ODE system is the 'blue sky catastrophe' model [JH02] defined by the following equations:

$$\frac{\mathrm{dx}}{\mathrm{dt}} = y$$
$$\frac{\mathrm{dy}}{\mathrm{dt}} = x - x^3 - 0.25y + A\sin t$$

with parameter $A = 0.2645$ and initial conditions $x(0) = 0.9$, $y(0) = 0.4$. Run to $t = 500$ with $\Delta t = 0.01$ and plot $x(t)$ and $y(x)$ (OpenCOR link). Also try with $A = 0.265$ to see how sensitive the solution is to small changes in parameter values.

1.7.6 Code generation

It is sometimes required to export CellML models to various procedural formats to make use of a given model with existing tools. OpenCOR currently uses the CellML Language Export Definition Service provided by the CellML API to achieve this (see this article for details). This service takes an XML file containing a conversion definition and uses that to export a CellML model to the defined format.

The OpenCOR distribution packages include definition files for C, Fortran 77, Python, and Matlab. These definition files are available in the `formats` folder of your OpenCOR installation or can be downloaded and used directly using the previous links.

The C and Fortran code generated using these definition files contain functions suitable for inclusion in DAE/ODE simulation codes. Whereas the Python and Matlab code generated are complete scripts that use standard Python or Matlab methods to actually perform an *default* simulation. The default simulation is probably not what is needed, so the generated code can be modified or reused to meet the specific usage requirements.

Exporting CellML to code

The steps to generate code from OpenCOR are given below.

1. Load the desired CellML model into OpenCOR (both CellML 1.0 and 1.1 models can be used)
2. From the OpenCOR menu, choose *Tools* → *CellML File Export To* → *User-Defined Format*.
3. The first file selection dialog is to provide the conversion definition file (as above).

4. The second file selection dialog is to provide the file to save the generated code to.

This conversion can also be performed using OpenCOR as a command line client. In this case the command is:

```
$ ./OpenCOR -c CellMLTools::export myfile.cellml myformat.xml
```

or for a remote model:

```
$ ./OpenCOR -c CellMLTools::export http://mydomain.com/myfile.cellml myformat.xml
```

where `myformat.xml` can be one of the standard definition files described above.

Generated code in PMR

The Physiome Model Repository uses the same code generation service from the CellML API to generate code in the above formats for all exposures containing CellML models. These are available from the *Generated Code* view for CellML models. See here for an example.

1.7.7 Model annotation

One of the most powerful features of CellML is its ability to import models. This means that complex models can be built up by combining previously defined models. There is a potential problem with this process, however, since the imported models (often developed by completely different modellers) may represent the same biological or biophysical entity with different expressions. The potassium channel model in *A model of the potassium channel: Introducing CellML components and connections*, for example, represents the intracellular concentration of potassium as ‘Ki’ (see the *CellML Text* code *Potassium_ion_channel.cellml*) but another model involving the intracellular potassium concentration may use a different expression.

The solution to this dilemma is to annotate the CellML variables with names from controlled vocabularies that have been agreed upon by the relevant scientific community. In this case we may simply want to annotate Ki as ‘*the concentration of potassium in the cytosol*’. This expression, however, refers to three distinct entities: *concentration*, *potassium* and *cytosol*. We might also want to specify that we are referring to the cytosol of a neuron ... and that the neuron comes from a particular part of a giant squid (the experimental animal used by Hodgkin and Huxley). Annotations can clearly get very complicated!

What comes to our rescue here is that most scientific communities have developed controlled vocabularies together with the relationships between the terms of that vocabulary – called **ontologies**. Furthermore relationships can always be expressed in the form subject-predicate-object. E.g. Ki is-the-concentration-of potassium is one relationship and potassium in-the cytosol is another. Each object can become the subject of another expression. We could continue, for example, with cytosol of-the neuron, neuron of-the squid and so on. The terms s-the-concentration-of, in-the and of-the are the predicates and these semantically rich expressions too have to come from controlled vocabularies. Each of these subject-predicate-object expressions is called an RDF **triple** and the World Wide Web consortium[1] has established a framework called the *Resource Description Framework* (RDF[2]) to support these.

CellML models therefore contain two parts, one dealing with **syntax** (the MathML definition of the models together with the structure of components, connections, groups, units, etc) as discussed in previous sections, and one dealing with **semantics** (the meanings of the terms used in the models) discussed in this section[3]. This latter is also referred to as *metadata* – i.e. data about data.

In the CellML metadata specification[4] the first RDF *subject* of a triple is a CellML element (e.g. a variable such as ‘Ki’), the RDF *predicate* is chosen from the Biomodels Biological Qualifiers[5] list, and the RDF *object* is a URI

[1] Referred to as W3C – see www.w3.org

[2] www.w3.org/RDF

[3] For details on the annotation plugin see http://opencor.ws/user/plugins/editing/CellMLAnnotationView.html

[4] See http://www.cellml.org/specifications/metadata/ and http://www.cellml.org/specifications/metadata/mcdraft

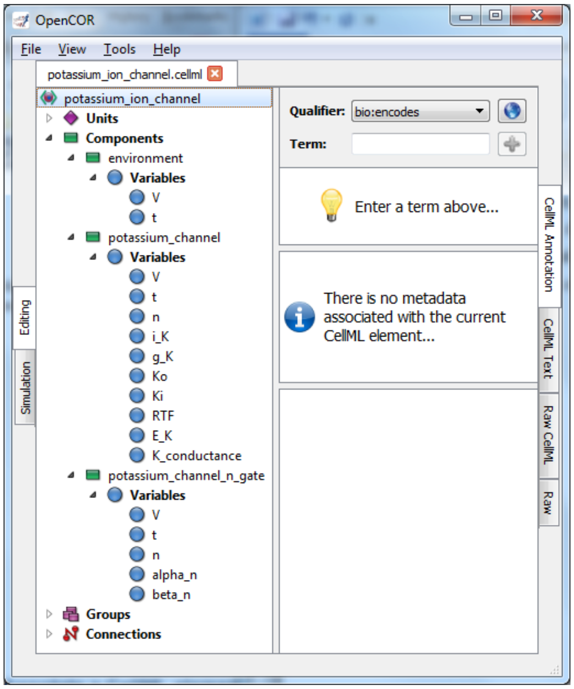
[5] http://co.mbine.org/standards/qualifiers

(the string of characters used to identify the name of a resource[6]). Establishing these RDF links to biological and biophysical meaning is the goal of annotation.

Note the different types of subject/object used in the RDF triples: *the concentration* is a biophysical entity, *potassium* is a chemical entity, *the cytosol* is an anatomical entity. In fact, to cover all the terminology used in the models, CellML uses five separate ontologies:

- ChEBI (Chemical Entities of Biological Interest) www.ebi.ac.uk/chebi
- GO (Gene Ontology) www.geneontology.org
- FMA (Foundation Model of Anatomy) fma.biostr.washington.edu/projects/fm/
- Cell type ontology code.google.com/p/cell-ontology
- OPB sbp.bhi.washington.edu/projects/the-ontology-of-physics-for-biology-opb

These ontologies are available through OpenCOR's annotation facilities as explained below.



Enter a term above...

There is no metadata associated with the current CellML element...

Fig. 1.186: Clicking on *CellML Annotation* lists the CellML components with their variables ready for annotation.

If we now go back to the potassium ion channel CellML model and, under *Editing*, click on *CellML Annotation*, the various elements of the model (Units, Components, Variables, Groups and Connections) are displayed (see Fig. 1.186). If you right click on any of them a popup menu will appear, which you can use to expand/collapse all the child nodes, as well as remove the metadata associated with the current CellML element or the whole CellML file. Expanding *Components* lists all the components and their variables. To annotate the potassium channel component, select it and specify a *Qualifier* from the list displayed:

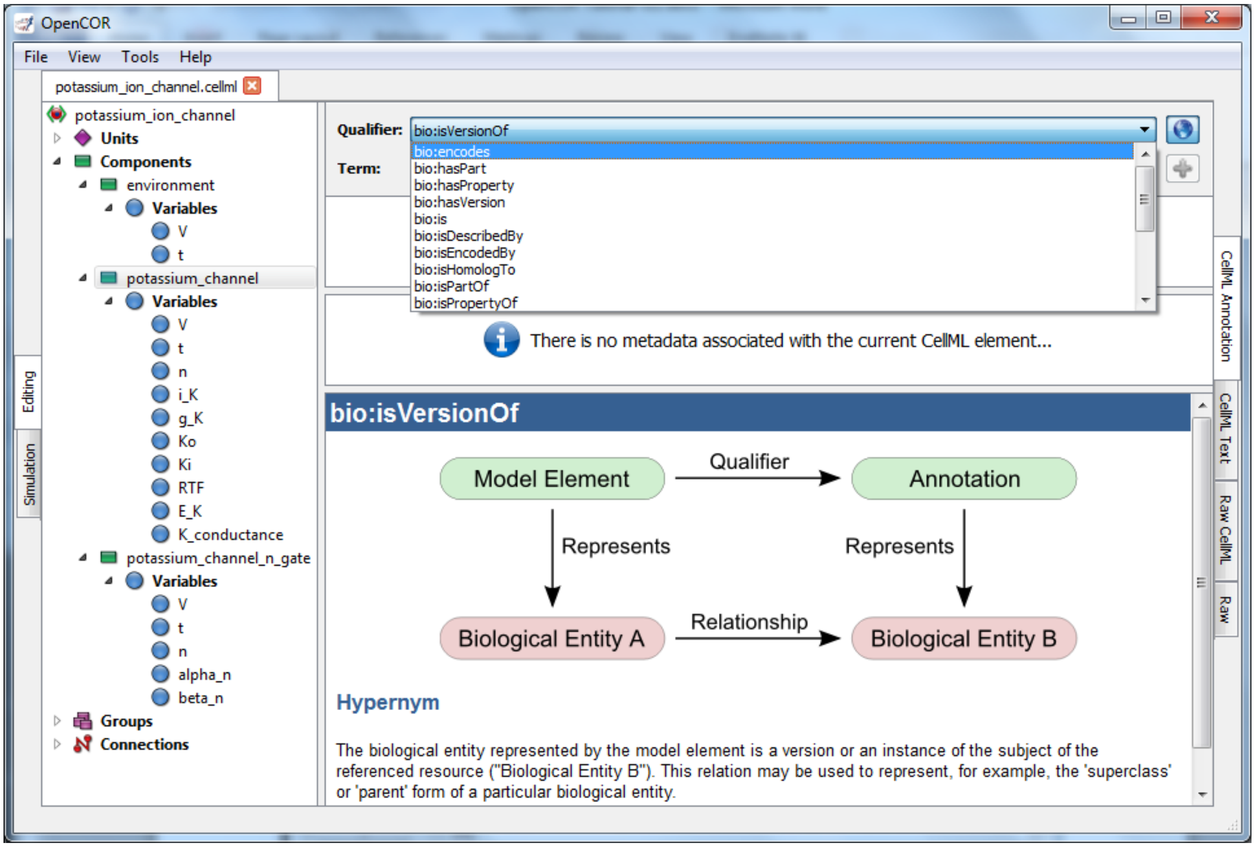
```
bio:encodes,          bio:isPropertyOf
bio:hasPart,          bio:isVersionOf
bio:hasProperty,      bio:occursIn
bio:hasVersion,       bio:hasTaxon
bio:is,               model:is
bio:isDescribedBy,    model:isDerivedFrom
bio:isEncodedBy,      model:isDescribedBy
bio:isHomologTo,      model:isInstanceOf
bio:isPartOf,         model:hasInstance
```

If you do not know which qualifier to use, click on the ● button to get some information about the current qualifier (you must be connected to the internet) and go through the list of qualifiers until you find the one that best suits your needs. Here, we will say that you want to use bio:isVersionOf. Fig. 1.187 shows the information displayed about this qualifier.

Now you need to retrieve some possible ontological terms to describe the *potassium_channel* component. For this you must enter a search term, which in our case is 'potassium channel' (note that regular expressions are supported[7]). This returns 24 possible ontological terms as shown in Fig. 1.188. The *voltage-gated potassium channel complex* is the most appropriate. Clicking on the GO identifier link shown provides more information about this term (see Fig. 1.189).

[6] http://en.wikipedia.org/wiki/Uniform_resource_identifier

[7] http://en.wikipedia.org/wiki/Regular_expression



Qualifier: bio:isVersionOf

Term:

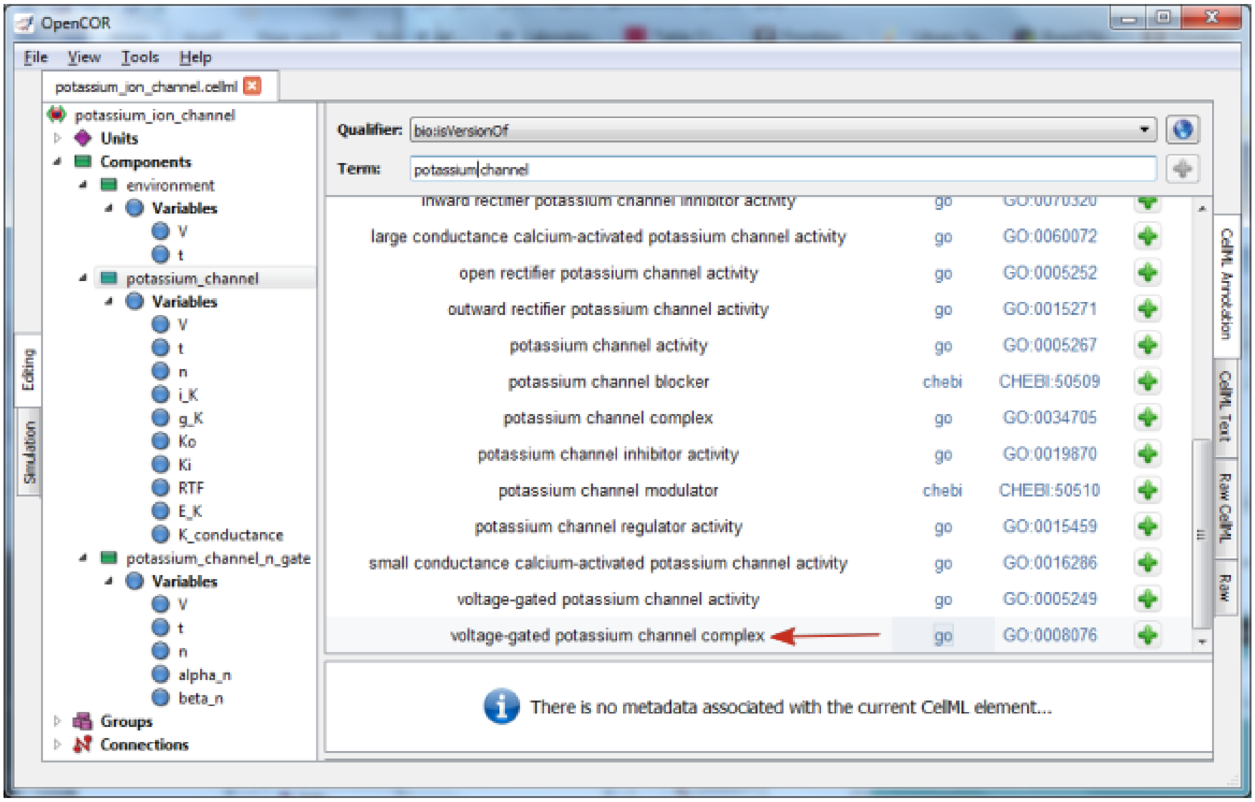
There is no metadata associated with the current CellML element...

bio:isVersionOf

Hypernym

The biological entity represented by the model element is a version or an instance of the subject of the referenced resource ("Biological Entity B"). This relation may be used to represent, for example, the 'superclass' or 'parent' form of a particular biological entity.

Fig. 1.187: The qualifiers are displayed from the top right menu. Clicking on the most appropriate one (bio:isVersionOf) gives more information about this qualifier in the bottom panel.



Qualifier: bio:isVersionOf

Term: potassium channel

inward rectifier potassium channel inhibitor activity	go	GO:0070320	+
large conductance calcium-activated potassium channel activity	go	GO:0060072	+
open rectifier potassium channel activity	go	GO:0005252	+
outward rectifier potassium channel activity	go	GO:0015271	+
potassium channel activity	go	GO:0005267	+
potassium channel blocker	chebi	CHEBI:50509	+
potassium channel complex	go	GO:0034705	+
potassium channel inhibitor activity	go	GO:0019870	+
potassium channel modulator	chebi	CHEBI:50510	+
potassium channel regulator activity	go	GO:0015459	+
small conductance calcium-activated potassium channel activity	go	GO:0016286	+
voltage-gated potassium channel activity	go	GO:0005249	+
voltage-gated potassium channel complex	go	GO:0008076	+

There is no metadata associated with the current CellML element...

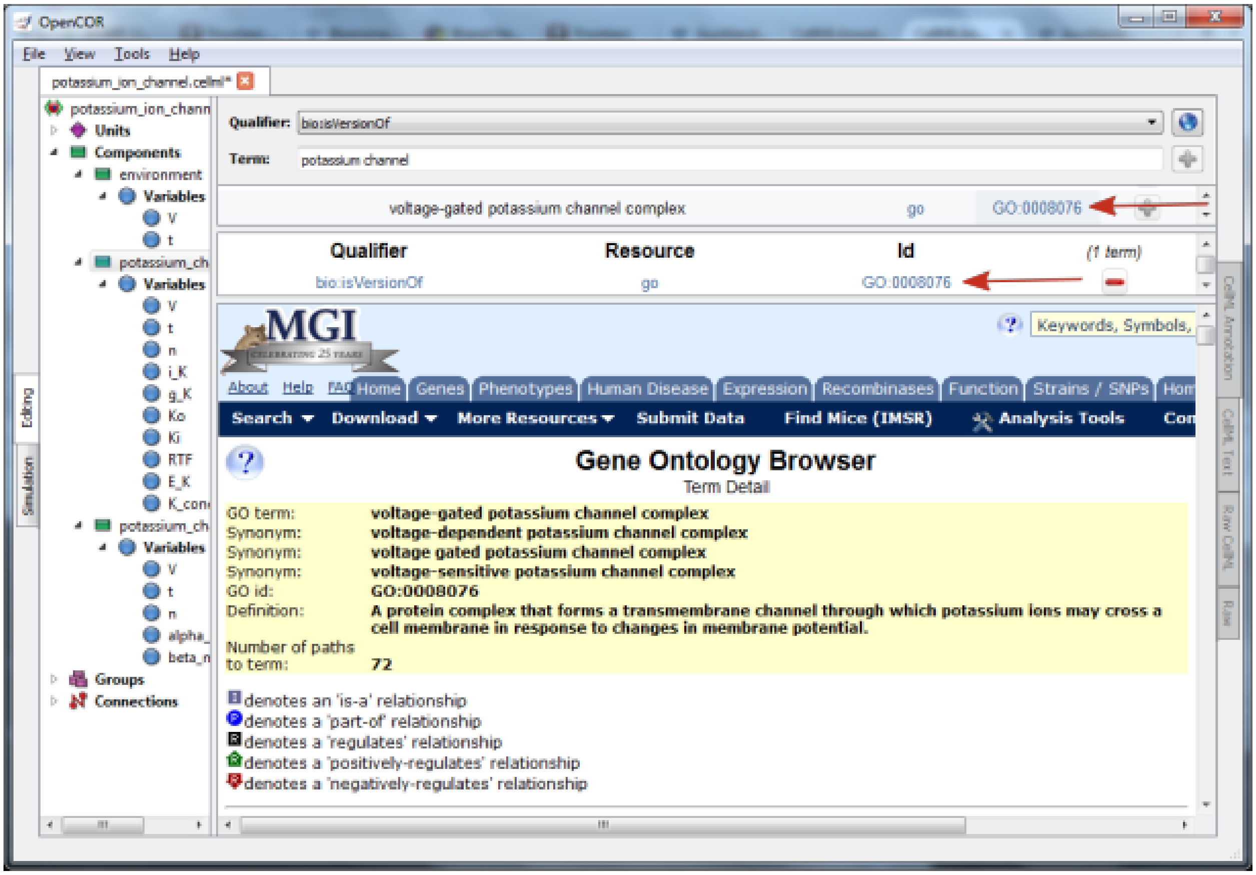
Fig. 1.188: The ontological terms listed when 'potassium channel' is entered into the search box next to *Term*.

Now, assuming that you are happy with your choice of ontological term, you can associate it with the *potassium_channel* component by clicking on its corresponding + button which then displays the qualifier, resource and ID information in the middle panel as shown in Fig. 1.188. If you make a mistake, this can be removed by clicking on the − button.

The first level annotation of the *potassium_channel* component has now been achieved. The content of the three terms in the RDF triple are shown in Fig. 1.190, along with the annotation for the variables *Ki* and *Ko*.

```
def comp {id_000000001} potassium_channel as
   var V: millivolt {pub: in, priv: out};
   var t: millisec {pub: in, priv: out};
   var n: dimensionless {priv: in};
   var i_K: microA_per_cm2 {pub: out};
   var g_K: milliS_per_cm2 {init: 36};
   var {id_000000002} Ki: mM {init: 90};
   var {id_000000003} Ko: mM {init: 3};
   var RTF: millivolt {init: 25};
   var E_K: millivolt;
   var K_conductance: milliS_per_cm2 {pub: out};

   E_K = RTF*ln(Ko/Ki);
   K_conductance = g_K*pow(n, 4{dimensionless});
   i_K = K_conductance*(V-E_K);
enddef;
```

When saved (the *CellML Annotation* tag will appear un-grayed), the result of these annotations is to add metadata to the CellML file. If you switch to the *CellML Text* view you will see that the elements that have been annotated appear



Qualifier: bio:isVersionOf

Term: potassium channel

voltage-gated potassium channel complex go GO:0008076

Qualifier	Resource	Id	(1 term)
bio:isVersionOf	go	GO:0008076	

Gene Ontology Browser

Term Detail

GO term:	**voltage-gated potassium channel complex**
Synonym:	**voltage-dependent potassium channel complex**
Synonym:	**voltage gated potassium channel complex**
Synonym:	**voltage-sensitive potassium channel complex**
GO id:	**GO:0008076**
Definition:	**A protein complex that forms a transmembrane channel through which potassium ions may cross a cell membrane in response to changes in membrane potential.**
Number of paths to term:	**72**

I denotes an 'is-a' relationship
P denotes a 'part-of' relationship
R denotes a 'regulates' relationship
+ denotes a 'positively-regulates' relationship
- denotes a 'negatively-regulates' relationship

Fig. 1.189: The qualifier, resource & ID information in the middle panel appears when you click on the + button next to the selected term in Fig.32. GO identifier details are listed when either of the arrowed links are clicked.

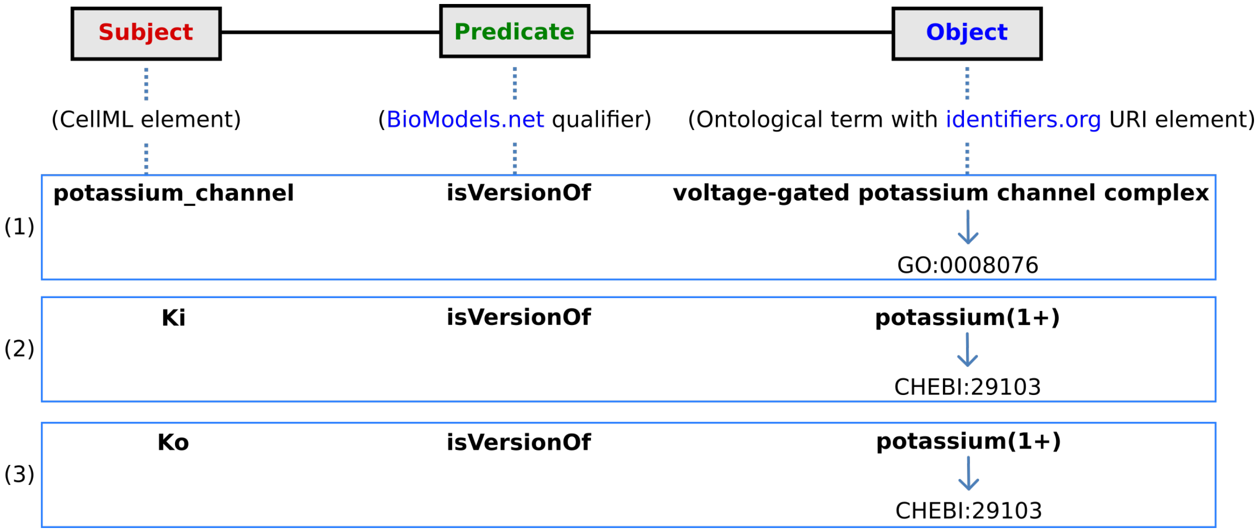


Fig. 1.190: The RDF triple used in CellML metadata to link a CellML element (component or variable) with an ontological term from one of the five ontologies accessed via identifiers.org, using a predicate qualifier from BioModels.net. The three examples of annotated CellML model elements shown are for (1) the *potassium_channel* component (this points to a GO identifier), (2) the variable *Ki*, and (3) the variable *Ko*. These two variables are defined within the *potassium_channel* component of the model and point to CHEBI identifiers. A further annotation is needed to identify the cellular location of those variables (since one is intracellular and one is extracellular).

with ID numbers, as shown above. These point to the corresponding metadata contained in the CellML file for this model and are displayed under the qualifier-resource-Id headings in the annotation window when you click on the element in the editing window.

Note that the three annotations added above are all biological annotations. Many of the other components and variables in the CellML potassium channel model deal with biophysical entities and these require the use of the OPB ontology (yet to be implemented in OpenCOR). The use of composite annotations is also being developed[8], such as “Ki is-the concentration of potassium in-the cytosol of-the neuron of-the giant-squid”, where *concentration*, *potassium*, *cytosol*, *neuron* and *giant-squid* are defined by the ontologies OPB, ChEBI, GO, FMA and a species ontology, respectively.

1.7.8 SED-ML, functional curation and Web Lab

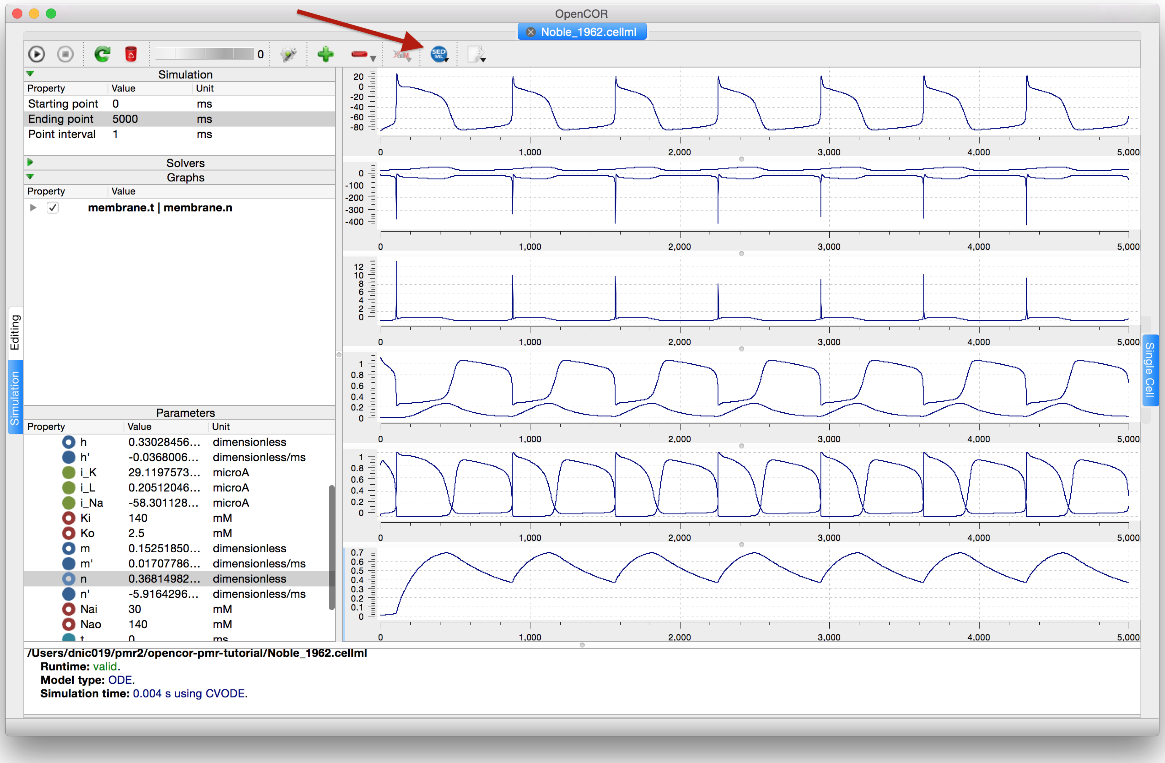
In the same way that CellML models can be defined unambiguously, and shared easily, in a machine- readable format, there is a need to do the same thing with ‘protocols’ - i.e. to define what you have to do to replicate/simulate an experiment, and to analyse the results. An XML standard for this called SED-ML[1] is being developed by the CellML/SBML community and preliminary support for SED-ML has been implemented in OpenCOR in order to allow precise and reproducible control over the OpenCOR simulation and graphical output (e.g., see Fig. 1.76).

The current snapshot release (2016-02-27) of OpenCOR supports exporting the *Simulation view* configuration to a SED-ML file, which can then be read back into OpenCOR to reproduce a given simulation experiment, illustrated in Fig. 1.191.

Support for SED-ML will also facilitate the curation of models according to their functional behaviour under a range of experimental scenarios. The key idea behind functional curation is that, when mathematical and computational models are being developed, a primary goal should be the continuous comparison of those models against experimental data.

[8] This is a project being carried out at the University of Washington, Seattle, using an annotation tool called SEMGEN (. . .).

[1] The ‘Simulation Experiment Description Markup Language’: sed-ml.org



Simulation

Property	Value	Unit
Starting point	0	ms
Ending point	5000	ms
Point interval	1	ms

Solvers

Graphs

Property	Value
☑ membrane.t | membrane.n	

Parameters

Property	Value	Unit
h	0.33028456...	dimensionless
h'	-0.0368006...	dimensionless/ms
i_K	29.1197573...	microA
i_L	0.20512046...	microA
i_Na	-58.301128...	microA
Ki	140	mM
Ko	2.5	mM
m	0.15251850...	dimensionless
m'	0.01707786...	dimensionless/ms
n	0.36814982...	dimensionless
n'	-5.9164296...	dimensionless/ms
Nai	30	mM
Nao	140	mM
t	0	ms

/Users/dnic019/pmr2/opencor-pmr-tutorial/Noble_1962.cellml
Runtime: valid.
Model type: ODE.
Simulation time: 0.004 s using CVODE.

Fig. 1.191: Once you are happy with the configuration of the *Simulation view* in OpenCOR, clicking the SED-ML button (highlighted) will prompt for a file to save the SED-ML document to. This document can be loaded back into OpenCOR to reproduce the simulation, or shared with collaborators so they can reproduce the simulation.

