DTP Computational Physiology Documentation

Release 2016.03

University of Auckland

March 18, 2016

Contents

Note: This is release 2016.03 of the DTP Computational Physiology module documentation. 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

Welcome to the Computational Physiology module of the Doctoral Training Programme from the MedTech CoRE. In this series of tutorials, you will be exposed to 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, geometric model building, simulation, and visualisation. You will 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. Finally, the course ends with an introduction to some of the key tools that are used in this field locally.

CHAPTER 1

Clinical Workflows

In Fig. 1.1 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); and 5) the preceeding steps are wrapped into a customised user interface which can be provided to the clincians for them to execute these steps in the clinc.

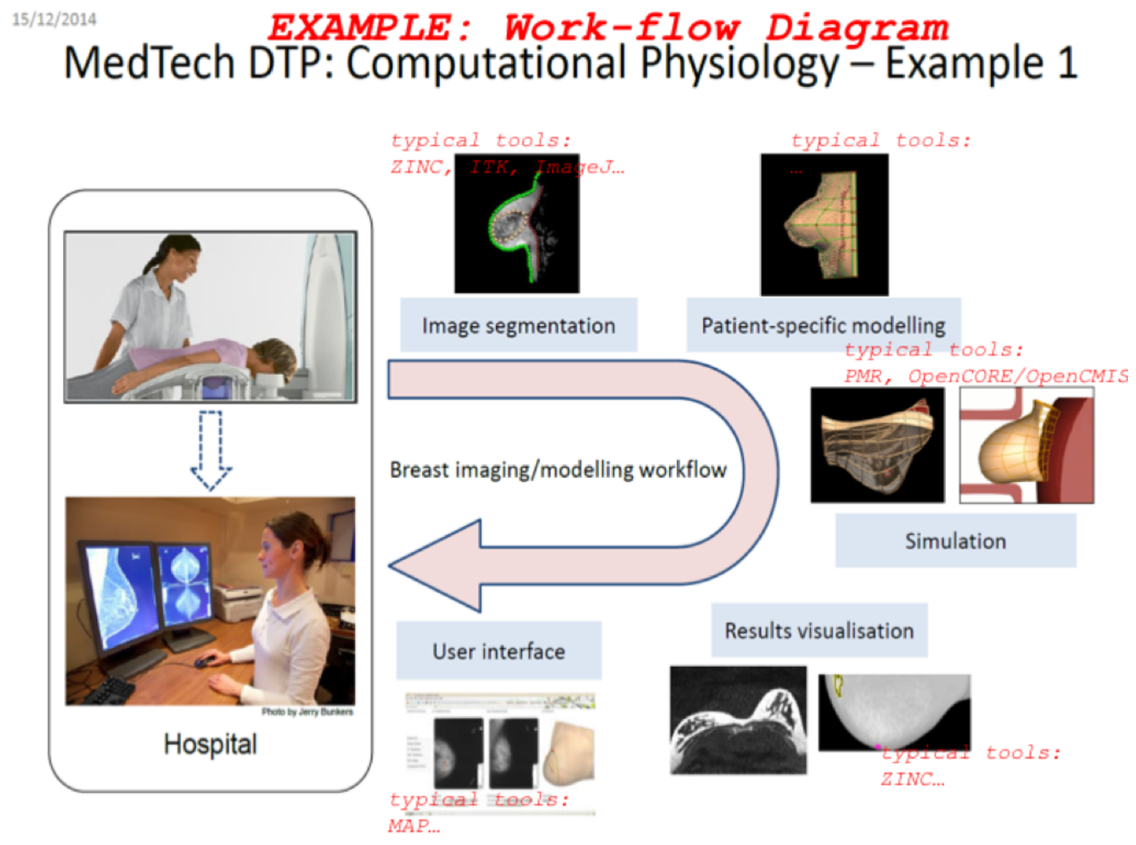


Fig. 1.1: An example of a clinical workflow, showing the steps in the process which form the basis for the Computational Physiology module.

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. 1.1. 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.

1.1 Computational Physiology - Segmentation

Contents:

1.1.1 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 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.

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 the Digital Imaging and Communications in Medicine (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. In the 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

Task one 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 Segmentation Task 1' workflow, with the workflow loaded you should see something like Fig. 1.2.

In this workflow there are three steps, the first step is the image source step from which we will load the DICOM images. The second step is the DICOM header query step which will be used to query tags from the DICOM image. The third stick is a simple Python dict serialisation step so that we may store queries for future reference.

When we execute this workflow we are presented with Fig. 1.3 an interface for querying DICOM headers.

In Fig. 1.3 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 below from which we can select one to perform a query 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

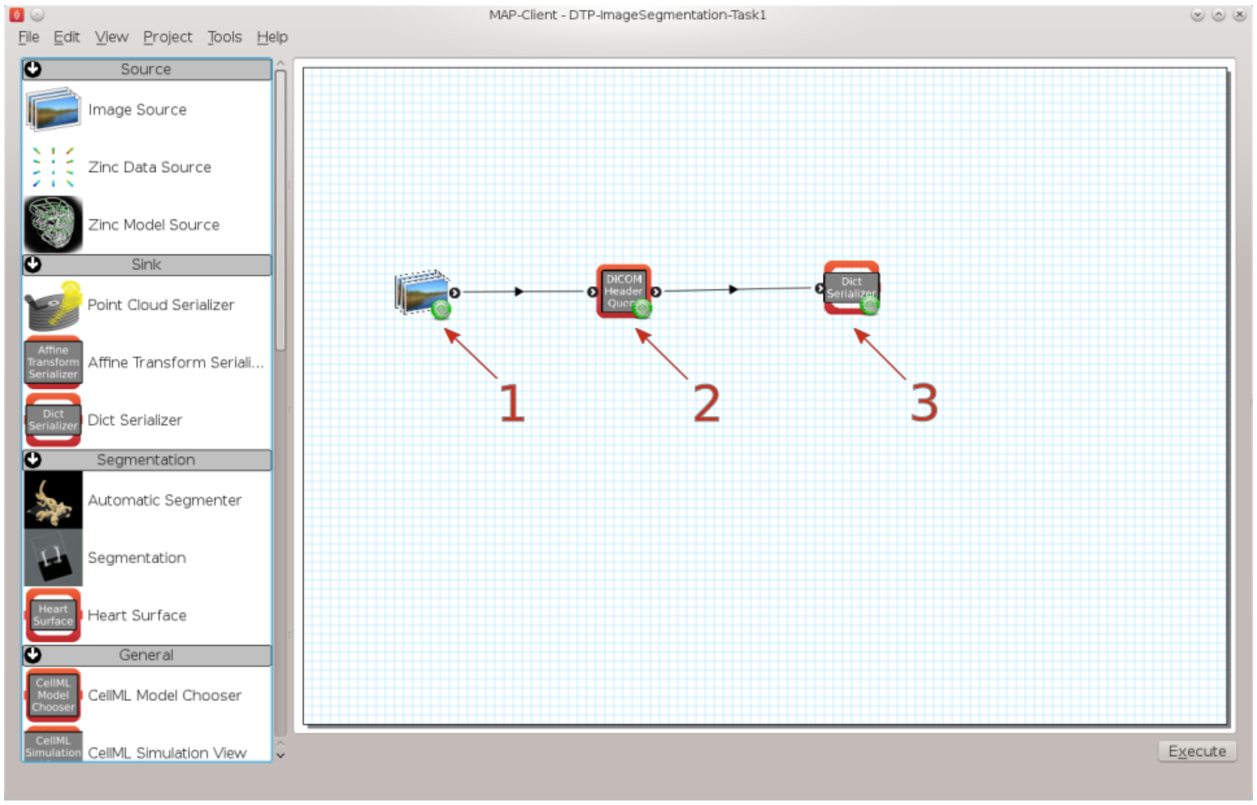
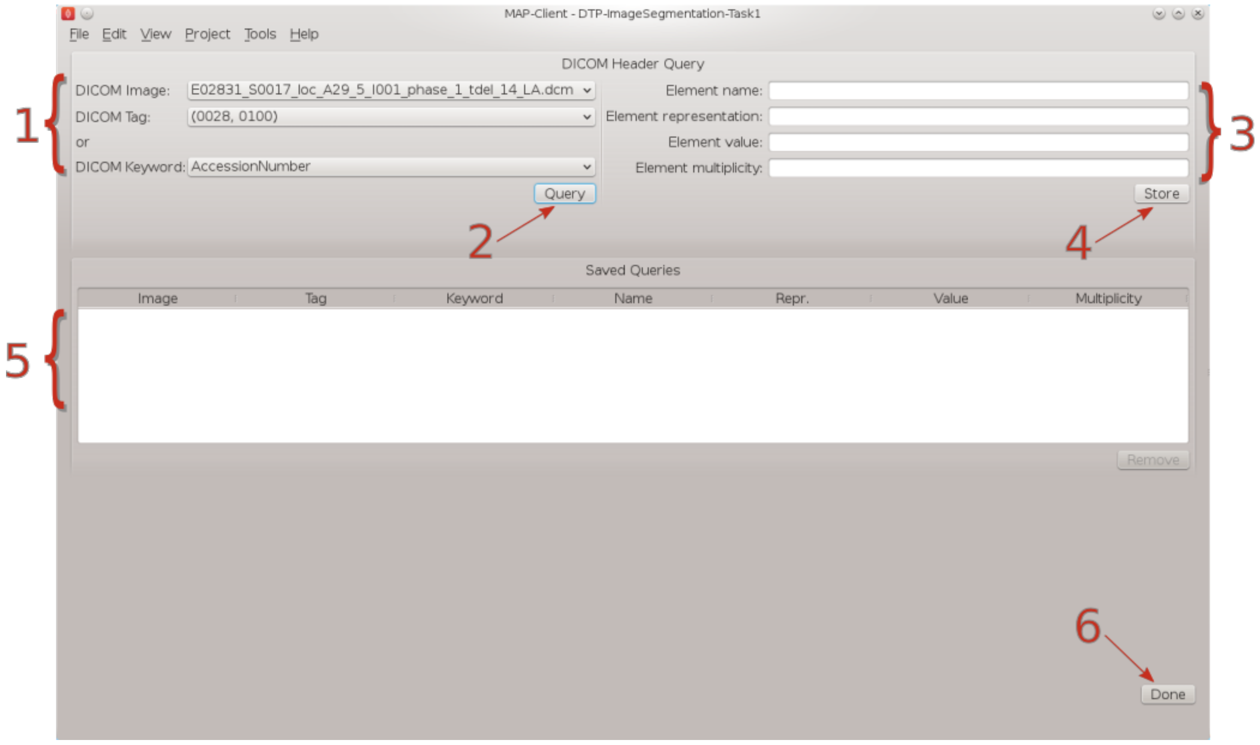


Fig. 1.2: Task 1 workflow



DICOM Image: E02831_S0017_loc_A29_5_I001_phase_1_tdel_14_LA.dcm

DICOM Tag: (0028, 0100)

or

DICOM Keyword: AccessionNumber

Element name:

Element representation:

Element value:

Element multiplicity:

Fig. 1.3: DICOM header query interface

have four text boxes 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 edit 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 is 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 though 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?
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 Segmentation Task 2' workflow. In this workflow we see five steps there are two image source steps, the 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. 1.4.

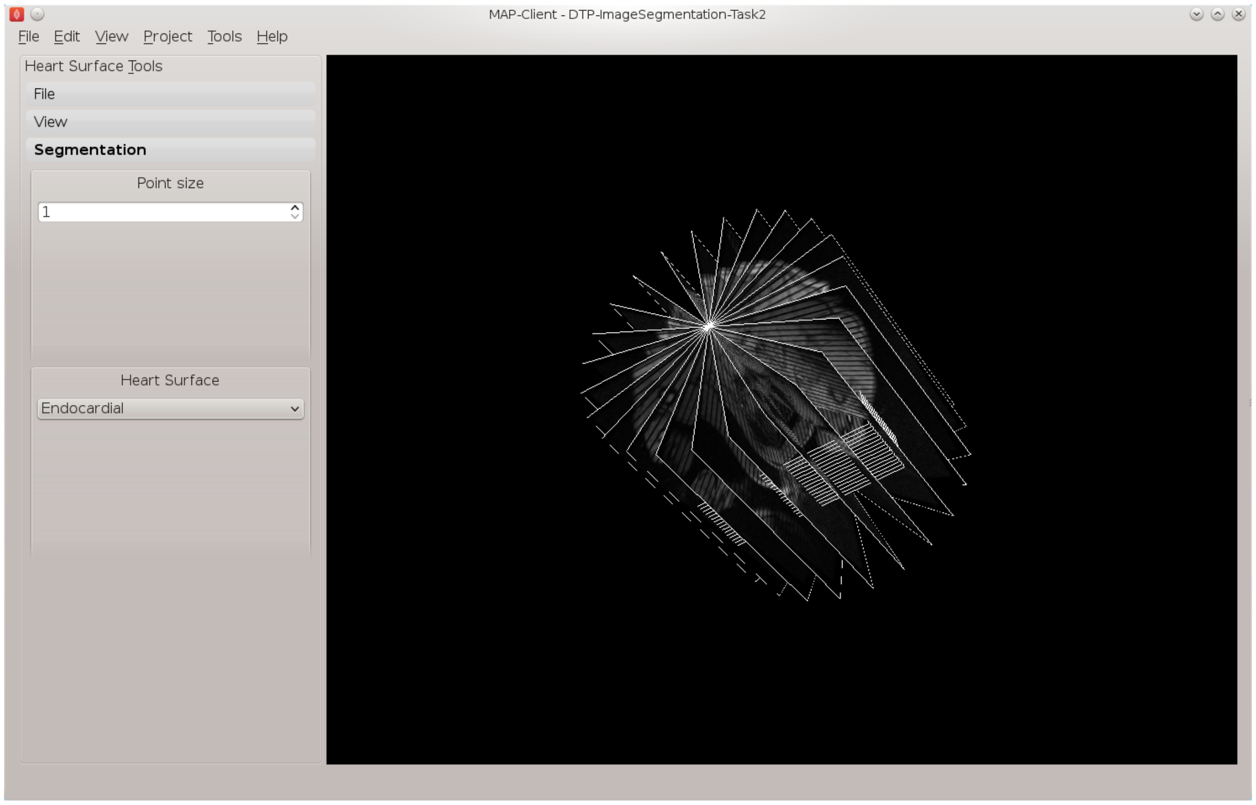
In Fig. 1.4 we can see on the left a toolbox that allows us to change the state of this segmentation tool, on the right hand side we can see a three-dimensional view of the two sets of DICOM images. To create this view we have used the

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

information from the DICOM header. 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 for box we can load and save our progress. The done button is also in the file toolbox for when we are finished segmenting.

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. 1.5.


MAP-Client - DTP-ImageSegmentation-Task2
File Edit View Project Tools Help
Heart Surface Tools
File
View
Segmentation

Point size: 1

Heart Surface: Endocardial

Fig. 1.4: Heart segmentation interface initial state

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 are set up to segment. In this view the 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. 1.6.

Continue segmenting the left ventricle using the long axis images to check for consistency. The end result should look like Fig. 1.7.

Using the save button from the file toolbox save your progress and click the done 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.

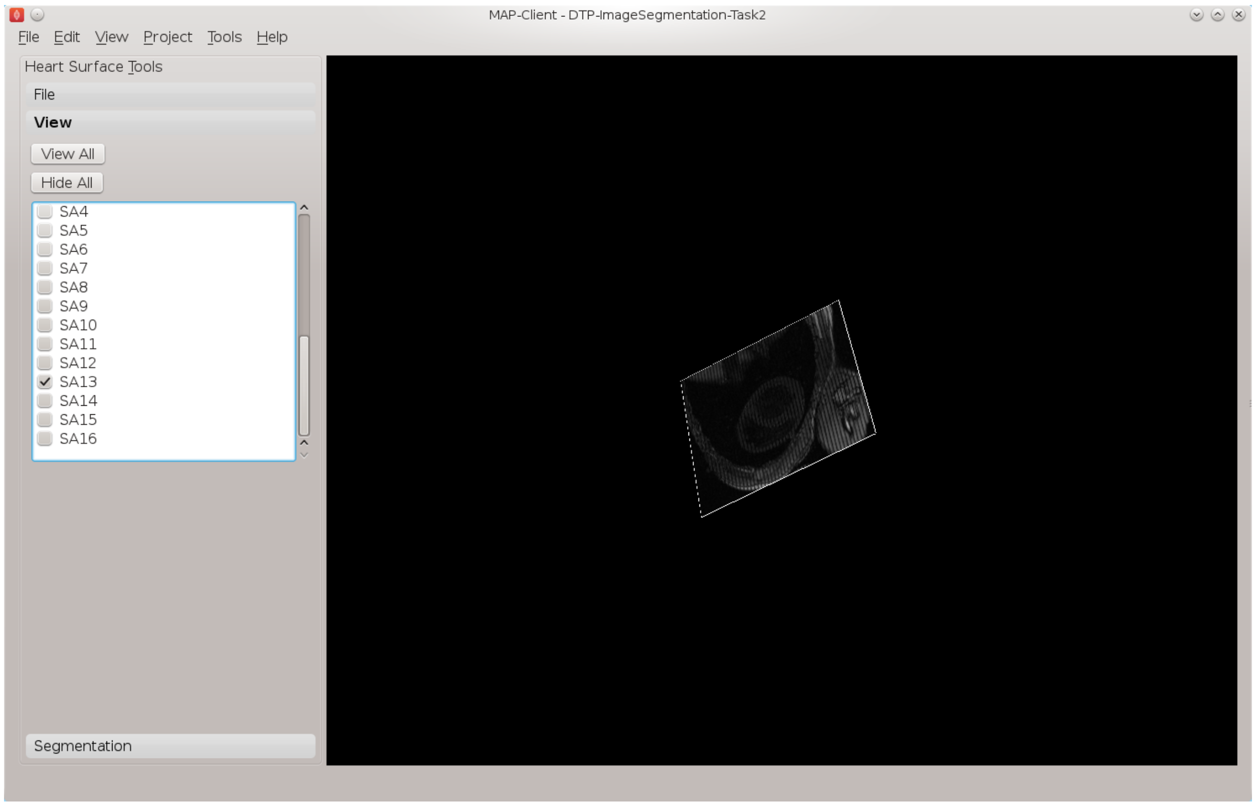


Fig. 1.5: View of the thirteenth short axis image plane

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 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. 1.8 shows the apex point.

Make only the 13th image plane visible, on this image plane place the landmarks for the base point and the RV point. 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. 1.9 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. 1.10 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 then click the done button to complete this workflow.

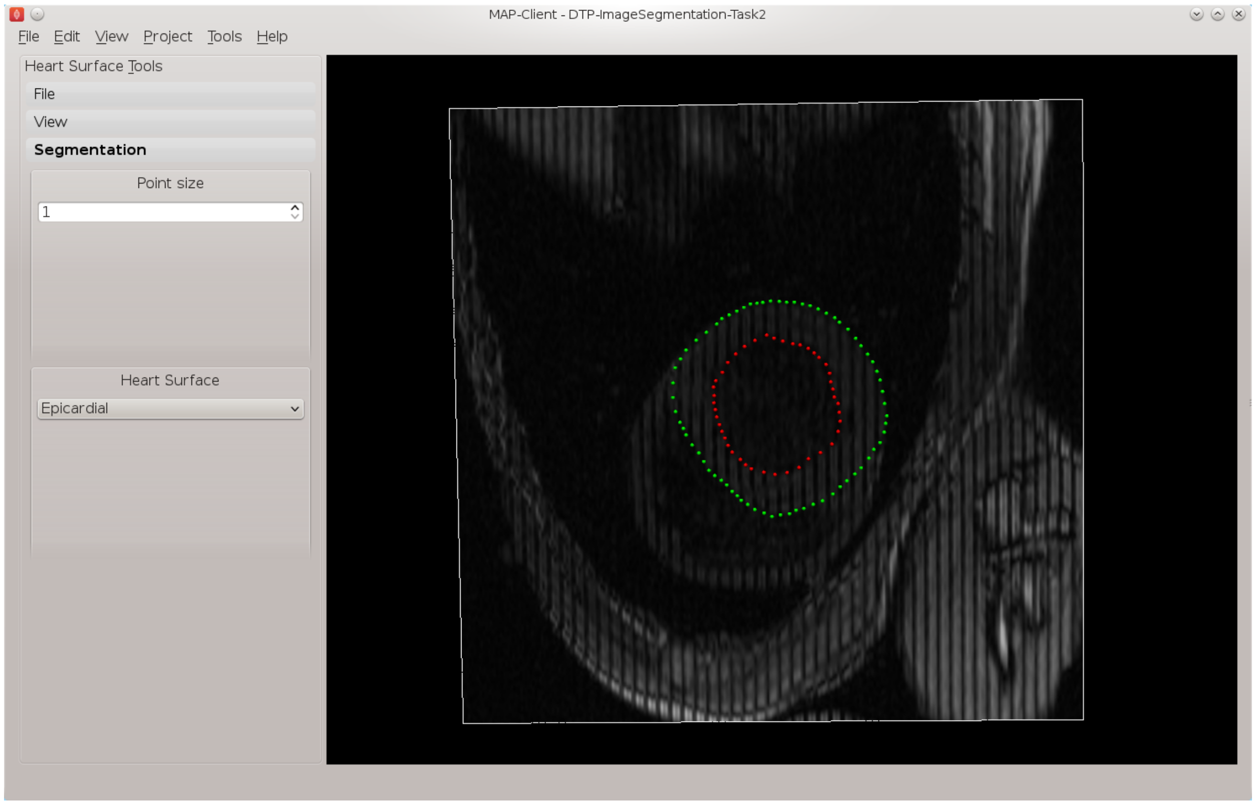

MAP-Client - DTP-ImageSegmentation-Task2
File Edit View Project Tools Help
Heart Surface Tools
File
View
Segmentation
Point size
1
Heart Surface
Epicardial

Fig. 1.6: View of the segmented thirteenth short axis image plane

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.