When computational models are being re-used in new studies, it is similarly important to check that they behave appropriately in the new situation to which you're applying them. To achieve this goal, a pre-requisite is to be able to replicate in-silico precisely the same protocols used in an experiment of interest. A language for describing rich 'virtual experiment' protocols and software for running these on compatible models is being developed in the Computational Biology Group at Oxford University[2]. An online system called Web Lab[3] is also being developed that supports definition of experimental protocols for cardiac electrophysiology, and allows any CellML model to be tested under these protocols [CJ15]. This enables comparison of the behaviours of cellular models under different experimental protocols: both to characterise a model's behaviour, and comparing hypotheses by seeing how different models react under the same protocol (Fig. 1.192 adapted from [CJ15]).

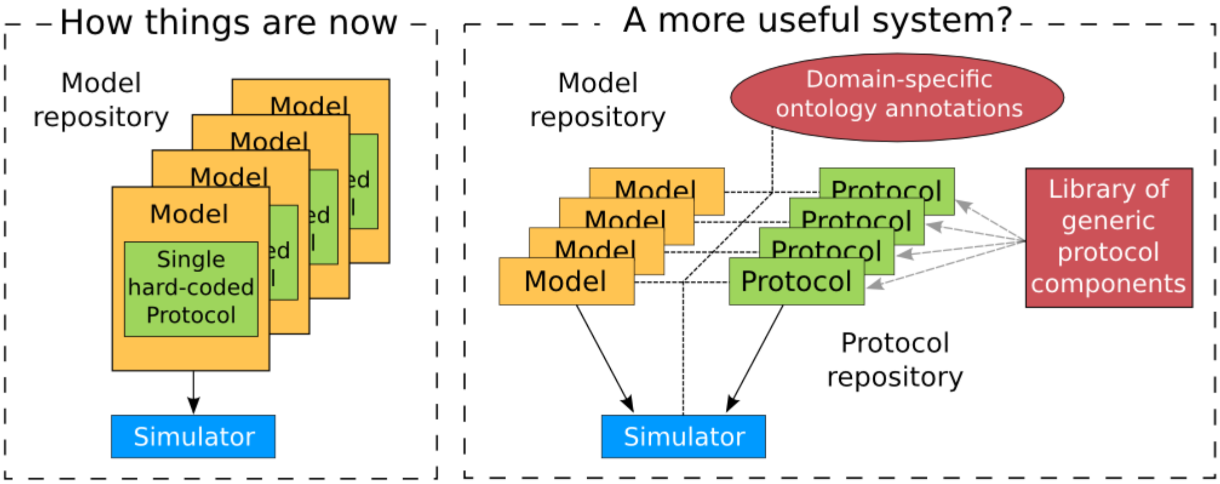


Fig. 1.192: A schematic of the way we organise model and protocol descriptions. Web Lab provides an interface to a Model/Protocol Simulator, storing and displaying the results for cardiac electrophysiology models.

The Web Lab website provides tools for comparing how two different cardiac electrophysiology models behave under the same experimental protocols. Note that Web Lab demonstration for CellML models of cardiac electrophysiology is a prototype for a more general approach to defining simulation protocols for all CellML models.

1.7.9 Speed comparisons with MATLAB

Solution speed is important for complex computational models and here we compare the performance of OpenCOR with MATLAB[1]. Nine representative CellML models were chosen from the PMR model repository. For the MATLAB tests we used the MATLAB code, generated automatically from CellML, that is available on the PMR site. These comparisons are based on using the default solvers (listed below) available in the two packages.

Testing environment

- MacBook Pro (Retina, Mid 2012).
- Processor: 2.6 GHz Intel Core i7.
- Memory: 16 GB 1600 MHz DDR3.

[2] travis.cs.ox.ac.uk/FunctionalCuration/about.html This initiative is led by Jonathan Cooper and Gary Mirams.
[3] travis.cs.ox.ac.uk/FunctionalCuration.
[1] www.mathworks.com/products/matlab

- Operating system: OS X Yosemite 10.10.3.

OpenCOR

- Version: 0.4.1.
- Solver: CVODE with its default settings, except for its Maximum step parameter, which is set to the model's stimulation duration, if needed.

MATLAB

- Version: R2013a.
- Solver: ode15s (i.e. a solver suitable for stiff problems and which has low to medium order of accuracy) with both its RelTol and AbsTol parameters set to 1e-7 and its MaxStep parameter set to the stimulation duration, if needed.

Testing protocol

- Run a model for a given simulation duration.
- Generate simulation data every milliseconds.
- Only keep track of all the simulation data (i.e. no graphical output).
- Run a model 7 times, discard the 2 slowest runs (to account for unpredictable slowdowns of the testing machine) and average the resulting computational times.
- Computational times are obtained directly from OpenCOR and MATLAB (through a couple of calls to cputime in the case of MATLAB).

Results

CellML model (from PMR on 18/6/2015)	**Duration** (s)	**OpenCOR time** (s)	**MATLAB time** (s)	**Time ratio** (MATLAB/OpenCOR)
Bondarenko et al. 2004	10	1.16	140.14	121
Courtemanche et al. 1998	100	0.998	45.720	46
Faber & Rudy 2000	50	0.717	29.010	40
Garny et al. 2003	100	0.996	48.180	48
Luo & Rudy 1991	200	0.666	70.070	105
Noble 1962	1000	1.42	310.02	218
Noble et al. 1998	100	0.834	42.010	50
Nygren et al. 1998	100	0.824	31.370	38
ten Tusscher & Panfilov 2006	100	0.969	59.080	61

*The value of membrane.stim_end was increased so as to get action potentials for the duration of the simulation

Conclusions

For this range of tests, OpenCOR is between 38 and 218 times faster than MATLAB. A more extensive evaluation of these results is available on GitHub[2].

[2] https://github.com/opencor/speedcomparison. These tests were carried out by Alan Garny.

1.7.10 Future developments

Both CellML and OpenCOR are continuing to be developed. These notes will be updated to reflect new features of both. The next release of OpenCOR (0.5) will include

- the SED-ML API which means that all the variables controlling the simulation and its output can be specified in a file for that simulation
- the BioSignalML API which will allow experimental data to be read into OpenCOR in a standardised way
- colour plots, to better distinguish overlapping traces in the output windows

Priorities for later releases of OpenCOR include the incorporation of GIT into OpenCOR to enable the upload of models to PMR, graphical rendering of the model structure (using SVG), model building templates, such as templates for creating Markov models, tools for parameter estimation and tools for analysing model outputs.

The next release of CellML (1.2) will include the ability to specify a probability distribution for a parameter value. Together with SED-ML, this will allow OpenCOR to generate error bounds on the solutions, corresponding to the specified parameter uncertainty.

Link to weblab.

These notes are currently being extended to include

- a discussion of system identification and parameter estimation
- more extensive discussion of membrane protein models
- CellML modules for signal transduction pathways

1.7.11 References

See https://www.cellml.org/getting-started/tutorials.

Latest version rendered at: http://tutorial-on-cellml-opencor-and-pmr.readthedocs.io/en/latest/.

Todo:

- Colour background of *CellML Text*
- Annotate screen shots with svg for same look and feel
- *CellML Text* code is not highlighted for all display situations, currently only in environments that are using an adapted version of pygments
- Tidy up citations and BiBTeX source (possibly use Zotero to manage?)
- Make horizontal line for footnotes only visible in html output
- Check external references markup
- Consider a more suitable theme (may require changes to an existing one to get a good result)
- Must check over output (and models) from screenshots to make sure that it matches the current release of OpenCOR, especially against running experiments for the first time.

1.8 Auckland Physiome Repository

The documentation found here is mainly aimed towards providing information to users of the Auckland Physiome 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.

1.8.1 Quickstart guides

Here we are creating some quickstart guides for some common tasks. If you have a task that you think would benefit from such a guide, please let us know.

Contents

Uploading a model to the repository

While it is desirable to use some kind of version control system to manage the development of any work performed on a computer, it is often the case that a user has a model, in CellML for example, that they simply wish to upload to the model repository. It is hoped that soon tools such as OpenCOR will natively support such a task, but for now users are currently required to learn a bit of *Git* in order to upload such models themselves. An alternative is to simply contact us with your model and we will upload it for you, but this will loose any direct association or connection with the original author of the model. We therefore highly recommend that the following process is followed at this time.

1. create a *workspace*

   More detail is available at *Creating a new workspace*. Note the *teaching instance* that is available for testing things out if required.

2. use a *git* client to *clone* the workspace to your local machine

   To ensure an accurate record of your contributions is maintained, please see *Git username configuration*. A detailed description of cloning a workspace is available here: *Cloning your forked workspace*.

3. add your model and any other data to the workspace (such as a brief documentation page in HTML or reStructuredText and any images used in the documentation)

   See *Committing changes* for details on adding the files to the cloned workspace on your local machine.

4. push the changes back to the repository

   Details available here: *Pushing changes to the repository*.

5. [optional] share the workspace with any collaborators

   Your workspace will currently be private and only visible to yourself when you are logged in to the repository. If you would like to share the workspace with your collaborators while still keeping the work private, you are able to do so: *Working with collaborators*.

6. [optional] create an *exposure*

   If you have some documenation for your workspace or would like to easily link to a specific revision of your workspace, you are able to *create an exposure* for the workspace. Like the workspace, this will initially be private to yourself and the users you choose to share it with. For CellML models, see extra information here: *Creating CellML exposures*.

7. submit the workspace for "publication" - this will make the workspace publicly available

   In order to make your workspace publicly available, you need to submit it for publication. The publication process simply allows the repository administrators to quickly check the workspace for material that should not be distributed in the repository (once a workspace is published, it will be available *forever*). The process is the same as that described here: *Making your work publicly accessible*.

8. [optional] submit the exposure for publication (again, this makes it public)

   Once the workspace is published, you are able to submit any exposure(s) you might want public for publication.

1.8.2 Auckland Physiome Repository - an introduction

The Auckland Physiome Repository is the combination of what previously was the CellML Model Repository and FieldML repositories, and is powered by a suite of software packages that collectively is known as PMR2. This repository includes PMR2 relies on the distributed version control system Git, 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 Git client for your operating system, and become familiar with the basic functions of Git. There are many excellent resources available on the internet and *linked on the Git website*. A notable one is the Git Beginner's Guide for Dummies as it has detailed guides on installation procedures for the major operating systems and the overall guide is quite excellent.

1.8.3 Downloading and viewing models from the Auckland Physiome Repository

There are several ways of obtaining and using models from the Auckland Physiome 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 *Git* 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

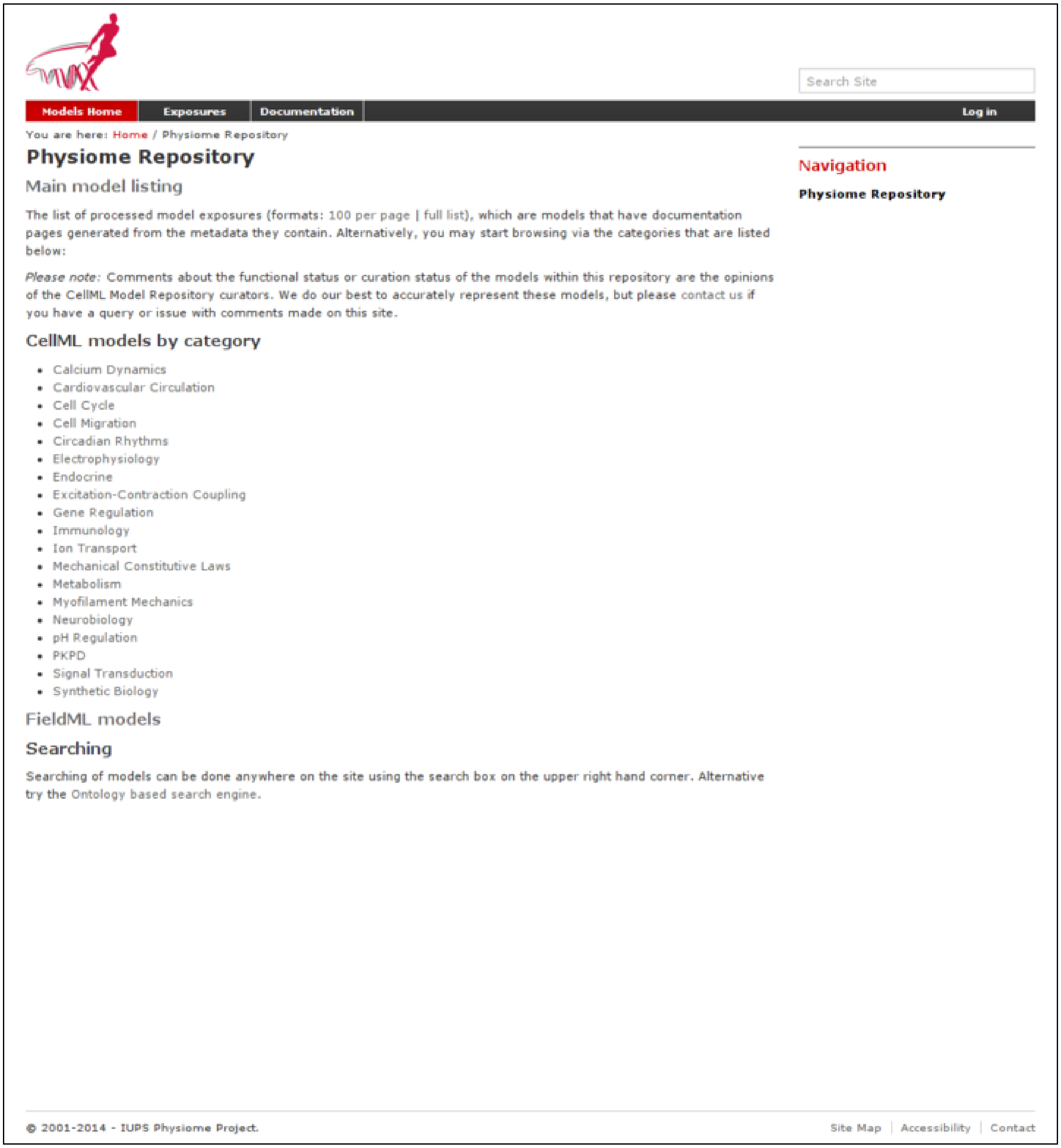
The Auckland Physiome Repository has a basic search function that can be accessed by typing search terms into the box at the top right hand side of the page. You can use keywords such as `cardiac` or `insulin`, author names, or any other terms relevant to the models you want to find.

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).

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 Auckland Physiome 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.



You are here: Home / Physiome Repository

Physiome 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

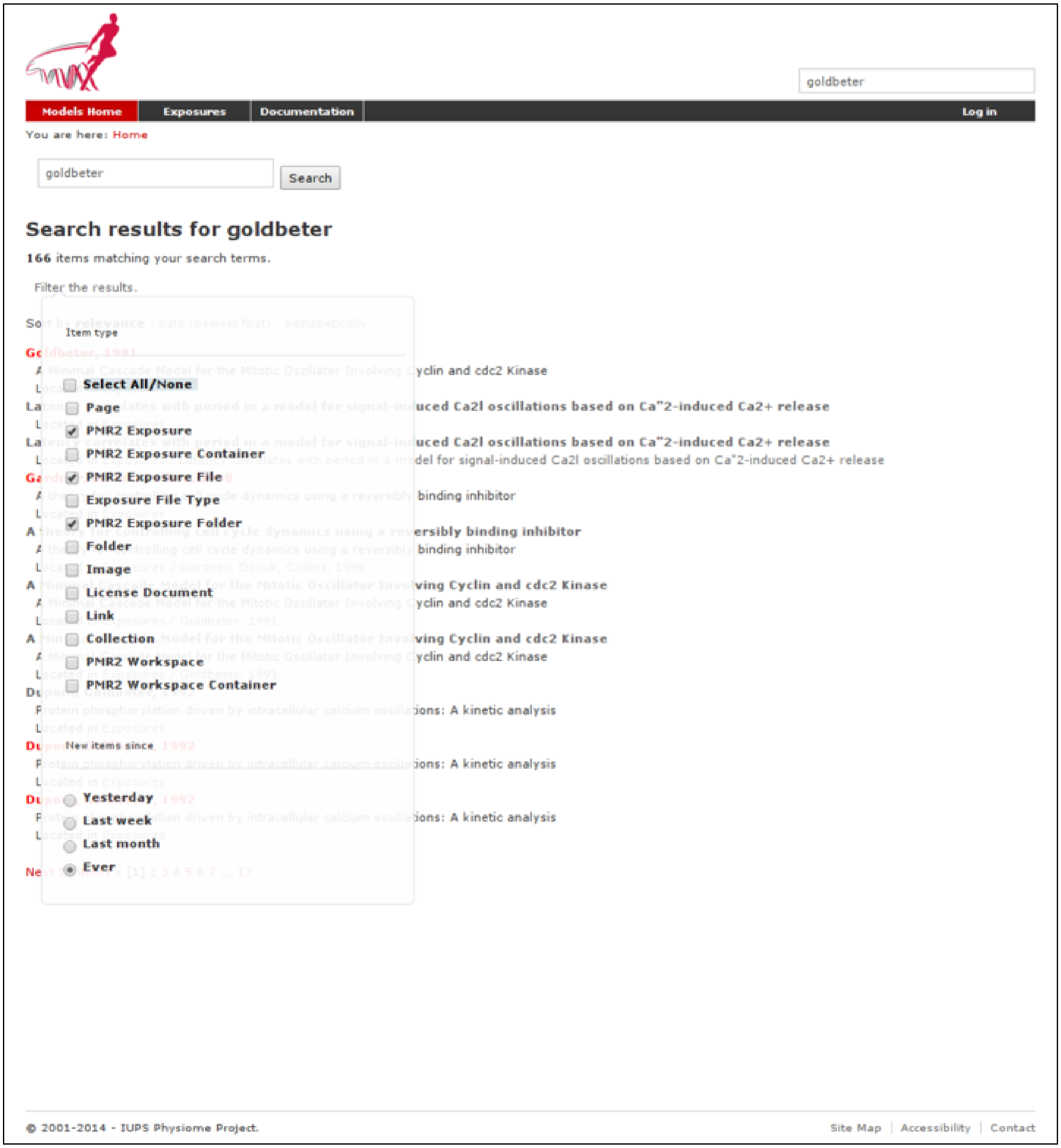
Searching

Searching of models can be done anywhere on the site using the search box on the upper right hand corner. Alternative try the Ontology based search engine.

© 2001-2014 - IUPS Physiome Project.

Site Map | Accessibility | Contact

Fig. 1.193: 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.



You are here: Home

goldbeter [Search]

Search results for goldbeter

166 items matching your search terms.

Filter the results.

Item 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

- ○ Yesterday
- ○ Last week
- ○ Last month
- ● Ever

Sort by relevance · date (newest first) · alphabetically

Goldbeter, 1991
A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
Located in Exposures

Latency correlates with period in a model for signal-induced Ca2l oscillations based on Ca"2-induced Ca2+ release
Located in Exposures

Latency correlates with period in a model for signal-induced Ca2l oscillations based on Ca"2-induced Ca2+ release
Located in Exposures / Latency correlates with period in a model for signal-induced Ca2l oscillations based on Ca"2-induced Ca2+ release

Gardner, Dolnik, Collins, 1998
A theory for controlling cell cycle dynamics using a reversibly binding inhibitor
Located in Exposures

A theory for controlling cell cycle dynamics using a reversibly binding inhibitor
A theory for controlling cell cycle dynamics using a reversibly binding inhibitor
Located in Exposures / Gardner, Dolnik, Collins, 1998

A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
Located in Exposures / Goldbeter, 1991

A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
Located in Exposures / Goldbeter, 1991

Dupont, Goldbeter, 1992
Protein phosphorylation driven by intracellular calcium oscillations: A kinetic analysis
Located in Exposures

Dupont, Goldbeter, 1992
Protein phosphorylation driven by intracellular calcium oscillations: A kinetic analysis
Located in Exposures

Dupont, Goldbeter, 1992
Protein phosphorylation driven by intracellular calcium oscillations: A kinetic analysis
Located in Exposures

Next 10 items » [1] 2 3 4 5 6 7 ... 17

© 2001-2014 - IUPS Physiome Project.

Fig. 1.194: In this search only exposure related items are to be shown in the 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 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:

Downloading models via Git

All data in the Auckland Physiome Repository are stored in *workspaces* and each *workspace* is a *Git* repository. The most comprehensive method of downloading content from Auckland Physiome Repository is to clone the workspace containing the desired data. In this manner you will have a local copy of the entire history of that data, including all provenance data, and the ability to step back through the history of the workspace to a state that may not be available via the download links in the exposure pages discussed above. If you would like to modify the contents of workspace, making use of Git will ensure accurate provenance records are maintained as well as all the other benefits of using a version control system.

As software tools like OpenCOR and MAP Client evolve, they will be able to hide a lot of the Git details and present the user with a user interface suitable for their specific application areas. Directly using Git is, however, currently the most powerful way to leverage the full capabilities of Auckland Physiome Repository.

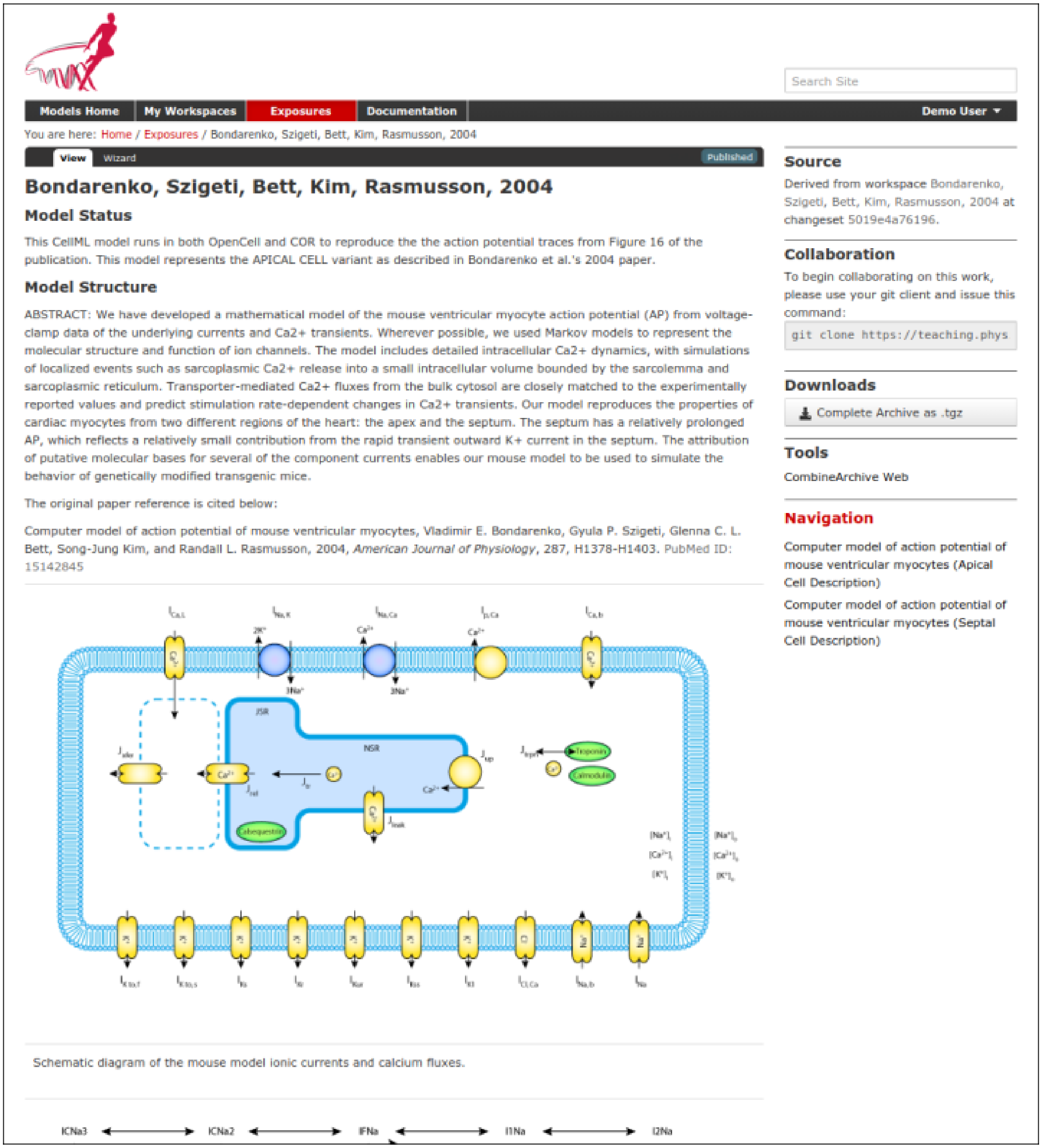
If you are using the command line Git client, you can easily clone the underlying repository for an exposure simply by selecting the text box inside the **Collaboration** portlet and paste that command into a terminal, or right click on the name of the workspace under the **Source** portlet and copy that URL and then paste that into your Git client.

Detailed instructions for working with Git can be found in the *CellML repository tutorial*.

1.8.4 Working with workspaces

All models in the Auckland Physiome Repository exist in *workspaces*, which are *Git* repositories that can be used to store any kind of file. Git is a distributed version control system (DVCS).

In order to create your own workspaces, you will first need to create a repository account by registering at models.physiomeproject.org. Near the top right of the repository page there will be links labelled *Log in* and *Register*. Click on the register link, and follow the instructions.



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.

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset 5019e4a76196.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone https://teaching.phys
```

Downloads

Complete Archive as .tgz

Tools

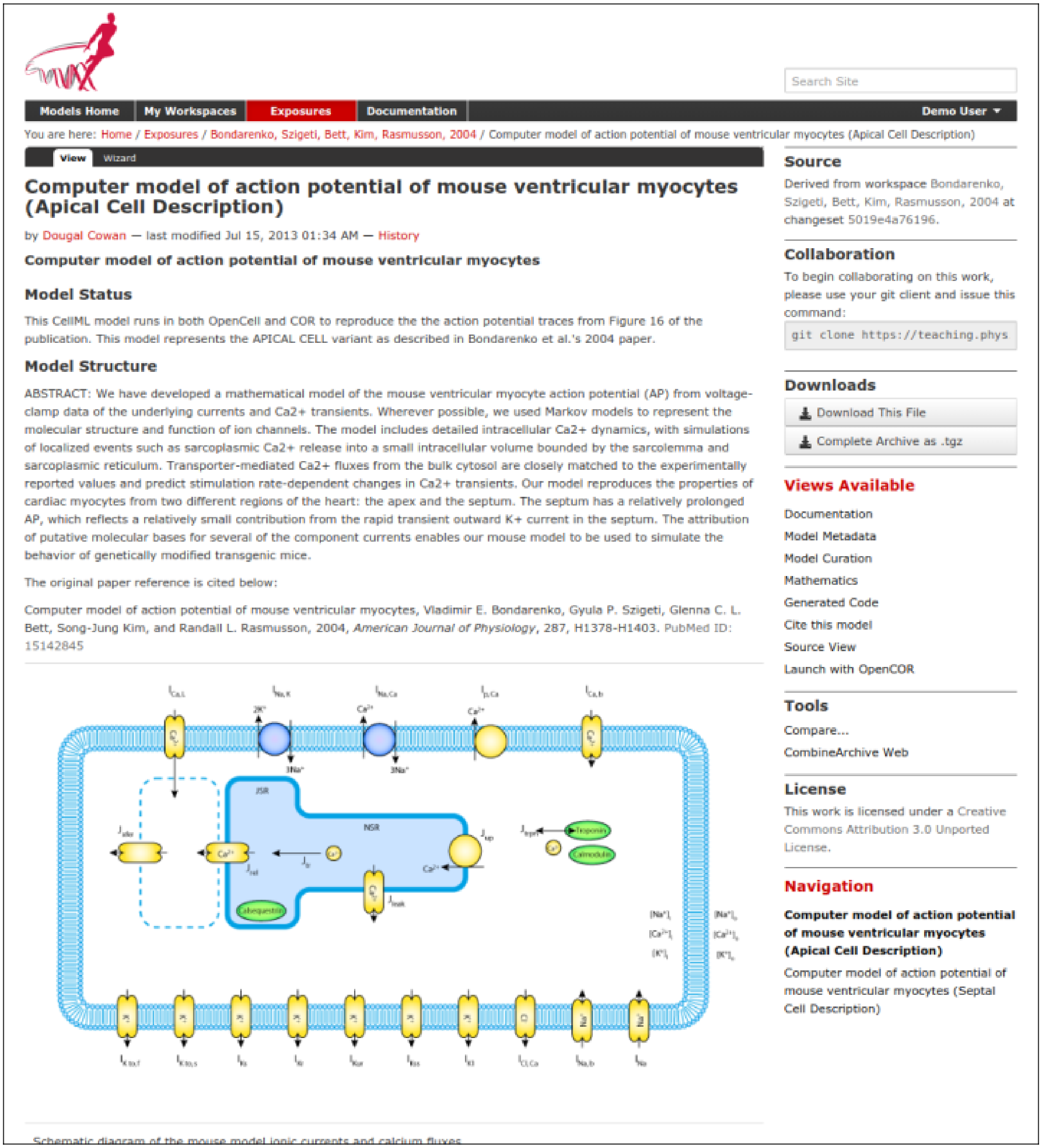
CombineArchive Web

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. 1.195: 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 Dougal Cowan — last modified Jul 15, 2013 01:34 AM — History

Computer model of action potential of mouse ventricular myocytes

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

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset 5019e4a76196.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone https://teaching.phys
```

Downloads

Views Available

Tools

License

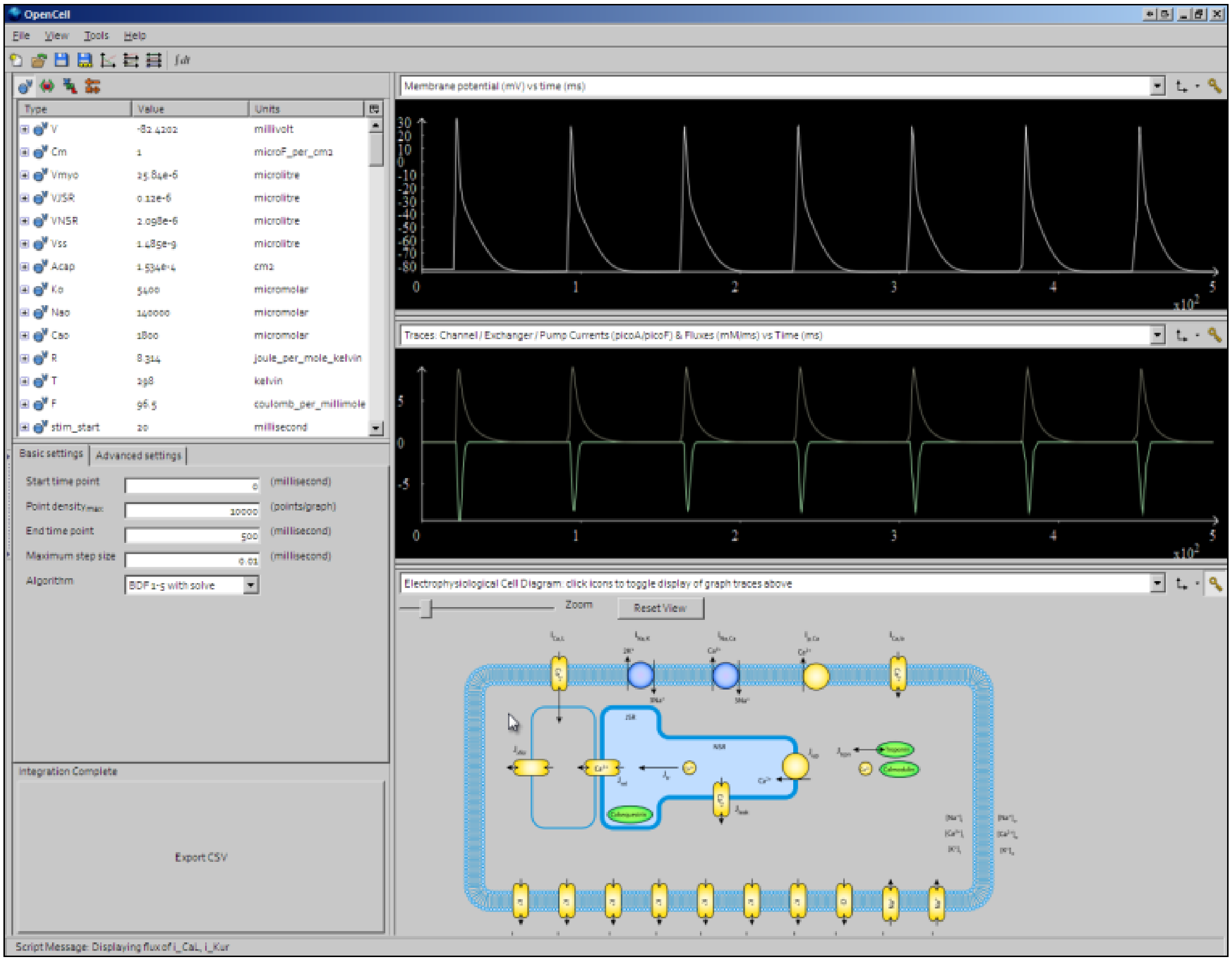
This work is licensed under a Creative Commons Attribution 3.0 Unported License.