1.1.2 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 a sphere around the scene and when you click that you're placing a point on the sphere so that when you drag the mouse it as if you are pulling the sphere around buy a piece of string attached to that point. This means that when the whole scene is visible and we click on the edge of the sphere containing the scene we should be able to achieve a pure roll movement of the scene.

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

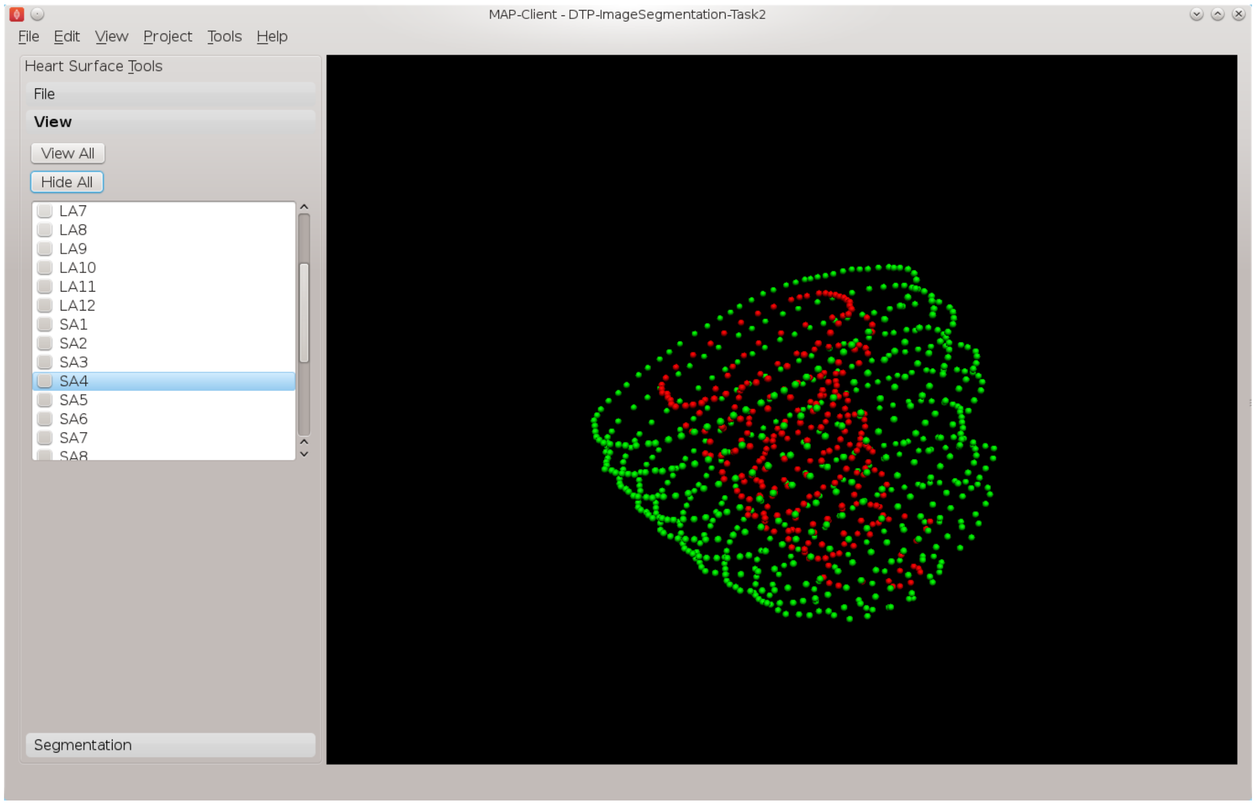


Fig. 1.7: View of the segmented left ventricle

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. 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. 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 Computational Physiology - Model Construction

Contents:

1.2.1 Model Construction

Model construction consists of:

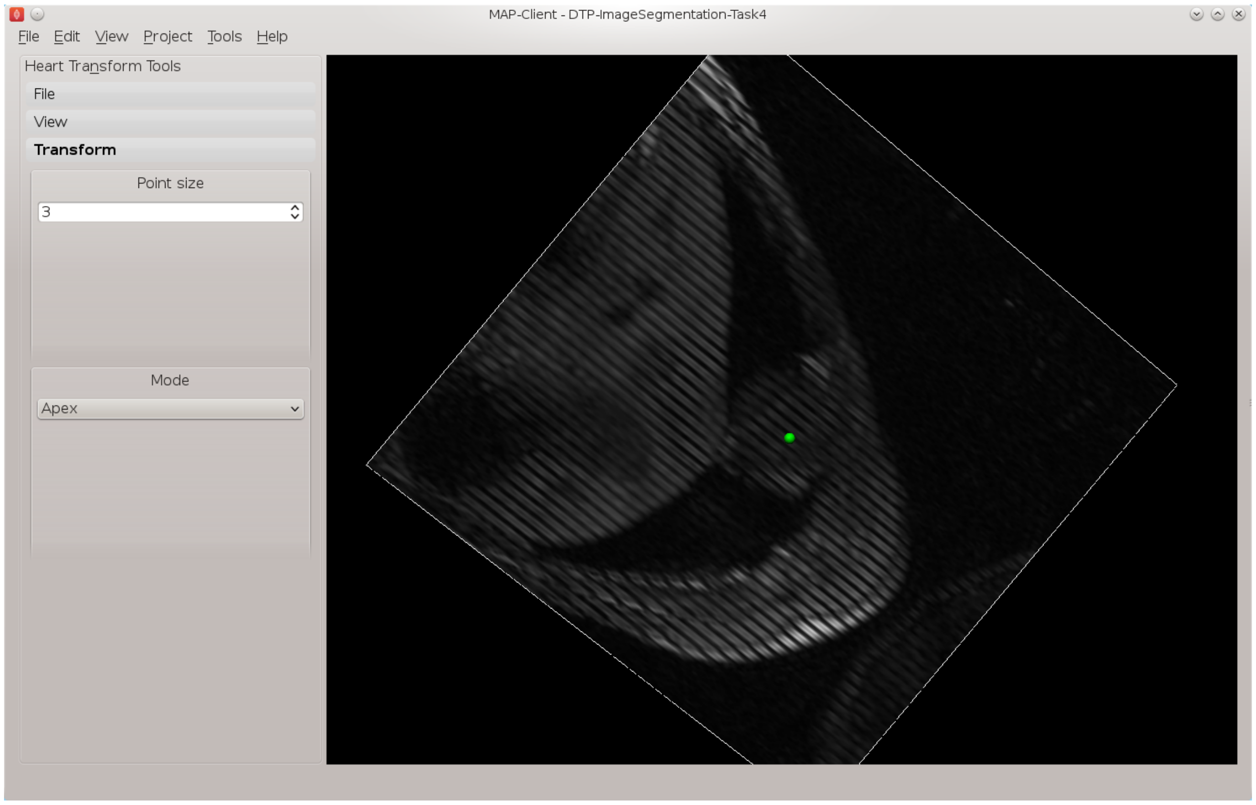

MAP-Client - DTP-ImageSegmentation-Task4
File Edit View Project Tools Help
Heart Transform Tools
File
View
Transform
Point size
3
Mode
Apex

Fig. 1.8: Apex point position in the left ventricle

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.

This tutorial currently concentrates on geometric fitting, but gives overviews of the other topics. It uses the breast surface as the model for fitting, which is appropriate as 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 the 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, then relying on fitting to the geometry. The models used in the geometric fitting steps below were all generated from a plate topology and the tutorial involves fitting their geometry.

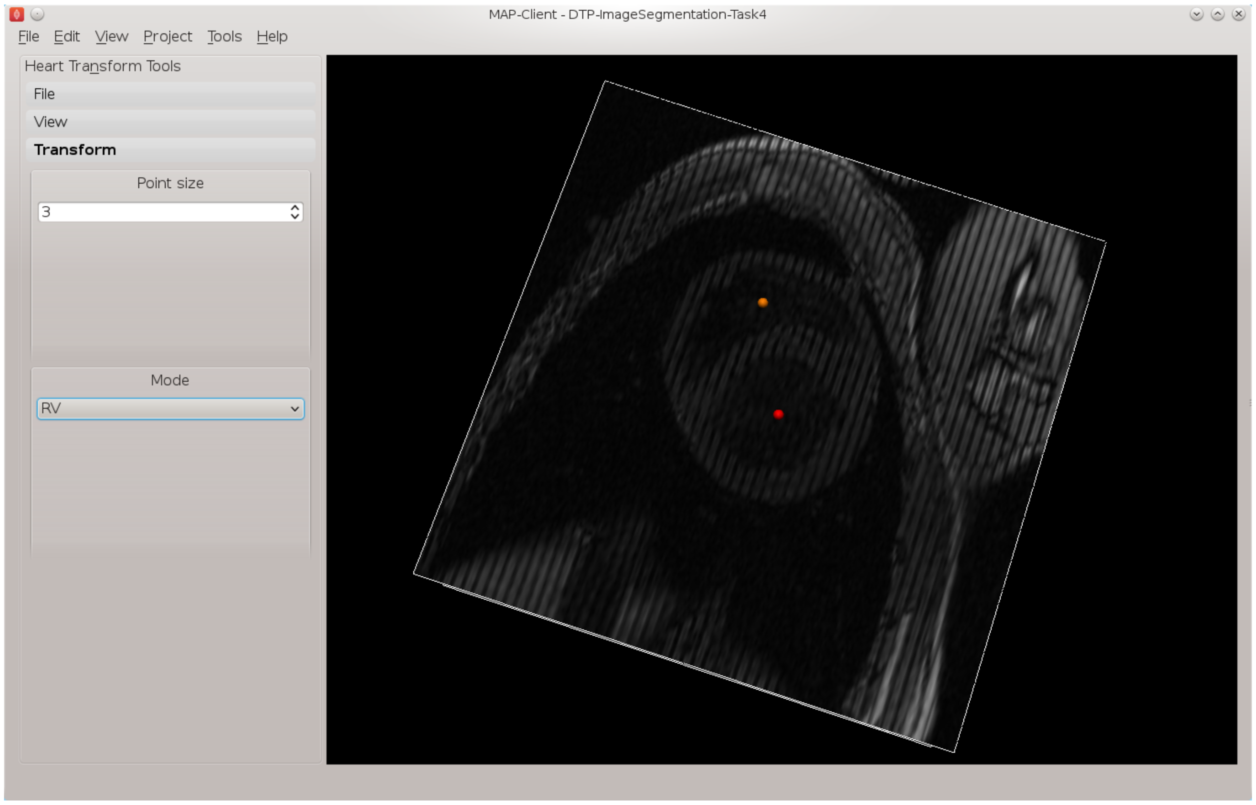

MAP-Client - DTP-ImageSegmentation-Task4
File Edit View Project Tools Help
Heart Transform Tools
File
View
Transform
Point size
3
Mode
RV

Fig. 1.9: Placement of the base point and RV point

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.

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 breast fitting 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.

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. 1.11 shows a time sequence of constructing a heart model from this example which you will be able to view interactively.

This example opens up a SimpleViz viewer for a model that shows stages in constructing a heart model. At the left of

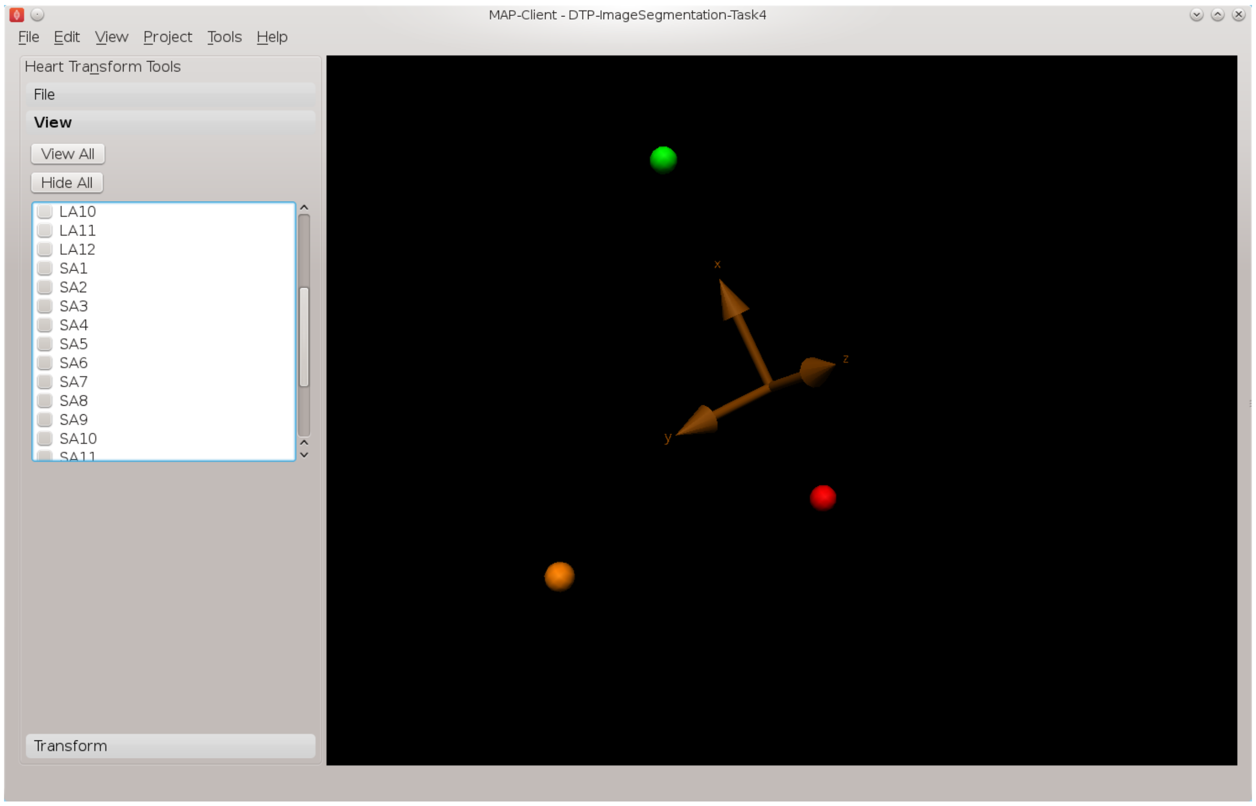


Fig. 1.10: Axes glyph representing the heart coordinate system

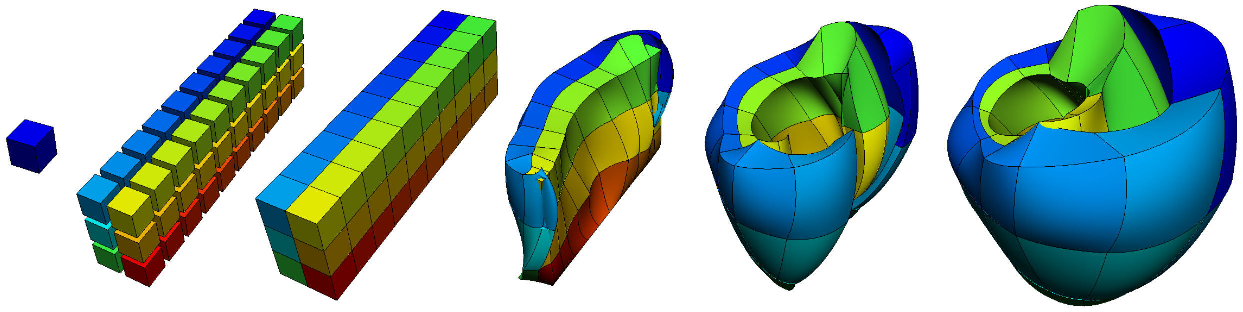


Fig. 1.11: Heart mesh construction stages: a single template element; multiple disconnected elements; elements merged into a connected mesh; mesh geometry gradually warped into the shape of the heart, closing up the sides and apex.

the window is a toolbox; switch to the 'time' page of the tool box and drag the time slider between 0 and 1. Rotate, pan and zoom into the view with the mouse as per the description for the Smoothfit Tool, below.

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.

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 workflow in the MAP client is shown in Fig. 1.12, and requires only the input files to be specified (and workflow step identifiers to be named):

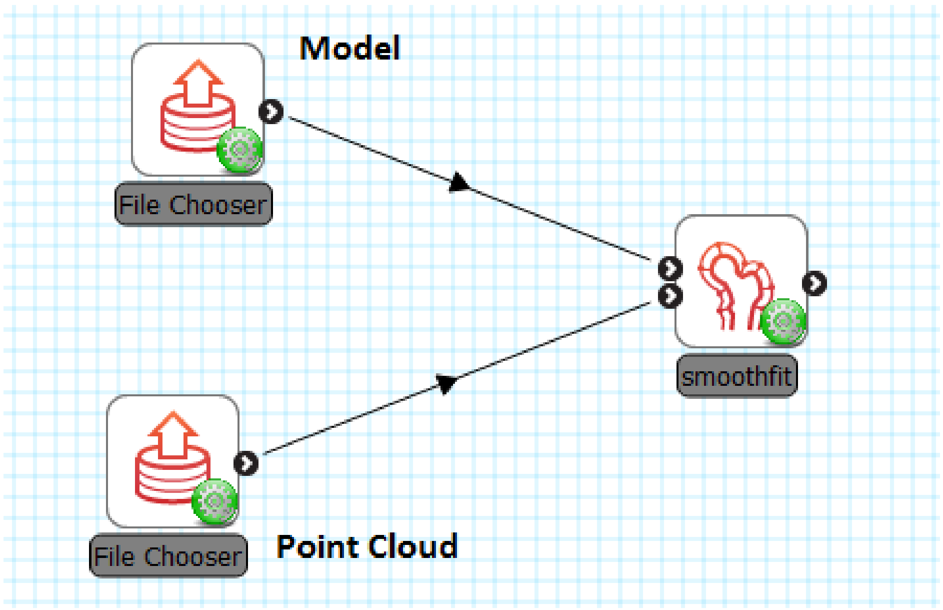
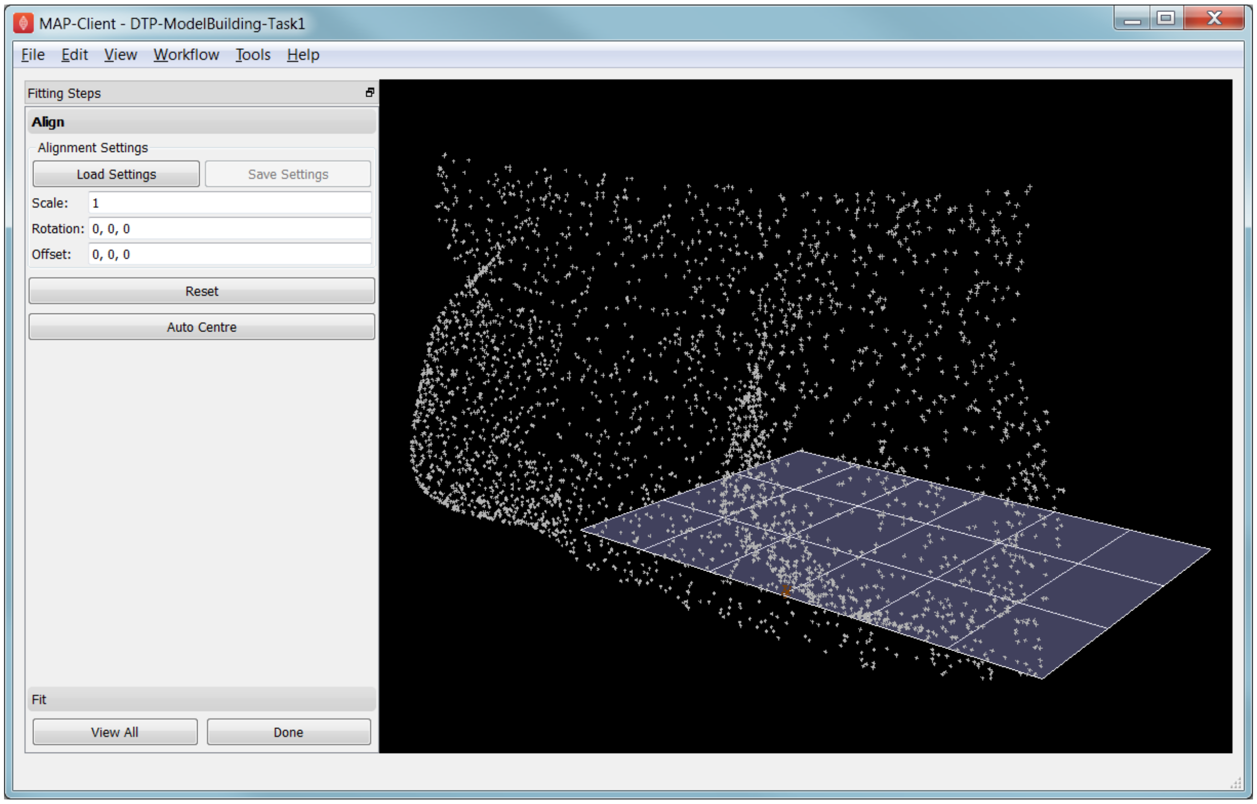


Fig. 1.12: Geometric fitting workflow in the MAP client framework.

When the 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. 1.13 shows the interface in its pre-fitting *Align* state.



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

Fig. 1.13: Interface for aligning the model with the data point cloud.

In any of the views 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 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

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 tool bar, 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 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.)

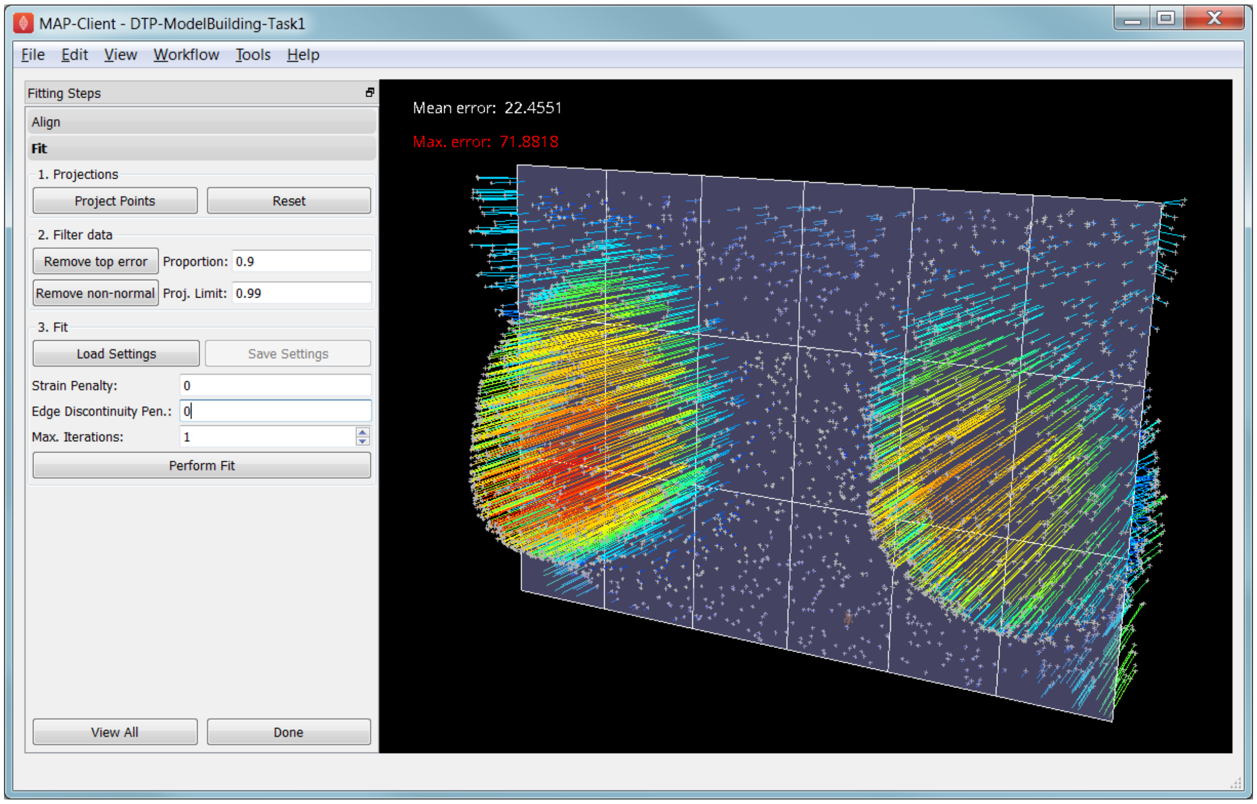
Often the shape of the model and point cloud make it pretty clear where to align to. Smoothfit uses manual alignment, but other tools may make it automatic (based on shape analysis) or semi-automatic (e.g. by identifying 3 or more points on the data cloud as being key points on the model, and automatically 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 tool bar 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. 1.14. 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.



Proportion: 0.9
Proj. Limit: 0.99
Strain Penalty: 0
Edge Discontinuity Pen.: 0
Max. Iterations: 1
Mean error: 22.4551
Max. error: 71.8818

Fig. 1.14: 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. 1.14 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. 1.15 shows what the view looks like after 2 iterations of fitting with a moderate strain penalty to keep the solution smooth.

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* 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.

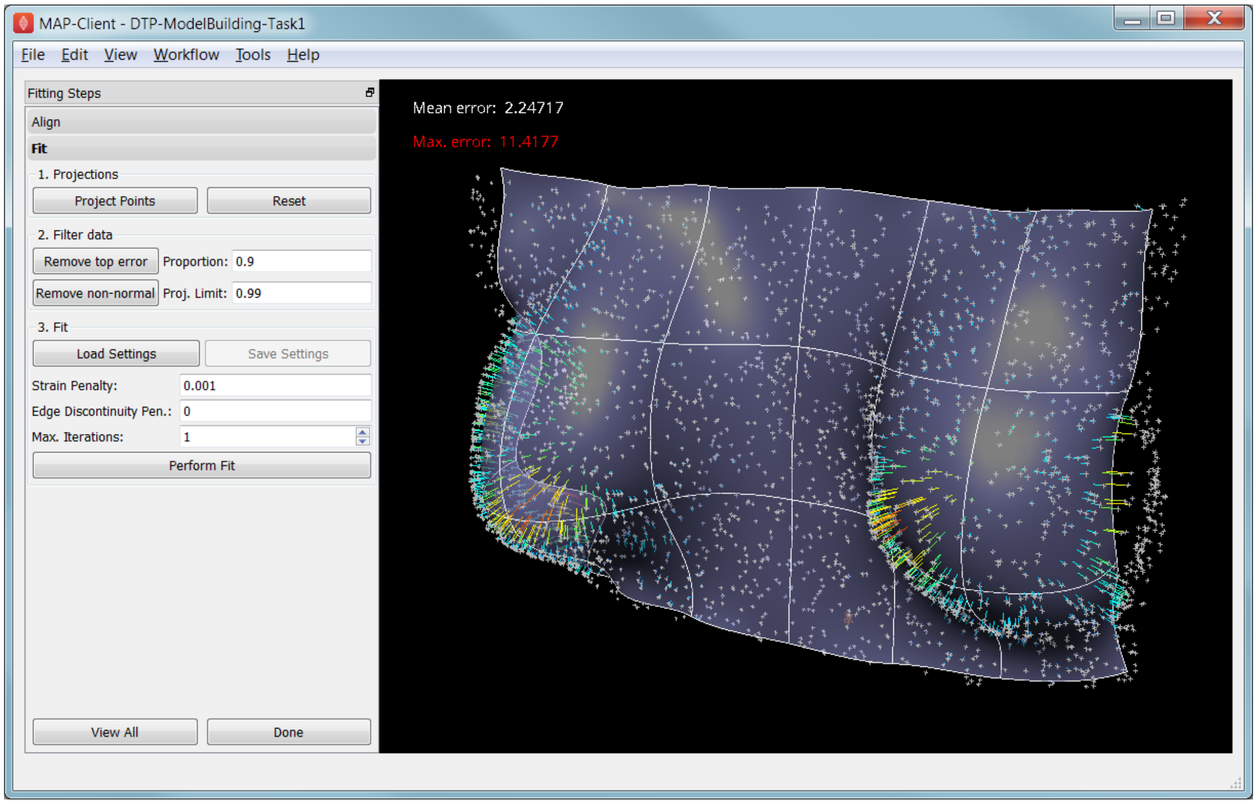
Beware that projections are not recalculated during the fitting: you must manually click on *Project Points* to do this, and you will probably want to filter some more points before re-fitting.

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

The penalty values allow you to smooth the fit by penalising particular deformations. The strain penalty limits excessive strain in the model so where there is absent or noisy data, solutions which minimise the deformation from the initial aligned state are favoured. The edge discontinuity penalty is only useful for non-C_1-continuous coordinate fields such as the final linear mesh example. Penalties always increase 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. Penalty values should be adjusted in orders of magnitude until a likeable result is obtained, then fine-tuned. 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.

Note that Smoothfit does not yet offer a curvature penalty which is one of the most powerful tools for dealing with noisy or sparse data. Using the strain penalty is the next best thing but isn't as good at dealing with excessive waviness in the solution, particularly since higher values capable of helping the waviness may considerably reduce the accuracy of the fit. This shortcoming will hopefully be rectified in a later version.

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.



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. 1.15: Display after gross fitting the breast model.

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

Task 1: Coarse plate model fitted to breast data

Open the *DTP-ModelBuilding-Task1* 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' to load a good alignment.
2. Switch to the Fit page, project points and click on 'Remove non-normal'.
3. Click 'Load' 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. 1.15.)
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 2: Fine plate model fitted to breast data

Open the *DTP-ModelBuilding-Task2* 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.

Task 3: Coarse breast model fitted to breast data

Open the *DTP-ModelBuilding-Task3* workflow and execute it. In this example the initial model is more breast-like in shape so when well-aligned the amount of fitting needed is reduced. You should be able to fit it with the lower strain penalty of 0.0001 directly in 2 iterations. Since the initial model is already so close, deformations will not be as great to get a close fit.

Task 4: Fine breast model fitted to noisy data

Open the *DTP-ModelBuilding-Task4* workflow and execute it. This example uses a fine model with a breast-like shape, 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.

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 5: Fine breast model fitted to sparse, noisy data

Open the *DTP-ModelBuilding-Task5* workflow and execute it. This example uses only 10% of the data points from the previous tasks and adds +/- 3mm error to each point.

Try fitting as before. The effect of sparse data with random noise makes it even harder to obtain a close fit. Using the successful regime from Task 1 gives a result that is quite wavy after the final less-stiff fitting. Curvature penalties would greatly assist such models.

Task 6: Bilinear model fitted to point cloud

Open the *DTP-ModelBuilding-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.

1.2.2 DTP-ModelConstruction

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

1.3 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 physioloyg. We will demonstrate these skills across a range of spatial scales and numerical methods.

Contents:

1.3.1 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.

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 follow 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. 1.16. The workflow is pre-configured so there is no configuration required.

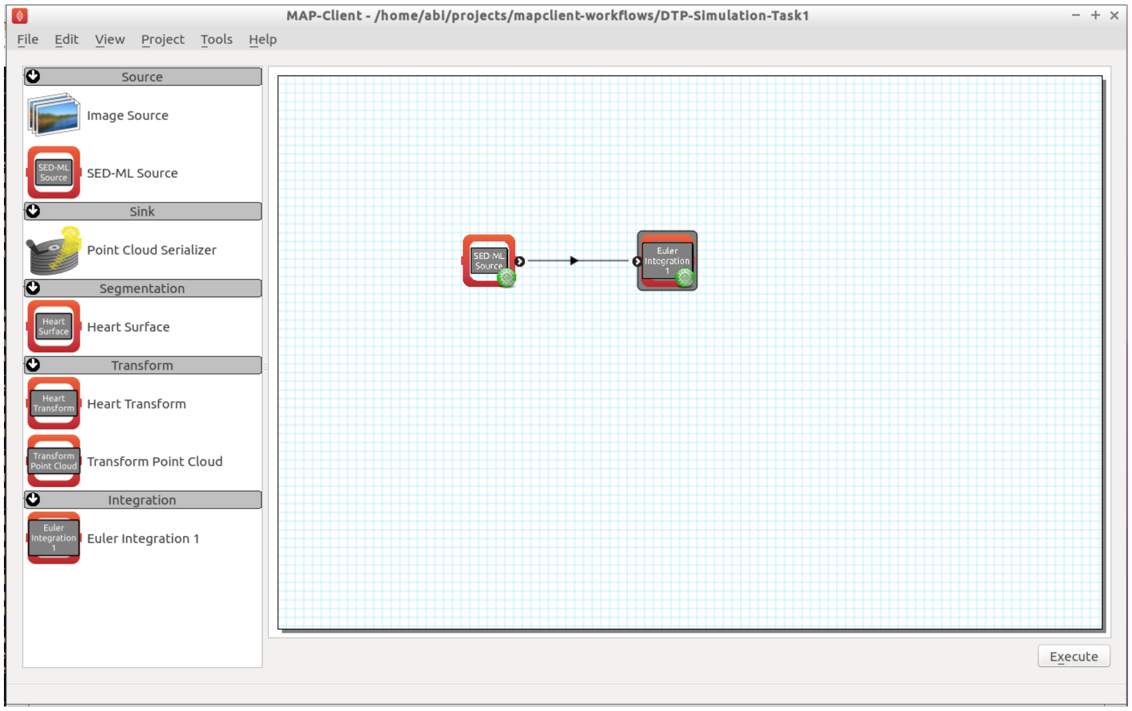


Fig. 1.16: The first Euler example as loaded.

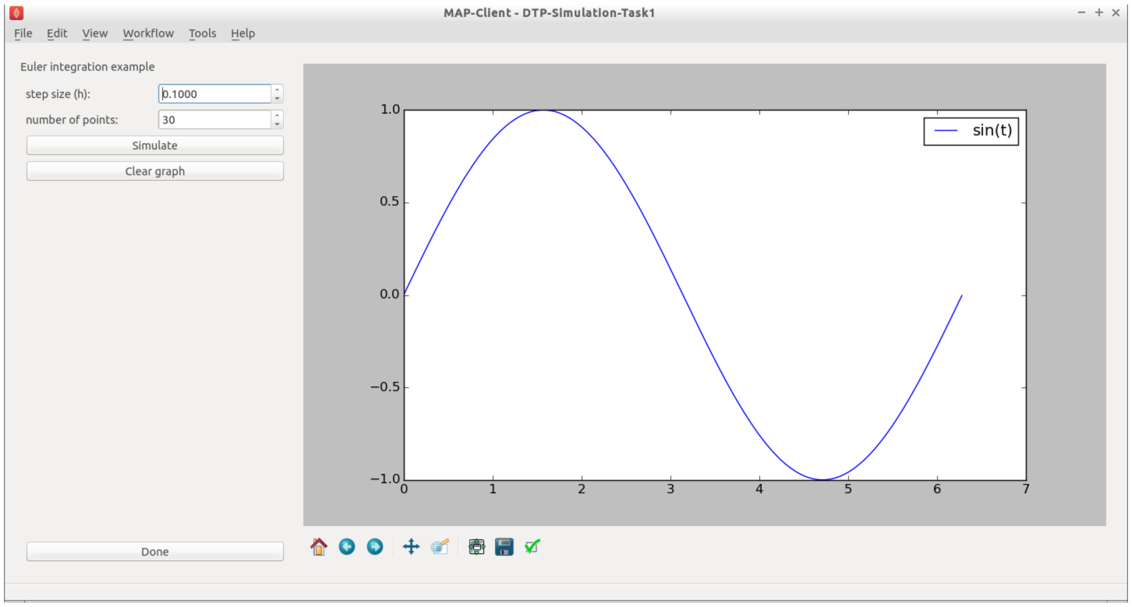


Fig. 1.17: The cool Euler integrator interface. In this simple interface, you will see the standard sine function, $sin(t)$, plotted in the right hand panel. The toolbar under the plot is self-explanatory, but provides access to some nifty features. At the top of the left hand panel you will see the 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.

3. Click the *Execute* button and you should get a widget displayed as per Fig. 1.17.
4. As described in Fig. 1.17, multiple simulations can be performed with varying values for the step size, h. Shown in Fig. 1.18 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.

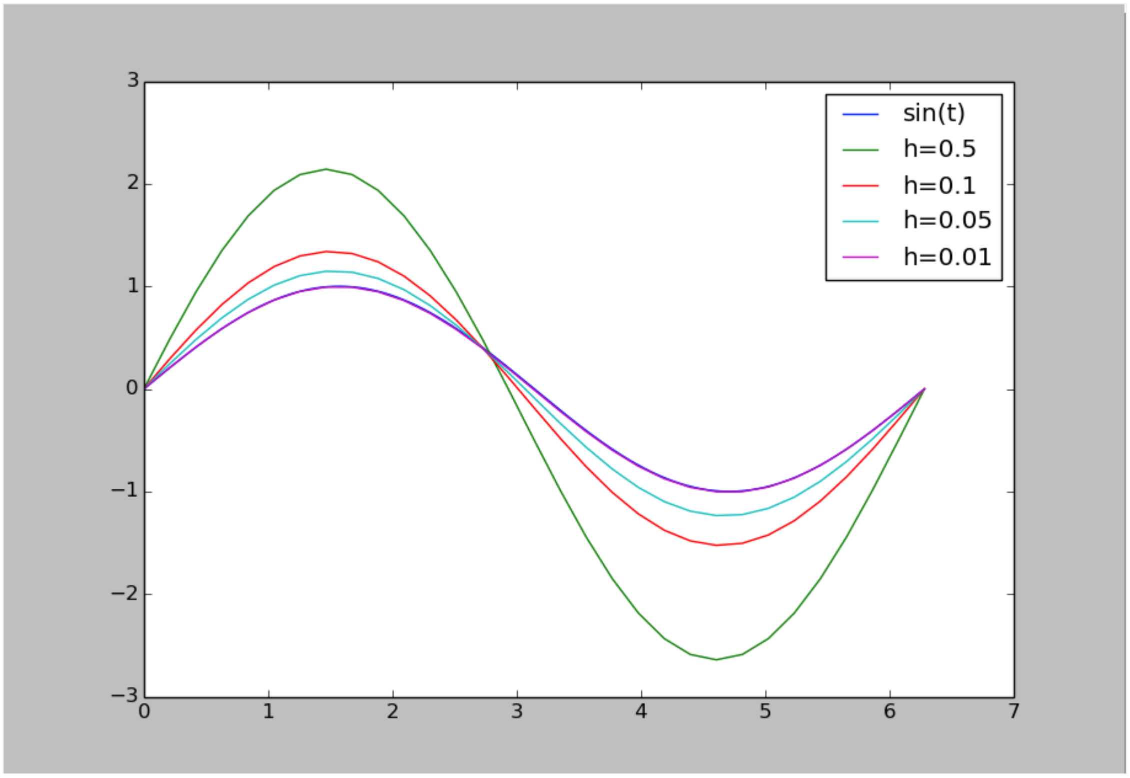


Fig. 1.18: Simulation results demonstrating the effect of step size, h, on the accuracy of Euler's method in approximating the solution of (1.3).

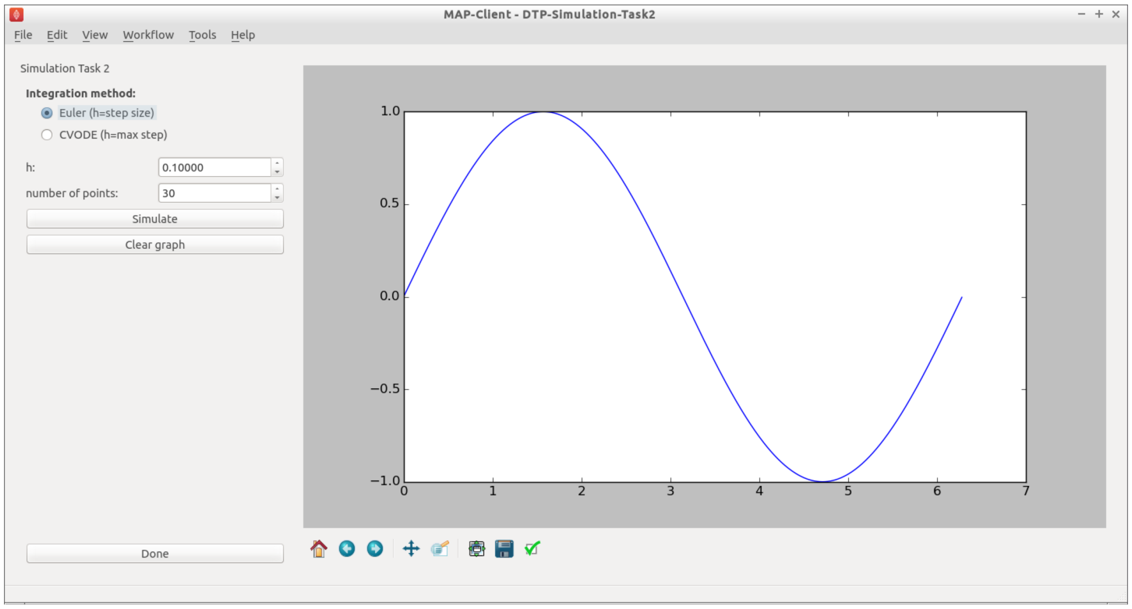
5. Now have a play with combining different values for the step size and the number of points to be obtained. See if you can answer the following.
   (a) How small should h be to accurately simulate a sine wave?
   (b) What do you think would happen beyond a single cycle?
   (c) Given $h = 1$, do you 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. 1.16). 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. 1.19.

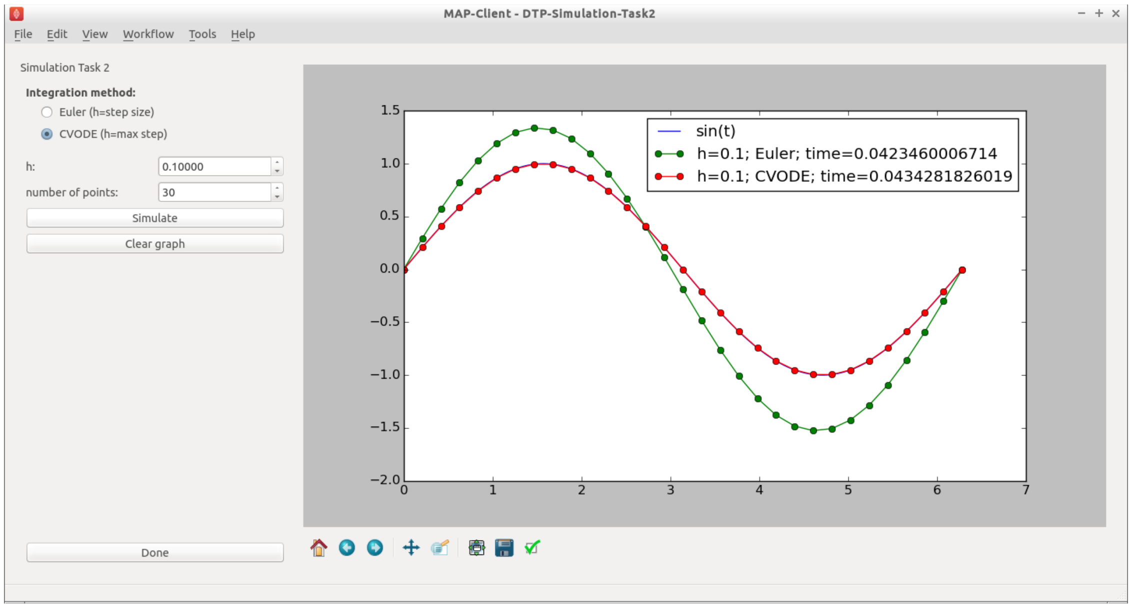


Integration method:
[✓ Euler (h=step size)]
CVODE (h=max step)
h: 0.10000
number of points: 30

Fig. 1.19: The user interface in this task is similar to that described in Fig. 1.17, 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. 1.20.
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.
   (a) 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?
   (b) How small does h need to be to get the same solution with Euler?
   (c) Are either of those a useful solution?
   (d) What is the minimum number of points required to capture an accurate sine wave?
   (e) 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 accumulation 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.