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. 1.196: An example of a CellML exposure sub-page.



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

Membrane potential (mV) vs time (ms)

Traces: Channel / Exchanger / Pump Currents (picoA/picoF) & Fluxes (mM/ms) vs Time (ms)

Electrophysiological Cell Diagram. click icons to toggle display of graph traces above

Script Message: Displaying flux of i_CaL, i_Kur

Fig. 1.197: 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.

Workspaces in the Auckland Physiome Repository are permanent once they are created. There is a teaching instance of the repository which may be used for *experimenting* with features of the software without worrying about creating permanent workspaces that might have errors in them. Users accounts and data from the Auckland Physiome Repository will be copied to the teaching instance periodically, overwriting all data there in the process, but users may register for an account just on the teaching instance if they prefer. Such accounts will need to be recreated each time the teaching instance is overwritten.

Note: The teaching instance of the repository is a mirror of the main repository site found at https://teaching.physiomeproject.org/, running the latest development version of *PMR2*. User accounts are periodicly synchronised from the main repository, but if you recently created an account on the main site you might need to also create a new account on the teaching instance.

Any changes you make to the contents of the teaching instance are not permanent, and will be overwritten with the contents of the main repository whenever the teaching instance is upgraded to a new release of PMR2. For this reason, you can feel free to experiment and make mistakes when pushing to the teaching instance. Please subscribe to the cellml-discussion mailing list to receive notifications of when the teaching instance will be refreshed.

See the section *Migrating content to the main repository* for instructions on how to migrate any content from the teaching instance to the main (permanent) Auckland Physiome Repository.

Creating a new workspace

Once a user is logged into Auckland Physiome Repository, they will be presented with a *My Workspaces* link in the top toolbar, as shown below:

The first paragraph includes a link to your dashboard to add a new workspace, shown below:

Currently *Git* is the only avialable option for the storage method for a new workspace, but this may be expanded to include other storage methods in future. A workspace should be given a meaningful title and a brief description to help locate the workspace using the repository search. Both these fields can be edited later, so don’t worry if you don’t get it perfect the first time.

Clicking the *Add* button with then create the workspace, which will initially be empty, as shown below:

In the figure above, the URI of the newly created workspace has been highlighted. This is the URI that will be used when operating on the workspace using Git.

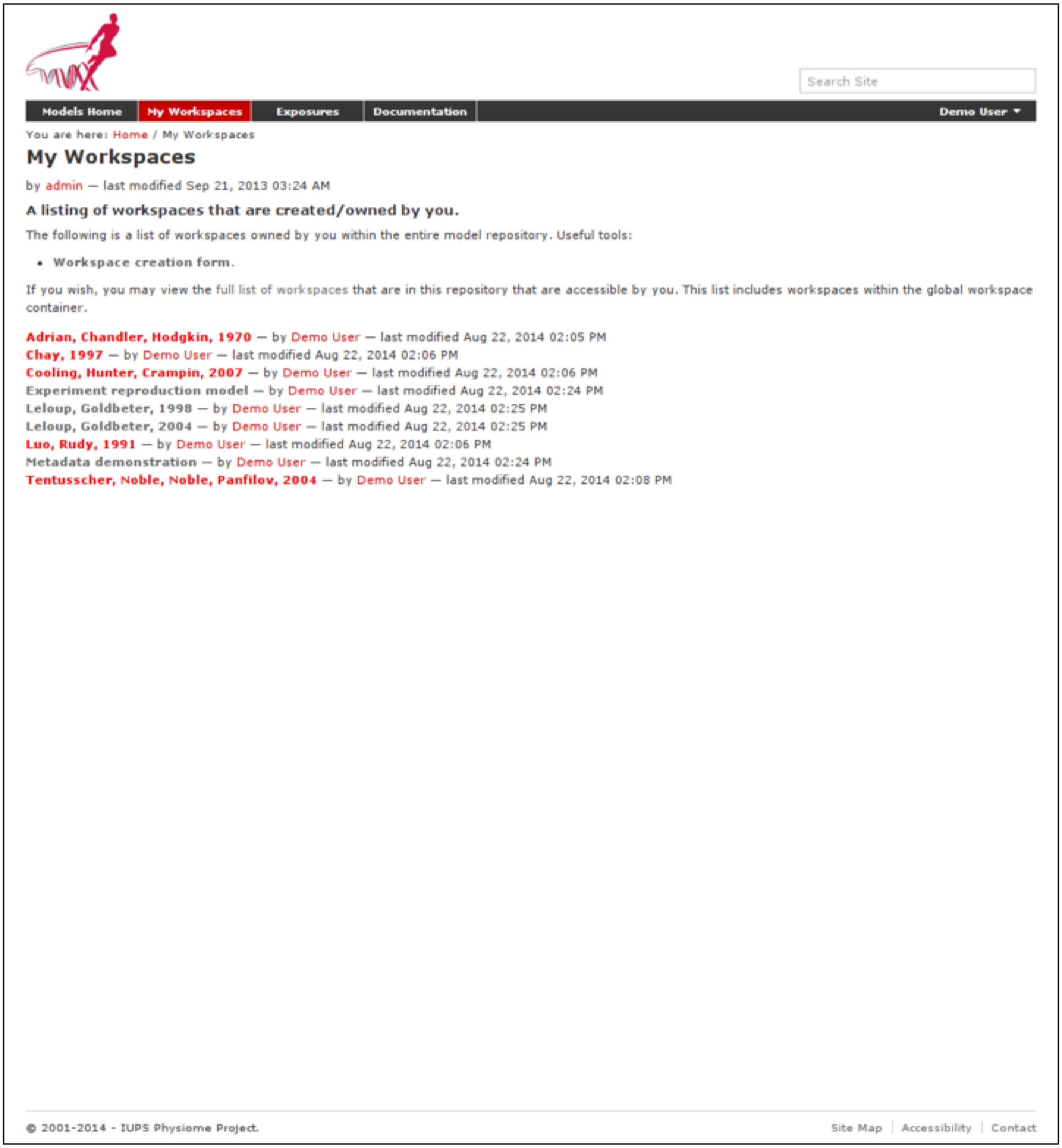
Working with collaborators

The repository makes use of *Git* to manage individual workspaces. Git is a Distributed Version Control System (DVCS), and as such encourages collaborative development of your model, dataset, results, *etc*. Using Git, each member of the development team is able to have their own clone of the workspace which can be kept synchronized with the other members of the development team, while ensuring that each team member’s contributions are accurately recorded in the workspace history.

Once a *workspace* has been published, any registered users (or members) of the repository is able to access and clone the workspace, including team members and the anonymous public. Only the owner and those with privileges granted by the owner are able to make changes to the workspace, including *pushing* changes into the Git repository. Private workspaces, however, can only be viewed by its owner and those with viewing privileges granted by its owner.

Auckland Physiome Repository provides access controls to manage the ability of its members and anonymous users to interact with workspaces. The access control is managed via the *Sharing* tab for a given workspace, as shown below.

By default, you will initially see that all logged-in user has the **Can add** permission. That is the inherited permission from the global workspace container, and does not imply that they can view your work as that is determined by the **Can view** permission. This also does not mean that they can add data to your workspace. This permission setting is



You are here: Home / My Workspaces

My Workspaces

by admin — last modified Sep 21, 2013 03:24 AM

A listing of workspaces that are created/owned by you.

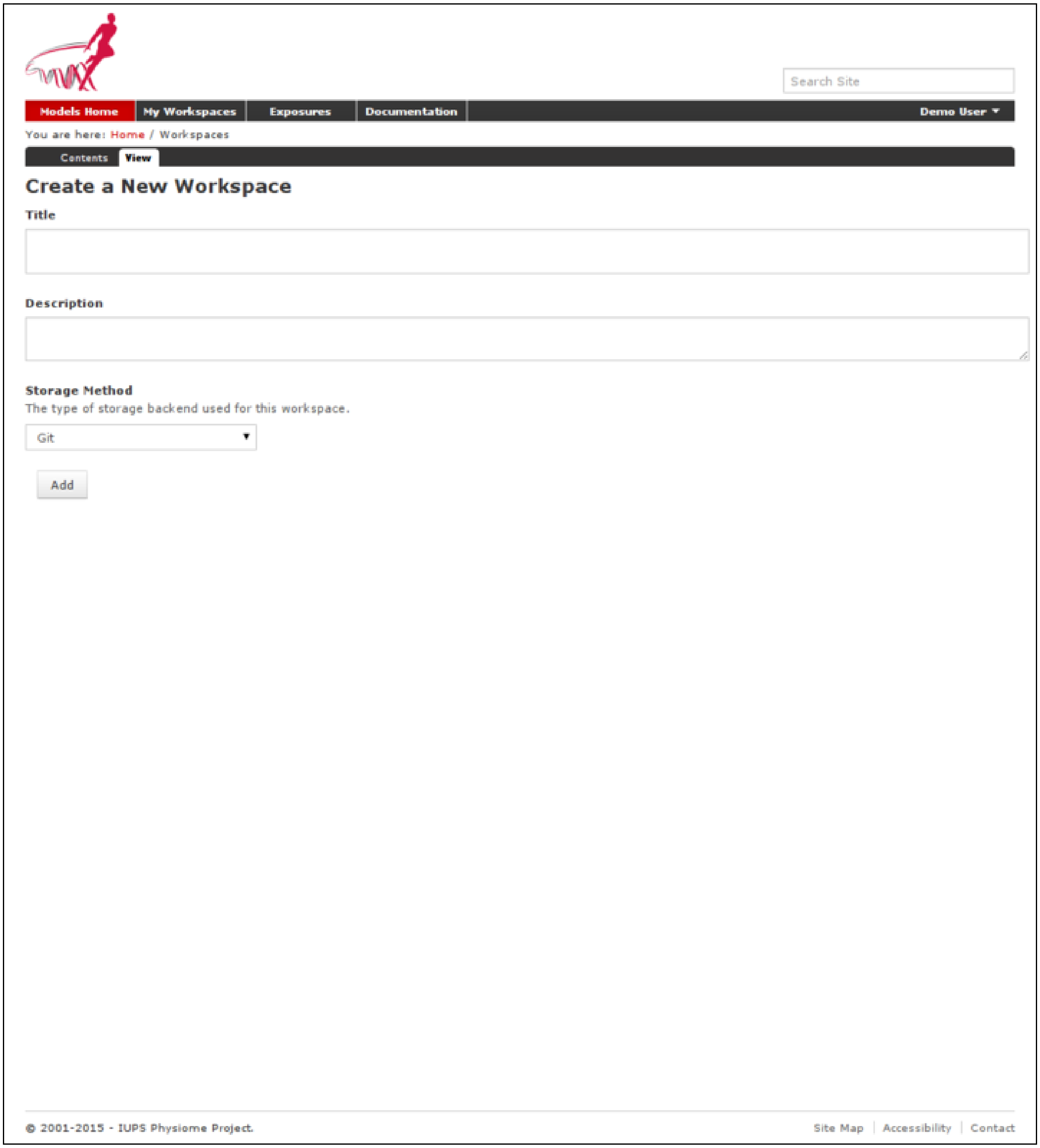
The following is a list of workspaces owned by you within the entire model repository. Useful tools:

- **Workspace creation form.**

If you wish, you may view the full list of workspaces that are in this repository that are accessible by you. This list includes workspaces within the global workspace container.

Adrian, Chandler, Hodgkin, 1970 — by Demo User — last modified Aug 22, 2014 02:05 PM
Chay, 1997 — by Demo User — last modified Aug 22, 2014 02:06 PM
Cooling, Hunter, Crampin, 2007 — by Demo User — last modified Aug 22, 2014 02:06 PM
Experiment reproduction model — by Demo User — last modified Aug 22, 2014 02:24 PM
Leloup, Goldbeter, 1998 — by Demo User — last modified Aug 22, 2014 02:25 PM
Leloup, Goldbeter, 2004 — by Demo User — last modified Aug 22, 2014 02:25 PM
Luo, Rudy, 1991 — by Demo User — last modified Aug 22, 2014 02:06 PM
Metadata demonstration — by Demo User — last modified Aug 22, 2014 02:24 PM
Tentusscher, Noble, Noble, Panfilov, 2004 — by Demo User — last modified Aug 22, 2014 02:08 PM

© 2001-2014 - IUPS Physiome Project.



Create a New Workspace

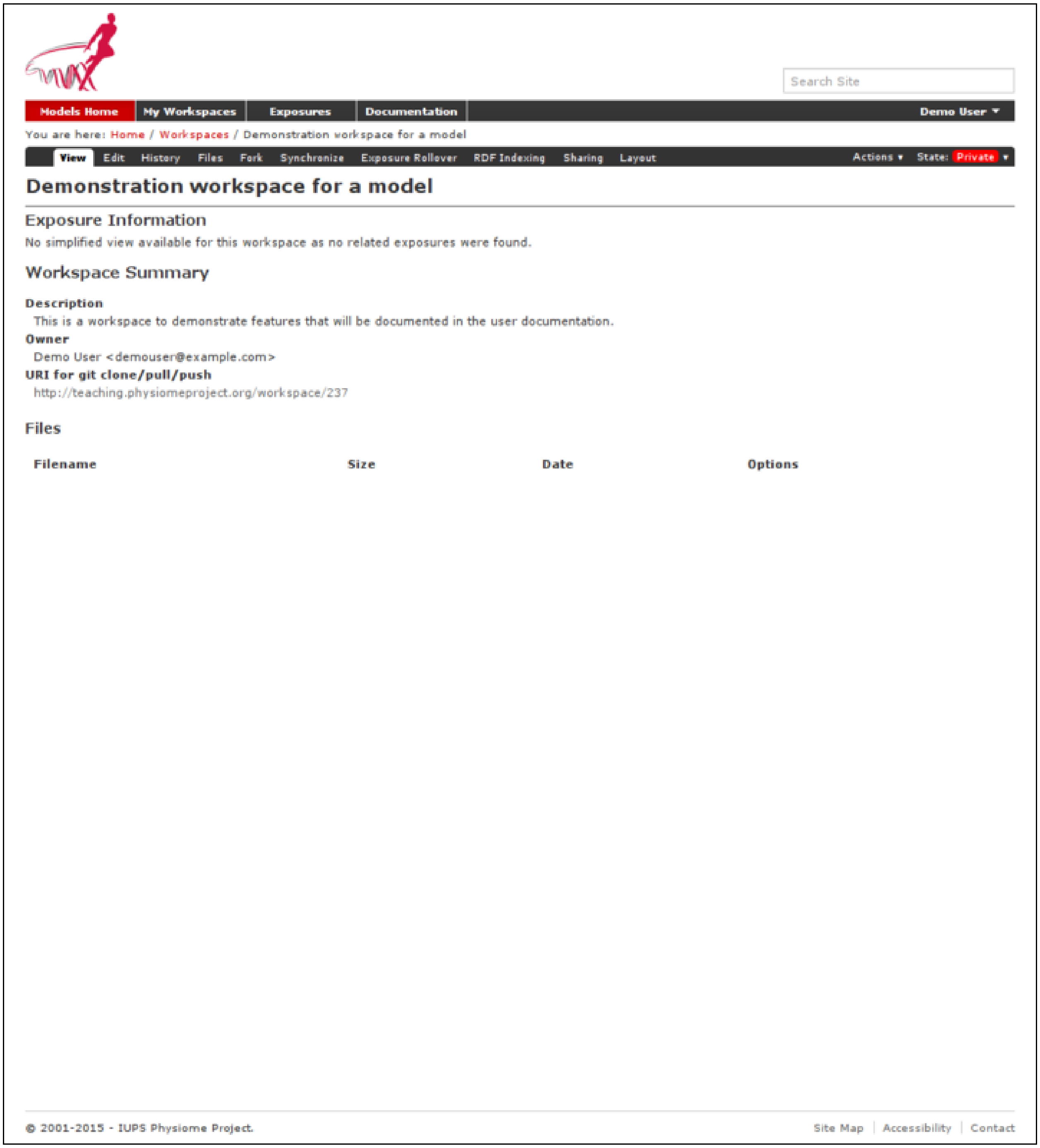
Title

Description

Storage Method

The type of storage backend used for this workspace.

Git



You are here: Home / Workspaces / Demonstration workspace for a model

Demonstration workspace for a model

Exposure Information

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

Workspace Summary

Description
This is a workspace to demonstrate features that will be documented in the user documentation.

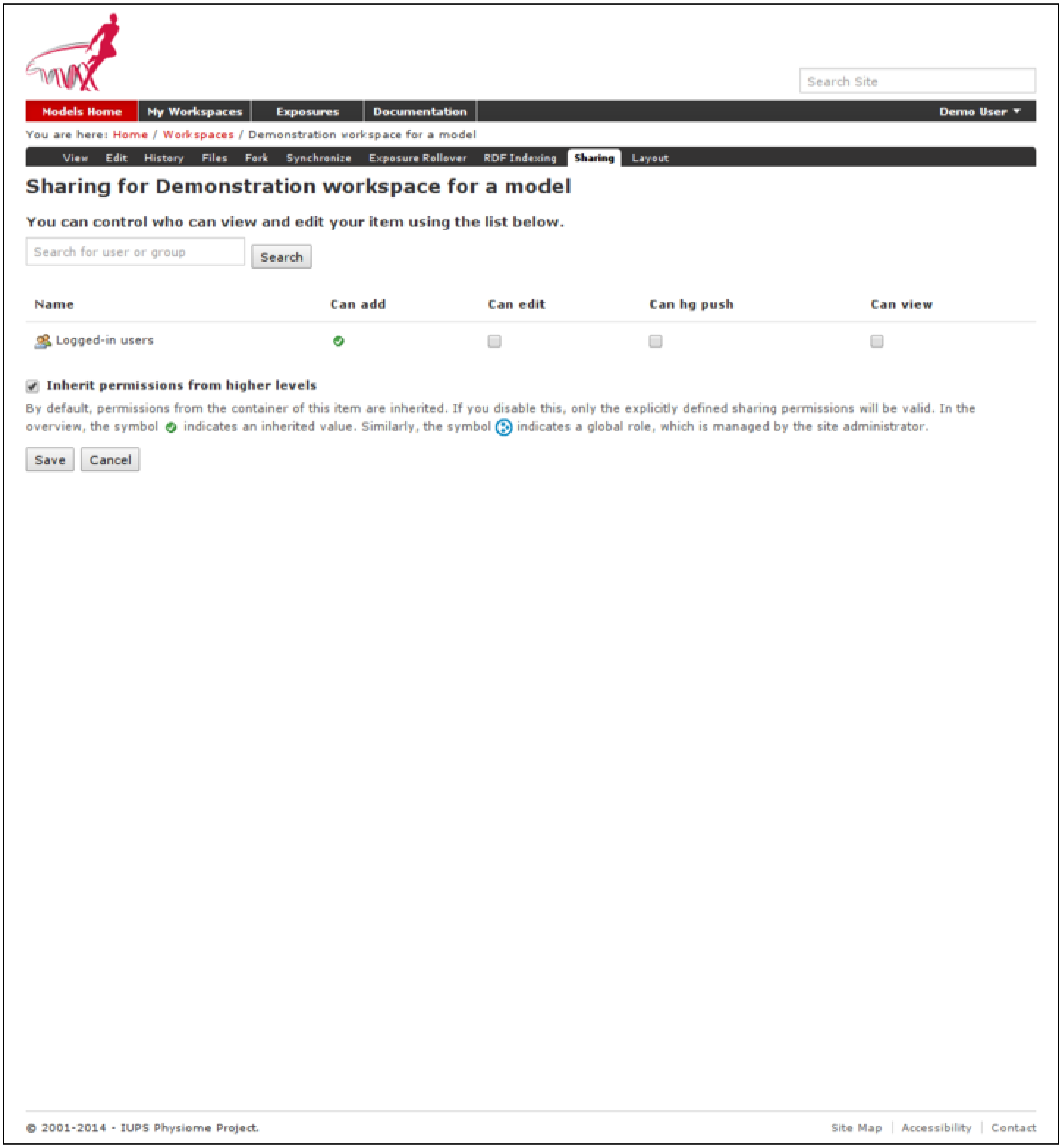
Owner
Demo User <demouser@example.com>

URI for git clone/pull/push
http://teaching.physiomeproject.org/workspace/237

Files

Filename	Size	Date	Options

© 2001-2015 - IUPS Physiome Project.



You are here: Home / Workspaces / Demonstration workspace for a model

Sharing for Demonstration workspace for a model

You can control who can view and edit your item using the list below.

Name	Can add	Can edit	Can hg push	Can view
Logged-in users	✓	□	□	□

☑ Inherit permissions from higher levels

By default, permissions from the container of this item are inherited. If you disable this, only the explicitly defined sharing permissions will be valid. In the overview, the symbol ✓ indicates an inherited value. Similarly, the symbol ⚙ indicates a global role, which is managed by the site administrator.

© 2001-2014 - IUPS Physiome Project.

applied to the default workspace container so that you and all other users of the system have the ability to create new workspaces.

PMR2 has the option to provide individual containers per user for their private workspaces, but this option is now disabled in the Auckland Physiome Repository.

You can disable the inherited higher level permissions from your workspace by unchecking the **Inherit permissions from higher levels** checkbox, if you wish, but the administrators of the repository can access your workspace regardless if you wish for them to aid you with your workspace. Using the *Sharing* tab you are able to search for other members, such as the names of people in your development team. These members would then appear in the list of members and you are able to set their access as required.

Using the *Sharing* controls there are currently four possible permissions that can be controlled. The **Can add** and **Can edit** permissions relate to the object that represents the workspace in the website database and are generally left in the default state. When selected for a given member, the **Can view** permission allows that member to view the workspace on the website, even if the workspace is private. Similarly, when the **Can push** permission is enabled the selected member is able to *push* into the workspace - this is the most important permission as enabling this allows members to add, modify, and delete the actual content of the workspace. One benefit of using Git means that even if one of the privileged members accidentally modifies the workspace in a detrimental manner, it is possible to revert the workspace back to the correct state.

When working in a collaborative team you would generally enable the **Can push** and **Can view** permissions for all team members and only enable the **Can add** and **Can edit** permissions for the team members responsible for the workspace presentation in the website.

Making your workspace public

If you wish to make your work available for searching by any users, including the ones who do not have an account with the repository, you may do so by changing the workflow state from "private" to "submit for publication". This will put your workspace into the reviewer queue and they will turn it into the "published" state.

Uploading files to your workspace

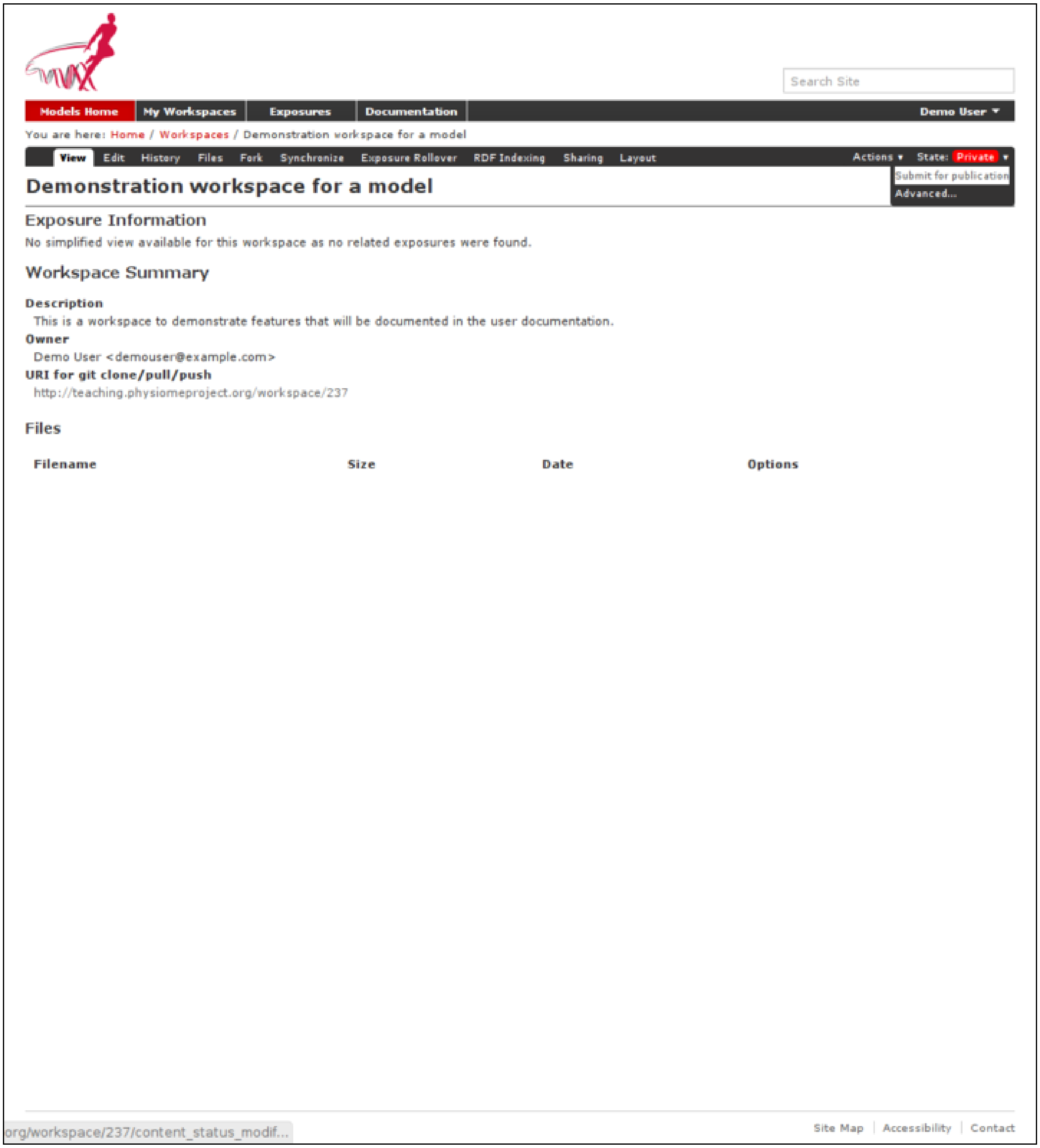
The basic process for adding content to a *workspace* consists of the following steps:

1. *Clone* the workspace to your local machine.
2. Add files to cloned workspace.
3. Commit the files using a *Git* client.
4. *Push* the workspace back to the repository.

An example demonstrating these steps can be found in in this tutorial step: Populate with content, or continue on to the *next section of this guide*.

Other hints and tips

- Avoid putting large binary data files into the workspace, as that will slow down users who wish to clone the repository. Especially archive of the data that are already in the workspace - the repository offers services to generate archive file download links.
- Documentation should be written in either HTML or reStructuredText, as these formats are currently supported by the repository for formatted rendering.



You are here: Home / Workspaces / Demonstration workspace for a model

Demonstration workspace for a model

Exposure Information

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

Workspace Summary

Description
This is a workspace to demonstrate features that will be documented in the user documentation.

Owner
Demo User <demouser@example.com>

URI for git clone/pull/push
http://teaching.physiomeproject.org/workspace/237

Files

Filename	Size	Date	Options

org/workspace/237/content_status_modif...

Troubleshoot

If during a `git push` something like this happened:

```
$ git push http://models.example.com/workspace/test
Counting objects: 101, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (101/101), done.
error: RPC failed; result=55, HTTP code = 0
fatal: The remote end hung up unexpectedly
Writing objects: 100% (101/101), 1.42 MiB | 0 bytes/s, done.
Total 101 (delta 59), reused 0 (delta 0)
fatal: The remote end hung up unexpectedly
Everything up-to-date
```

This is caused by an insufficiently large `http.postBuffer` provided by the default Git configuration (1 MiB). This needs to be in the `.gitconfig` file:

```
[http]
    postBuffer = 524288000
```

Issue this command to add it globally:

```
git config --global http.postBuffer 524288000
```

Alternatively, if TortoiseGit is used, edit one of the relevant configuration files as per **‘documentation‘_** on this.