MAP-Client - DTP-Simulation-Task2
File Edit View Workflow Tools Help
Simulate
Clear graph
Done

Simulation Task 2

Integration method:

Euler (h=step size) [✓ CVODE (h=max step)]

h: 0.10000

number of points: 30

sin(t)

h=0.1; Euler; time=0.0423460006714

h=0.1; CVODE; time=0.0434281826019

Fig. 1.20: Simulation results showing the comparison between the Euler and CVODE integrators.

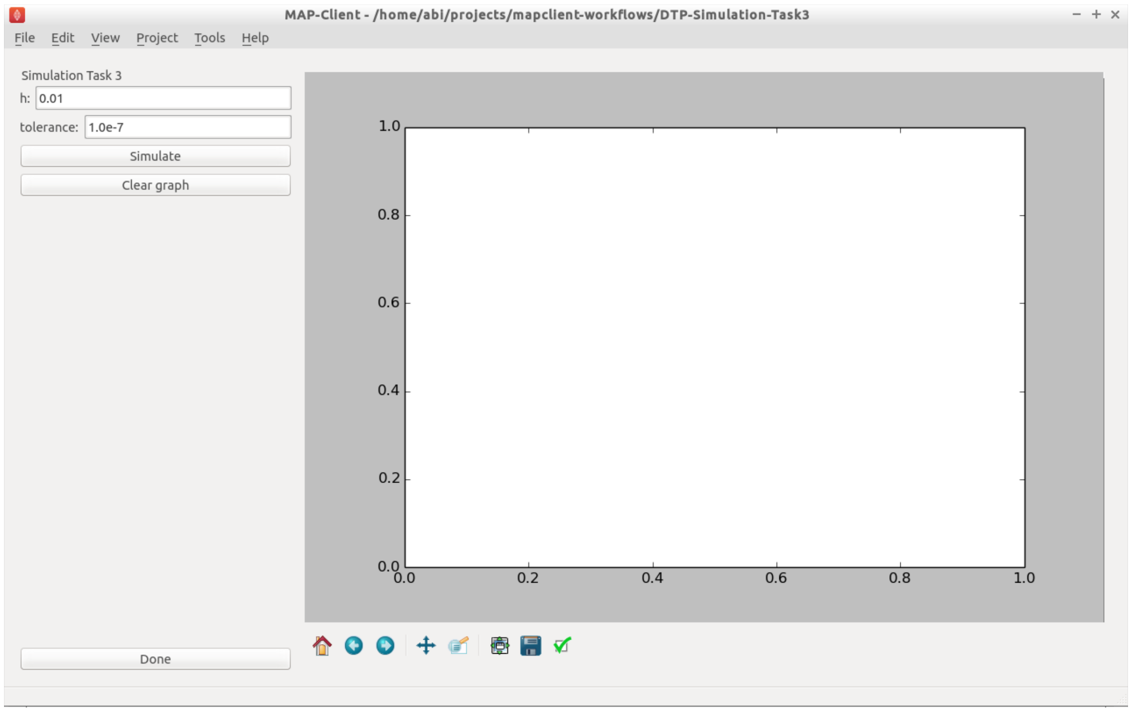
We use the recent biophysically based mathematical model of unitary potential activity in interstitial cells of Cajal. The interstitial cells of Cajal (ICC) are the pacemaker cells of the gastrointestinal tract and provide the electrical stimulus required to activate the contraction of smooth muscle 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. 1.16). 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. 1.21.
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. 1.22.
5. After exploring the effects of the integrator parameters and the simulated model behaviour, see if you can answer the following questions.
   (a) With $h = 0.0$, how loose can the tolerance be and still get an accurate solution?
   (b) How tight can you make the tolerance before the computational cost outweighs any improvement in solution accuracy?
   (c) Is there any value of h that will give an accurate solution for a tolerance of 0.01?

1.3.2 Biological Complexity

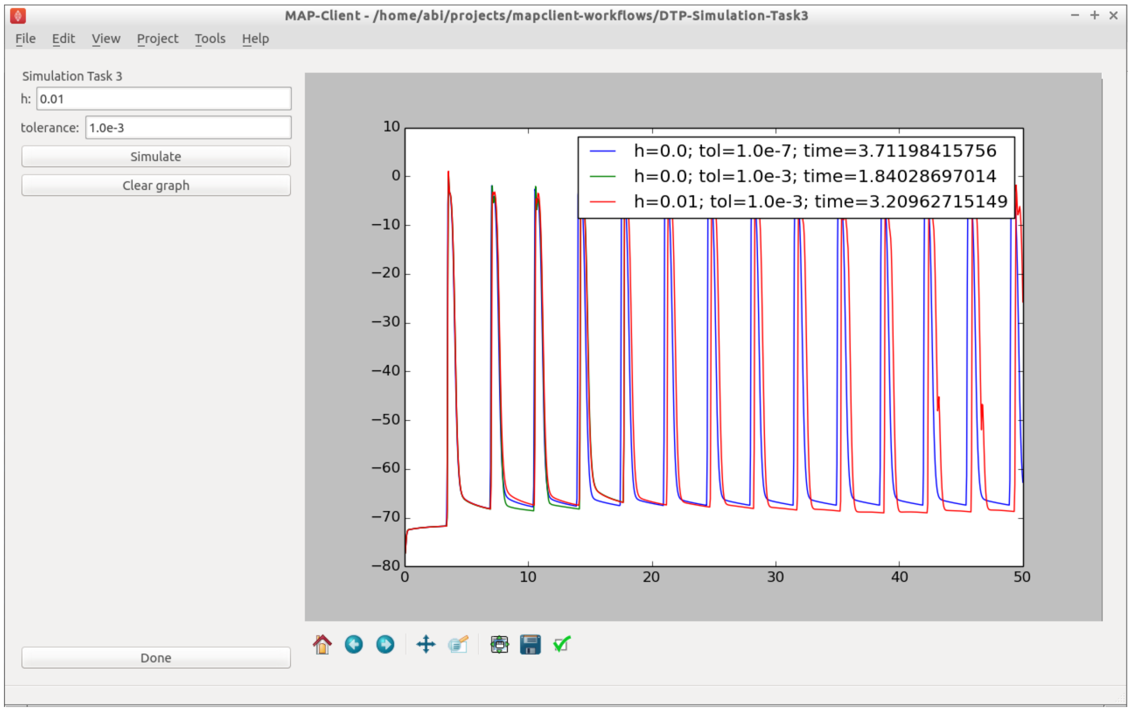
Mathematical models are a useful tool for investigating biological systems, but such models only ever approximate the actual biological system and are tuned to specific applications. Depending on your specific requirements, the inclusion of more or less biological complexity may be necessary.

Switch to Powerpoint slides :)


MAP-Client - /home/abi/projects/mapclient-workflows/DTP-Simulation-Task3
File
Edit
View
Project
Tools
Help
Simulation Task 3
Simulate
Clear graph
Done

h: 0.01
tolerance: 1.0e-7

Fig. 1.21: The user interface in this task is similar to that described in Fig. 1.17, 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.


MAP-Client - /home/abi/projects/mapclient-workflows/DTP-Simulation-Task3
File
Edit
View
Project
Tools
Help
Simulation Task 3
Simulate
Clear graph
Done

h: 0.01
tolerance: 1.0e-3

h=0.0; tol=1.0e-7; time=3.71198415756
h=0.0; tol=1.0e-3; time=1.84028697014
h=0.01; tol=1.0e-3; time=3.20962715149

Fig. 1.22: Simulation results for a selection of simulations of the ICC model using various configurations of the CVODE integratior.

1.3.3 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. 1.16). 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. 1.23.

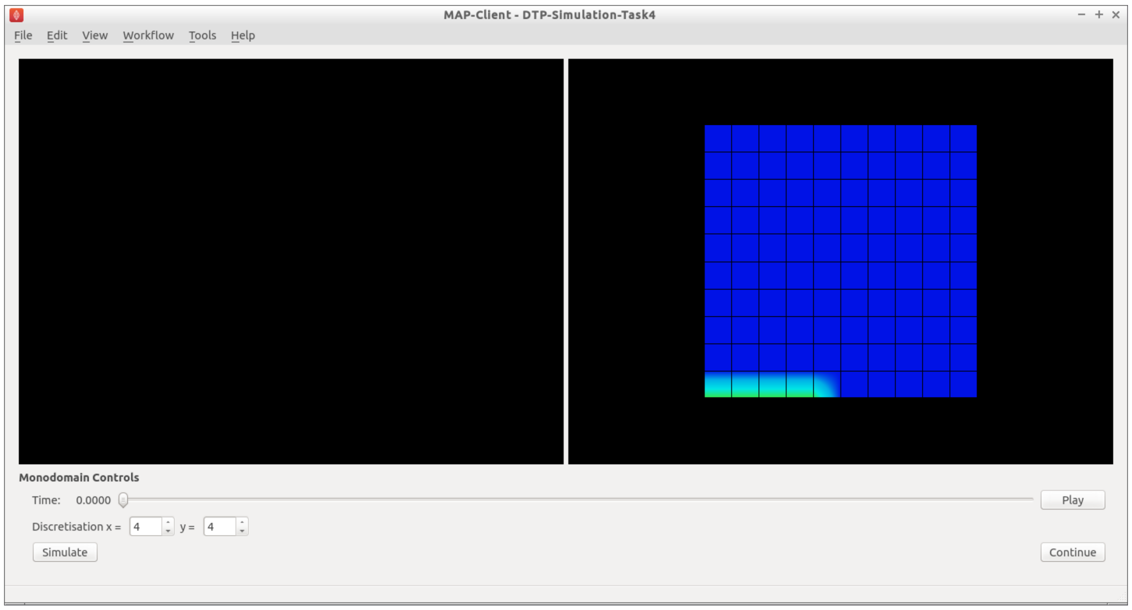


Fig. 1.23: The user interface in this task initially shows a "converged" solution on the right. The user is able to set the discretisation of the finite element mesh using the widgets at the bottom.

4. You can now investigate the effect of changing the spatial resolution. Example results are shown in Fig. 1.24.
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.
   (a) How long are you prepared to wait for a suitable simulation result?
   (b) Why do you need a higher mesh resolution in the x-direction than the y-direction to achieve a reasonable solution?
   (c) What is the lowest mesh resolution that gives a reasonable solution compared to the provided converged solution?

1.3.4 TL;DR

The quick highlights in this tutorial.

1. Introduce the concept of numerical integration of systems of differential equations.
   (a) Two specific algorithms presented: *Euler's method* and *CVODE*.

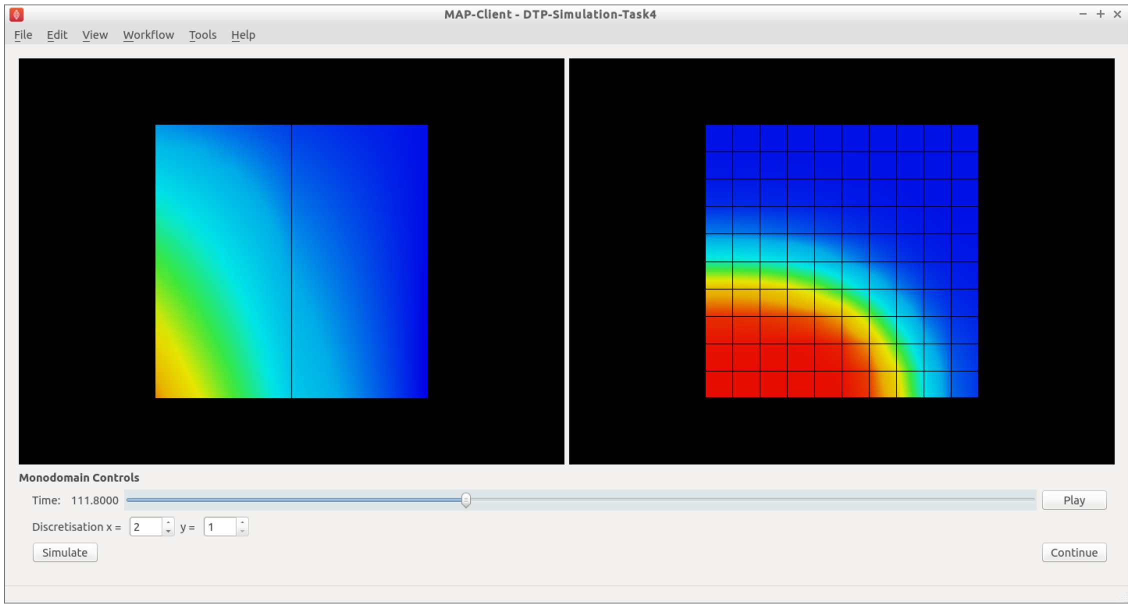


Fig. 1.24: Simulation results for a 2x1 mesh.

(b) Investigate the impact of choosing the correct algorithm control parameters (*Task 1 - the effect of step size*).

(c) Observe the limitations and the computational cost associated with numerical method choices (*Task 2 - fixed vs adaptive stepping*) and algorithm parameterisation (*Task 3 - error control*).

2. How much biological detial is too much?
3. Integrating cellular models into organ models.

1.4 Computational Physiology - Visualisation

Contents:

1.4.1 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.

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. 1.25:

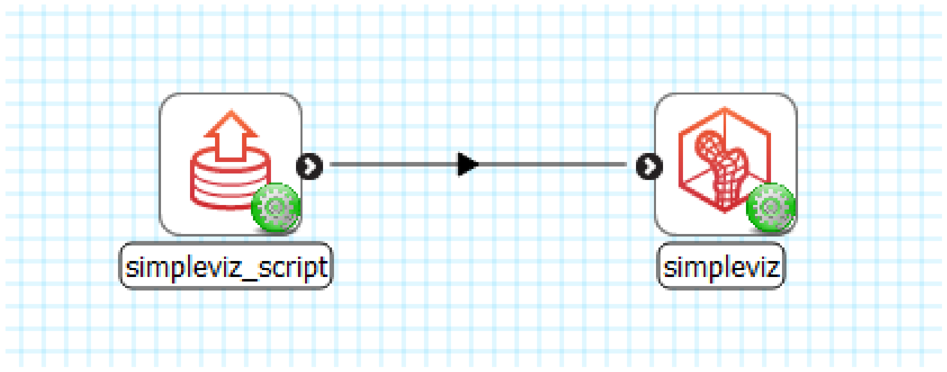


Fig. 1.25: Visualisation workflow using SimpleViz in the MAP client framework.

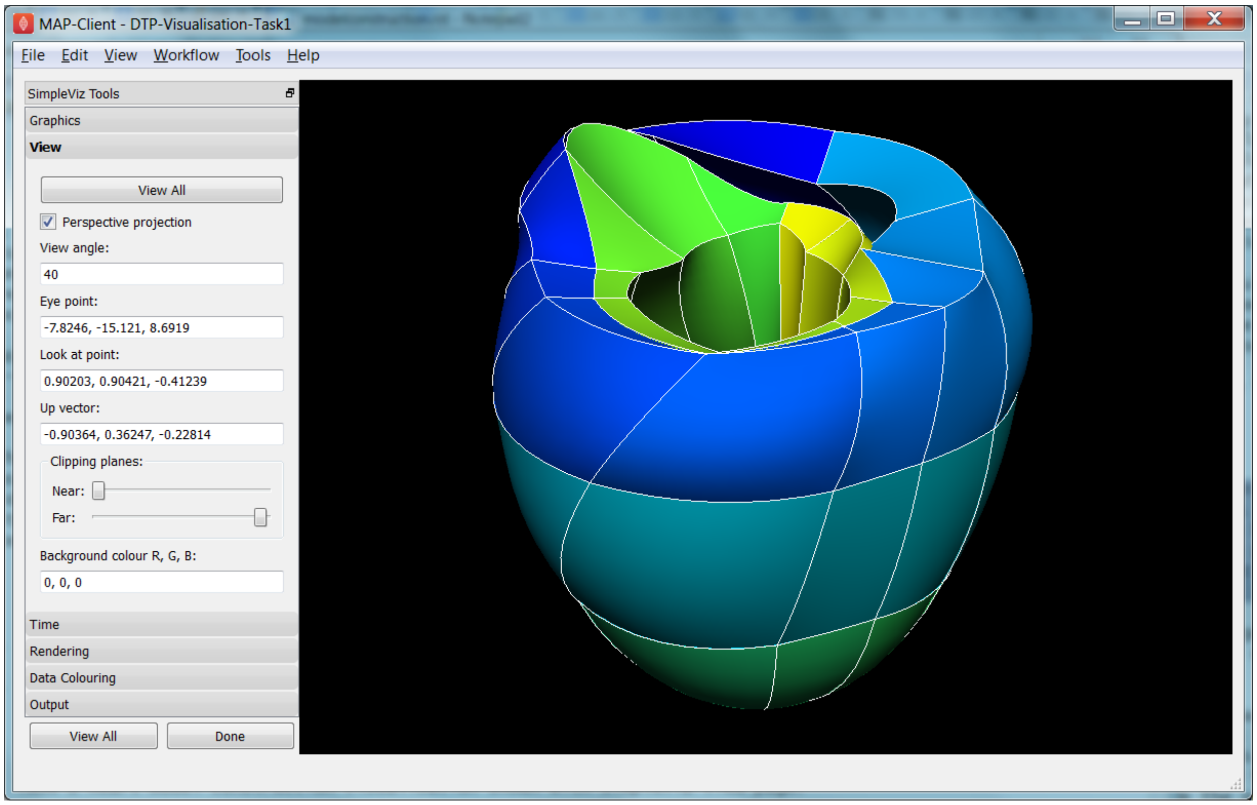
As the name suggests, SimpleViz presents a simplified interface for performing key aspects of interactive visualisation including results output. As shown in Fig. 1.26 its interface consists of a large 3-D graphics view and a toolbar with a series of pages for performing key functions. These are described 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. 1.26.

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:



[✓] 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
Background colour R, G, B: 0, 0, 0

Fig. 1.26: SimpleViz heart model construction visualisation, with view controls.

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.

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. 1.26). 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. 1.27 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.)

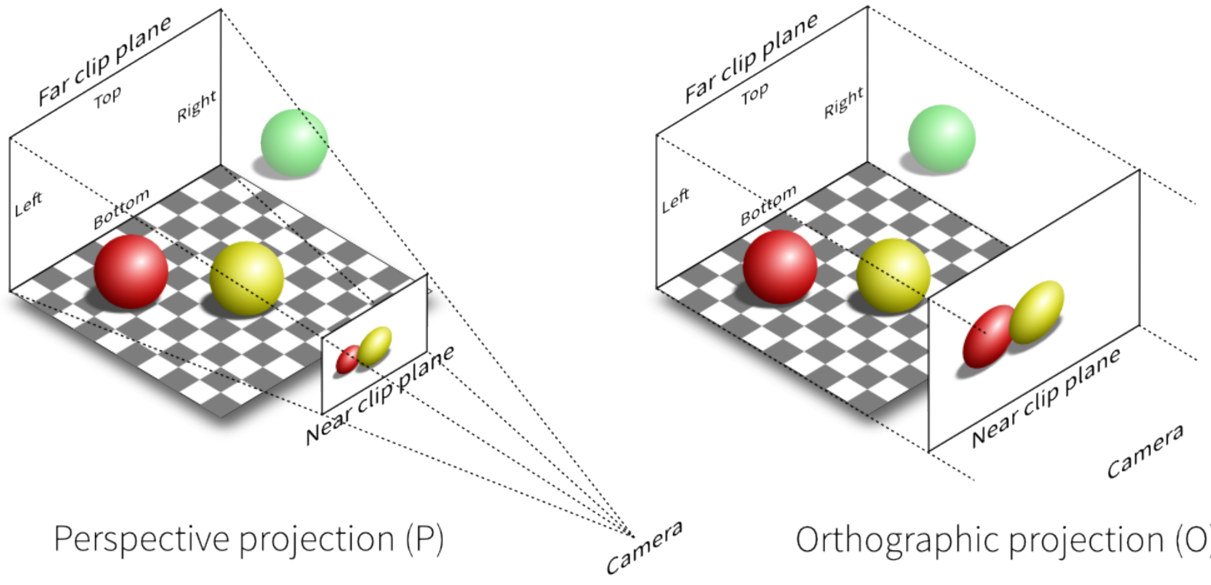


Fig. 1.27: Computer graphics perspective and orthographic/parallel projections. Original illustration from Nicolas P. Rougier, licensed under CC BY 4.0.

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.