1.8.5 Tutorial on using CellML with Auckland Physiome Repository

About this tutorial

The Auckland Physiome Repository provides extensive support for CellML model and related files. Previously it was called the CellML Model Repository, this has since been merged completely along with the FieldML Model Repository into the unified repository. The underlying software is *PMR2*, which in turn relies on the distributed version control system *Git*, 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 TortoiseGit, which provides a Windows explorer integrated system for working with Git repositories.

```
Brief mention of the equivalent command line versions of the
TortoiseGit 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 Git client. Optionally, a GUI client such as TortoiseGit.
- The OpenCell CellML modelling environment.
- A text editor such as Notepad++ or gedit.

Basic concepts

The Auckland Physiome 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 Git 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 Git DVCS has a range of terms that are useful to know, and definitions of these terms can be found in the Git Reference http://git-scm.com/docs.

Working with the repository web interface

This part of the tutorial will teach you how to find models in the Auckland Physiome Repository https://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.

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.

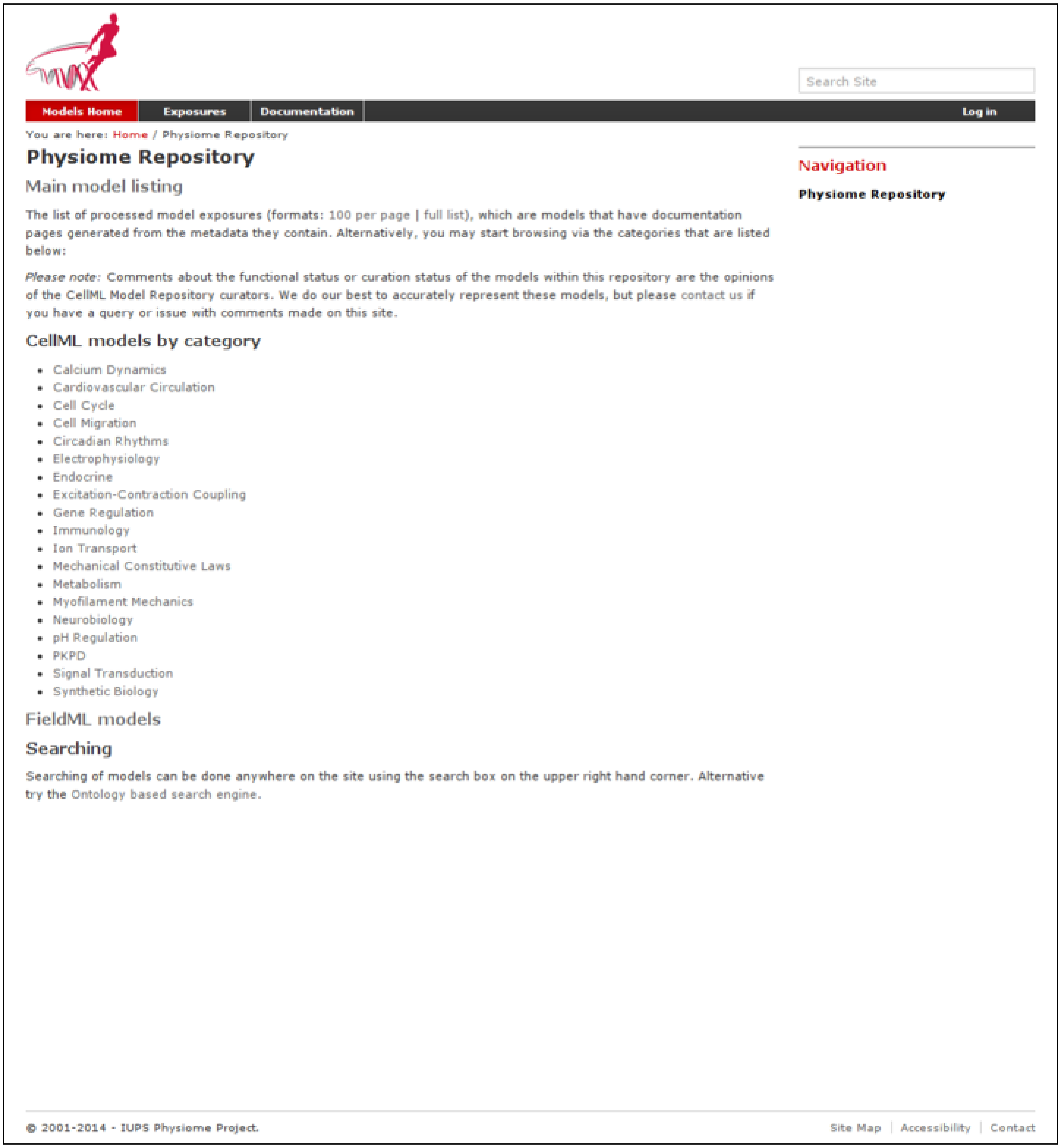
Searching the repository

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 by where they are located, as they will be located inside **Workspaces**. In the following screenshot, the first two results are workspaces, and the remainder are exposures. Note that red links are exposures that are marked as expired.

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 Git

This part of the tutorial will teach you how to *clone* a workspace from the model repository using a Git 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.



You are here: Home / Physiome Repository

Physiome 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

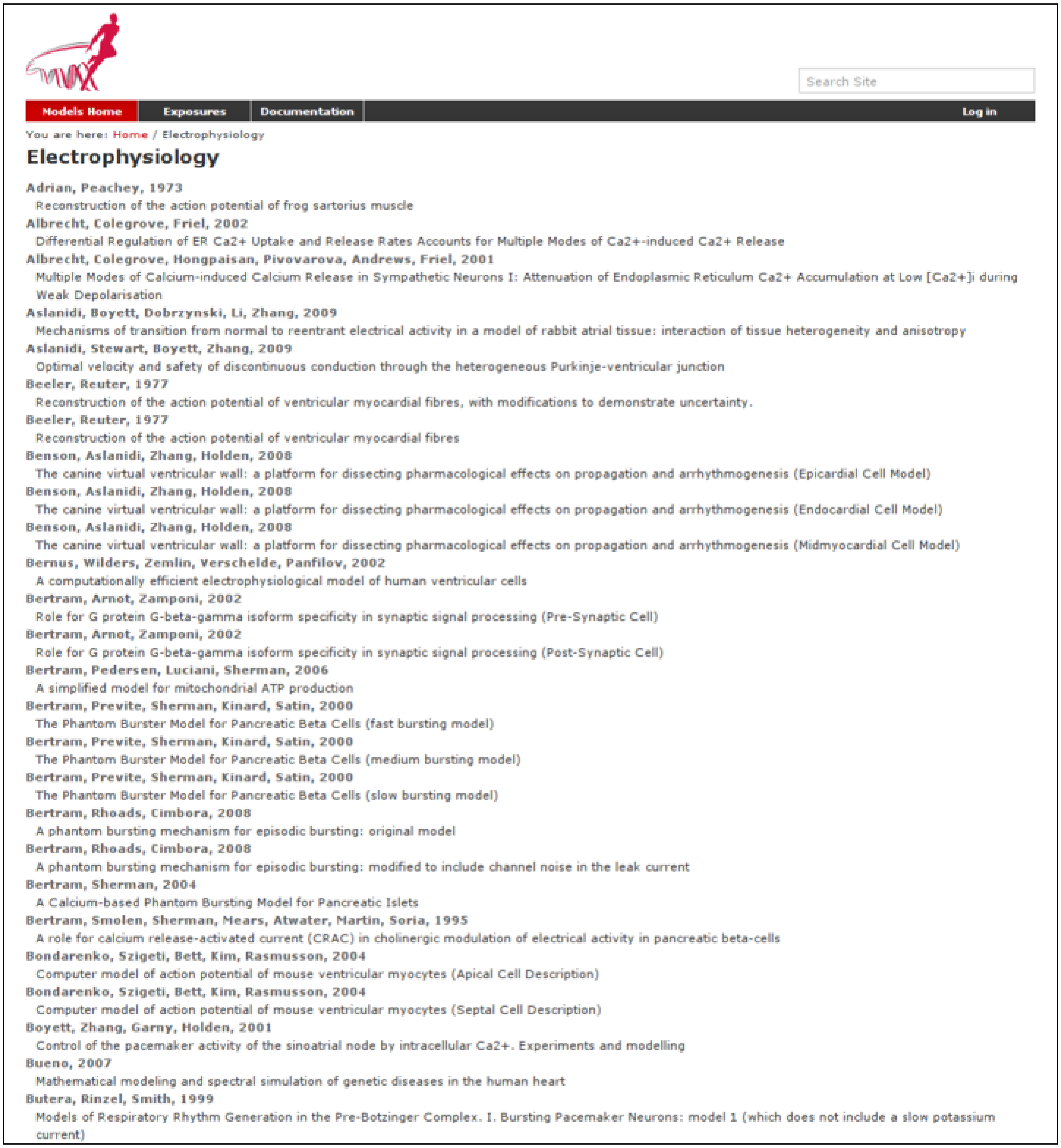
FieldML models

Searching

Searching of models can be done anywhere on the site using the search box on the upper right hand corner. Alternative try the Ontology based search engine.

© 2001-2014 - IUPS Physiome Project.

Fig. 1.198: The front page of the Auckland Physiome Repository.



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, with modifications to demonstrate uncertainty.

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

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

Bertram, Arnot, Zamponi, 2002
Role for G protein G-beta-gamma isoform specificity in synaptic signal processing (Pre-Synaptic Cell)

Bertram, Arnot, Zamponi, 2002
Role for G protein G-beta-gamma isoform specificity in synaptic signal processing (Post-Synaptic Cell)

Bertram, Pedersen, Luciani, Sherman, 2006
A simplified model for mitochondrial ATP production

Bertram, Previte, Sherman, Kinard, Satin, 2000
The Phantom Burster Model for Pancreatic Beta Cells (fast bursting model)

Bertram, Previte, Sherman, Kinard, Satin, 2000
The Phantom Burster Model for Pancreatic Beta Cells (medium bursting model)

Bertram, Previte, Sherman, Kinard, Satin, 2000
The Phantom Burster Model for Pancreatic Beta Cells (slow bursting model)

Bertram, Rhoads, Cimbora, 2008
A phantom bursting mechanism for episodic bursting: original model

Bertram, Rhoads, Cimbora, 2008
A phantom bursting mechanism for episodic bursting: modified to include channel noise in the leak current

Bertram, Sherman, 2004
A Calcium-based Phantom Bursting Model for Pancreatic Islets

Bertram, Smolen, Sherman, Mears, Atwater, Martin, Soria, 1995
A role for calcium release-activated current (CRAC) in cholinergic modulation of electrical activity in pancreatic beta-cells

Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004
Computer model of action potential of mouse ventricular myocytes (Apical Cell Description)

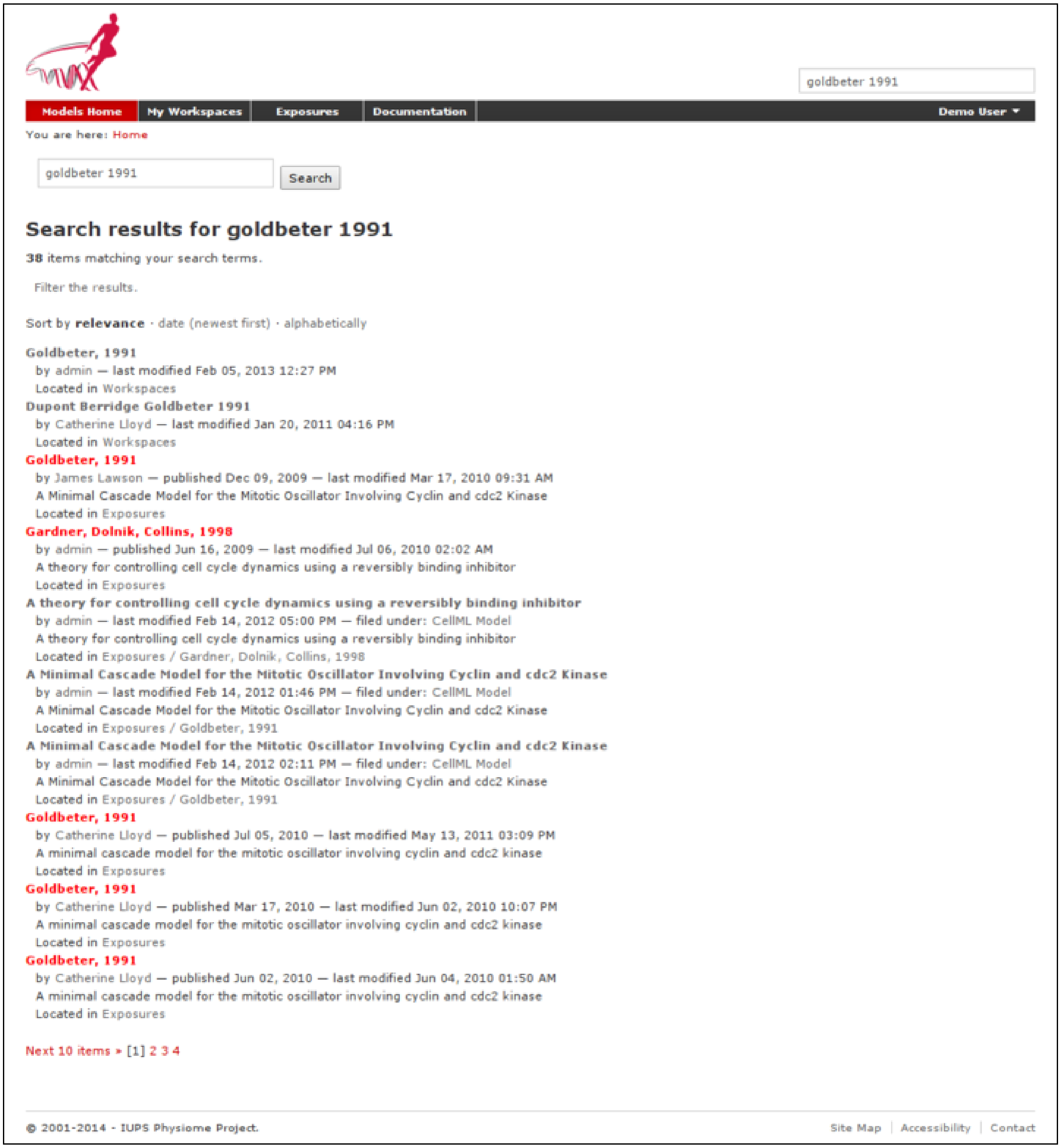
Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004
Computer model of action potential of mouse ventricular myocytes (Septal Cell Description)

Boyett, Zhang, Garny, Holden, 2001
Control of the pacemaker activity of the sinoatrial node by intracellular Ca2+. Experiments and modelling

Bueno, 2007
Mathematical modeling and spectral simulation of genetic diseases in the human heart

Butera, Rinzel, Smith, 1999
Models of Respiratory Rhythm Generation in the Pre-Botzinger Complex. I. Bursting Pacemaker Neurons: model 1 (which does not include a slow potassium current)

Fig. 1.199: A list of models in the electrophysiology category.



Search results for goldbeter 1991

38 items matching your search terms.

Filter the results.

Sort by **relevance** · date (newest first) · alphabetically

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

Dupont Berridge Goldbeter 1991
by Catherine Lloyd — last modified Jan 20, 2011 04:16 PM
Located in Workspaces

Goldbeter, 1991
by James Lawson — published Dec 09, 2009 — last modified Mar 17, 2010 09:31 AM
A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
Located in Exposures

Gardner, Dolnik, Collins, 1998
by admin — published Jun 16, 2009 — last modified Jul 06, 2010 02:02 AM
A theory for controlling cell cycle dynamics using a reversibly binding inhibitor
Located in Exposures

A theory for controlling cell cycle dynamics using a reversibly binding inhibitor
by admin — last modified Feb 14, 2012 05:00 PM — filed under: CellML Model
A theory for controlling cell cycle dynamics using a reversibly binding inhibitor
Located in Exposures / Gardner, Dolnik, Collins, 1998

A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
by admin — last modified Feb 14, 2012 01:46 PM — filed under: CellML Model
A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
Located in Exposures / Goldbeter, 1991

A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
by admin — last modified Feb 14, 2012 02:11 PM — filed under: CellML Model
A Minimal Cascade Model for the Mitotic Oscillator Involving Cyclin and cdc2 Kinase
Located in Exposures / Goldbeter, 1991

Goldbeter, 1991
by Catherine Lloyd — published Jul 05, 2010 — last modified May 13, 2011 03:09 PM
A minimal cascade model for the mitotic oscillator involving cyclin and cdc2 kinase
Located in Exposures

Goldbeter, 1991
by Catherine Lloyd — published Mar 17, 2010 — last modified Jun 02, 2010 10:07 PM
A minimal cascade model for the mitotic oscillator involving cyclin and cdc2 kinase
Located in Exposures

Goldbeter, 1991
by Catherine Lloyd — published Jun 02, 2010 — last modified Jun 04, 2010 01:50 AM
A minimal cascade model for the mitotic oscillator involving cyclin and cdc2 kinase
Located in Exposures

Next 10 items » [1] 2 3 4

© 2001-2014 - IUPS Physiome Project.

Fig. 1.200: A search results listing on the Auckland Physiome Repository.

Registering an account and logging in

First, navigate to the teaching instance of the Auckland Physiome Repository at https://teaching.physiomeproject.org/.

Note: The teaching instance of the repository is a mirror of the main repository site found at https://teaching.physiomeproject.org/, running the latest development version of *PMR2*. User accounts are periodicly synchronised from the main repository, but if you recently created an account on the main site you might need to also create a new account on the teaching instance.

Any changes you make to the contents of the teaching instance are not permanent, and will be overwritten with the contents of the main repository whenever the teaching instance is upgraded to a new release of PMR2. For this reason, you can feel free to experiment and make mistakes when pushing to the teaching instance. Please subscribe to the cellml-discussion mailing list to receive notifications of when the teaching instance will be refreshed.

See the section *Migrating content to the main repository* for instructions on how to migrate any content from the teaching instance to the main (permanent) Auckland Physiome Repository.

In order to make changes to models in the CellML repository, you must first register for an account. 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.

To register an account, access the *Log in* link and access the *registration form* from there. Enter your desired username and 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 Git.

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 link is also replaced by your username and a Log out link.

Git username configuration

Important: Username setup for Git

Since you are about to make changes, your name and email address must be recorded as part of the workspace revision history. When commit your changes using Git, they exist independently from the Auckland Physiome Repository. This means that you have to set-up your username for the Git client software, even though you have registered a username on Auckland Physiome Repository.

You only need to do this once.

Steps for TortoiseGit:

- From the right-click menu in Windows Explorer (which is accessible by right-clicking anywhere on the file selection area) you can select the Settings by *TortoiseGit* → *Settings*.
- On the settings screen, under the Git section, enter your details into the respective sections.

Steps for command line:

The configurations are saved in the `.gitconfig` file in the user's home directory, and the recommended way to do so is:

```
$ git config --global user.name "<Your Name>"
$ git config --global user.email "<Your Email address>"
```

Forking an existing workspace

Important: It is essential to use a Git client to obtain models from the repository for editing. The Git 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 Git 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: https://teaching.physiomeproject.org/workspace/beeler_reuter_1977.

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

You will be asked to confirm the *fork* action by clicking the *Fork* button. 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 Git clone/pull/push as shown below:

In Windows explorer, navigate the folder where you want to create the clone of the workspace. Then use the right-click menu and select *Git 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

```
git clone [URI]
```

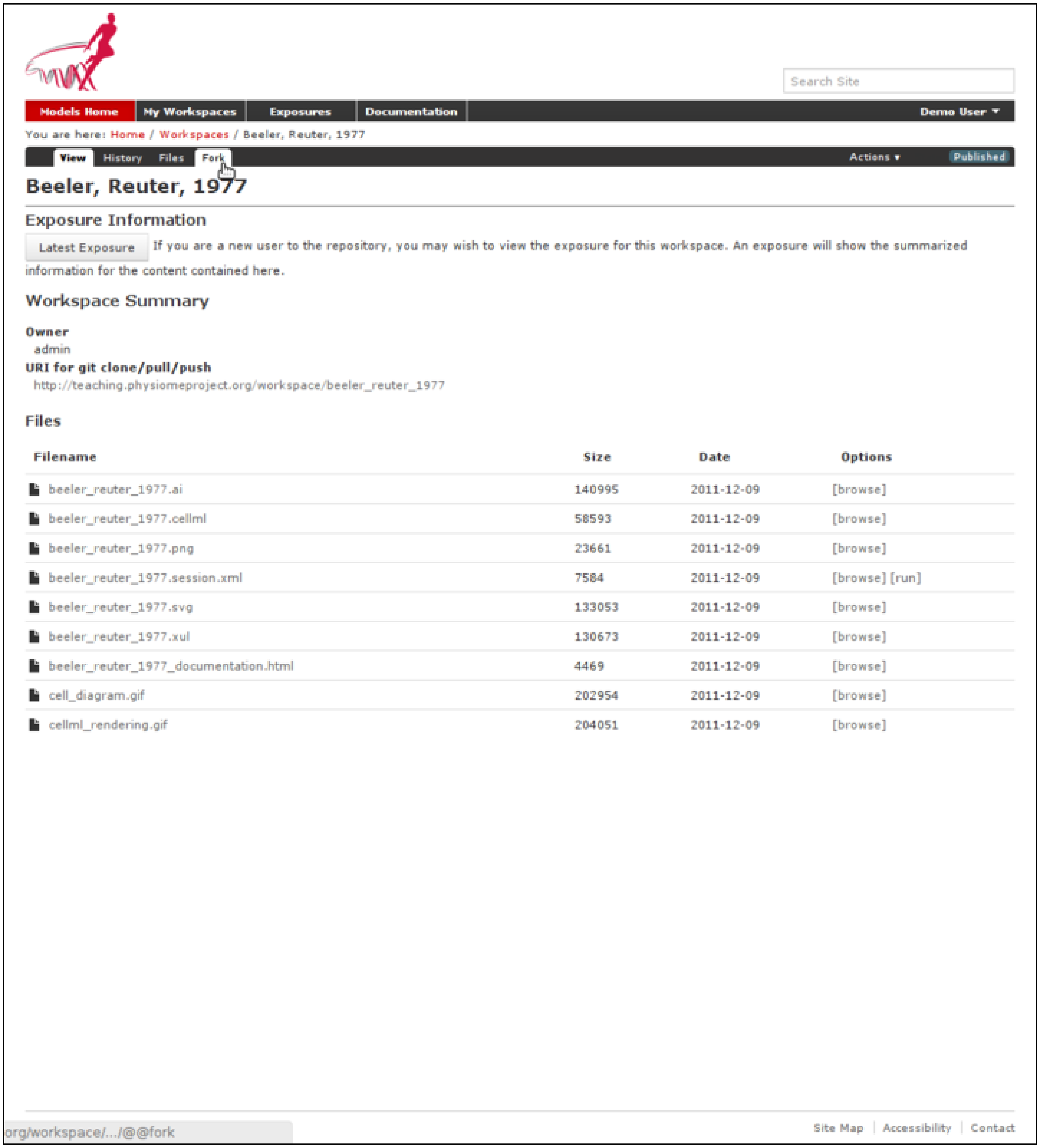
You will need to enter your username and password to clone the workspace, as the fork will be set to *private* when it is created.

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:

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



Beeler, Reuter, 1977

Exposure Information

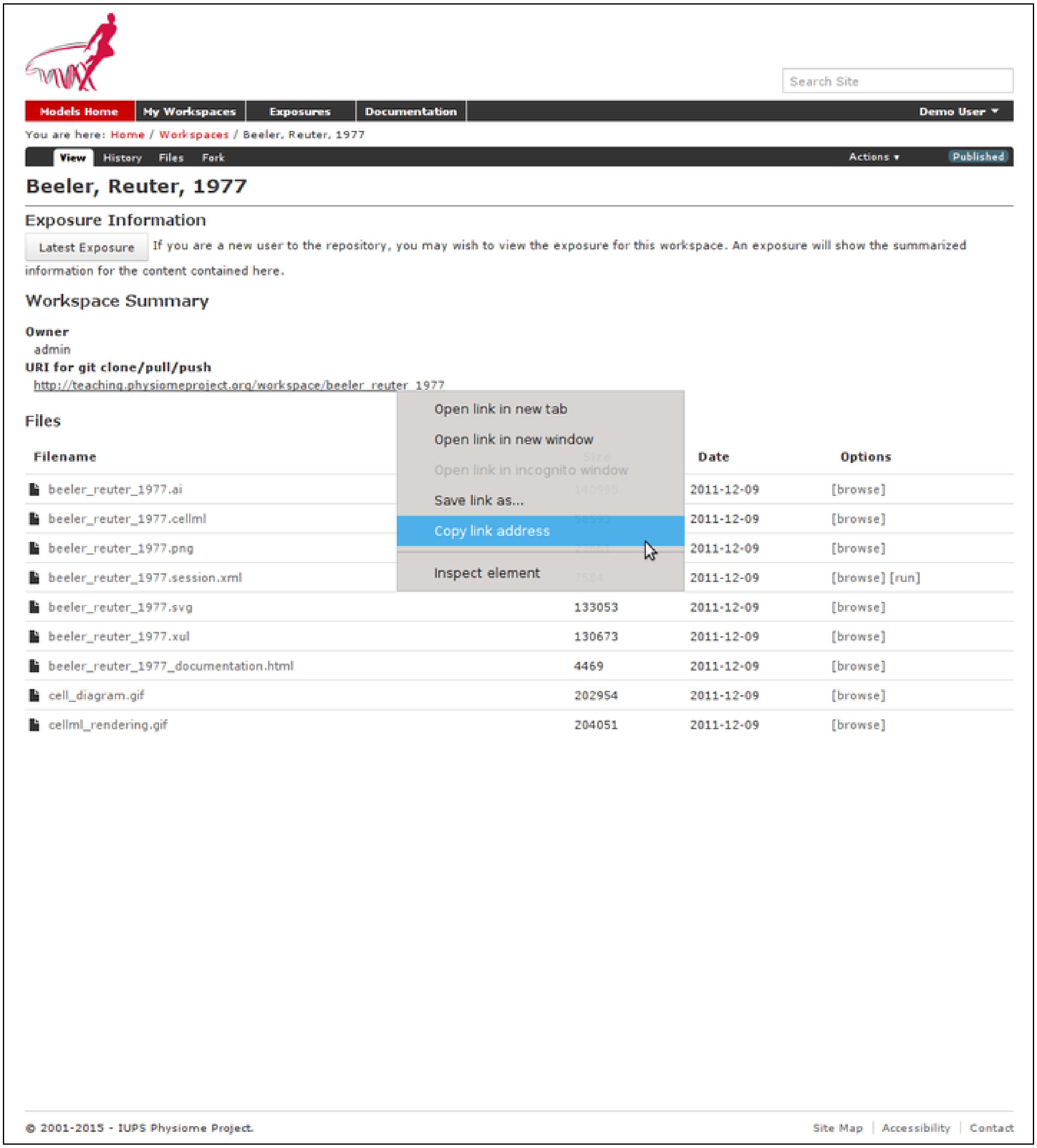
If you are a new user to the repository, you may wish to view the exposure for this workspace. An exposure will show the summarized information for the content contained here.

Workspace Summary

Owner
admin
URI for git clone/pull/push
http://teaching.physiomeproject.org/workspace/beeler_reuter_1977

Files

Filename	Size	Date	Options
beeler_reuter_1977.ai	140995	2011-12-09	[browse]
beeler_reuter_1977.cellml	58593	2011-12-09	[browse]
beeler_reuter_1977.png	23661	2011-12-09	[browse]
beeler_reuter_1977.session.xml	7584	2011-12-09	[browse] [run]
beeler_reuter_1977.svg	133053	2011-12-09	[browse]
beeler_reuter_1977.xul	130673	2011-12-09	[browse]
beeler_reuter_1977_documentation.html	4469	2011-12-09	[browse]
cell_diagram.gif	202954	2011-12-09	[browse]
cellml_rendering.gif	204051	2011-12-09	[browse]



You are here: Home / Workspaces / Beeler, Reuter, 1977

Beeler, Reuter, 1977

Exposure Information

If you are a new user to the repository, you may wish to view the exposure for this workspace. An exposure will show the summarized information for the content contained here.

Workspace Summary

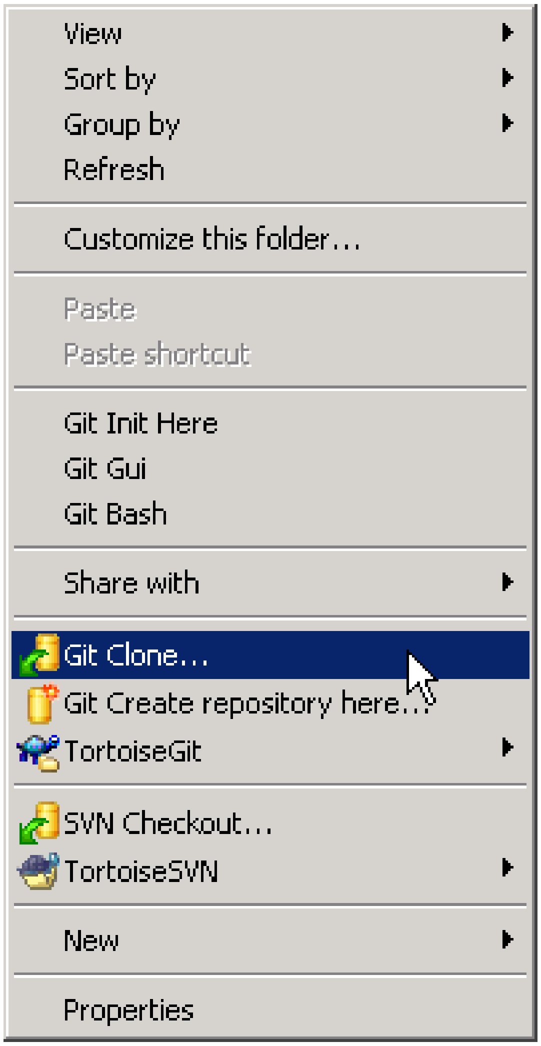
Owner
admin
URI for git clone/pull/push
http://teaching.physiomeproject.org/workspace/beeler_reuter_1977

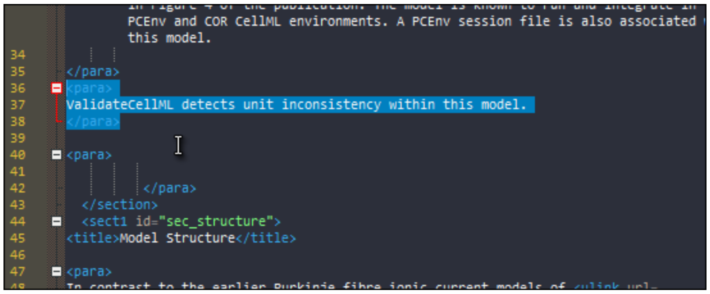
Files

Filename	Size	Date	Options
beeler_reuter_1977.ai	140995	2011-12-09	[browse]
beeler_reuter_1977.cellml	58593	2011-12-09	[browse]
beeler_reuter_1977.png	23561	2011-12-09	[browse]
beeler_reuter_1977.session.xml	7584	2011-12-09	[browse] [run]
beeler_reuter_1977.svg	133053	2011-12-09	[browse]
beeler_reuter_1977.xul	130673	2011-12-09	[browse]
beeler_reuter_1977_documentation.html	4469	2011-12-09	[browse]
cell_diagram.gif	202954	2011-12-09	[browse]
cellml_rendering.gif	204051	2011-12-09	[browse]

© 2001-2015 - IUPS Physiome Project.

Fig. 1.201: Copying the URI for cloning your workspace.



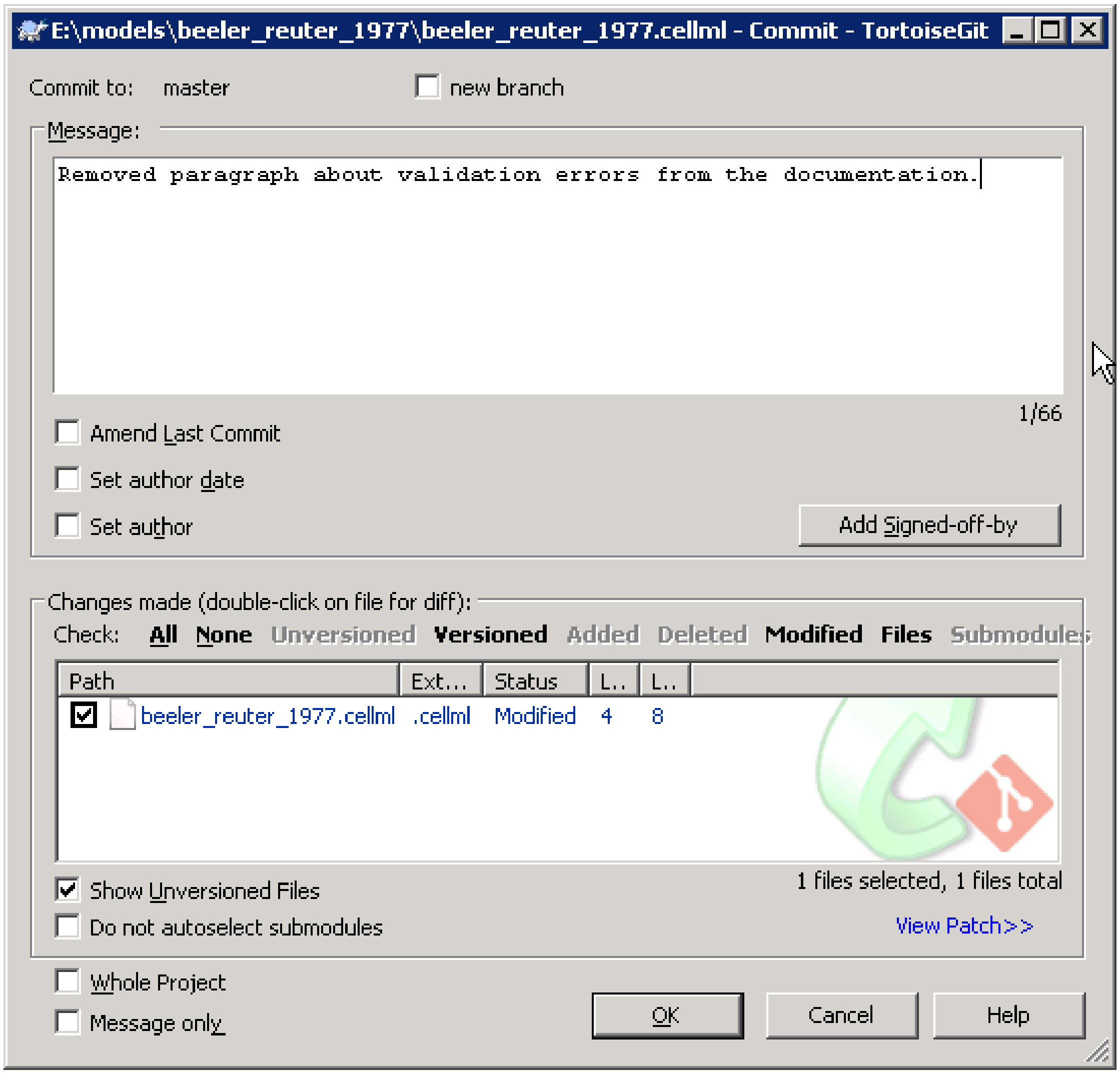


```
        in Figure 4 of the publication. The model is known to run and integrate in
        PCEnv and COR CellML environments. A PCEnv session file is also associated
        this model.