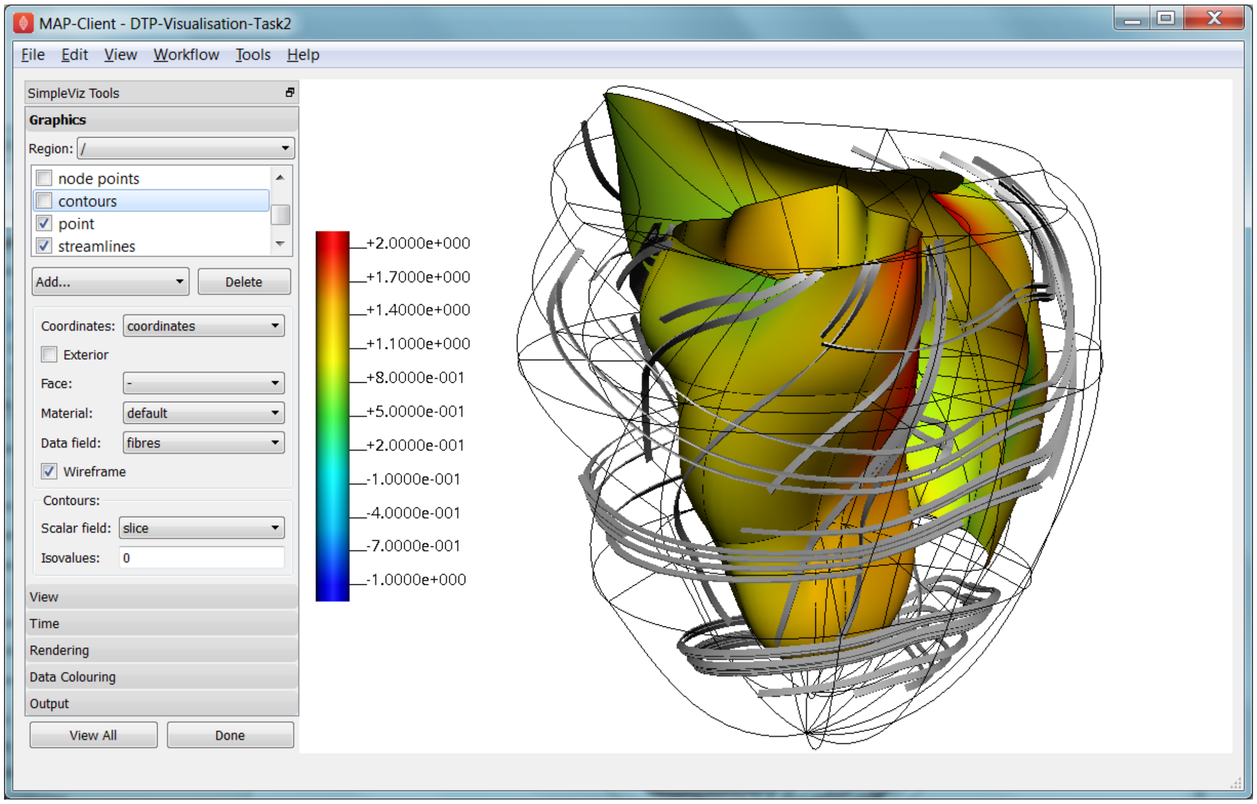
On the view page, drag the near clipping plane until close parts of the model disappear; when you are close you can hover over the slider and rotate the mouse wheel which moves it with more precision. Similar clipping occurs if you zoom in close enough to the model since you can't see things behind you. 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. 1.28 shows the model visualised how we want at the end of the task, and shows the graphics page controls in SimpleViz.



Region: /

☐ node points
☐ contours
☑ point
☑ streamlines

Coordinates: coordinates

☐ Exterior

Face: -

Material: default

Data field: fibres

☑ Wireframe

Contours:

Scalar field: slice

Isovalues: 0

Fig. 1.28: 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 scale 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 bases 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.