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

<para>

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

<para>
In contrast to the earlier Purkinje fibre ionic current models of <ulink url=
```

Committing changes

If you are using TortoiseGit, bring up the shell menu for the altered file and select *Git Commit -> "master"*. 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.



E:\models\beeler_reuter_1977\beeler_reuter_1977.cellml - Commit - TortoiseGit

Commit to: master ☐ new branch

Message:

```
Removed paragraph about validation errors from the documentation.
```

1/66

☐ Amend Last Commit

☐ Set author date

☐ Set author

Changes made (double-click on file for diff):

Path	Ext...	Status	L..	L..
☑ beeler_reuter_1977.cellml	.cellml	Modified	4	8

☑ Show Unversioned Files

☐ Do not autoselect submodules

1 files selected, 1 files total

☐ Whole Project

☐ Message only

Add Signed-off-by
Check: All None Unversioned Versioned Added Deleted Modified Files Submodules
View Patch>>
OK
Cancel
Help

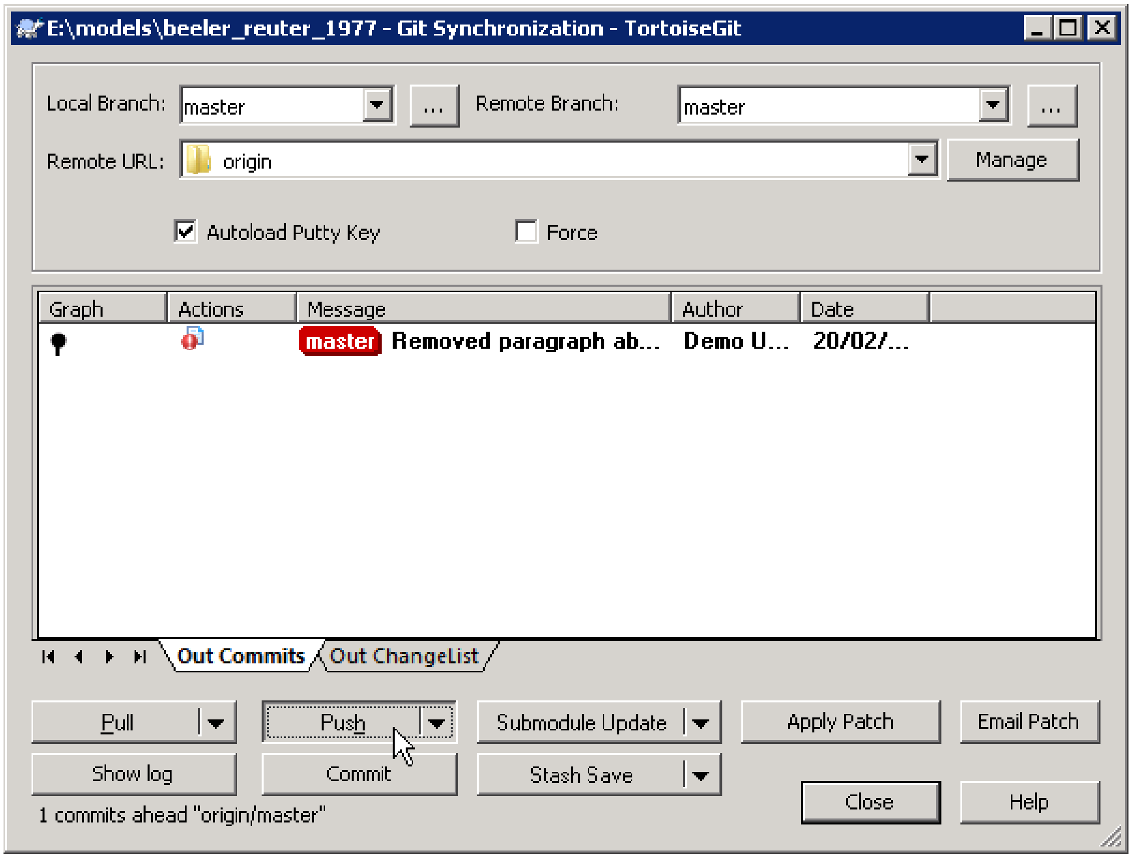
Command line equivalent

```
git 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 *Git Sync…* 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.



E:\models\beeler_reuter_1977 - Git Synchronization - TortoiseGit

Local Branch: master

Remote Branch: master

Remote URL: origin

☑ Autoload Putty Key

□ Force

Graph	Actions	Message	Author	Date
		master Removed paragraph ab...	Demo U...	20/02/...

1 commits ahead "origin/master"

Command line equivalent

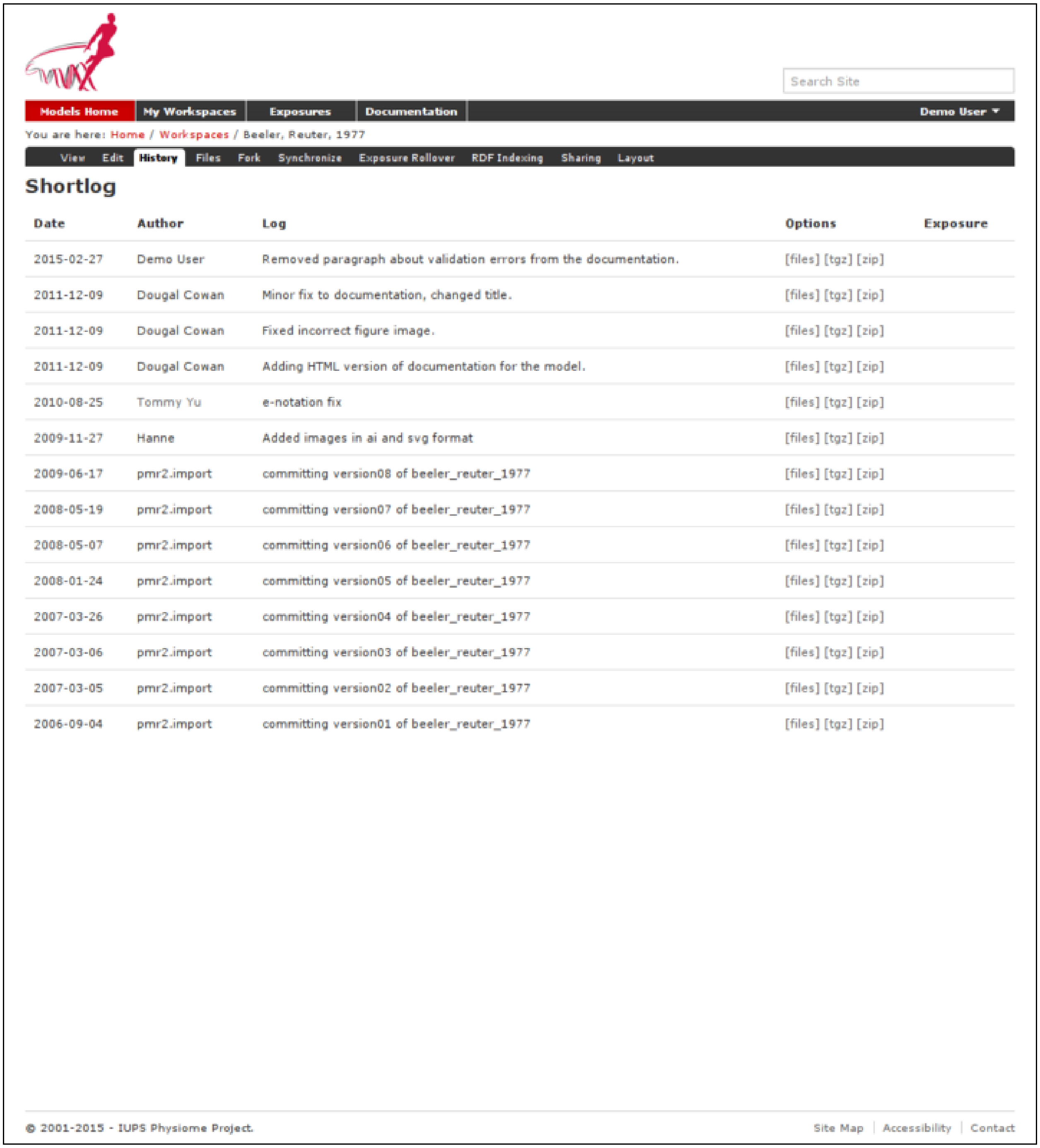
```
git push
```

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:

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



Shortlog

Date	Author	Log	Options	Exposure
2015-02-27	Demo User	Removed paragraph about validation errors from the documentation.	[files] [tgz] [zip]	
2011-12-09	Dougal Cowan	Minor fix to documentation, changed title.	[files] [tgz] [zip]	
2011-12-09	Dougal Cowan	Fixed incorrect figure image.	[files] [tgz] [zip]	
2011-12-09	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]	

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*.

Migrating content to the main repository

As noted above, the teaching instance used in this tutorial is not suitable for permanent storage of your work. One of the advantages of using a distributed version control system to manage *workspaces* is that it is straightforward to move the entire workspace, including the full history and provenance record, from one location to another. PMR2 also provides a feature that exports exposures so that they can then be imported into another PMR2 instance.

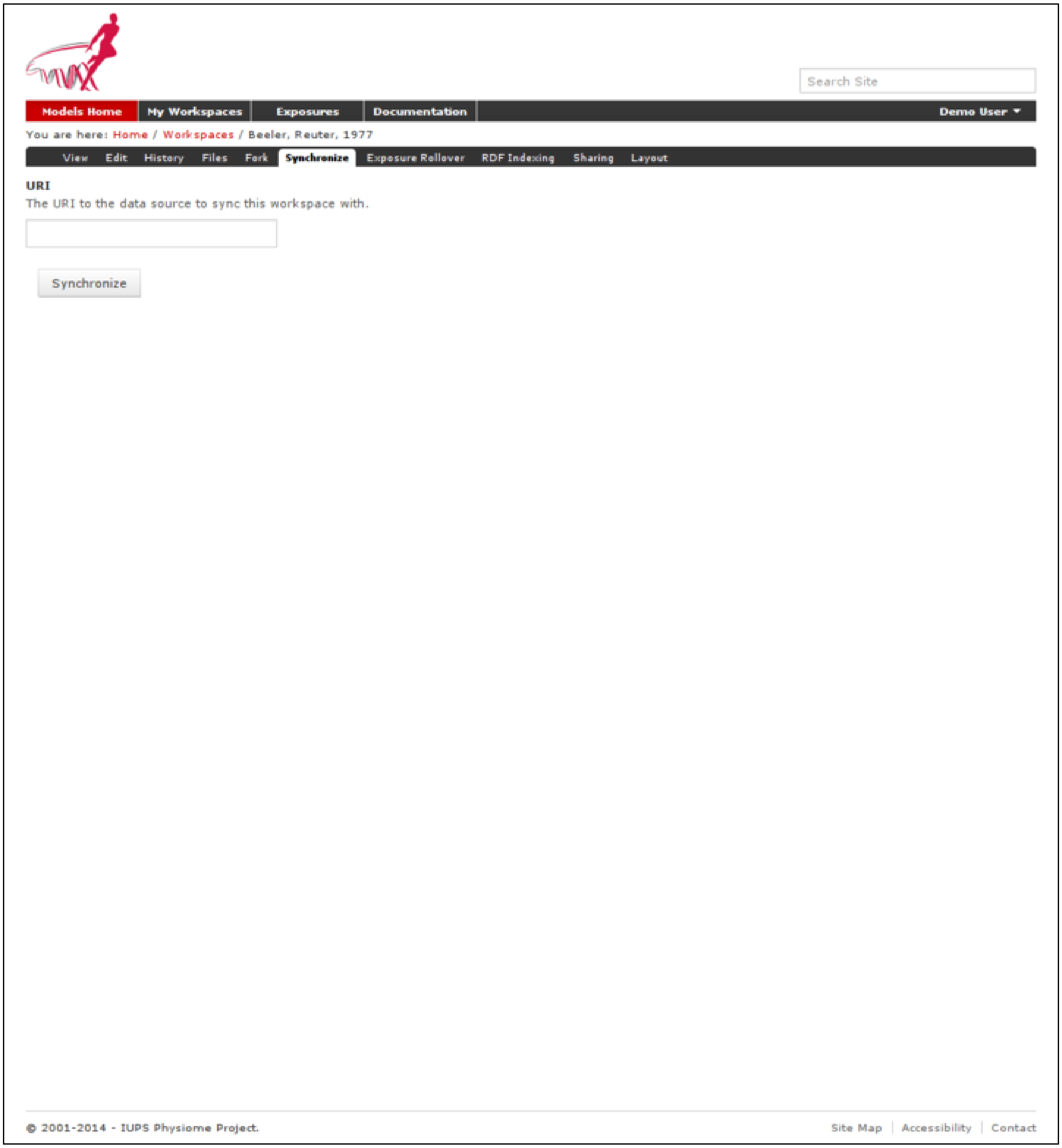
For example: if you would like to move your work from any publicly accessible Git repository (such as a published workspace on the teaching repository or a public repository on GitHub.com) to a *new* workspace on the Auckland Physiome Repository, you should follow these steps:

1. Ensure that you have pushed all your commits to the source instance;
2. *Create the new workspace* in the destination repository;
3. Navigate to the workspace created and choose the *synchronize* action from the workspace toolbar, as shown below.
4. Fill in the URI of your workspace on the source instance (*e.g.,* https://teaching.physiomeproject.org/w/andre/cortassa-ECME-2006)
5. Click the *Synchronize* button.

If you would like to move your work from a private workspace on the teaching instance (or any non-publicly accessible Git repository), you will need to use your Git client to directly *push* your repository up to the newly created workspace. This is because you will need to make use of the relevant authentication to access the private workspace or repository.

In a similar manner, you are able to copy *exposures* you might have made on the teaching instance over to the main repository, or from the main to the teaching instance if you want to test things out. Follow these steps to migrate an *exposure* from one repository to another.

1. Navigate to the exposure you would like to migrate in the source repository.
2. Choose the *wizard* item from the toolbar as shown below.
3. In the destination repository, navigate to the desired revision of the (published) workspace and choose the *Create exposure* action as described in the directions for *creating an exposure from scratch*
4. Rather than building a new exposure, choose the *Exposure Import via URI* tab in the exposure creation wizard, as shown below.
5. Copy and paste the URI from the source exposure wizard, highlighted above, into the *Exposure Export URI* field in the exposure creation wizard shown above.
6. Click the *Add* button. This will take you back to the standard *exposure build page*, but now with all the fields pre-populated from the source exposure.
7. Navigate to the bottom of the page and click the *Build* button to actually build the exposure pages. You are free to reconfigure the exposure if desired, some *help is available* for this if needed.

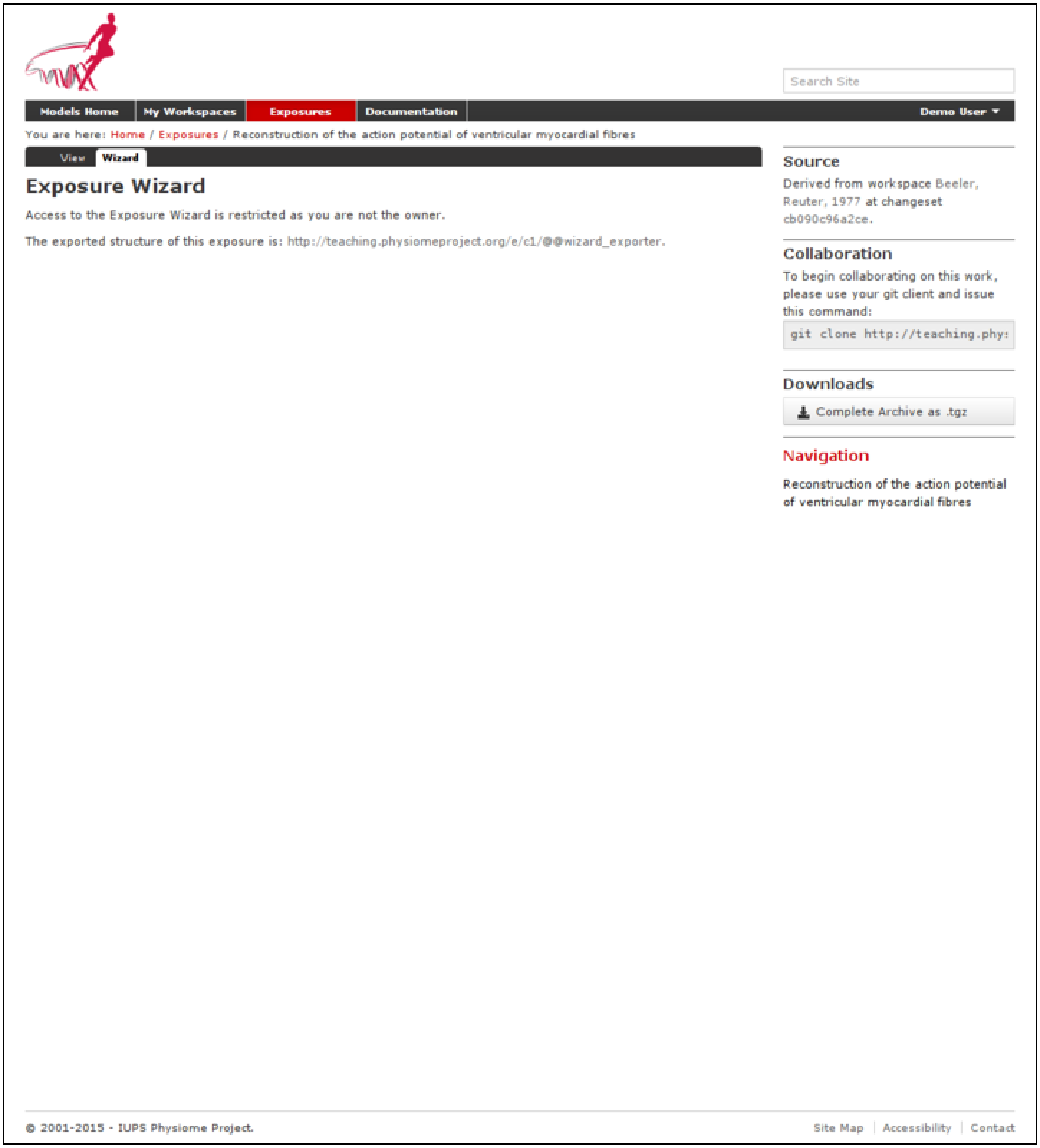


You are here: Home / Workspaces / Beeler, Reuter, 1977

URI

The URI to the data source to sync this workspace with.

© 2001-2014 - IUPS Physiome Project.



You are here: Home / Exposures / Reconstruction of the action potential of ventricular myocardial fibres

Exposure Wizard

Access to the Exposure Wizard is restricted as you are not the owner.

The exported structure of this exposure is: http://teaching.physiomeproject.org/e/c1/@@wizard_exporter.

Source

Derived from workspace Beeler, Reuter, 1977 at changeset cb090c96a2ce.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone http://teaching.phys
```

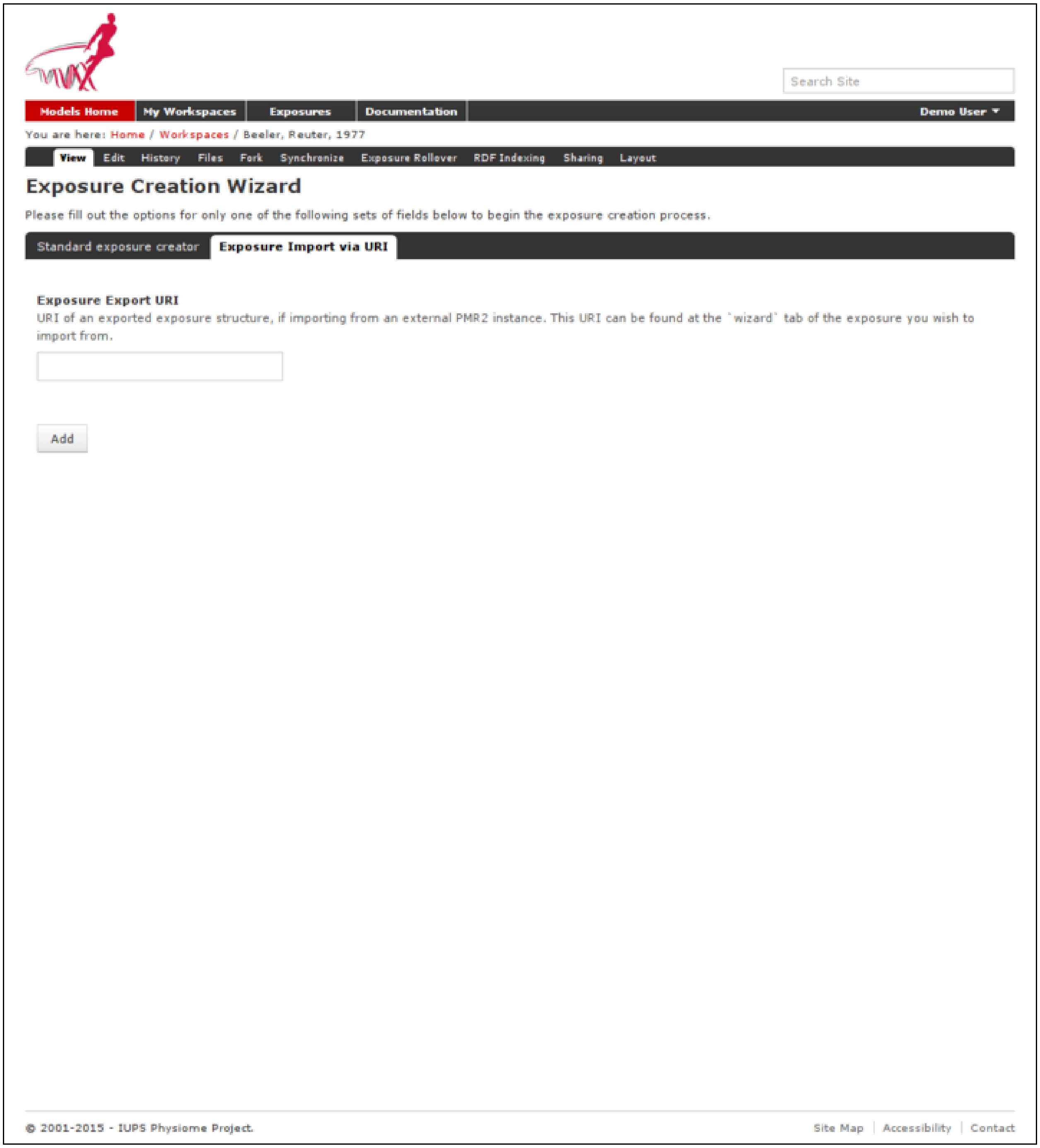
Downloads

Complete Archive as .tgz

Navigation

Reconstruction of the action potential of ventricular myocardial fibres

© 2001-2015 - IUPS Physiome Project.



Exposure Creation Wizard

Please fill out the options for only one of the following sets of fields below to begin the exposure creation process.

Exposure Export URI

URI of an exported exposure structure, if importing from an external PMR2 instance. This URI can be found at the `wizard` tab of the exposure you wish to import from.

1.8.6 Working with semantic metadata

PMR2 release 8 and 9 brought in the support for semantic metadata, which allows users to add whatever metadata and annotations they might have stored into the repository into the underlying metadata semantic engine, which then allows them to be retrieved using search queries. In this section, we will go over how to use OpenCOR to annotate a model, and how to add the metadata to the underlying metadata engine then query for the results.

Preparation

In this section, we assume that you have already run through the *tutorial on using the repository* for the basic operations of the repository.

For the tutorial, we will use a *fork* of the the Hodgkin, Huxley, 1952 workspace. If you need a quick reminder on how you might do this, please see *this section of the tutorial*.

Once you forked that workspace, you should now clone that workspace onto your system. If you need help on this, please refer to *this help on cloning a workspace*.

Using OpenCOR for model annotation

Use OpenCOR to open your local clone of your model file, specifically the `hodgkin_huxley_1952.cellml` file.

Select the `sodium_channel` component under the list of components, then click on the helpful link to remove the existing metadata for that node.

In the dropdown menu of *Qualifiers*, select bio:isVersionOf

In the textbox *Term*, type in “sodium channel”, as the component is named so. Wait for the possible terms to be retrieved and populated by OpenCOR.

Once that is done, hit the green ‘+’ button for the “sodium channel complex” (GO:0034706) to denote that the component is a version of this term.

Now select the `potassium_channel` component, and repeat the processs to annotate this with the “potassium channel complex” term.

Once you are done, save your changes, commit and push your work back into your private fork. Again, refer to the tutorial linked in the preparation section if you need a primer.

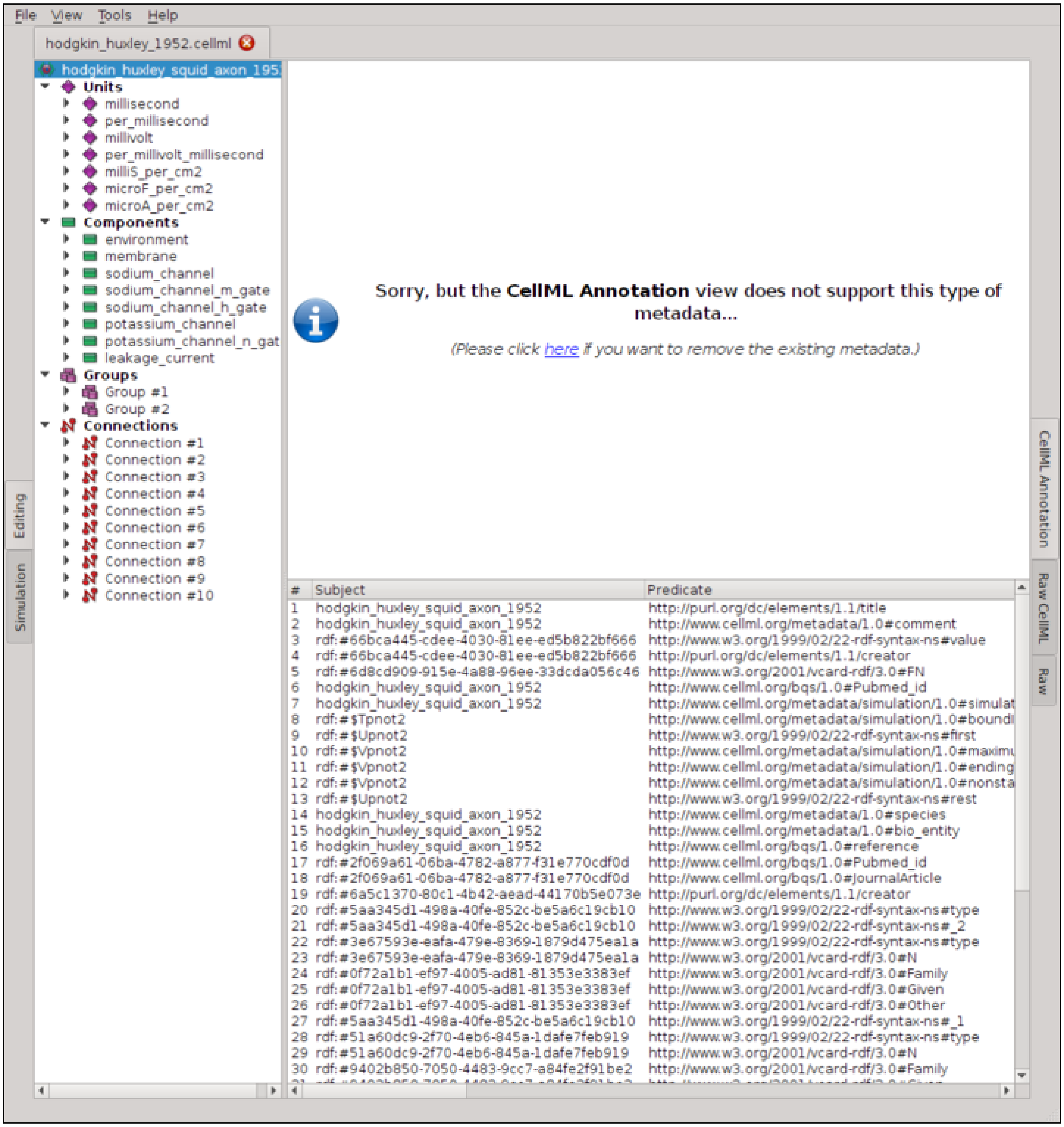
Getting your workspace indexed by the repository

In order to have your annotations pulled into the semantic engine, you need to inform the repository which resources in your workspace should be indexed. Once you have done this, all future versions of the indexed resources will be used to update the semantic engine.

Now that your changes have been pushed back, go back to the page for your fork of the model and select the “RDF Indexing” tab.

Scroll down the left-handed list until you see the `hodgkin_huxley_1952.cellml` file, select it, then push the button with the right arrow on it to add it onto the list of paths to be tracked, then select the “Apply Changes and Export To RDF Store” button.

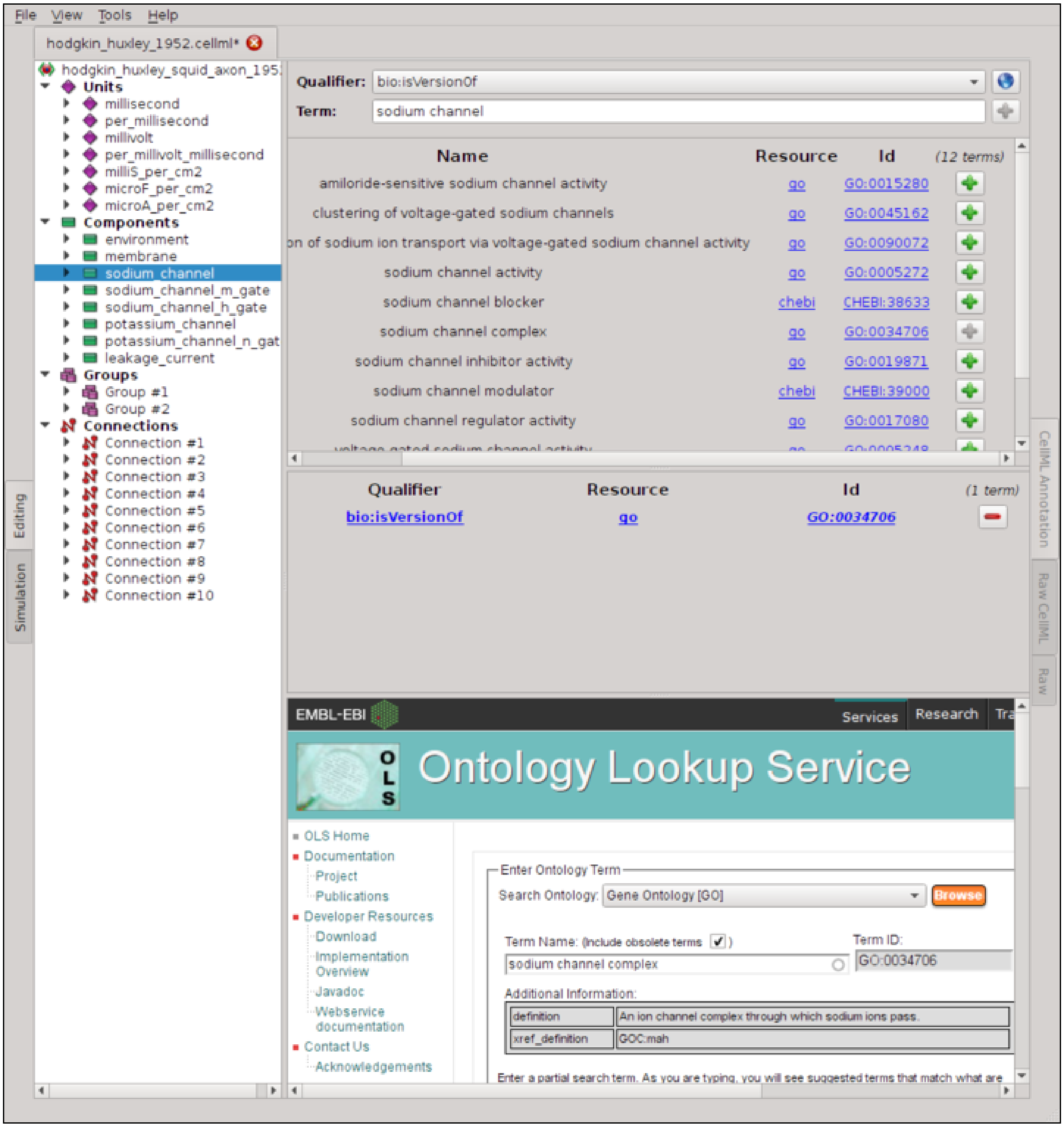
Go back to the main page, select the “Ontology based search engine” link at the bottom, then enter the relevant search term. As there are limited reasoning capabilities built into the current iteration of the search engine, you may enter a term one level up above the terms we annotated the model with. For our example, please enter “cation channel complex” into the search box, select the term ending with (GO_0034703). The search indicator will give a green



Sorry, but the **CellML Annotation** view does not support this type of metadata...

(Please click here if you want to remove the existing metadata.)

#	Subject	Predicate
1	hodgkin_huxley_squid_axon_1952	http://purl.org/dc/elements/1.1/title
2	hodgkin_huxley_squid_axon_1952	http://www.cellml.org/metadata/1.0#comment
3	rdf:#66bca445-cdee-4030-81ee-ed5b822bf666	http://www.w3.org/1999/02/22-rdf-syntax-ns#value
4	rdf:#66bca445-cdee-4030-81ee-ed5b822bf666	http://purl.org/dc/elements/1.1/creator
5	rdf:#6d8cd909-915e-4a88-96ee-33dcda056c46	http://www.w3.org/2001/vcard-rdf/3.0#FN
6	hodgkin_huxley_squid_axon_1952	http://www.cellml.org/bqs/1.0#Pubmed_id
7	hodgkin_huxley_squid_axon_1952	http://www.cellml.org/metadata/simulation/1.0#simulat
8	rdf:#$Tpnot2	http://www.cellml.org/metadata/simulation/1.0#boundl
9	rdf:#$Upnot2	http://www.w3.org/1999/02/22-rdf-syntax-ns#first
10	rdf:#$Vpnot2	http://www.cellml.org/metadata/simulation/1.0#maximu
11	rdf:#$Vpnot2	http://www.cellml.org/metadata/simulation/1.0#ending
12	rdf:#$Vpnot2	http://www.cellml.org/metadata/simulation/1.0#nonsta
13	rdf:#$Upnot2	http://www.w3.org/1999/02/22-rdf-syntax-ns#rest
14	hodgkin_huxley_squid_axon_1952	http://www.cellml.org/metadata/1.0#species
15	hodgkin_huxley_squid_axon_1952	http://www.cellml.org/metadata/1.0#bio_entity
16	hodgkin_huxley_squid_axon_1952	http://www.cellml.org/bqs/1.0#reference
17	rdf:#2f069a61-06ba-4782-a877-f31e770cdf0d	http://www.cellml.org/bqs/1.0#Pubmed_id
18	rdf:#2f069a61-06ba-4782-a877-f31e770cdf0d	http://www.cellml.org/bqs/1.0#JournalArticle
19	rdf:#6a5c1370-80c1-4b42-aead-44170b5e073e	http://purl.org/dc/elements/1.1/creator
20	rdf:#5aa345d1-498a-40fe-852c-be5a6c19cb10	http://www.w3.org/1999/02/22-rdf-syntax-ns#type
21	rdf:#5aa345d1-498a-40fe-852c-be5a6c19cb10	http://www.w3.org/1999/02/22-rdf-syntax-ns#_2
22	rdf:#3e67593e-eafa-479e-8369-1879d475ea1a	http://www.w3.org/1999/02/22-rdf-syntax-ns#type
23	rdf:#3e67593e-eafa-479e-8369-1879d475ea1a	http://www.w3.org/2001/vcard-rdf/3.0#N
24	rdf:#0f72a1b1-ef97-4005-ad81-81353e3383ef	http://www.w3.org/2001/vcard-rdf/3.0#Family
25	rdf:#0f72a1b1-ef97-4005-ad81-81353e3383ef	http://www.w3.org/2001/vcard-rdf/3.0#Given
26	rdf:#0f72a1b1-ef97-4005-ad81-81353e3383ef	http://www.w3.org/2001/vcard-rdf/3.0#Other
27	rdf:#5aa345d1-498a-40fe-852c-be5a6c19cb10	http://www.w3.org/1999/02/22-rdf-syntax-ns#_1
28	rdf:#51a60dc9-2f70-4eb6-845a-1dafe7feb919	http://www.w3.org/1999/02/22-rdf-syntax-ns#type
29	rdf:#51a60dc9-2f70-4eb6-845a-1dafe7feb919	http://www.w3.org/2001/vcard-rdf/3.0#N
30	rdf:#9402b850-7050-4483-9cc7-a84fe2f91be2	http://www.w3.org/2001/vcard-rdf/3.0#Family



Qualifier: bio:isVersionOf

Term: sodium channel

Name	Resource	Id	(12 terms)
amiloride-sensitive sodium channel activity	go	GO:0015280	+
clustering of voltage-gated sodium channels	go	GO:0045162	+
on of sodium ion transport via voltage-gated sodium channel activity	go	GO:0090072	+
sodium channel activity	go	GO:0005272	+
sodium channel blocker	chebi	CHEBI:38633	+
sodium channel complex	go	GO:0034706	+
sodium channel inhibitor activity	go	GO:0019871	+
sodium channel modulator	chebi	CHEBI:39000	+
sodium channel regulator activity	go	GO:0017080	+
voltage-gated sodium channel activity	go	GO:0005248	+

Qualifier	Resource	Id	(1 term)
bio:isVersionOf	go	GO:0034706	−

Enter Ontology Term

Search Ontology: Gene Ontology [GO]

Term Name: (include obsolete terms ☑)

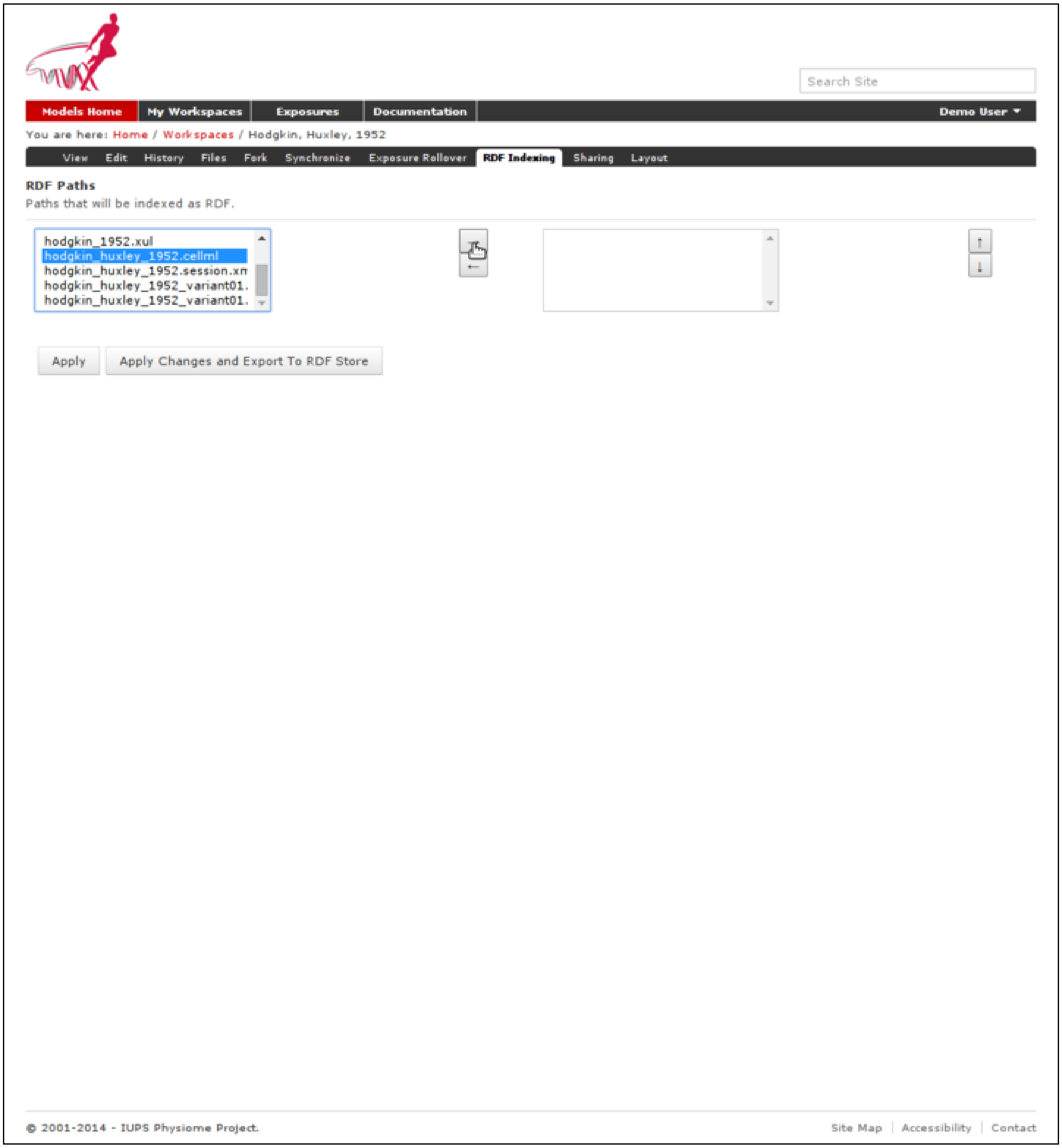
sodium channel complex

Term ID: GO:0034706

Additional Information:

definition	An ion channel complex through which sodium ions pass.
xref_definition	GOC:mah

Enter a partial search term. As you are typing, you will see suggested terms that match what are



You are here: Home / Workspaces / Hodgkin, Huxley, 1952

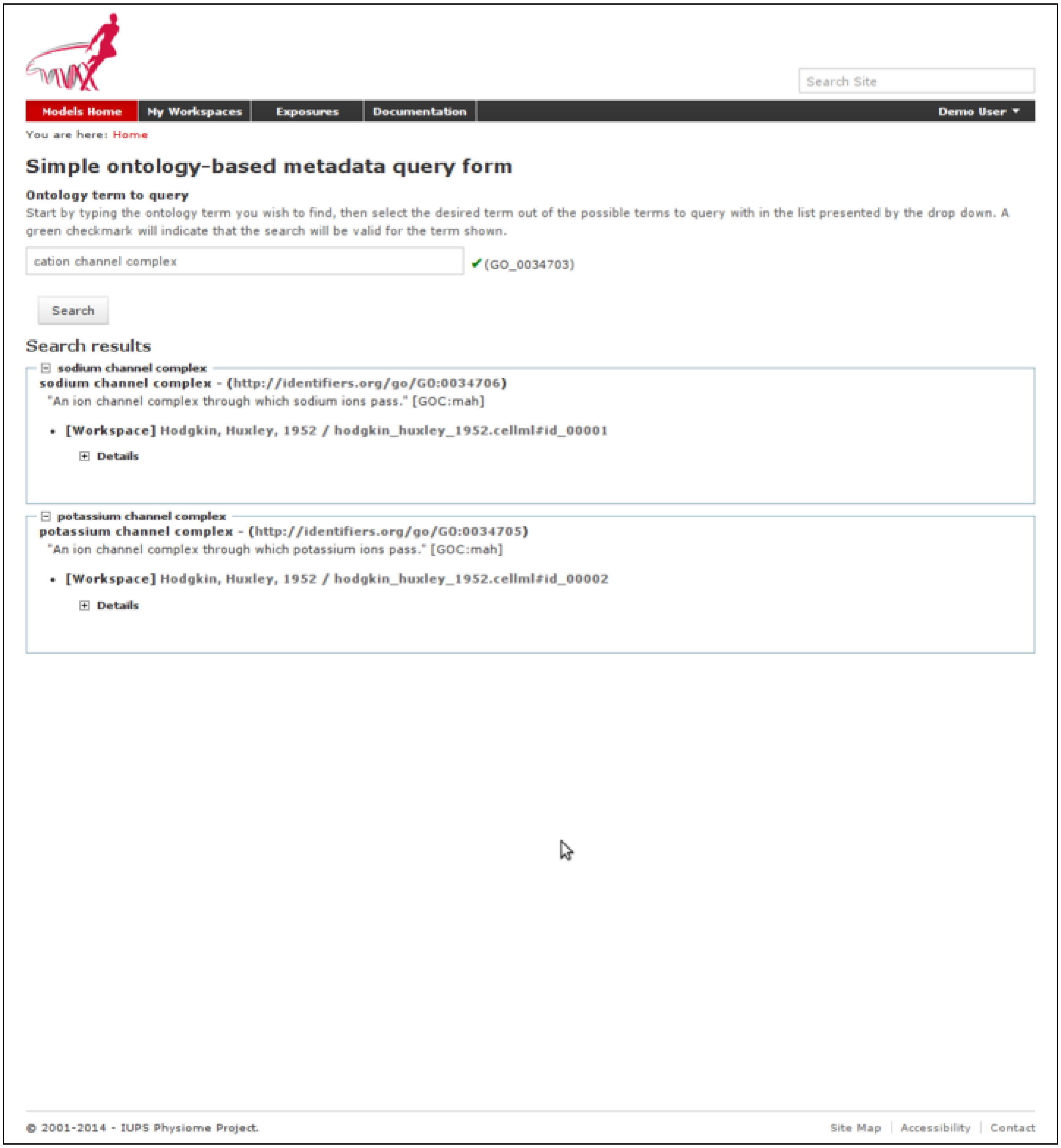
RDF Paths

Paths that will be indexed as RDF.

hodgkin_1952.xul
hodgkin_huxley_1952.cellml
hodgkin_huxley_1952.session.xm
hodgkin_huxley_1952_variant01.
hodgkin_huxley_1952_variant01.

© 2001-2014 - IUPS Physiome Project.

checkmark and now you may select the "Search" button. The search result will now list the workspace and the file that contain this annotation.



Simple ontology-based metadata query form

Ontology term to query

Start by typing the ontology term you wish to find, then select the desired term out of the possible terms to query with in the list presented by the drop down. A green checkmark will indicate that the search will be valid for the term shown.

cation channel complex ✔(GO_0034703)

Search results

⊟ sodium channel complex

sodium channel complex - (http://identifiers.org/go/GO:0034706)

"An ion channel complex through which sodium ions pass." [GOC:mah]

- **[Workspace]** Hodgkin, Huxley, 1952 / hodgkin_huxley_1952.cellml#id_00001

⊞ Details

⊟ potassium channel complex

potassium channel complex - (http://identifiers.org/go/GO:0034705)

"An ion channel complex through which potassium ions pass." [GOC:mah]

- **[Workspace]** Hodgkin, Huxley, 1952 / hodgkin_huxley_1952.cellml#id_00002

⊞ Details

© 2001-2014 - IUPS Physiome Project.

1.8.7 Creating CellML exposures

CellML models in the Auckland Physiome 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

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 Auckland Physiome Repository. For more information on this feature, refer to the information on features or collaboration, or see the *relevant section of the tutorial*.

At this point you are recommended to submit the workspace for publication, using the *state:* menu at the top right of the workspace view page. This is especially important if you decide to make an exposure public, as having a private workspace for a public exposure will impede access of linked data, such as images for the introduction to that particular exposure.

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.

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.

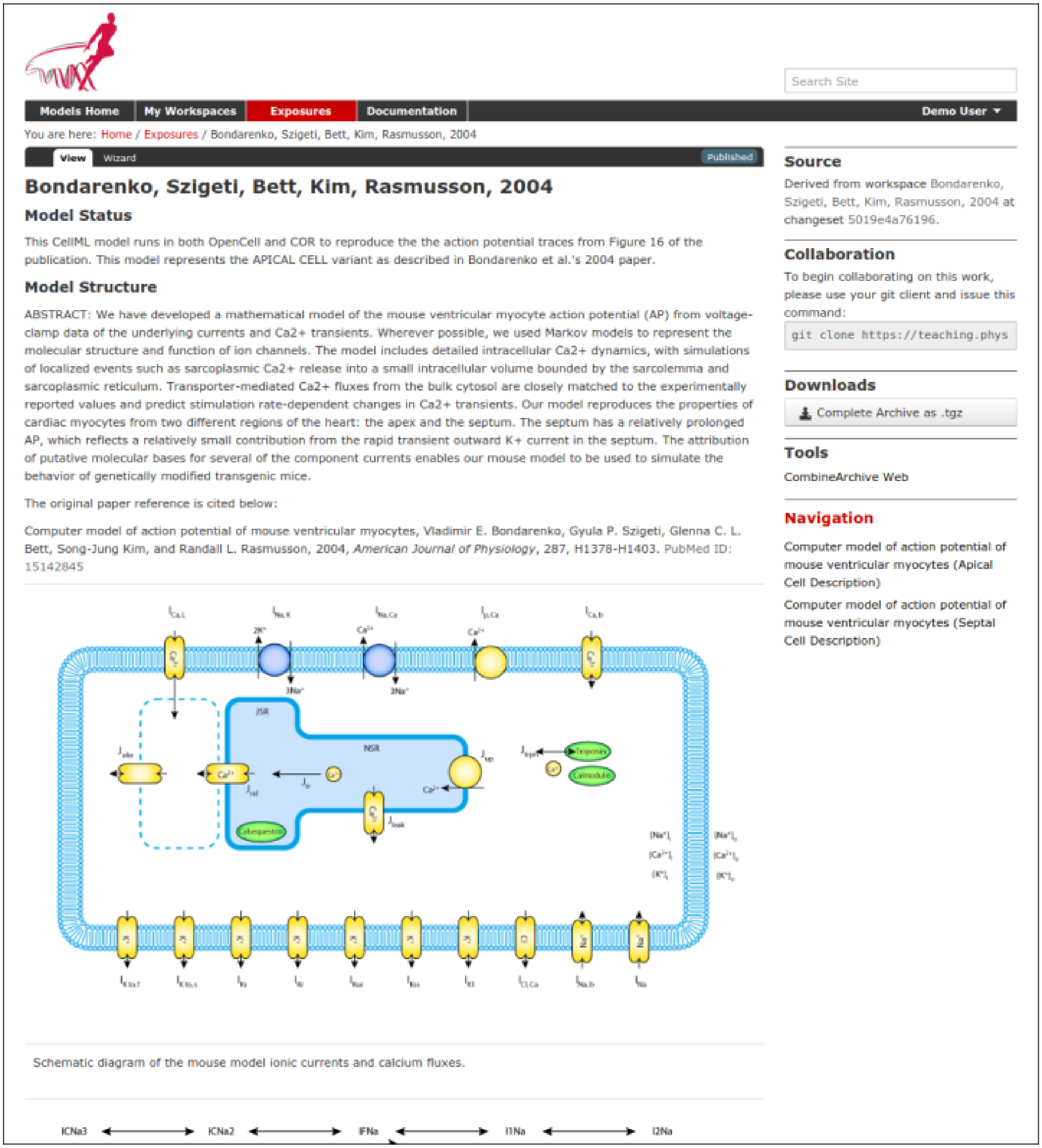
After selecting the revision you wish to expose, click on the *workspace actions* menu at the far right end of the menu bar and select *create exposure*.

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.

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



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.

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset 5019e4a76196.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone https://teaching.phys
```

Downloads

Tools

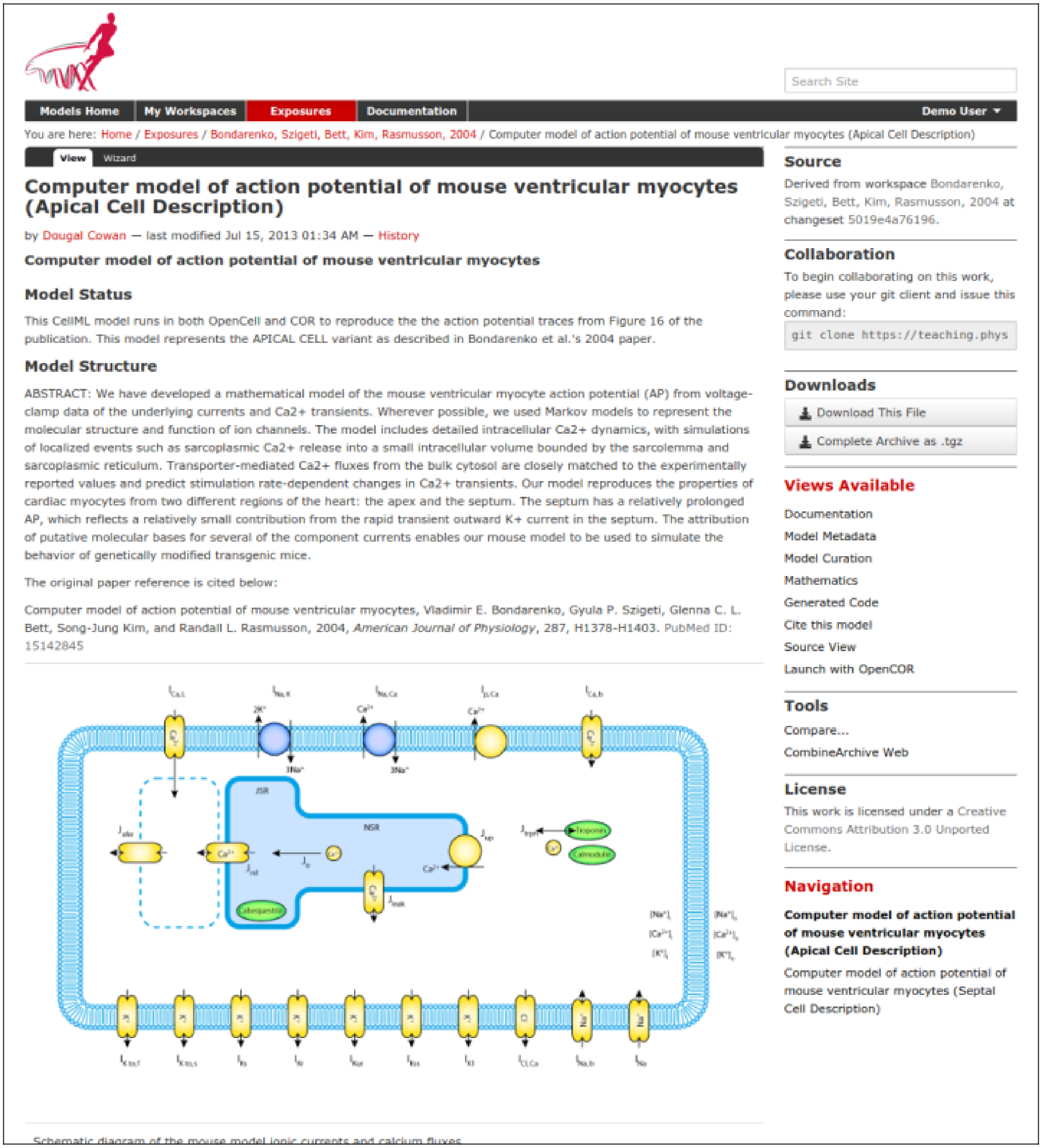
CombineArchive Web

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. 1.202: **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 Dougal Cowan — last modified Jul 15, 2013 01:34 AM — History