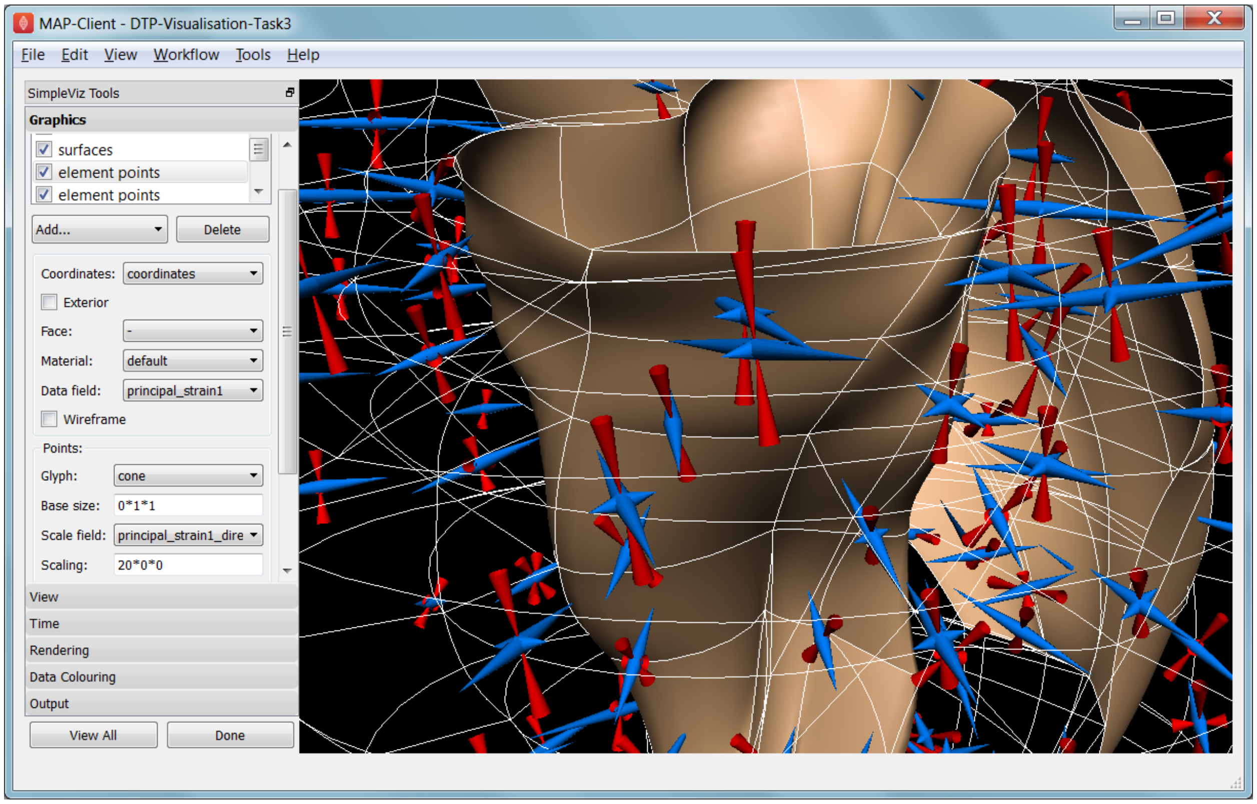
Printed Output

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. 1.29 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. 1.29: 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
```

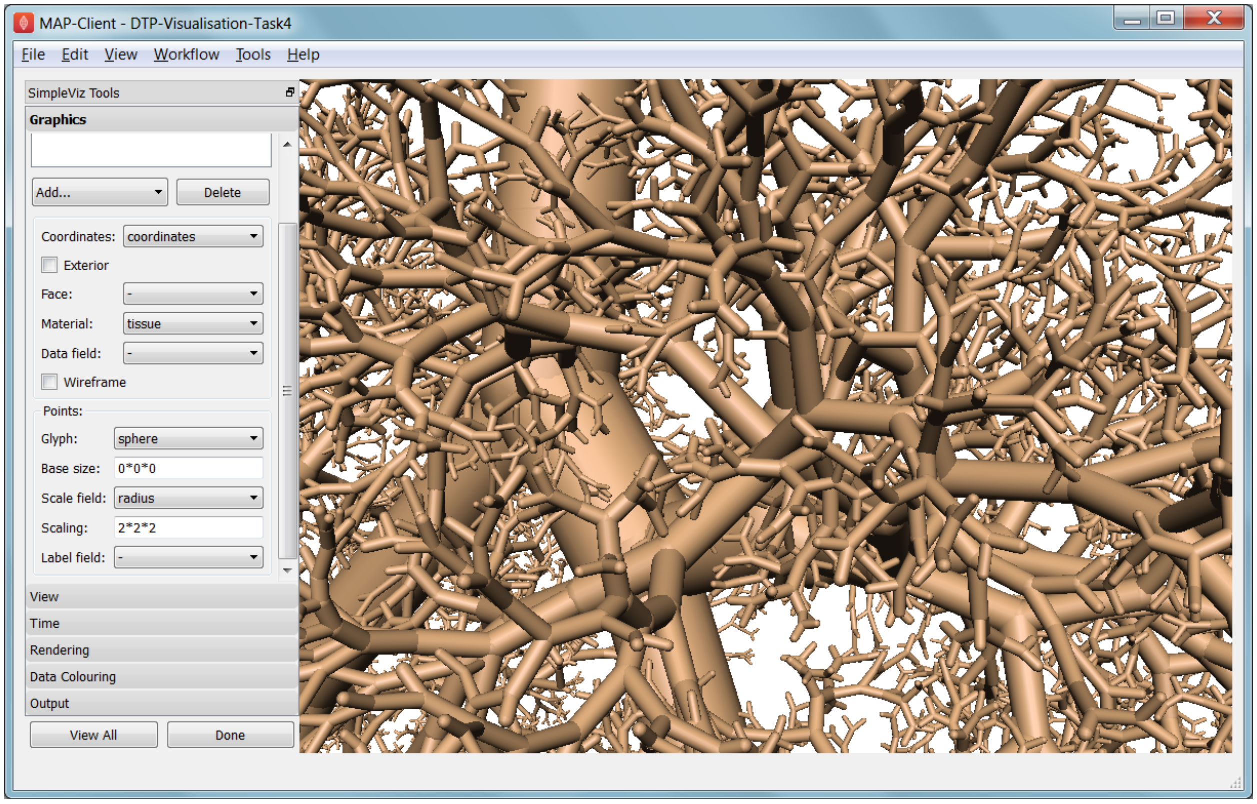
It should display the model as a slowly deforming heart, which you can view from different directions just as in SimpleViz. This technology is relatively new and there is still much to be exploited, but it shows one of the ways visualisations will be shared in the future.

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. 1.30 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



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

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.

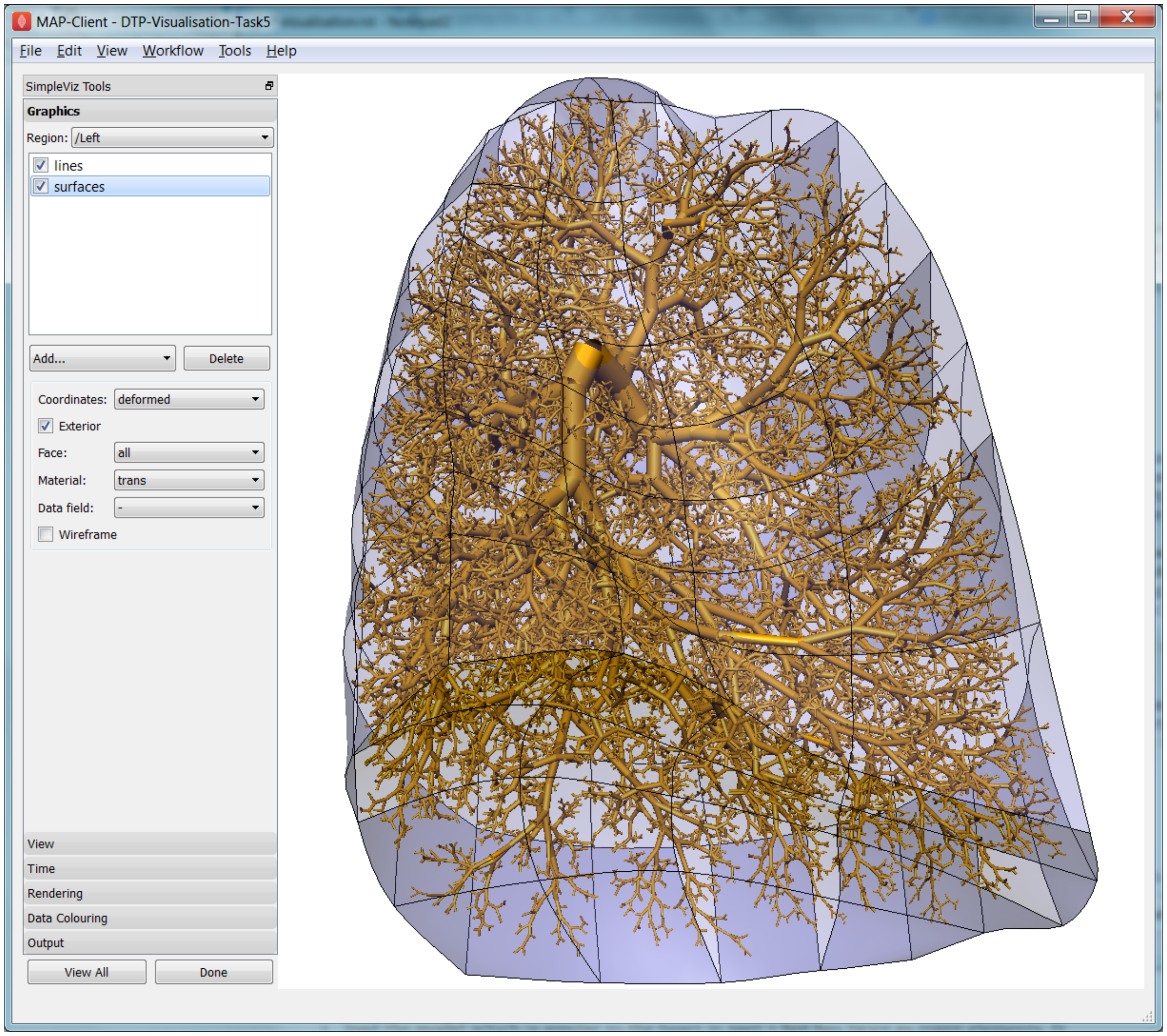
For any of these models it may be helpful to see where the global x, y, z axes are. Add a new *point* graphics, set the material to 'black', change the glyph to 'axes_xyz' and set the base size to 50. Surprisingly, the origin is quite far from the model; you may need to zoom out or click on 'View All'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

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. 1.31 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.



Region: /Left

- [✓] lines
- [✓] surfaces

Coordinates: deformed
[✓] Exterior
Face: all
Material: trans
Data field: -
[ ] Wireframe

Fig. 1.31: Left Lung with embedded airways.

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.

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 coordinate 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. 1.32 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.

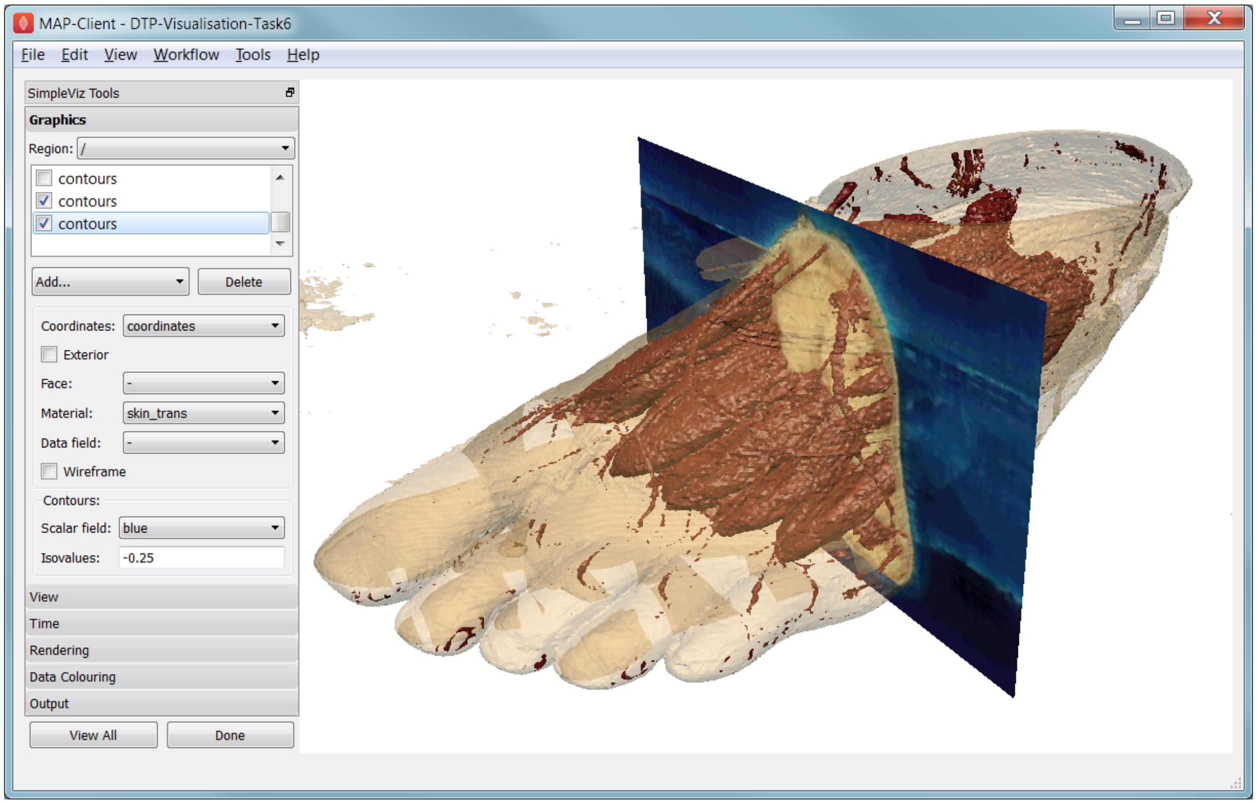
Initially two perpendicular slices of the 3-D images – contours of x and y – are drawn, plus two contours graphics segmenting surfaces of the skin and interior red tissue including muscles and larger vessels. An initially hidden contours graphics shows segmented bone surfaces, but is not so clean and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



Region: /

- [ ] contours
- [✓] contours
- [✓] contours

Coordinates: coordinates

- [ ] Exterior

Face: -

Material: skin_trans

Data field: -

- [ ] Wireframe

Contours:

Scalar field: blue

Isovalues: -0.25

Fig. 1.32: Segmented skin, muscles and vessels of the foot image.

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.

1.5 Computational Physiology - Complete Workflow

Contents:

1.5.1 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.

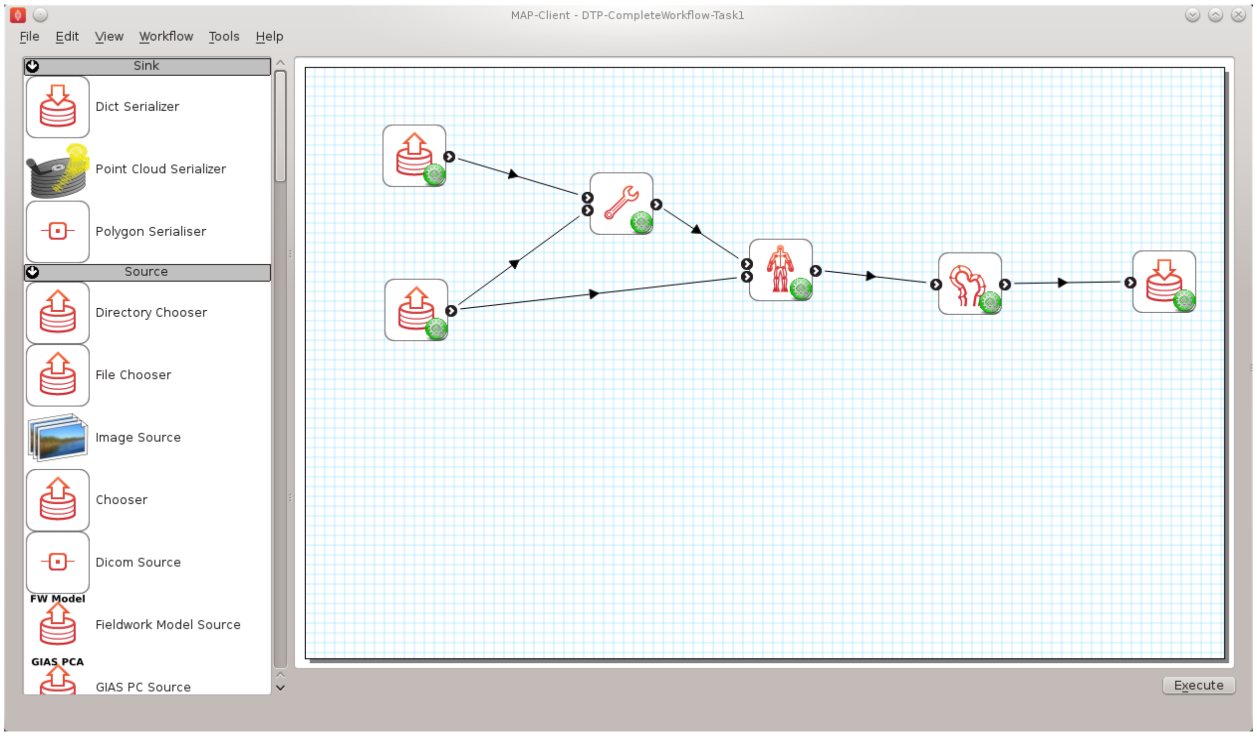


Fig. 1.33: Task 1 workflow horse fetlock characterisation

In this workflow [shown in Fig. 1.33] 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.

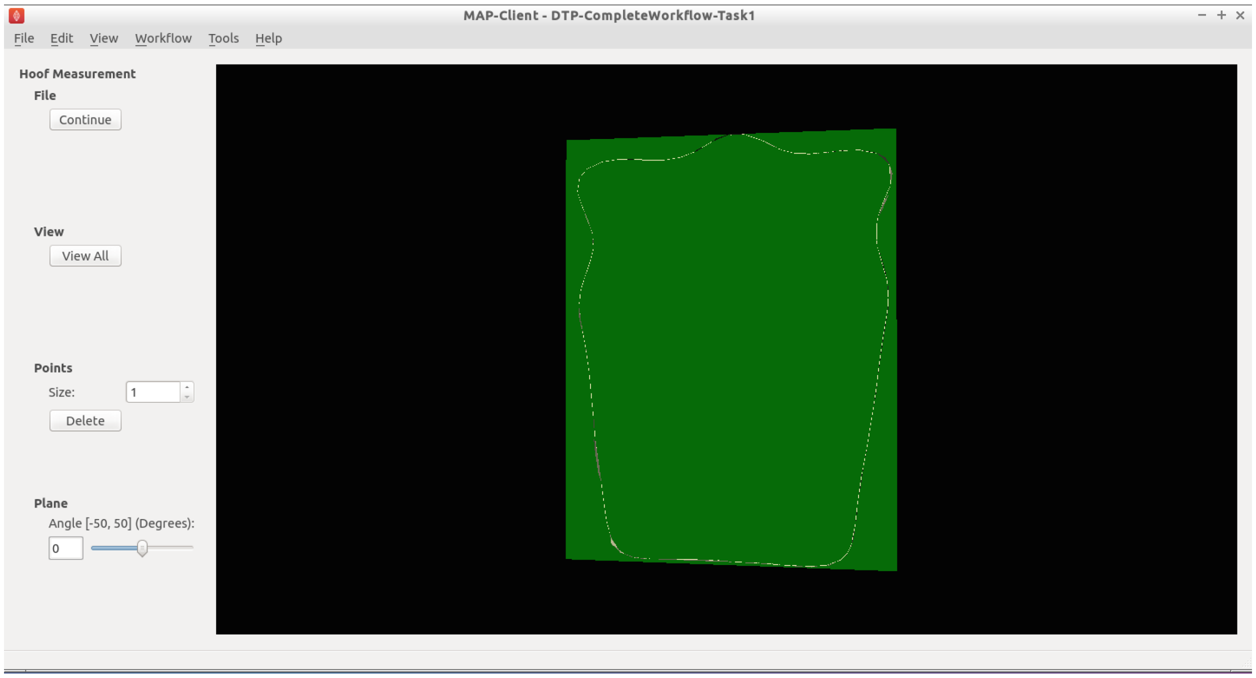

MAP-Client - DTP-CompleteWorkflow-Task1
File Edit View Workflow Tools Help
Hoof Measurement
File
Continue
View
View All
Points
Size:
1
Delete
Plane
Angle [-50, 50] (Degrees):
0

Fig. 1.34: Initial view of the 'Hoof measurement' step

Fig. 1.34 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. 1.35 shows seven points placed on the segmentation plane with the plane angle set at zero degrees.

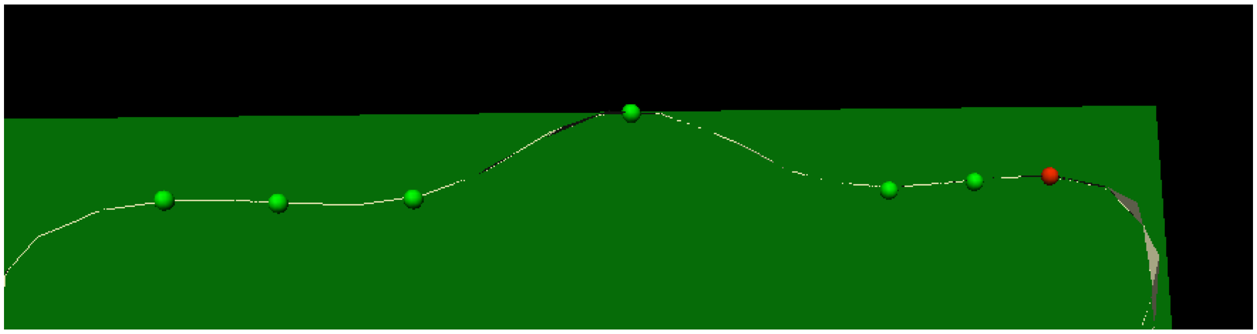


Fig. 1.35: Segmented ridge of fetlock joint at 0 degrees

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. 1.36 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.

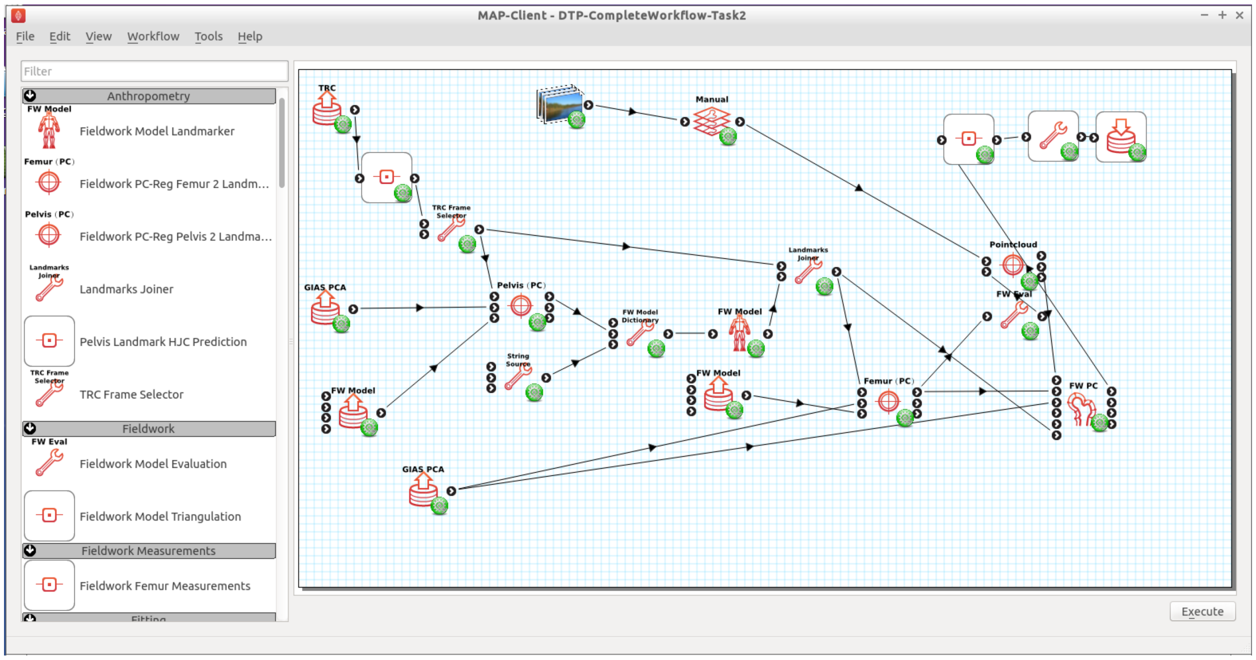


Fig. 1.36: Task 2 patient femur analysis workflow.

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.

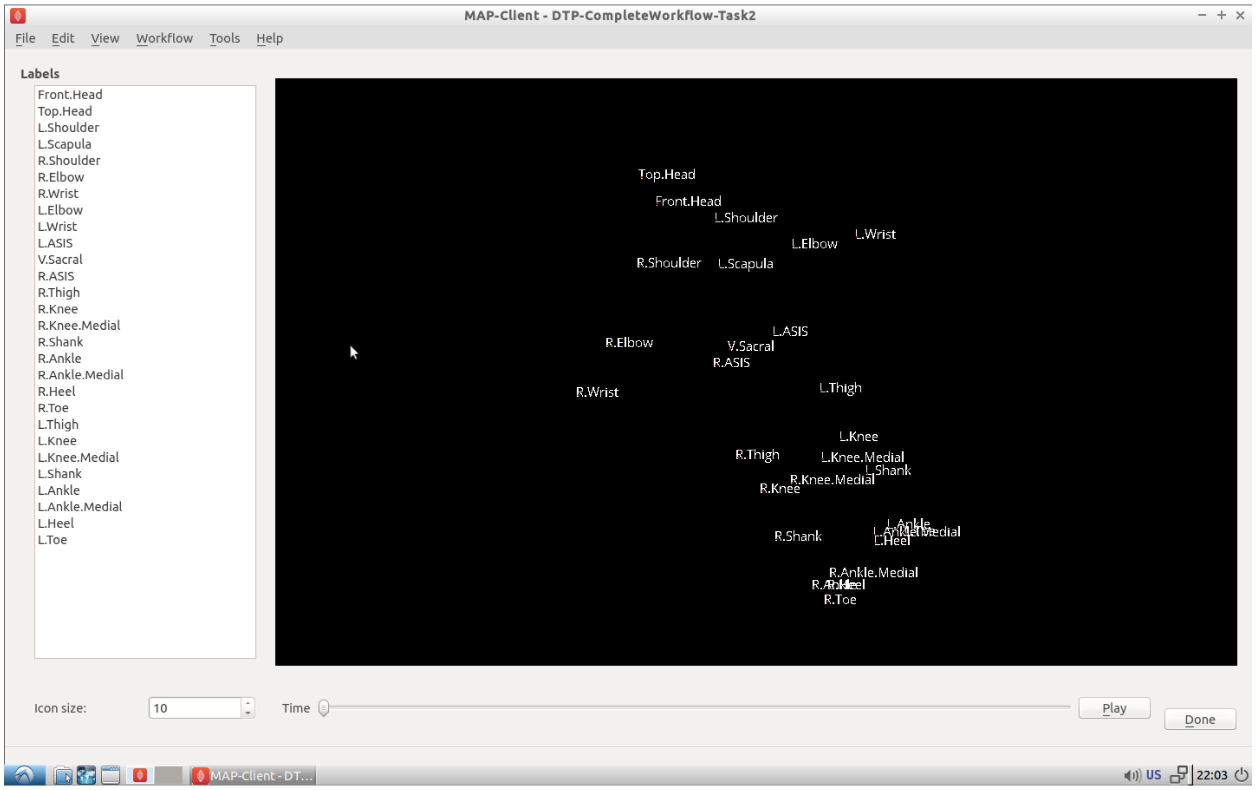


Fig. 1.37: MOCAP data viewer showing the fiducial markers location and their associated labels.

The first interactive step that occurs when executing the workflow is the MOCAP viewer step (shown in Fig. 1.37). 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. 1.38).

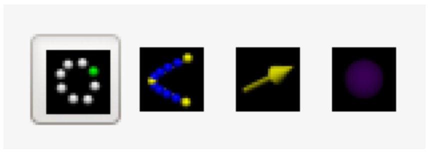


Fig. 1.38: Segmentation toolbar showing the icons for the different tools available to the user.

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.

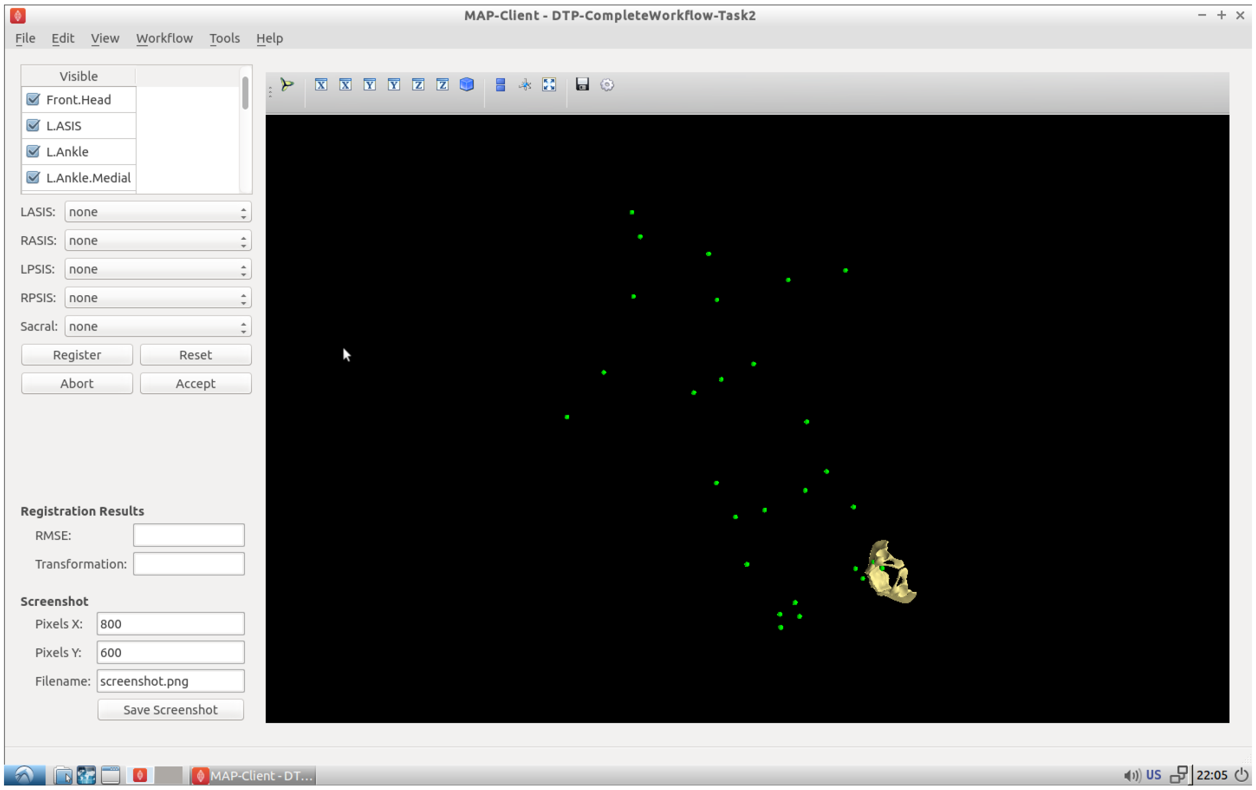
You are able to load a pre-prepared segmentation using the load button on under the file tab. It is also possible to save your own segmentation using the save button under the file tab. But beware that at this point you can only havea single saved segmention, so using the save button will overwrite the 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. 1.39 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.



Visible:
- [✓] Front.Head
- [✓] L.ASIS
- [✓] L.Ankle
- [✓] L.Ankle.Medial

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

RMSE:
Transformation:

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

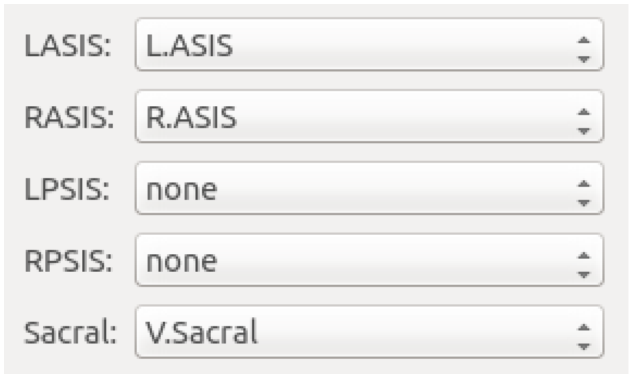
Fig. 1.39: Initial view of the register pelvis step.

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. 1.40 shows the correct associations.

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.

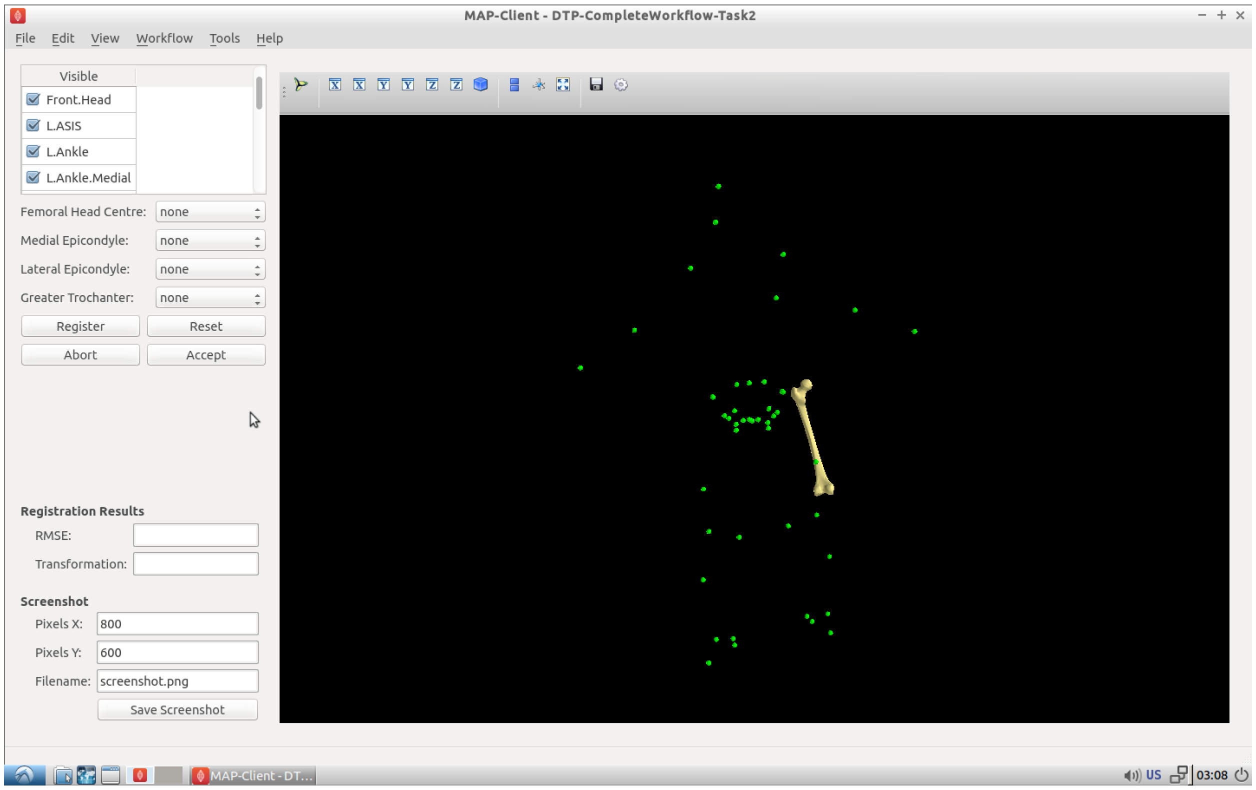
Press the register button to perform the registration, if the fit looks correct accept it to continue with the workflow.

Now it is time to register the femur to the subjects fedcial markers, in Fig. 1.41 we see a very similar interface to what was seen in the registration of the pelvis (Fig. 1.39). 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 fedcial 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.



LASIS:	L.ASIS
RASIS:	R.ASIS
LPSIS:	none
RPSIS:	none
Sacral:	V.Sacral

Fig. 1.40: Association of pelvis model landmarks to fiducial markers.



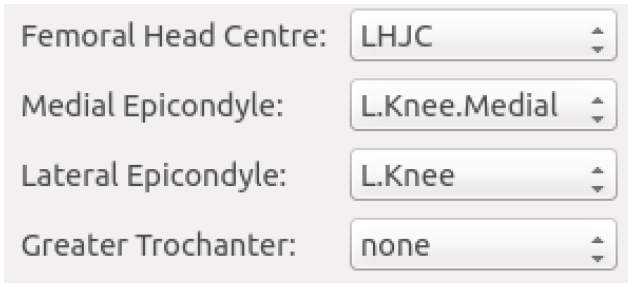
Femoral Head Centre: none
Medial Epicondyle: none
Lateral Epicondyle: none
Greater Trochanter: none

Registration Results
RMSE:
Transformation:

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

Fig. 1.41: 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. 1.42 shows the correct associations for the left femur model.

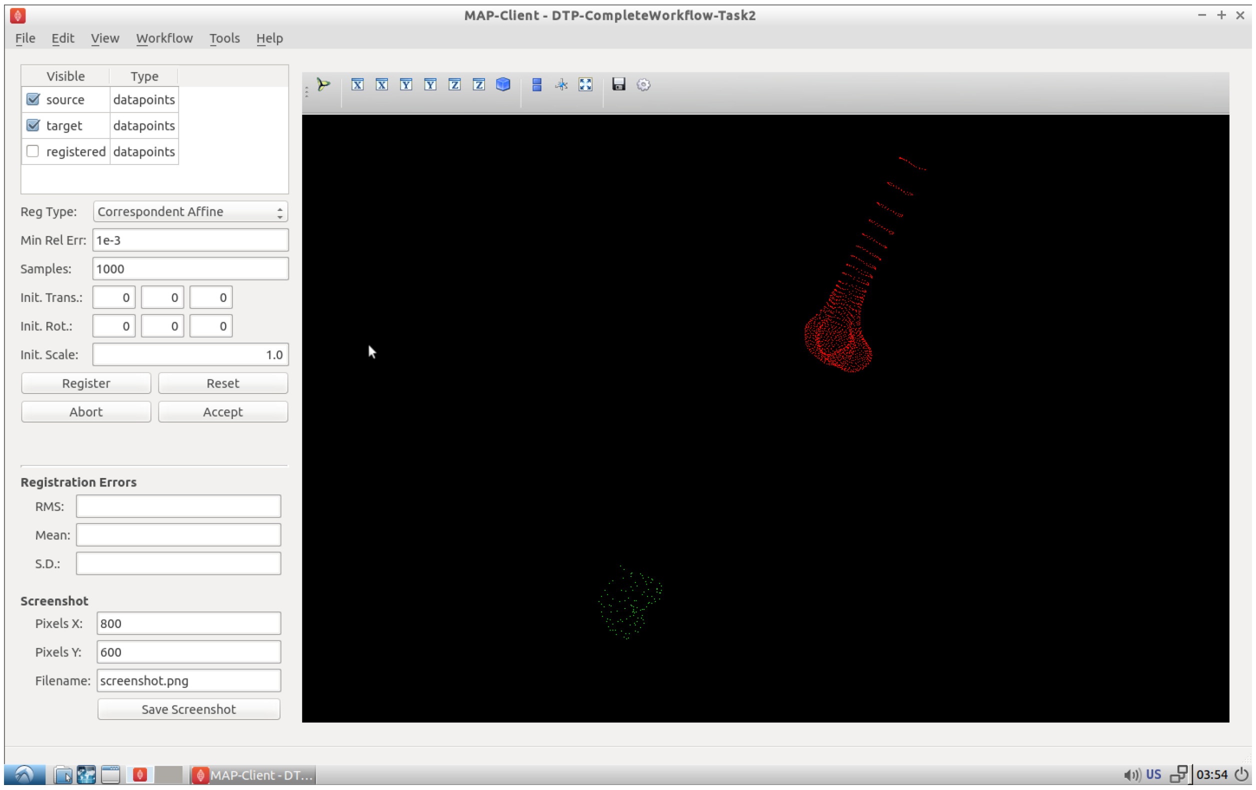


Femoral Head Centre: LHJC
Medial Epicondyle: L.Knee.Medial
Lateral Epicondyle: L.Knee
Greater Trochanter: none

Fig. 1.42: 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. 1.43



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. 1.43: 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. 1.44.

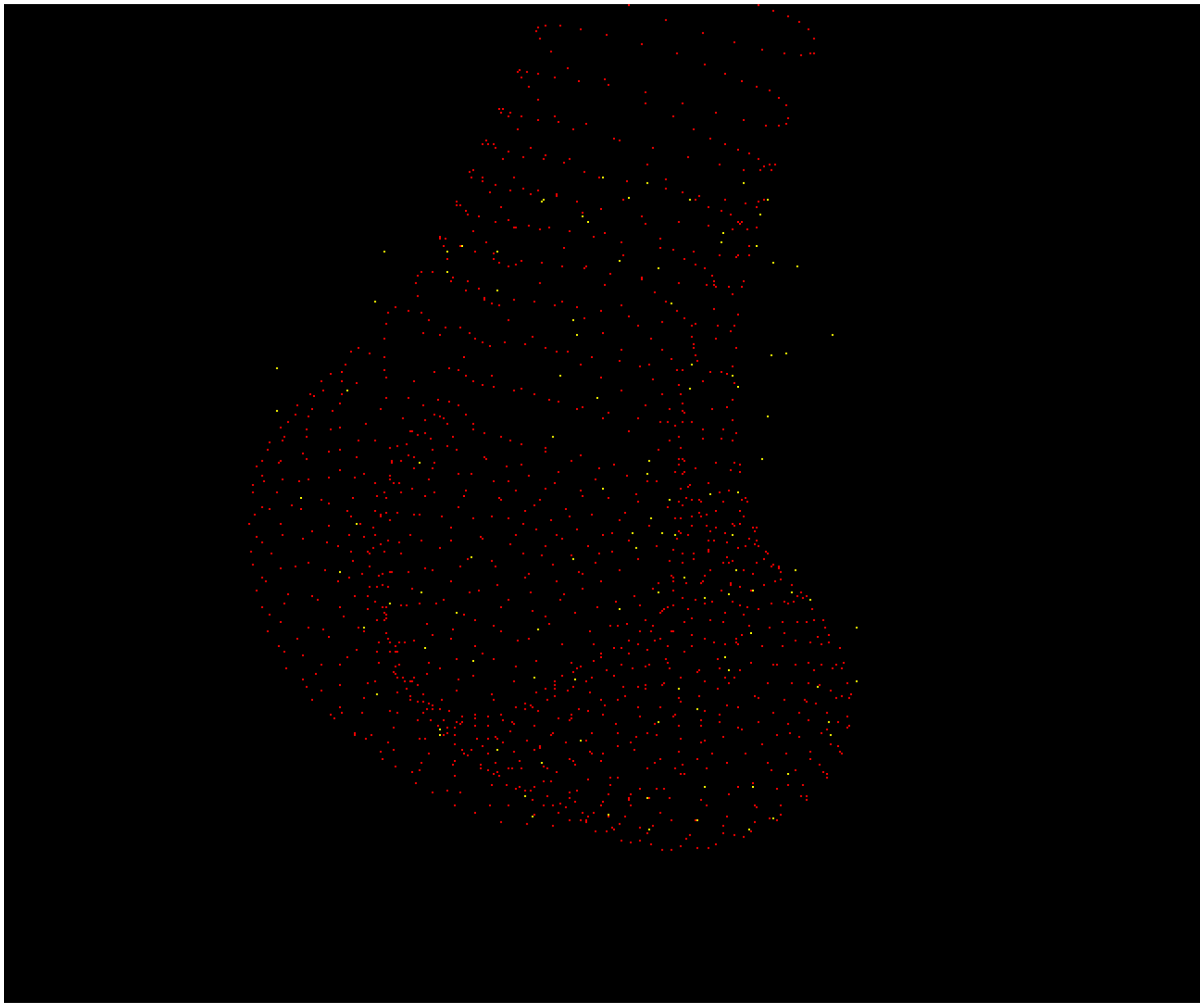


Fig. 1.44: Result of registering the segmented point cloud using ICP source-target registration method with the default settings.

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. 1.45 for the correct settings). We should now see a much better alignment of the two point clouds

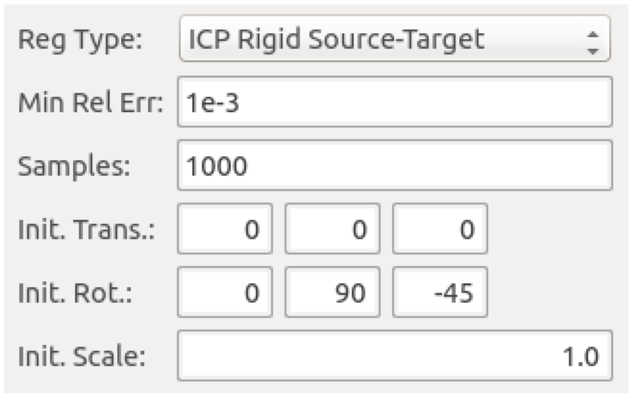

Reg Type: ICP Rigid Source-Target
Min Rel Err: 1e-3
Samples: 1000
Init. Trans.: 0 0 0
Init. Rot.: 0 90 -45
Init. Scale: 1.0

Fig. 1.45: Settings for the ICP source-target registration method.

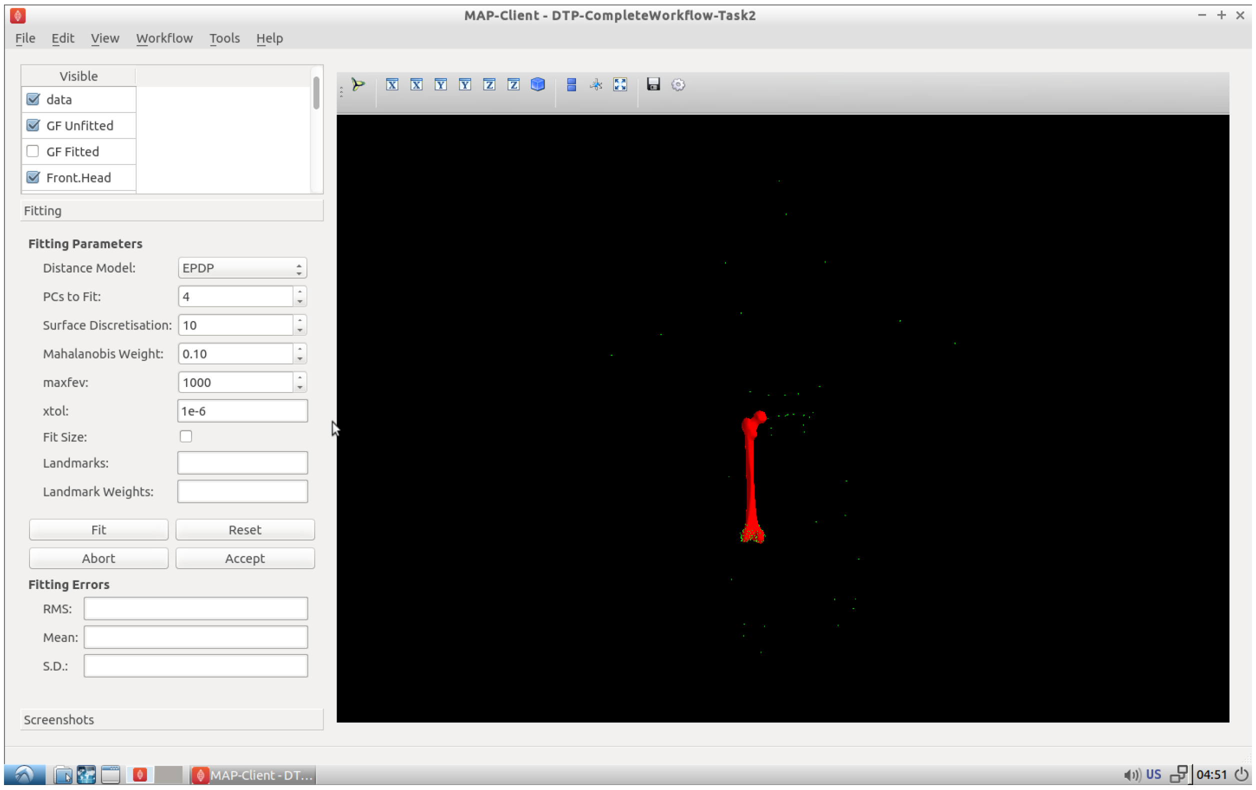
Have a look at the fit, if we do not have enough segmented points it will be difficult for the ICP registration to find a satisfactory fit to the model point cloud. In this situation, to get a satisfactory fit, we need to set the initial values so that they are very close to the final values making the 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. 1.46). 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. 1.47 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.



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. 1.46: Initial view of the model fitting step.

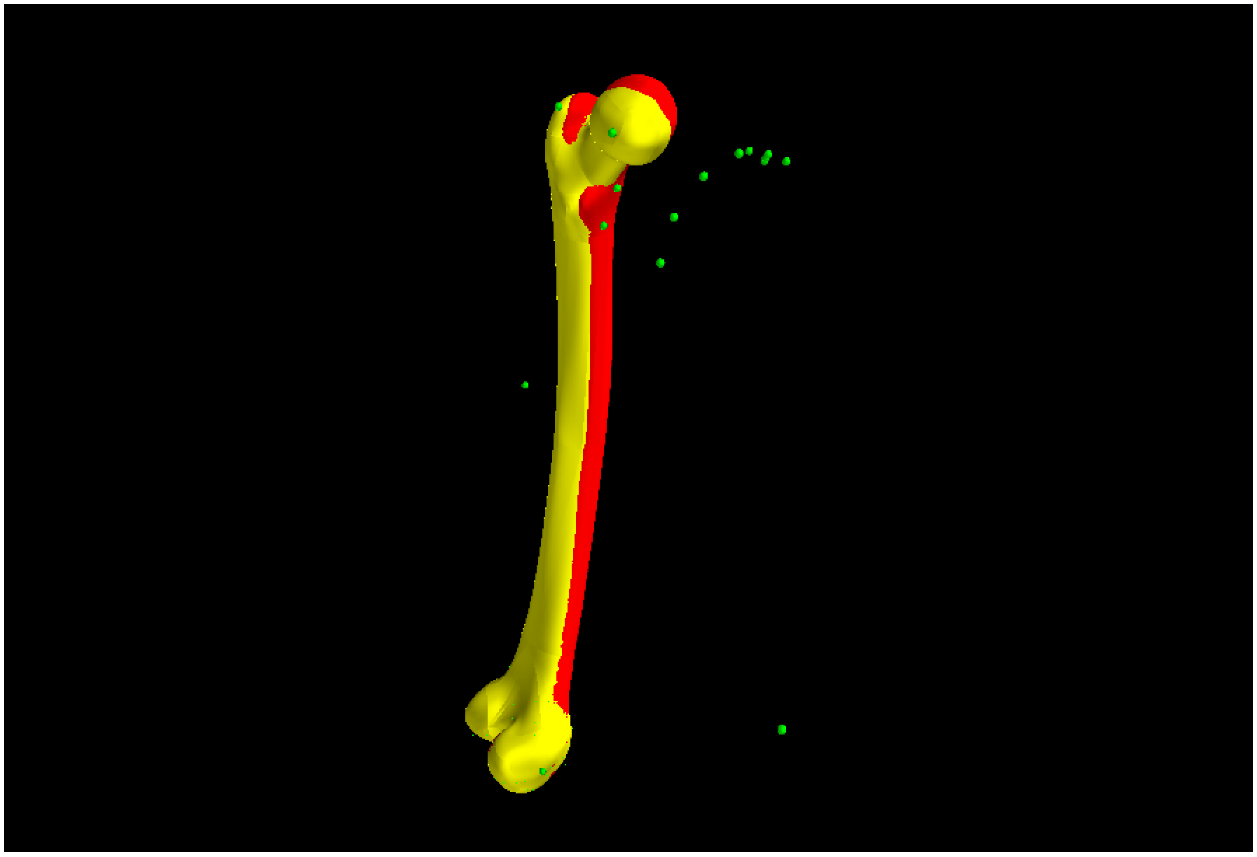


Fig. 1.47: The reference model (red) and the final fitted model (yellow).

CHAPTER 2

Introduction to (ODE) Modelling Best Practices

In this series of tutorials 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 exchange formats, but the principles are valid across the spectrum of computational physiology.

2.1 Tutorial on CellML, OpenCOR & the Physiome Model Repository

Note: This tutorial originated from a translation from a single Word document. Aspects of formatting and presentation may need further work. For reference, the original tutorial is available here: `OpenCOR-Tutorial-v17.pdf`.

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.

[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.

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]

2.1.1 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].

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.

[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.

[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.

IUPS Physiome Project
cellML
OpenCOR
VPH Institute
Building the Virtual Physiological Human

2.1.2 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 2016-02-27 (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. 2.1(a).

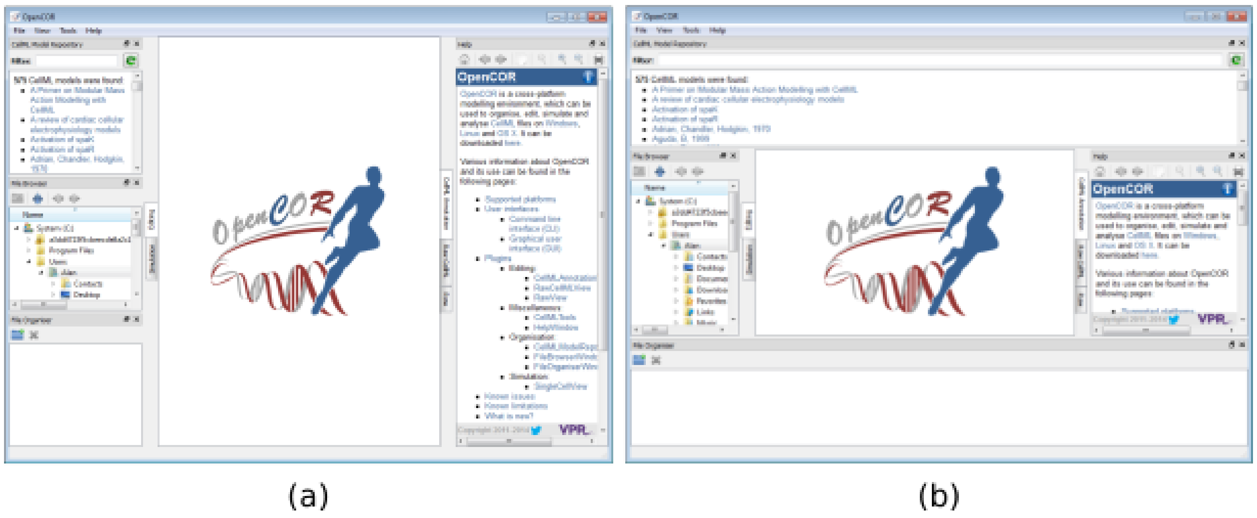


Fig. 2.1: OpenCOR application (a) Default positioning of dockable windows. (b) An alternative configuration achieved by dragging and dropping the dockable windows.

Dockable Windows

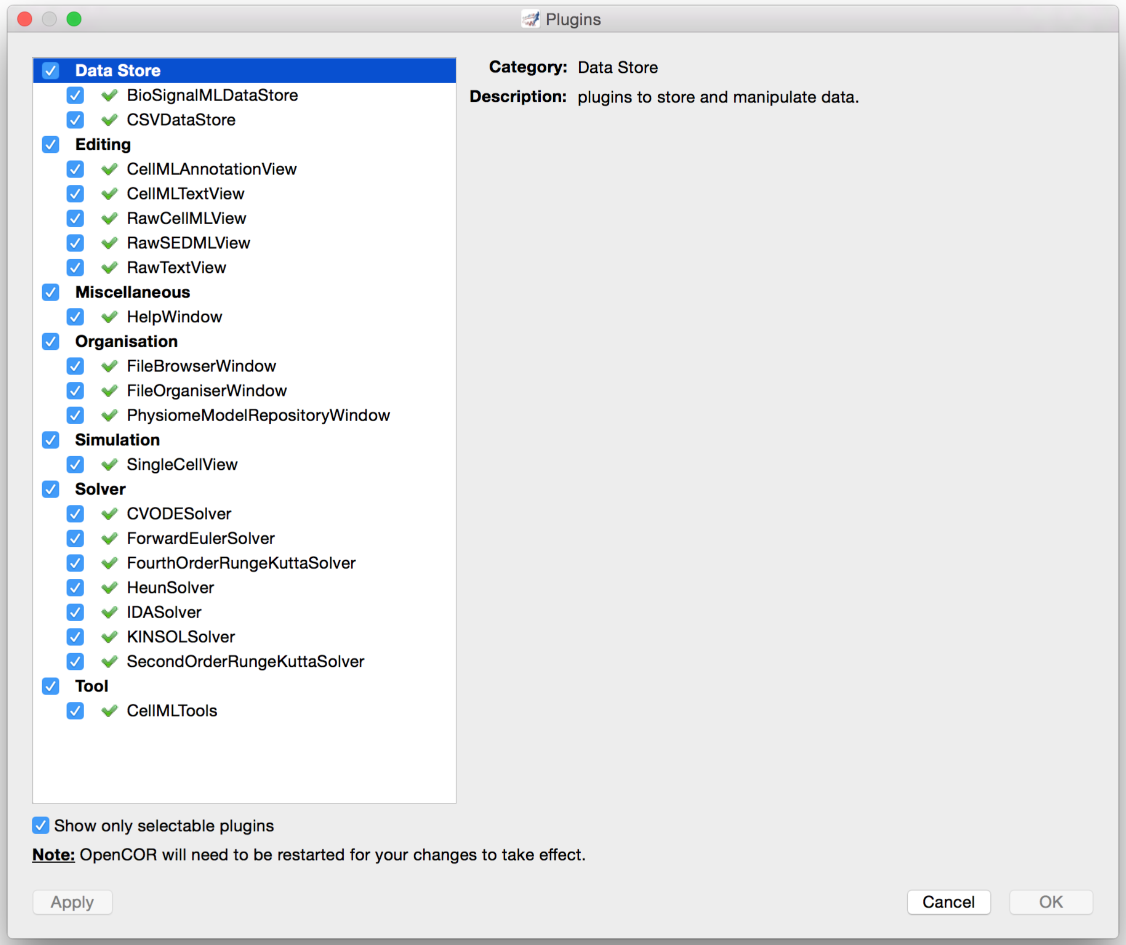
The central area is used to interact with files. By default, no files are open, hence the OpenCOR logo is shown instead. To the sides, there are dockable windows, which provide additional features. Those windows can be dragged and dropped to the top or bottom of the central area as shown in Figure 1(b) 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.

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. 2.2. Information about developing plugins for OpenCOR is also available.

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


Plugins

- ✓ **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. 2.2: OpenCOR tools menu showing the plugins that are selectable. Untick the box on the bottom left to show all plugins.

2.1.3 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:

$$\frac{\mathrm{dx}}{\mathrm{dt}} = y \tag{2.1}$$

$$\frac{\mathrm{dy}}{\mathrm{dt}} = \mu\left(1 - x^2\right)y - x \tag{2.2}$$

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:

```
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:

1. the closing semicolon at the end of each line (apart from the first two *def* statements that are opening a CellML construct);
2. the need to indicate dimensions for each variable and constant (all dimensionless in this example – but more on dimensions later);
3. the use of *ode(x,t)* to indicate a first order[4] ODE in *x* and *t*
4. the use of the squaring function *sqr(x)* for x^2, and
5. 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)

[1] http://en.wikipedia.org/wiki/Van_der_Pol_oscillator
[2] Equations (2.1) and (2.2) are equations that are implemented directly in OpenCOR.
[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.

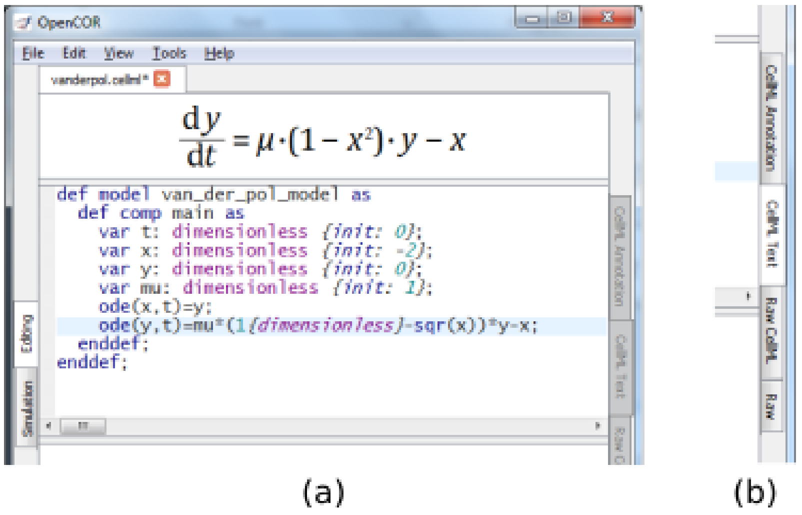
Table 1. Partial list of mathematical functions available for coding in OpenCOR.

Positioning the cursor over either of the ODEs renders the maths in standard form above the code as shown in Fig. 2.3.