Computer model of action potential of mouse ventricular myocytes

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

Source

Derived from workspace Bondarenko, Szigeti, Bett, Kim, Rasmusson, 2004 at changeset 5019e4a76196.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone https://teaching.phys
```

Downloads

- Download This File
- Complete Archive as .tgz

Views Available

- Documentation
- Model Metadata
- Model Curation
- Mathematics
- Generated Code
- Cite this model
- Source View
- Launch with OpenCOR

Tools

- Compare...
- CombineArchive Web

License

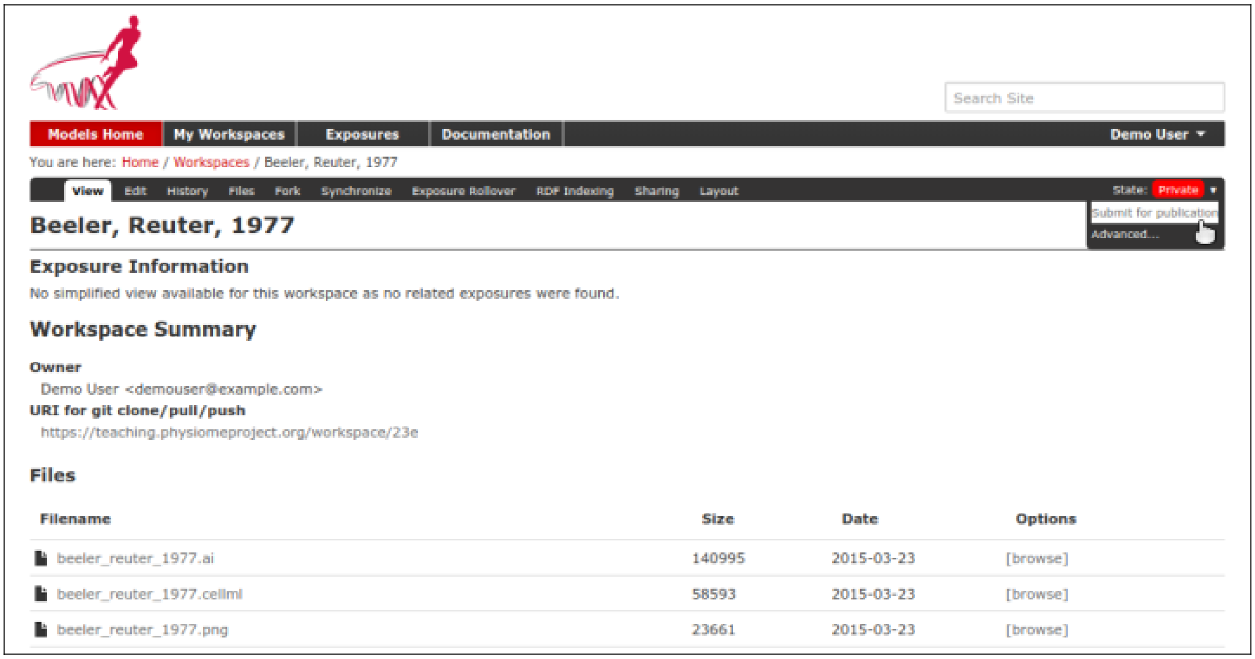
This work is licensed under a Creative Commons Attribution 3.0 Unported License.

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. 1.203: **Example of a model exposure page**



Beeler, Reuter, 1977

Exposure Information

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

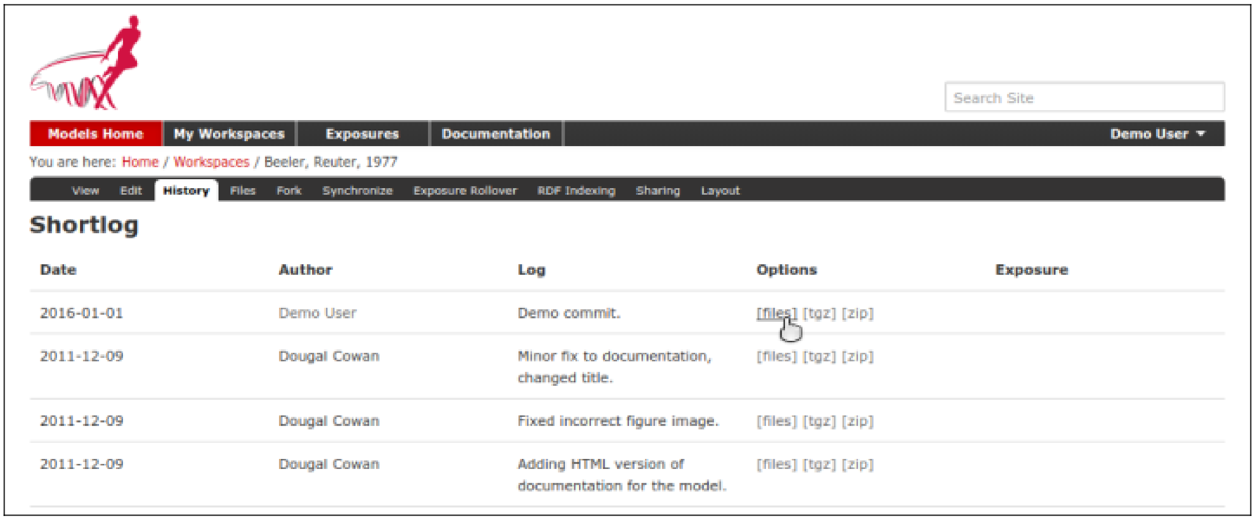
Workspace Summary

Owner
Demo User <demouser@example.com>
URI for git clone/pull/push
https://teaching.physiomeproject.org/workspace/23e

Files

Filename	Size	Date	Options
beeler_reuter_1977.ai	140995	2015-03-23	[browse]
beeler_reuter_1977.cellml	58593	2015-03-23	[browse]
beeler_reuter_1977.png	23661	2015-03-23	[browse]

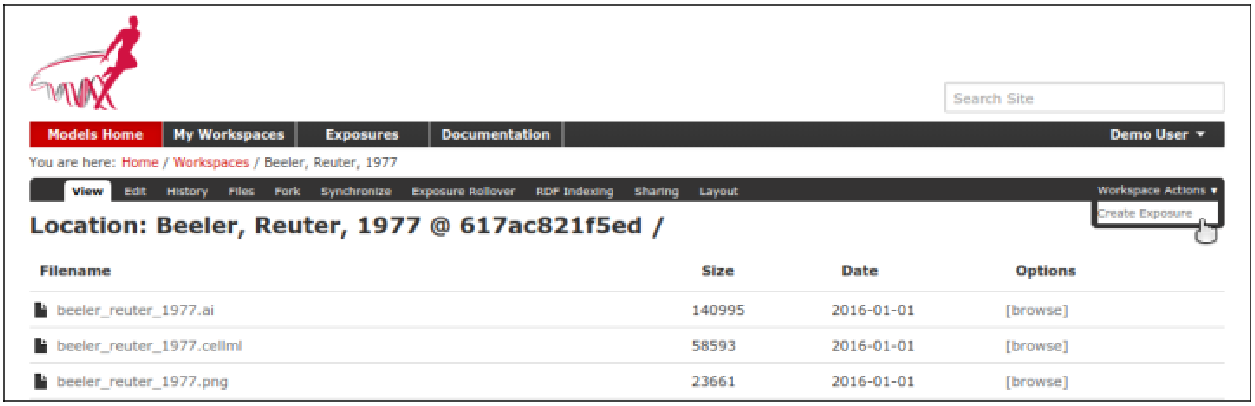
Fig. 1.204: **The state menu is used to submit objects such as workspaces for publication. Submitted items will be reviewed by site administrators and then published.**



Shortlog

Date	Author	Log	Options	Exposure
2016-01-01	Demo User	Demo commit.	[files] [tgz] [zip]	
2011-12-09	Dougal Cowan	Minor fix to documentation, changed title.	[files] [tgz] [zip]	
2011-12-09	Dougal Cowan	Fixed incorrect figure image.	[files] [tgz] [zip]	
2011-12-09	Dougal Cowan	Adding HTML version of documentation for the model.	[files] [tgz] [zip]	

Fig. 1.205: **The revision history of a fork of the Beeler Reuter 1977 workspace**

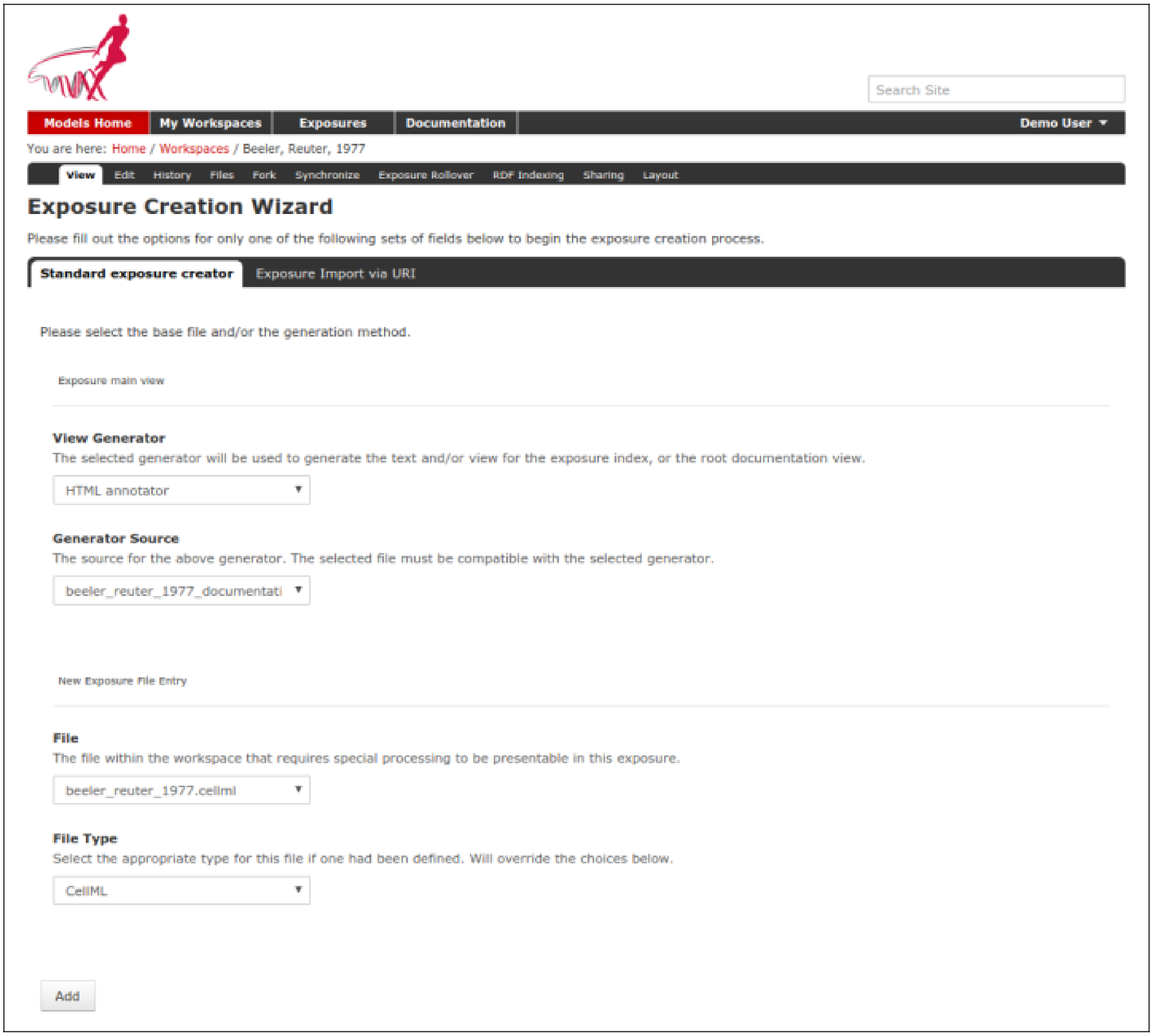


You are here: Home / Workspaces / Beeler, Reuter, 1977

Location: Beeler, Reuter, 1977 @ 617ac821f5ed /

Filename	Size	Date	Options
beeler_reuter_1977.ai	140995	2016-01-01	[browse]
beeler_reuter_1977.cellml	58593	2016-01-01	[browse]
beeler_reuter_1977.png	23661	2016-01-01	[browse]

Fig. 1.206: **Selecting the manifest of the revision to expose**



You are here: Home / Workspaces / Beeler, Reuter, 1977

Exposure Creation Wizard

Please fill out the options for only one of the following sets of fields below to begin the exposure creation process.

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_documentati

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. 1.207: **Selecting the main documentation and the first CellML model file**

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, in the cases where the above option is unsuitable.
- **Source Viewer**
  - Language Type - select xml
- **SedML file**
  - The SedML file associated with the given CellML file; if specified, the 'Launch with OpenCOR' link will open this file instead of the CellML file.
- **Manifest path**
  - A COMBINE archive manifest file associated with the CellML file; if specified, a COMBINE Archive download link will be enabled for the exposure page for this CellML file.

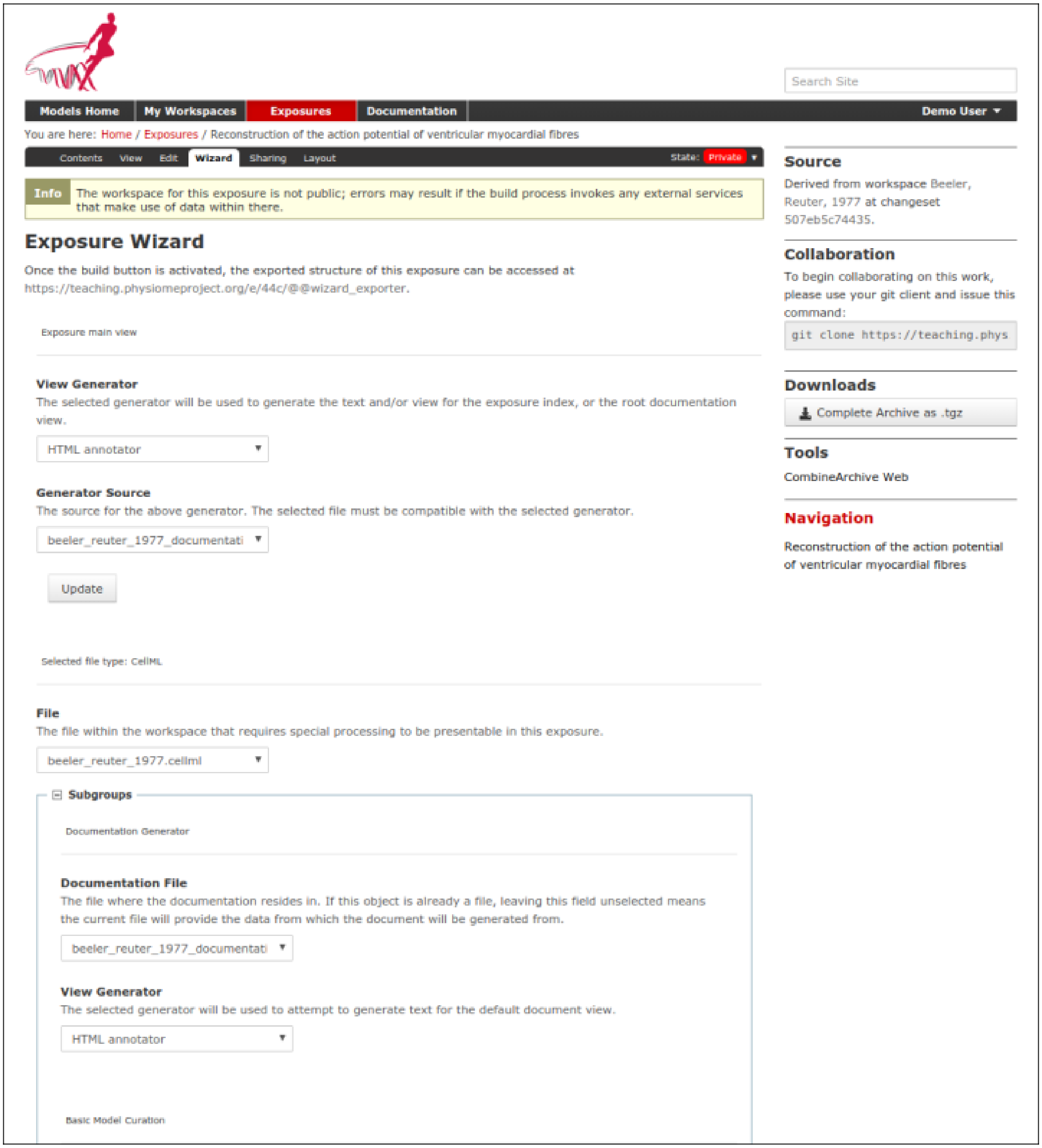
After selecting the subgroup options, you need to select the *Update* button to apply the chosen options for the exposure builder, as this is an independent subform to the main form. The options you selected will be ignored if this *Update* button is not selected, and the options will be replaced by the default options when you click *Build* before this was done.

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.

Making your work publicly accessible

In order to make the exposure visible and searchable, you will need to publish it. You can choose to submit your exposure for review, so that the repository administrators or curators will know to publish it for you. Naturally, if you have sufficient privileges you can publish it directly.



You are here: Home / Exposures / Reconstruction of the action potential of ventricular myocardial fibres

The workspace for this exposure is not public; errors may result if the build process invokes any external services that make use of data within there.

Exposure Wizard

Once the build button is activated, the exported structure of this exposure can be accessed at https://teaching.physiomeproject.org/e/44c/@@wizard_exporter.

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_documentati

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_documentati

View Generator

The selected generator will be used to attempt to generate text for the default document view.

HTML annotator

Basic Model Curation

Source

Derived from workspace Beeler, Reuter, 1977 at changeset 507eb5c74435.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone https://teaching.phys
```

Downloads

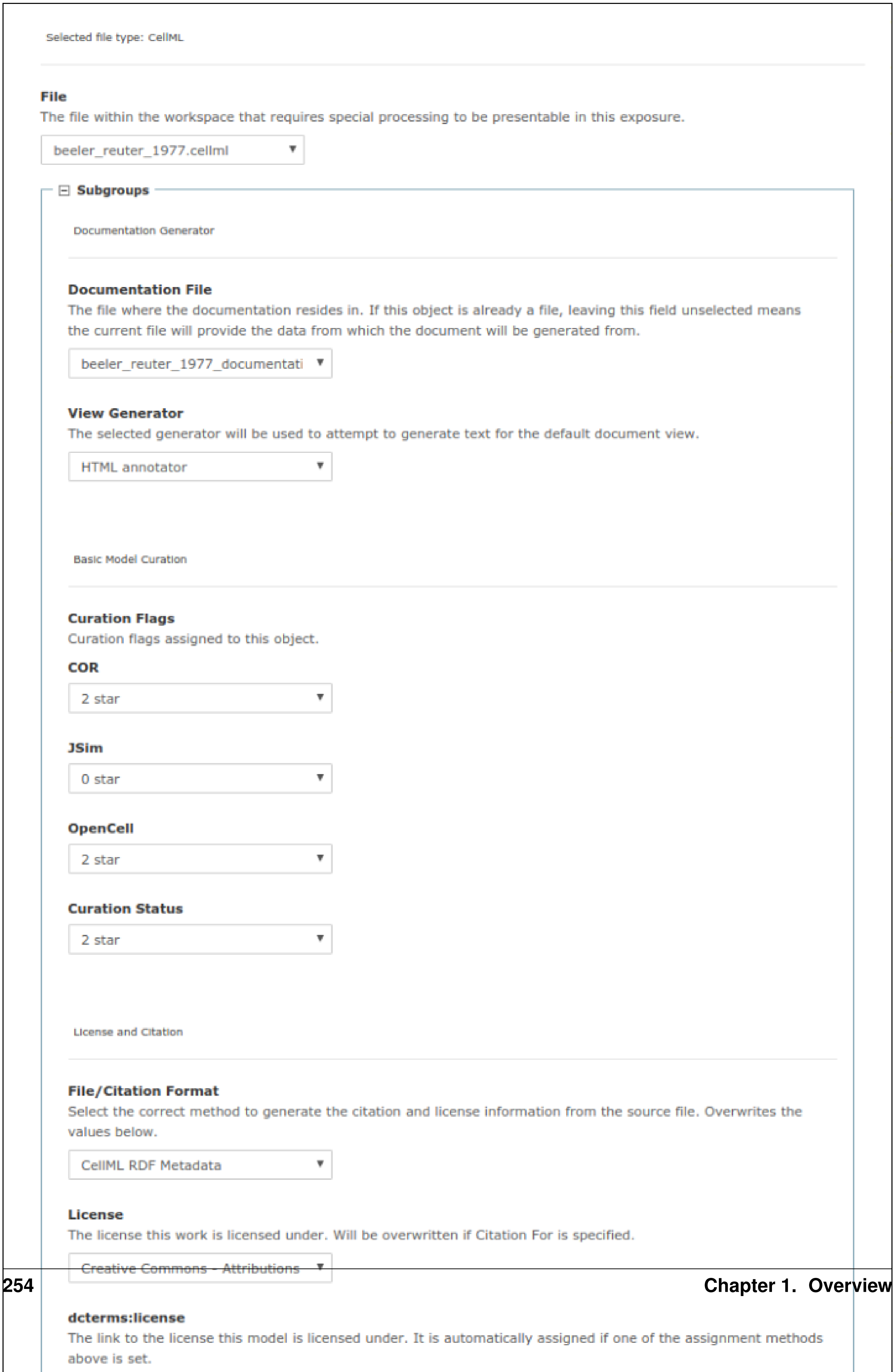
Tools

CombineArchive Web

Navigation

Reconstruction of the action potential of ventricular myocardial fibres

Fig. 1.208: Note that if your workspace is not publicly accessible, there will be an informative note for this which you can safely ignore as this warning was for a previous version of the CellML model loader that required the workspace to be publicly accessible. That said, if the exposure became public with the workspace remaining as private, issue will arise if visitors attempt to access the underlying workspace data.



Selected file type: CellML

File

The file within the workspace that requires special processing to be presentable in this exposure.

beeler_reuter_1977.cellml

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_documentati

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

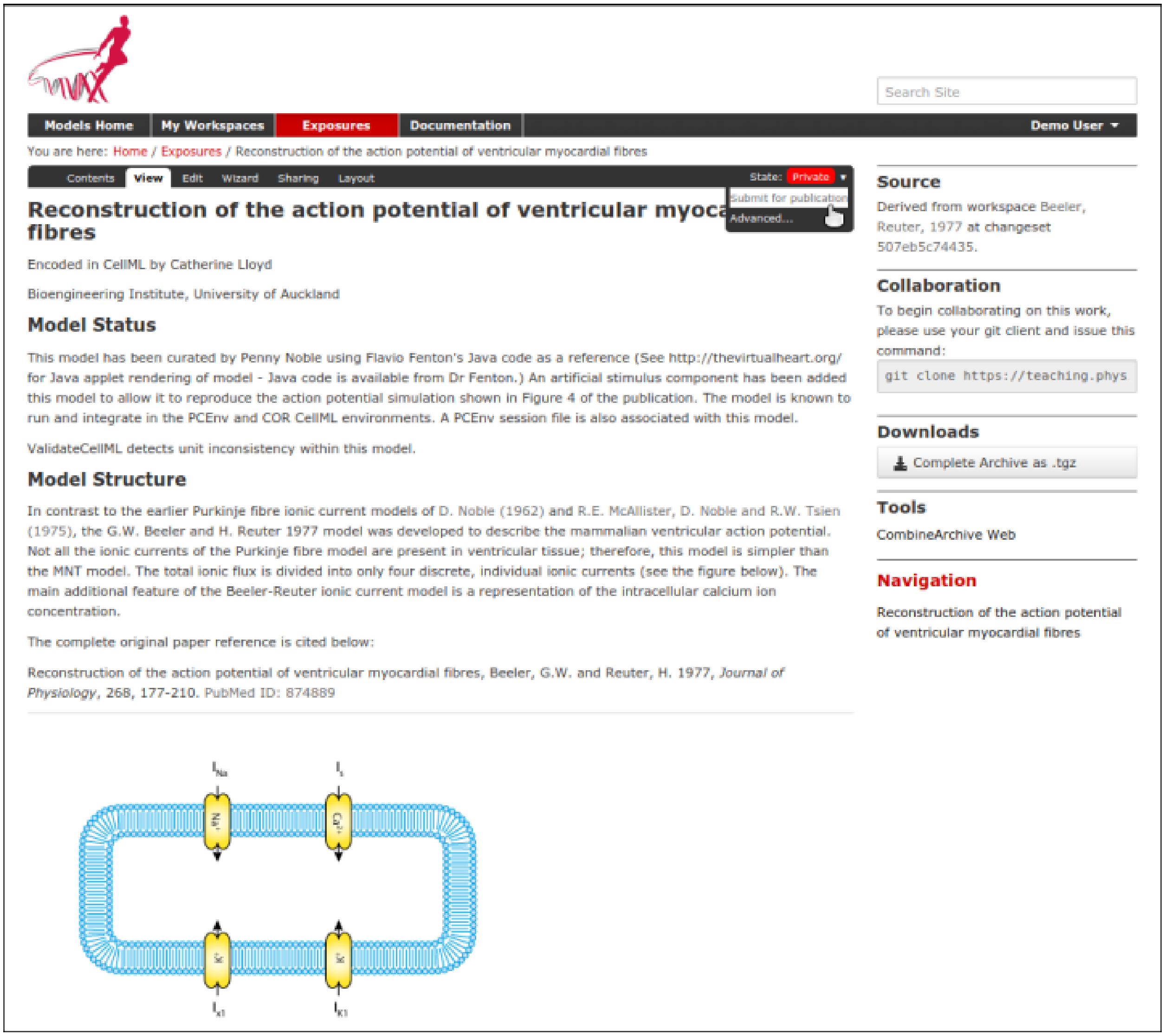
License

The license this work is licensed under. Will be overwritten if Citation For is specified.

Creative Commons - Attributions

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.



You are here: Home / Exposures / Reconstruction of the action potential of ventricular myocardial fibres

Reconstruction of the action potential of ventricular myoca 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 of the publication. The model is known to run and integrate in the PCEnv and COR CellML environments. A PCEnv session file is also associated with this model.

ValidateCellML detects unit inconsistency within this model.

Model Structure

In contrast to the earlier Purkinje fibre ionic current models of D. Noble (1962) and R.E. McAllister, D. Noble and R.W. Tsien (1975), the G.W. Beeler and H. Reuter 1977 model was developed to describe the mammalian ventricular action potential. Not all the ionic currents of the Purkinje fibre model are present in ventricular tissue; therefore, this model is simpler than the MNT model. The total ionic flux is divided into only four discrete, individual ionic currents (see the figure below). The main additional feature of the Beeler-Reuter ionic current model is a representation of the intracellular calcium ion concentration.

The complete original paper reference is cited below:

Reconstruction of the action potential of ventricular myocardial fibres, Beeler, G.W. and Reuter, H. 1977, *Journal of Physiology*, 268, 177-210. PubMed ID: 874889

Source

Derived from workspace Beeler, Reuter, 1977 at changeset 507eb5c74435.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone https://teaching.phys
```

Downloads

Complete Archive as .tgz

Tools

CombineArchive Web

Navigation

Reconstruction of the action potential of ventricular myocardial fibres

Fig. 1.210: **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 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

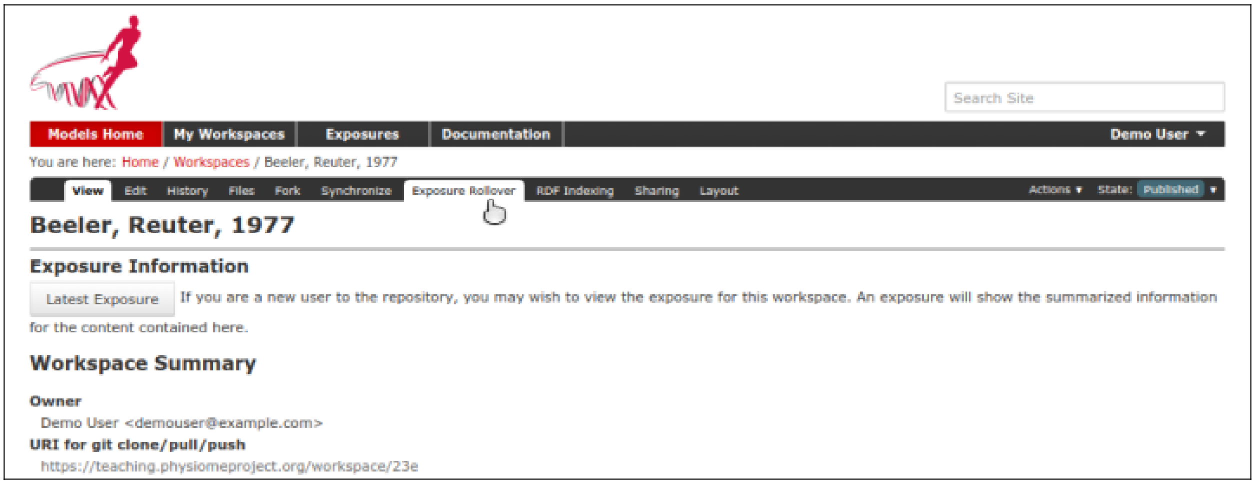
Making an exposure using "rollover"

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 has no linkages to any of the exposures that existed for the original workspace. However, you may navigate to the history of the original workspace and select any exposure, then select the wizard tab to the link to its exported structure, from which the exposure can be migrated over. Please see *the section on migrating exposure* for more details.

From the view page of your workspace, select "exposure rollover".



You are here: Home / Workspaces / Beeler, Reuter, 1977

Beeler, Reuter, 1977

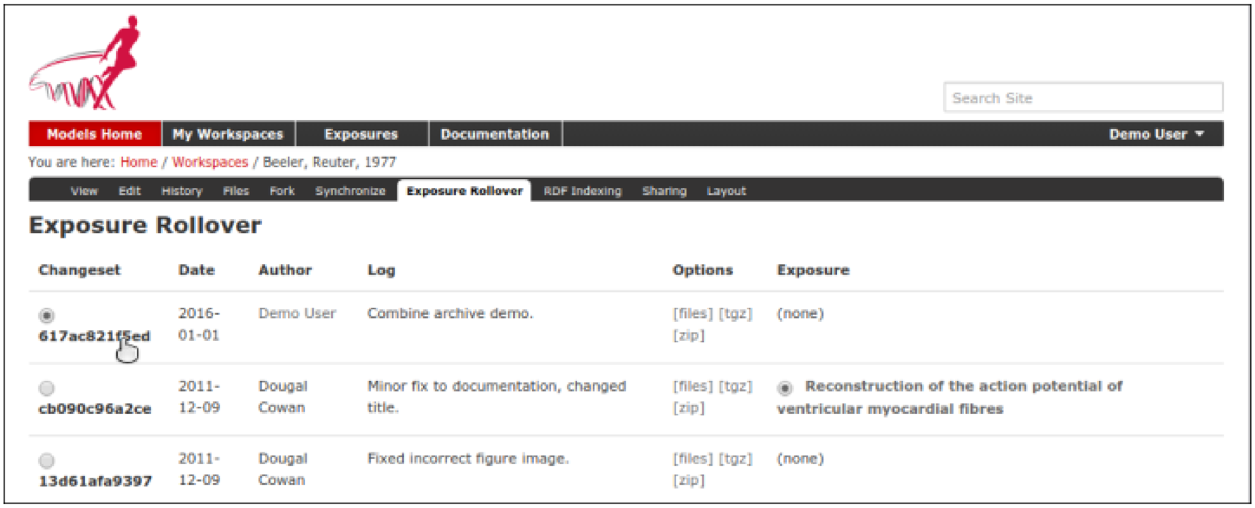
Exposure Information

If you are a new user to the repository, you may wish to view the exposure for this workspace. An exposure will show the summarized information for the content contained here.

Workspace Summary

Owner
Demo User <demouser@example.com>
URI for git clone/pull/push
https://teaching.physiomeproject.org/workspace/23e

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
● 617ac821f5ed	2016-01-01	Demo User	Combine archive demo.	[files] [tgz] [zip]	(none)
○ cb090c96a2ce	2011-12-09	Dougal Cowan	Minor fix to documentation, changed title.	[files] [tgz] [zip]	● Reconstruction of the action potential of ventricular myocardial fibres
○ 13d61afa9397	2011-12-09	Dougal Cowan	Fixed incorrect figure image.	[files] [tgz] [zip]	(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.

1.8.8 Creating FieldML exposures

FieldML models in the Auckland Physiome 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

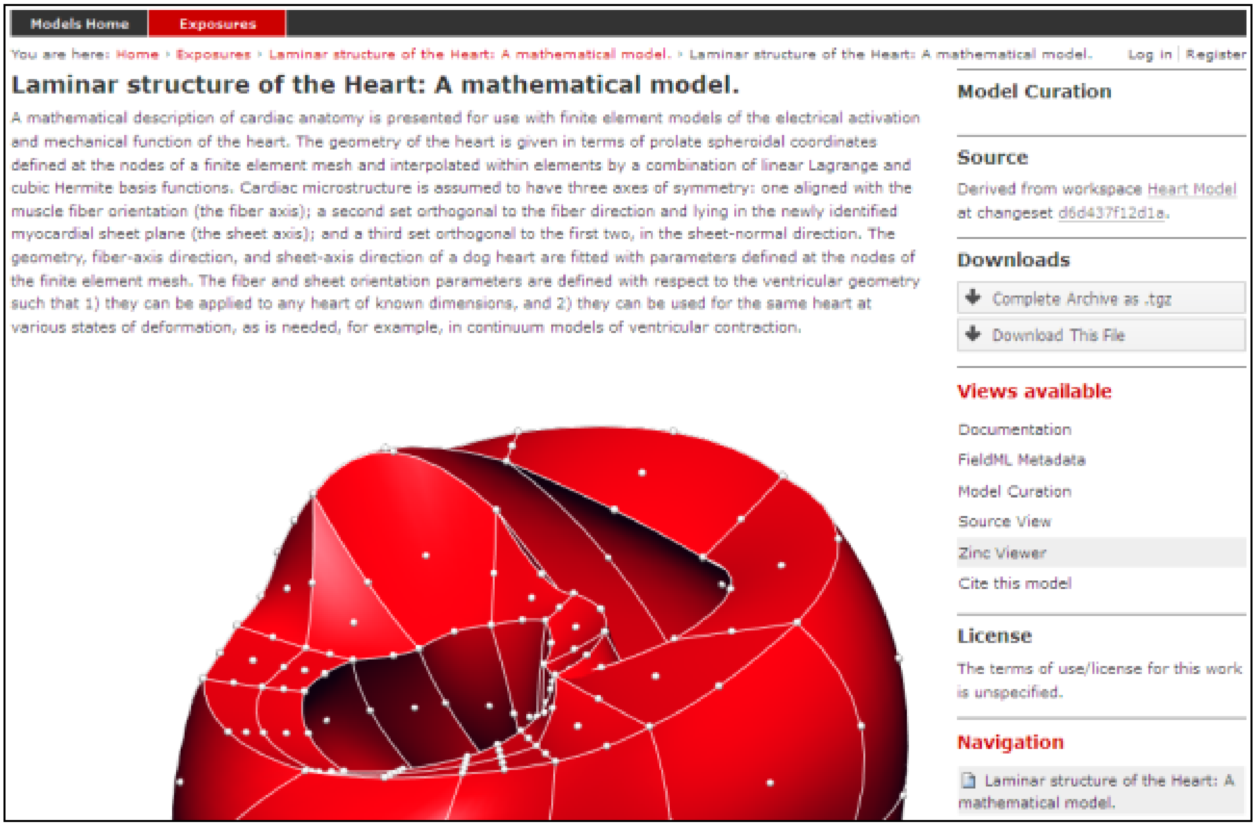
To create a FieldML exposure, the following files will need to be stored in a workspace in the repository:

- 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 repository:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
      xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

(continues on next page)


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

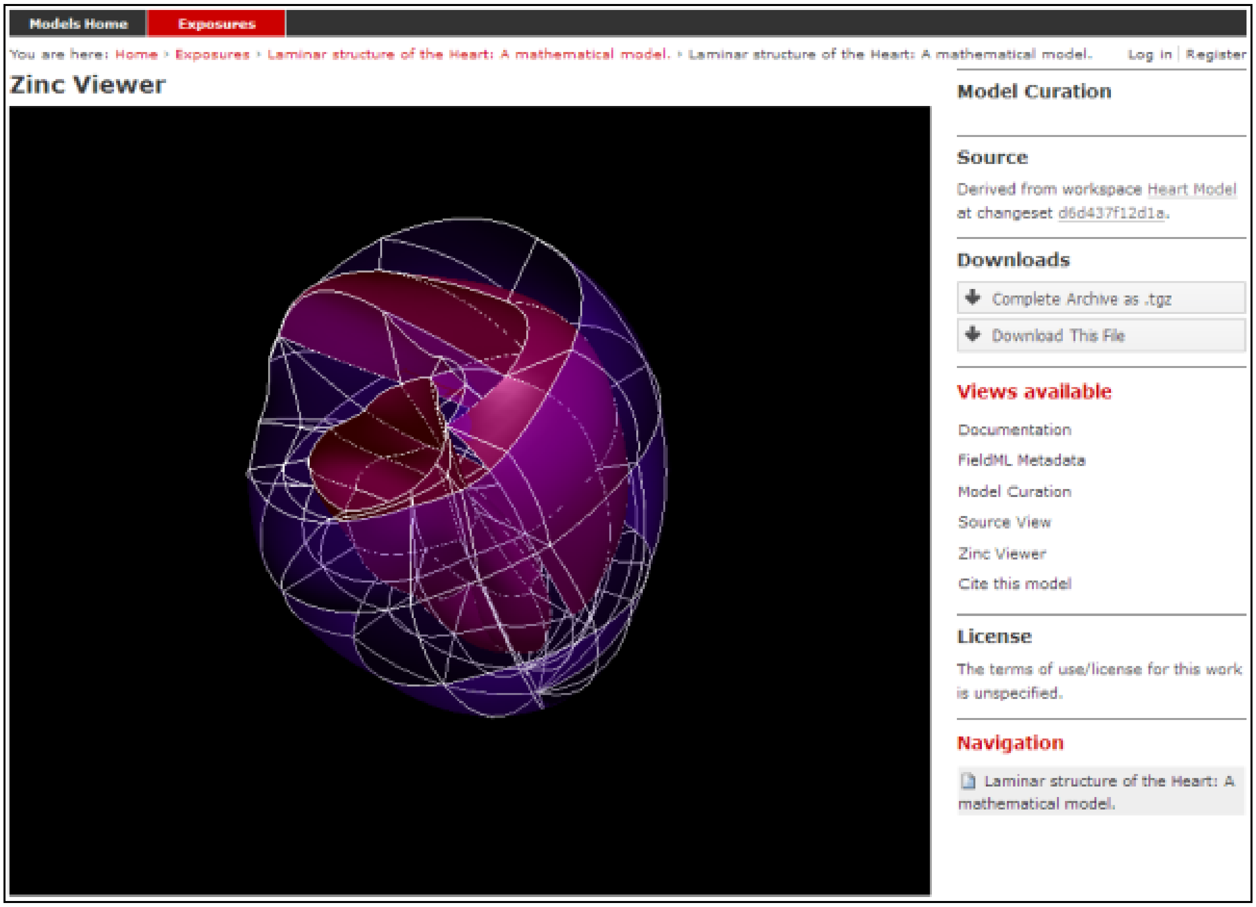
Derived from workspace Heart Model at changeset d6d437f12d1a.

Downloads

License

The terms of use/license for this work is unspecified.

Fig. 1.211: The main documentation view of a FieldML exposure



You are here: Home › Exposures › Laminar structure of the Heart: A mathematical model. › Laminar structure of the Heart: A mathematical model.

Zinc Viewer

Model Curation

Source

Derived from workspace Heart Model at changeset d6d437f12d1a.

Downloads

Views available

License

The terms of use/license for this work is unspecified.

Navigation

Fig. 1.212: The main Zinc viewer view of the same FieldML exposure

(continued from previous page)

```
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": [
```

(continues on next page)

(continued from previous page)

```
11          {
12              "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 *PMR2*.

Creating the exposure in the Auckland Physiome 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.

1.8.9 Embedded workspaces and their uses

Warning: This section is a work in progress, please send us any questions, comments, or issues.

Todo: This section needs more work. Need to add documentation on how to do the examples using a GUI client like TortoiseGit.

Workspaces in *PMR* are currently implemented as *Git* repositories. One Git feature that is quite useful in the context of the Auckland Physiome Repository is the ability to nest repositories using Git submodules. Using the more general *PMR2* concepts, we term such nesting as *embedded workspaces*.

Embedded workspaces:

- are intended to manage the separation of modules or sub-models which are integrated to create a model;
- facilitate the sharing and reuse of models and sub-models independently from specific instances of a model;
- enable the development of the modules to proceed independently, thus the version of the workspaces embedded is also tracked; and
- allow authors to make use of relative URIs when linking between data resources providing a file system agnostic method to describe complex module relationships in a portable manner.

Workspaces can be embedded at a specific revision or set to track the most recent revision of the source workspace. Changes made to the source workspace will not affect any embedding workspace until the author explicitly chooses to update the embedded workspace. This provides the author with the opportunity to review the changesets and make an informed decision regarding alterations to embedded revisions. Any alterations in the specific revision of an embedded workspace is data captured in a changeset in the embedding workspace – thus providing a clear provenance record of the entire dataset in the workspace.

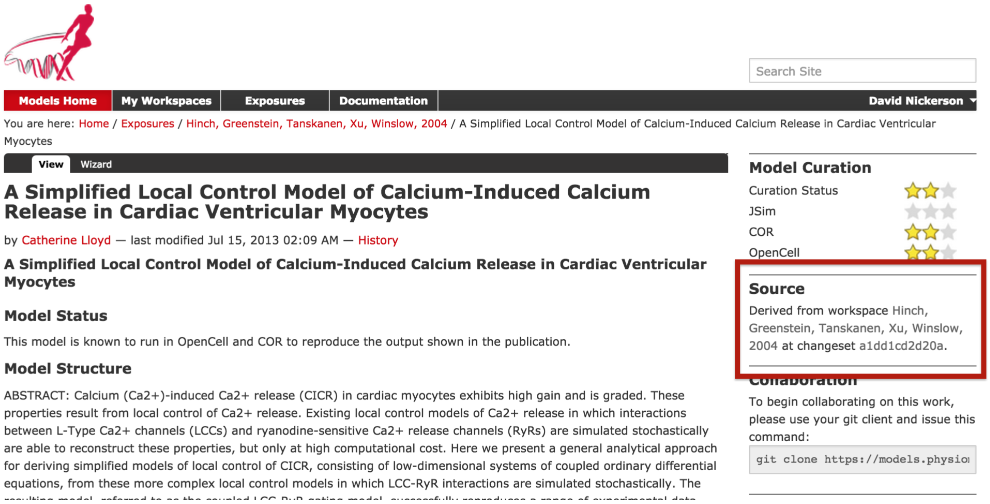
Uses

The Git submodules documentation provides detailed information on the use of Git submodules, and the documentation of the git-submodule command is also a useful resource. Here we provide some typical examples demonstrating the use of embedded workspaces with PMR.

Making use of a fixed version of a CellML model

A common use-case for creating an embedded workspace is when you wish to take a curated CellML model from PMR and use it in your own work. Here we demonstrate how to embed a specific version of a workspace within your own workspace. For our example we have our own workspace in which we are developing a cardiac mechanics model and we now want to couple our mechanics model to a model of cardiac calcium dynamics.

A popular cardiac calcium dynamics model is that of Hinch et al (2004). An encoding of this model in CellML can be found in PMR here. From this page, we are able to find two key pieces of information that we need in order to embed this version of the Hinch *et al* model into our workspace, as shown below, the *source workspace* and the *changeset* identifier.



You are here: Home / Exposures / Hinch, Greenstein, Tanskanen, Xu, Winslow, 2004 / A Simplified Local Control Model of Calcium-Induced Calcium Release in Cardiac Ventricular Myocytes

A Simplified Local Control Model of Calcium-Induced Calcium Release in Cardiac Ventricular Myocytes

by Catherine Lloyd — last modified Jul 15, 2013 02:09 AM — History

A Simplified Local Control Model of Calcium-Induced Calcium Release in Cardiac Ventricular Myocytes

Model Status

This model is known to run in OpenCell and COR to reproduce the output shown in the publication.

Model Structure

ABSTRACT: Calcium (Ca2+)-induced Ca2+ release (CICR) in cardiac myocytes exhibits high gain and is graded. These properties result from local control of Ca2+ release. Existing local control models of Ca2+ release in which interactions between L-Type Ca2+ channels (LCCs) and ryanodine-sensitive Ca2+ release channels (RyRs) are simulated stochastically are able to reconstruct these properties, but only at high computational cost. Here we present a general analytical approach for deriving simplified models of local control of CICR, consisting of low-dimensional systems of coupled ordinary differential equations, from these more complex local control models in which LCC-RyR interactions are simulated stochastically. The

Model Curation

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

Source

Derived from workspace Hinch, Greenstein, Tanskanen, Xu, Winslow, 2004 at changeset a1dd1cd2d20a.

Collaboration

To begin collaborating on this work, please use your git client and issue this command:

```
git clone https://models.physio
```

Fig. 1.213: The Hinch *et al* (2004) exposure page in PMR, highlighting the **Source** section of the CellML exposure page.

From the **Source** section highlighted above, right-clicking on the workspace link should allow you to easily copy the source workspace URL. You can then navigate to your local *clone* of your workspace you can embed the Hinch *et al* workspace as follows.

```
$ git submodule add https://models.physiomeproject.org/workspace/hinch_greenstein_
↪tanskanen_xu_winslow_2004
Cloning into 'hinch_greenstein_tanskanen_xu_winslow_2004'...
remote: dul-daemon says what
remote: counting objects: 29, done.
remote: how was that, then?
Unpacking objects: 100% (29/29), done.
Checking connectivity... done.
```

By default the embedded workspace (Git submodule) will have the same name as the source workspace and be placed in the root of your workspace, but you can provide a different path to the above command if desired.

When the embedded workspace is first created, it will be initialised to the latest version of the workspace, so we now need to *checkout* the specific version of the embedded workspace that we are after (shown in the figure above). We can do this as shown below.

```
$ cd hinch_greenstein_tanskanen_xu_winslow_2004
$ git checkout a1dd1cd2d20a
Note: checking out 'a1dd1cd2d20a'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at a1dd1cd... Tidied session file
```

We now have the Hinch *et al* workspace embedded and set the desired version. If we traverse back to our workspace root and check the status of the workspace:

```
$ cd ..
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

   new file:   .gitmodules
   new file:   hinch_greenstein_tanskanen_xu_winslow_2004

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   hinch_greenstein_tanskanen_xu_winslow_2004 (new commits)
```

we can see that the submodule has been added but that is has also been modified (due to changing from the latest revision to that specified in the exposure page). So as per standard Git usage, we add the change and then can commit the newly embedded workspace and push the changes in our workspace back to the repository.

```
$ git commit -m "Embedding Hinch et al (2004) model from exposure https://models.
↪physiomeproject.org/exposure/8e1a590fb82a2cab5284502b430c4a4f."
[master (root-commit) 563de87] Embedding Hinch et al (2004) model from exposure␣
↪https://models.physiomeproject.org/exposure/8e1a590fb82a2cab5284502b430c4a4f.
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 hinch_greenstein_tanskanen_xu_winslow_2004
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 489 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://teaching.physiomeproject.org/workspace/273
```

We can now confirm that we have the correct version of the Hinch *et al* model embedded by using our browser to navigate to our workspace in the repository (here we use the *teaching instance*). You should now see the embedded workspace listed in the view of your workspace, and clicking on the embedded workspace should take you directly to the source workspace at the correct revision, namely: https://models.physiomeproject.org/workspace/hinch_greenstein_tanskanen_xu_winslow_2004/file/a1dd1cd2d20a4f1d00c69ce6cd1b968ea0836659/.

Note: *PMR2* does some clever redirects to resolve the embedded workspaces, so the acutal link displayed for the Hinch *et al* workspace in your workspace will not directly point to the source workspace and revision.

Updating to a newer revision

Once you have created an embedded workspace, it can be used as an independent Git repository within your workspace - you can make changes, commit them, and, if you have permission, push the changes back to the original source workspace. In the example described above, we embedded the Hinch *et al* (2004) calcium model into our workspace. We specifically embedded the revision of the workspace which matched a given exposure in PMR. If we look at the history of the source workspace, we can see that the Hinch workspace has progressed since that exposure was made.

Upon examining the changes in the workspace, we decide in this example that it would be beneficial to our work to update our embedded version to match the latest changes in the source Hinch workspace. This can be accomplished as follows.

The first step is to update our embedded workspace to the latest revision.

```
$ cd hinch_greenstein_tanskanen_xu_winslow_2004/
$ git checkout master
Previous HEAD position was a1dd1cd... Tidied session file
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git pull
Already up-to-date.
```

(The final *git pull* is simply to confirm there have been no further changes.)

We can now see that in our own workspace that the embedded workspace has changed:

```
$ cd ..
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   hinch_greenstein_tanskanen_xu_winslow_2004 (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

Committing that change will then update the embedded workspace to the latest revision:

```
$ git add hinch_greenstein_tanskanen_xu_winslow_2004
$ git commit -m "updating embedded version of the Hinch calcium model to the latest␣
↪revision."
[master 1b74217] updating embedded version of the Hinch calcium model to the latest␣
↪revision.
 1 file changed, 1 insertion(+), 1 deletion(-)
```

(continues on next page)

(continued from previous page)

```
$ git push
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 294 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://teaching.physiomeproject.org/workspace/273
   563de87..1b74217  master -> master
```

Following the commit, and if you are curious, you can see that the actual changeset committed is an update of the revision ID of the embedded workspace:

```
-Subproject commit a1dd1cd2d20a4f1d00c69ce6cd1b968ea0836659
+Subproject commit a55b3f2eb14e937a816b3f975722e44d1d3915bd
```

and browsing your workspace in PMR should link the embedded workspace to the updated version of the Hinch model.

Cloning a workspace containing embedded workspace(s)

In you workspace stored in PMR, embedded workspaces are simply stored as links to the actual source workspace - *not* the actual contents. Thus, if you simply *clone* your workspace from PMR you will get that link and not the actual contents (which is usually what you really want). If you check the Git book you can see how to deal with this case in the general case. But for most purposes it is easiest to simply perform a recursive clone of your workspace. This can be done on the command line with the following.

```
$ git clone --recursive [workspace url]
```

where `[workspace url]`` should be replaced with your actual workspace URL (e.g., `https://models.physiomeproject.org/workspace/NNNN`).

Best practice

1.8.10 CellML Curation in the legacy Physiome Model Repository

As the Auckland Physiome Repository contains much of the data ported over from the legacy software products that powered what was called the CellML Model Repository, the curation system from that system was ported to Auckland Physiome Repository 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 Auckland Physiome 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).

1.8.11 Webservice for Physiome Repository

Overview

This document describes how the webservices are provided by the Physiome Repository. Generally, using the webservice is simply providing the `Accept` header to any URL within the instance.

Current Version

The webservice communicates using JSON, and by default sending the header `Accept:  application/json` to resource endpoints that support this protocol will return the requested content as JSON. However, it is highly encouraged that consumers provide version specific header, in this case it will be:

```
Accept: application/vnd.physiome.pmr2.json.1
```

In other words, there are no dedicated API servers or alternative URLs to access resources within the Physiome Repository as a webservice. Clients have to negotiate the intended content type they desire to consume through the usage of this `Accept` header.

Schema/Payload format

All data are communicated using the Collection+JSON hypermedia type. Please refer to the specification as to how this might be used. As this standard is designed as a proper hypermedia type a browser that sends and interpret Collection+JSON can be used to navigate the repository much like how an end-user might navigate the repository with their web browser. One such example is through the Collection+JSON Browser with the root endpoint of the repository (however, the example link uses the URL to the welcome page as the browser does not support redirection to that on its own).

However, as implemented the restricted views are not necessarily readily available by those links, which PMR2 implements an alternative end point specific for this purpose. The PMR2 Dashboard was created for this purpose but in the future it may be linked directly to the root end point of the repository.

Authentication

For third-party applications that are interested in accessing private content belonging to its users, Physiome Repository supports the use OAuth for client authentication. Applications must be registered manually by the system administrators. An example demo application (using the previous version of the JSON-based protocol) is provided by the package pmr2.client.

Example

Generally, all end points that can be accessible by a web browser should also be accessible by the web service. For instance, loading the front page of the official repository can be done like so, using `curl` with the follow location flag (`-L`):

```
$ curl -sL -H 'Accept: application/vnd.physiome.pmr2.json.1' \
>     https://staging.physiomeproject.org/ |python -m json.tool
```

The result of the above command should show something like so:

```
{
    "collection": {
        "href": "https://staging.physiomeproject.org/welcome/document_view",
        "items": [
            {
                "data": [
                    {
                        "name": "contents",
                        ...
                    }
                ],
            }
        ],
        "links": [
            {
                "href": "https://staging.physiomeproject.org/exposure",
                "prompt": "Main model listing",
                "rel": "bookmark"
            },
            ...
        ]
    }
}
```

Your client can simply follow the `href` inside any of the sections, especially the `links` section, for the links with a `bookmark` relationship. For instance:

```
$ curl -sL -H 'Accept: application/vnd.physiome.pmr2.json.1' \
>     https://staging.physiomeproject.org/exposure |python -m json.tool
```

Output may look something like:

```
{
    "collection": {
        "href": "https://staging.physiomeproject.org/exposure/listing/pmr1_folder_
↪listing",
        "links": [
            {
                "href": "https://staging.physiomeproject.org/e/105",
                "prompt": "\r\n      A Quantitative Model of Human Jejunal Smooth␣
↪Muscle Cell Electrophysiology\r\n     ",
                "rel": "bookmark"
```

(continues on next page)

(continued from previous page)

```
            },
            ...
        ]
    }
}
```

An exposure may look like so:

```
$ curl -sL -H 'Accept: application/vnd.physiome.pmr2.json.1' \
>     https://staging.physiomeproject.org/e/c1 |python -m json.tool
{
    "collection": {
        "href": "https://staging.physiomeproject.org/e/c1/exposure_info",
        "links": [
            {
                "href": "https://staging.physiomeproject.org/e/c1/beeler_reuter_1977.
↪cellml/view",
                "prompt": "Reconstruction of the action potential of ventricular␣
↪myocardial fibres",
                "rel": "bookmark"
            },
            {
                "href": "https://staging.physiomeproject.org/workspace/beeler_reuter_
↪1977",
                "prompt": "Workspace URL",
                "rel": "via"
            }
        ],
        "version": "1.0"
    }
}
```

Note that there are two links, one of them containing a link to the object inside (TODO: note that v2 will change this to items), and also a `"rel": "via"` highlighting the fact that this link is derived from the associated `href`, in this case it would be the source workspace URL. Many other details within the repository can be gathered by progressively navigating through these links.

One thing that is currently not linked is the search form, which users of the site through the web browser will find on the upper-right hand corner as a search bar. This function is also replicated by navigating directly to that link:

```
$ curl -sL -H 'Accept: application/vnd.physiome.pmr2.json.1' \
>     https://staging.physiomeproject.org/search |python -m json.tool
```

The result will be a template that looks like this:

```
{
    "collection": {
        "href": "https://staging.physiomeproject.org/search",
        "template": [
            {
                "name": "SearchableText",
                "prompt": "SearchableText",
                "value": ""
            },
            {
                "name": "Title",
                "prompt": "Title",
```

(continues on next page)

(continued from previous page)

```
                "value": ""
            },
            {
                "name": "Description",
                "prompt": "Description",
                "value": ""
            },
            {
                "name": "Subject",
                "options": [
                    {
                        "value": "CellML Model"
                    },
                    {
                        "value": "FieldML Model"
                    }
                ],
                "prompt": "Subject",
                "value": ""
            },
            {
                "name": "portal_type",
                "options": [
                    {
                        "value": "Exposure"
                    },
                    ...
                ],
                "prompt": "portal_type",
                "value": ""
            }
        ],
        "version": "1.0"
    }
}
```

Submission is done using the Collection+JSON format also. For example:

```
$ curl -sL -H 'Accept: application/vnd.physiome.pmr2.json.1' \
>     https://staging.physiomeproject.org/search -d '{
>         "template": {"data": [
>             {"name": "SearchableText", "value": "Beeler"}
>         ]}
>     }' | python -mjson.tool
```

Results will be contained in the `links` section:

```
{
    "collection": {
        "href": "https://staging.physiomeproject.org/search",
        "links": [
            {
                "href": "https://staging.physiomeproject.org/exposure/
↪8123cbe00754b718a39fed5eb9bfb4a2/skouibine_trayanova_moore_1999.cellml/view",
                "prompt": "Anode/cathode make and break phenomena in a model of␣
↪defibrillation",
                "rel": "bookmark"
```

(continues on next page)

(continued from previous page)

```
            },
            {
                "href": "https://staging.physiomeproject.org/w/miller/beeler_reuter_
↪1977_uncertexample",
                "prompt": "Beeler, Reuter, 1977",
                "rel": "bookmark"
            },
            ...
        ]
    }
}
```

This does the same raw text-based search, however with this service it is possible to make use of the other fields to further refine the results by using them, for example, to search for only workspaces with `Beeler` in its title:

```
$ curl -sL -H 'Aecept: application/vnd.physiome.pmr2.json.1' \
>     https://staging.physiomeproject.org/search -d '{
>         "template": {"data": [
>             {"name": "Title", "value": "Beeler"},
>             {"name": "portal_type", "value": "Workspace"}
>         ]}
>     }' | python -mjson.tool
```

Results sould look like so:

```
{
    "collection": {
        "href": "https://staging.physiomeproject.org/search",
        "links": [
            {
                "href": "https://staging.physiomeproject.org/w/miller/beeler_reuter_
↪1977_uncertexample",
                "prompt": "Beeler, Reuter, 1977",
                "rel": "bookmark"
            },
            {
                "href": "https://staging.physiomeproject.org/w/tommy/my_beeler_model",
                "prompt": "My Beeler Model",
                "rel": "bookmark"
            },
            {
                "href": "https://staging.physiomeproject.org/workspace/beeler_reuter_
↪1977",
                "prompt": "Beeler, Reuter, 1977",
                "rel": "bookmark"
            },
            ...
        ]
    }
}
```

Note how the results is much more specific. For all othe searches and forms/endpoints that display a `template` section, you can follow the standard as such to submit data. This includes authenticated forms that one can access after being granted write-access through OAuth and such to create workspaces and exposures.

1.8.12 Glossary

Checkout Switch branches or restore working tree files.

Clone The term used with distributed version control systems (DVCS) to describe the process for making complete copy of another repository, usually hosted at a different site. This is done in order to have a local copy of a repository to work in.

Embedded workspace

Embedded workspaces Essentially workspace or workspaces that are nested inside another one through mechanisms offered by distributed version control systems. In *Git* it will be through the use of Git submodules.

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.

Git Git is a distributed version control system currently used by the Physiome Model Repository software to maintain a history of changes to files in *workspaces*. See a tour of the Git Beginner's Guide for some good introductory material.

PMR An instance of *PMR2* known as the Physiome Model Repository. The official URL of PMR is https://models.physiomeproject.org. A *teaching instance* is also available for testing and teaching purposes.

PMR2 The software that powers the Auckland Physiome Repository.

Pull

Pulling The term used with distributed version control systems for the action of pulling changes from one clone of the repository into another. With PMR, this usually implies pulling from a workspace in the model repository into a clone of the workspace on your local machine.

Push

Pushing The term used with distibuted version control systems for the action of pushing changes from one clone of the repository into another. With PMR, this usually implies pushing from a workspace clone on your local machine back to the workspace in the model repository, but could be into any other clone of the workspace. See a tour of the Git Beginner's Guide for some good introductory material.

Python Python is a programming language that lets you work more quickly and integrate your systems more effectively. See http://python.org for all the details.

Synchronize Used to pull the contents or changes from other repositories into a workspace via a URI. The remote repository have to be of the same DVCS protocol of the corresponding workspace..

Teaching instance An instance of *PMR2* that allows users to explore and test the functionality of PMR2 without making permanent *workspaces* or *exposures*. The teaching instance is regularly cleared of content and re-populated from *PMR*. It is also used for testing the latest PMR2 features prior to their deployment on PMR, so the user interface may not always exactly match that of PMR.

Workspace

Workspaces A *Git* 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 Git.

Note: The teaching instance of the repository is a mirror of the main repository site found at https://teaching.physiomeproject.org/, running the latest development version of *PMR2*. User accounts are periodicly synchronised from the main repository, but if you recently created an account on the main site you might need to also create a new account on the teaching instance.

Any changes you make to the contents of the teaching instance are not permanent, and will be overwritten with the contents of the main repository whenever the teaching instance is upgraded to a new release of PMR2. For this reason, you can feel free to experiment and make mistakes when pushing to the teaching instance. Please subscribe to the cellml-discussion mailing list to receive notifications of when the teaching instance will be refreshed.

See the section *Migrating content to the main repository* for instructions on how to migrate any content from the teaching instance to the main (permanent) Auckland Physiome Repository.

1.8.13 Using SED-ML to specify simulations

Hopefully PMR will support SED-ML simulations as part of the CellML views.

Todo:

- Update all 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.

Note: Anonymous feedback on any part of this module can be left at: http://goo.gl/forms/0OOhzkwnFo

Bibliography

[Bre84] Pieter Cornelis Breedveld. Physical systems theory terms of bond graphs. *PhD thesis, University of Twente, Faculty of Electrical Engineering*, 1984.

[GC14] Peter J Gawthrop and Edmund J Crampin. Energy-based analysis of biochemical cycles using bond graphs. In *Proc. R. Soc. A*, volume 470, 20140459. The Royal Society, 2014.

[GC16] Peter J Gawthrop and Edmund J Crampin. Modular bond-graph modelling and analysis of biomolecular systems. *IET systems biology*, 10(5):187–201, 2016.

[GCC15] Peter J Gawthrop, Joseph Cursons, and Edmund J Crampin. Hierarchical bond graph modelling of biochemical networks. In *Proc. R. Soc. A*, volume 471, 20150642. The Royal Society, 2015.

[GSK+15] Peter J Gawthrop, Ivo Siekmann, Tatiana Kameneva, Susmita Saha, Michael R Ibbotson, and Edmund J Crampin. The energetic cost of the action potential: bond graph modelling of electrochemical energy transduction in excitable membranes. *arXiv preprint arXiv:1512.00956*, 2015.

[OPK71] George Oster, Alan Perelson, and Aharon Katchalsky. Network thermodynamics. *Nature*, 234(5329):393–399, 1971.

[Pay61] Henry M Paynter. *Analysis and design of engineering systems*. MIT press, 1961.

[APJ15] Garny A. and Hunter P.J. Opencor: a modular and interoperable approach to computational biology. *Frontiers in Physiology*, 2015.

[AYY] Non A. Www.biomodels.org <http://www.biomodels.org>. YYYY.

[AAF52] Hodgkin AL and Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[CPJ09] Christie R. Nielsen P.M.F. Blackett S. Bradley C. and Hunter P.J. Fieldml: concepts and implementation. *Philosophical Transactions of the Royal Society (London)*, A367(1895):1869–1884, 2009.

[CMEJ08] Hunter PJ Cooling M and Crampin EJ. Modeling biological modularity with cellml. *IET Systems Biology*, 2:73–79, 2008.

[CJ15] Waltemath D. Cooper J, Vik JO. A call for virtual experiments: accelerating the scientific process. *Progress in Biophysics and Molecular Biolog*, 117:99–106, 2015.

[D62] Noble D. A modification of the hodgkin-huxley equations applicable to purkinje fibre action and pacemaker potentials. *Journal of Physiology*, 160:317–352, 1962.

[DPPJ03] Cuellar A.A. Lloyd C.M. Nielsen P.F. Halstead M.D.B. Bullivant D.P. Nickerson D.P. and Hunter P.J. An overview of cellml 1.1, a biological model description language. *SIMULATION: Transactions of the Society for Modeling and Simulation*, 79(12):740–747, 2003.

[ea13] Hunter P.J. et al. A vision and strategy for the virtual physiological human: 2012 update. *Interface Focus*, 2013.

[eal11] Yu T. et al. The physiome model repository 2. *Bioinformatics*, 27:743–744, 2011.

[J97] Wigglesworth J. Energy and life. *Taylor & Francis Ltd*, 1997.

[JH02] Thompson JMT and Stewart HB. Nonlinear dynamics and chaos. *Wiley*, 2002.

[LCPF08] Hunter PJ Lloyd CM, Lawson JR and Nielsen PF. The cellml model repository. *Bioinformatics*, 24:2122–2123, 2008.

[P13] Britten R.D. Christie G.R. Little C. Miller A.K. Bradley C. Wu A. Yu T. Hunter P.J. Nielsen P. Fieldml, a proposed open standard for the physiome project for mathematical model representation. *Med. Biol. Eng. Comput.*, 51(11):1191–1207, 2013.

[PJ04] Hunter P.J. The iups physiome project: a framework for computational physiology. *Progress in Biophysics and Molecular Biology*, 85:551–569, 2004.

[VarYY] Various. See www.cellml.org/about/publications for a more extensive list of publications on cellml and opencor. *Various*, YYYY.

Index