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.



$$\frac{dy}{dt} = \mu \cdot (1 - x^2) \cdot y - x$$

```
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. 2.3: (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.

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 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.

[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.

- 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:

⭕ Editable constant	⭕ Editable state variable
🔴 Computed constant	🔵 Rate variable
● Variable of integration	🟢 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. 2.4 shows the results. Use the *interval delay* wheel

[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.

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.

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.

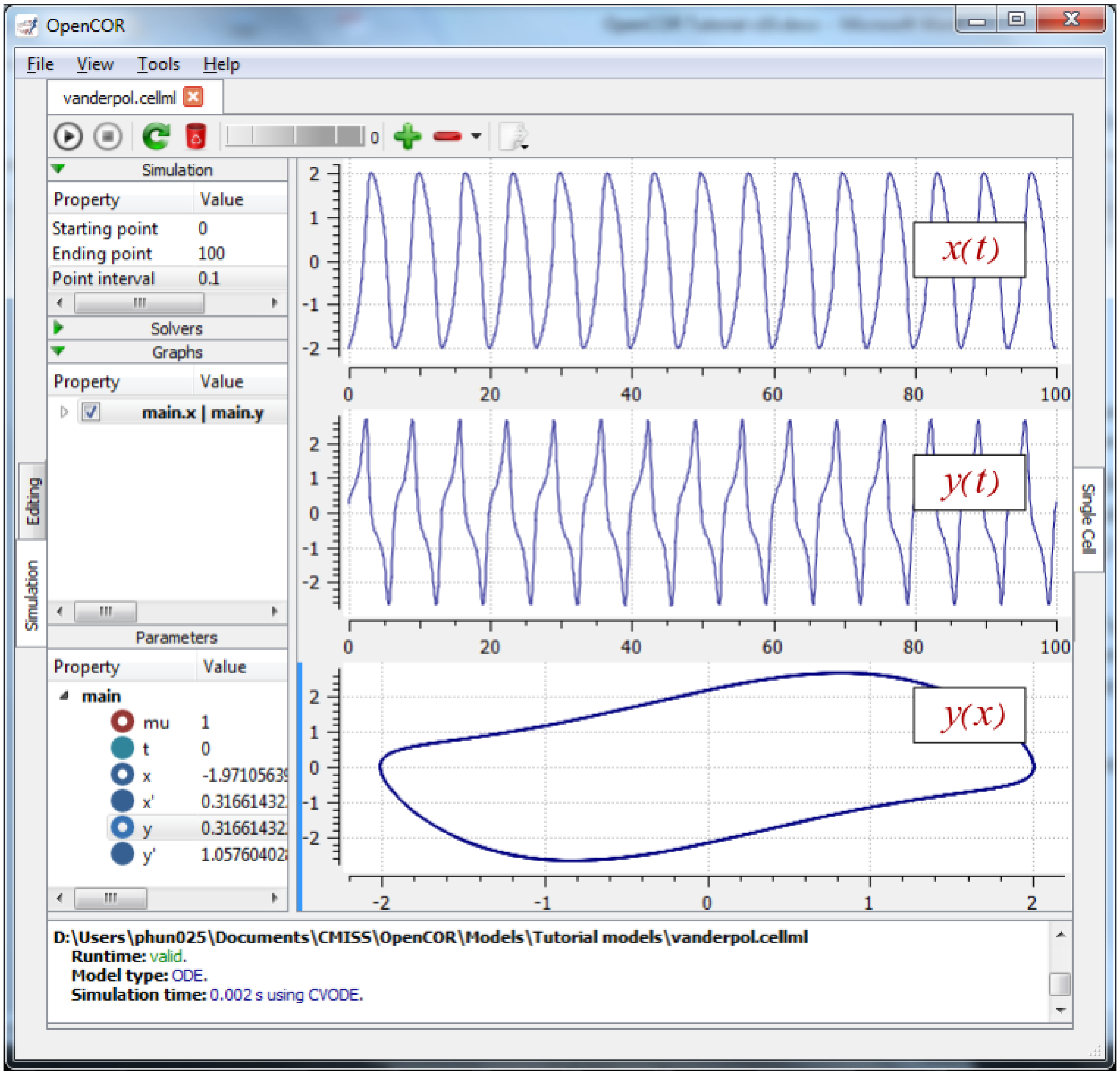
The various plugins used by OpenCOR can be viewed under the Tools menu. A French language version of OpenCOR is also available under the *Tools* menu. An option under the *File* menu allows a file to be locked (also 'CTRL-L'). To indicate that the file is locked, the background colour switches to pink in the *CellML Text* and *Raw CellML* views and a lock symbol appears on the filename tab. Note that OpenCOR text is case sensitive.

2.1.4 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. 2.1(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. 2.5 - 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.



Property	Value
Starting point	0
Ending point	100
Point interval	0.1

Property	Value
✓	main.x | main.y

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. 2.4: 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.

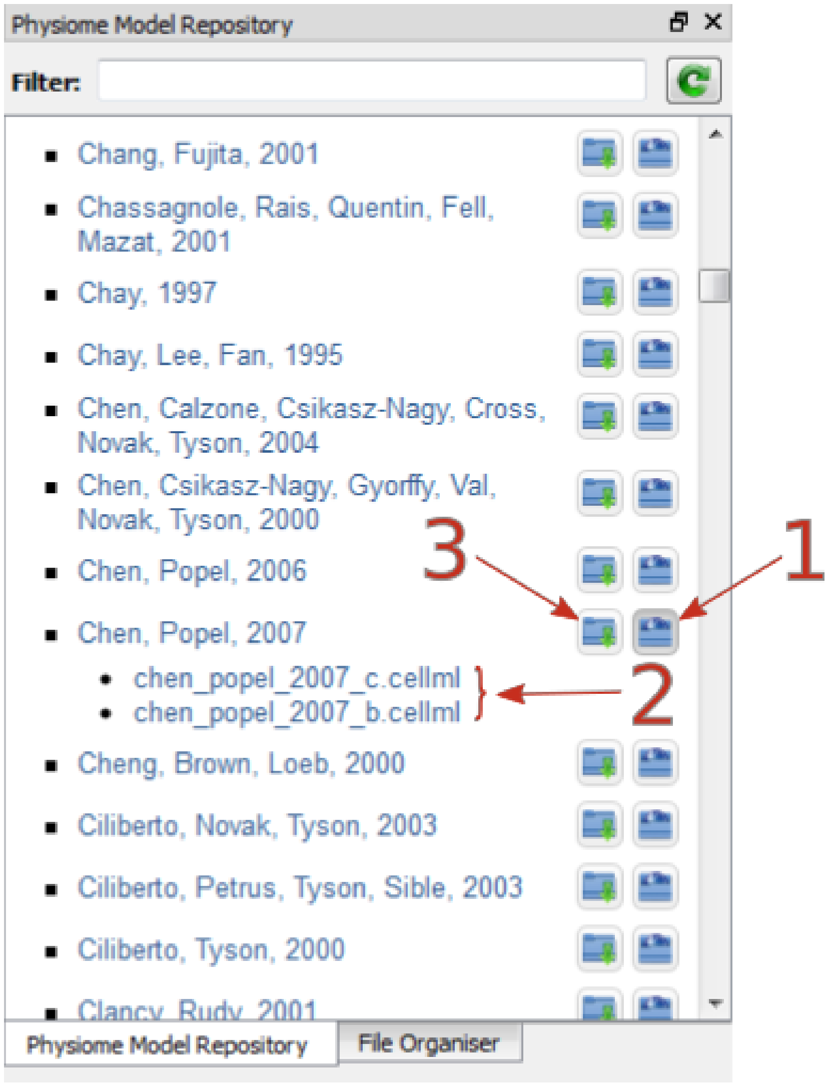

Physiome Model Repository
Filter:
Chang, Fujita, 2001
Chassagnole, Rais, Quentin, Fell, Mazat, 2001
Chay, 1997
Chay, Lee, Fan, 1995
Chen, Calzone, Csikasz-Nagy, Cross, Novak, Tyson, 2004
Chen, Csikasz-Nagy, Gyorffy, Val, Novak, Tyson, 2000
Chen, Popel, 2006
Chen, Popel, 2007
chen_popel_2007_c.cellml
chen_popel_2007_b.cellml
Cheng, Brown, Loeb, 2000
Ciliberto, Novak, Tyson, 2003
Ciliberto, Petrus, Tyson, Sible, 2003
Ciliberto, Tyson, 2000
Clancy, Rudy, 2001
3
1
2
Physiome Model Repository
File Organiser

Fig. 2.5: 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.

In the PMR window (Fig. 2.5) 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.

2.1.5 A simple first order ODE

The simplest example of a first order ODE is

$$\frac{\mathrm{dy}}{\mathrm{dt}} = -ay + b$$

with the solution

$$y\left(t\right) = \frac{b}{a} + \left(y\left(0\right) - \frac{b}{a}\right).e^{-at},$$

where $y\left(0\right)$ or y_0, the value of $y\left(t\right)$ at $t = 0$, is the *initial condition*. The final steady state solution as $t \to \infty$ is $y\left(\left.t\right|_{\infty}\right) = y_{\infty} = \frac{b}{a}$ (see Figure 6). Note that $t = \tau = \frac{1}{a}$ is called the *time constant* of the exponential decay, and that

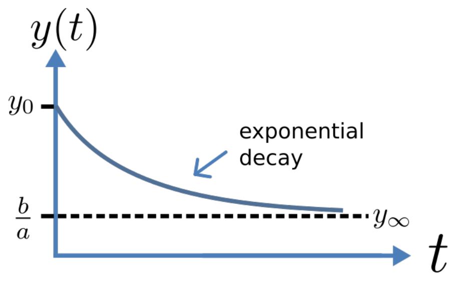


Fig. 2.6: Solution of 1st order equation.

$$y\left(\tau\right) = \frac{b}{a} + \left(y\left(0\right) - \frac{b}{a}\right).e^{-1}.$$

At $t = \tau$, $y\left(t\right)$ has therefore fallen to $\frac{1}{e}$ (or about 37%) of the difference between the initial ($y\left(0\right)$) and final steady state ($y\left(\infty\right)$) values[1].

Choosing parameters $a = \tau = 1; b = 2$ and $y\left(0\right) = 5$, 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. 2.7(a) for these parameters (a decaying exponential) and in Fig. 2.7(b) for parameters $a = 1; b = 5$ and $y\left(0\right) = 2$ (an inverted decaying exponential). Note the simulation panel with *Ending point*=10, *Point interval*=0.1. Try putting $a = -1$.

These two solutions have the same exponential time constant ($\tau = \frac{1}{a} = 1$) but different initial and final (steady state) values.

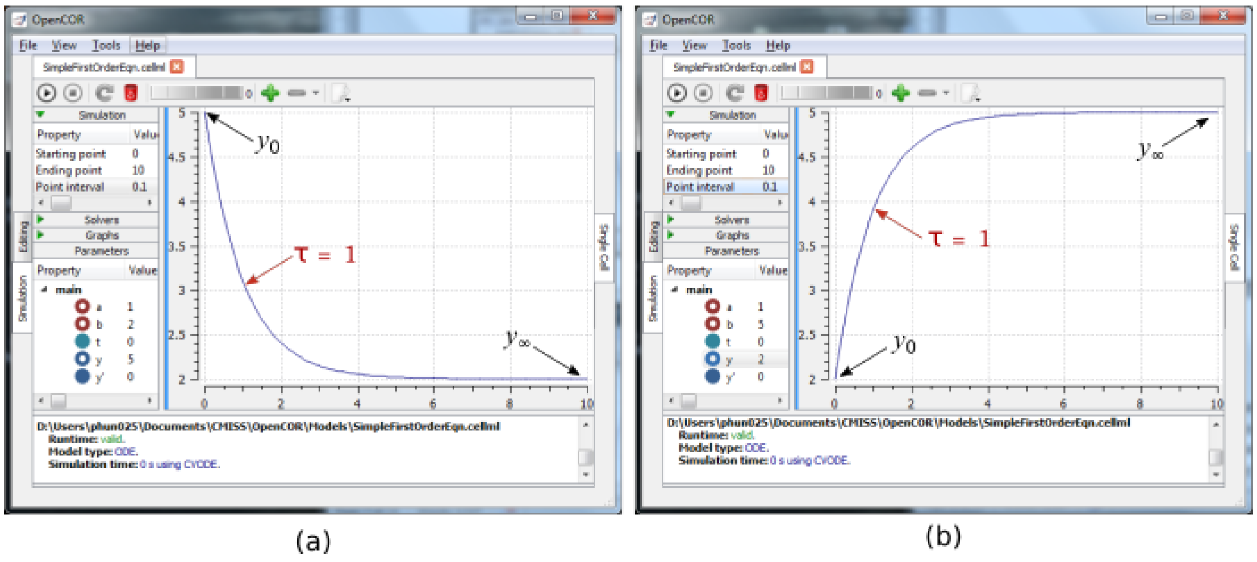
The exponential decay curve shown on the left in Fig. 2.7 is a common feature of many models and in the case of radioactive decay (for example) is a statement that the **rate of decay** ($-\frac{\mathrm{dy}}{\mathrm{dt}}$) is proportional to the **current amount of substance** (y). This is illustrated on the NZ$100 note (should you be lucky enough to possess one), shown in Figure 8.

2.1.6 The Lorenz attractor

An example of a third order ODE system (i.e. three 1st order equations) is the *Lorenz equations*[1].

[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: $y\left(t\right) = y_{\infty} + \left(y_0 - y_{\infty}\right).e^{-\frac{t}{\tau}}$.

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



(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. 2.7: 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.

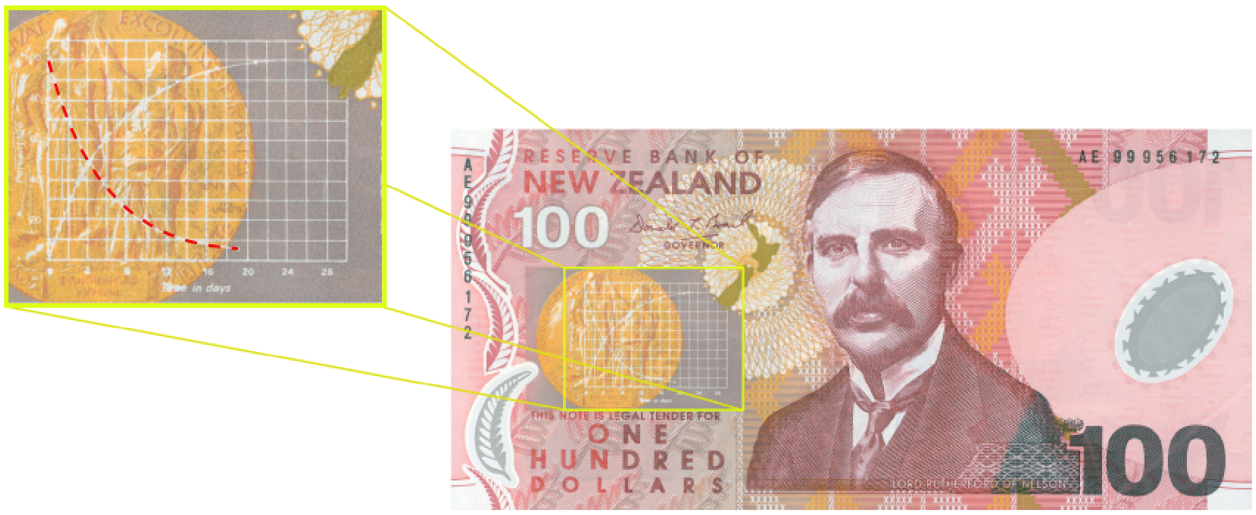


Fig. 2.8: The **exponential curve** representing the naturally occurring radioactive decay explained by the New Zealand Noble laureate Sir Ernest Rutherford - best known for ‘splitting the atom’. This may be the only bank note depicting the mathematical solution of a first order ODE.

This system has three equations:

$$\frac{\mathrm{dx}}{\mathrm{dt}} = \sigma\left(y - x\right)$$
$$\frac{\mathrm{dy}}{\mathrm{dt}} = x\left(\rho - z\right) - y$$
$$\frac{\mathrm{dz}}{\mathrm{dt}} = xy - \beta z$$

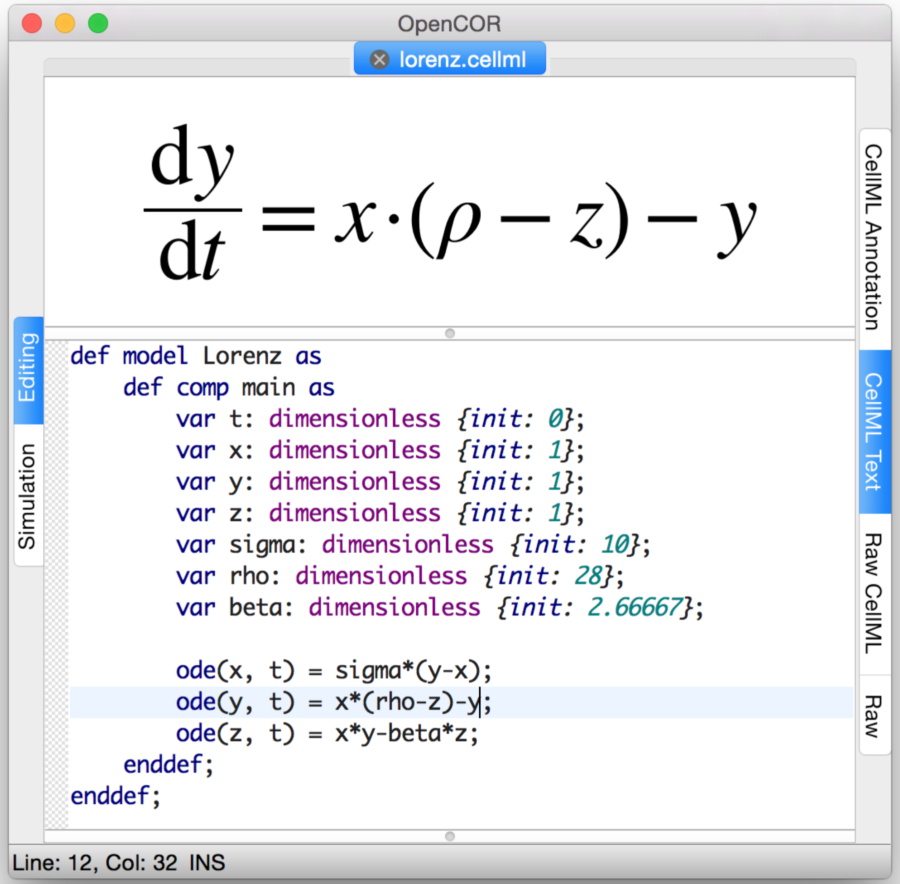
where σ, ρ and β are parameters.

The *CellML Text* code entered for these equations is shown in Fig. 2.9 with parameters

$\sigma = 10$, $\rho = 28$, $\beta = 8/3 = 2.66667$

and initial conditions

$x\left(0\right) = y\left(0\right) = z\left(0\right) =1.$



$$\frac{\mathrm{dy}}{\mathrm{dt}} = x \cdot (\rho - z) - y$$

```
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

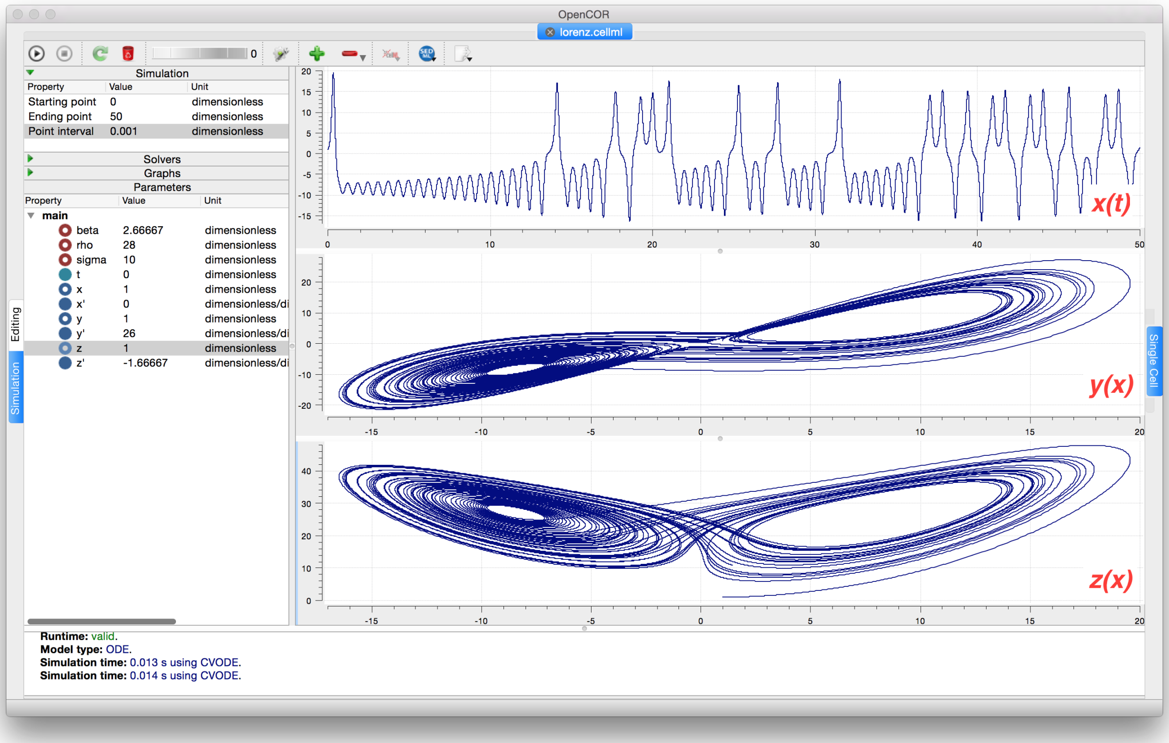
Fig. 2.9: *CellML Text* code for the Lorenz equations.

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. 2.10. 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.

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. 2.11).

[2] http://www.math.auckland.ac.nz/~hinke/crochet/



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. 2.10: 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.

Fig. 2.11: 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. 2.10 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.

2.1.7 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*.

Entropy S $(J.K^{-1})$ is a measure of the number of microstates available to a system, as defined by Boltzmann's equation $S = k_B \mathrm{lnW}$, 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. 2.12; 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

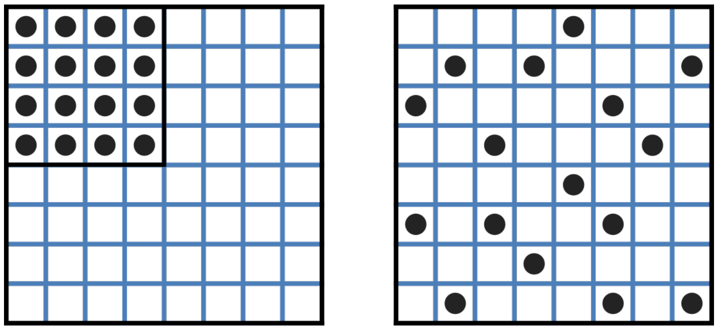


Fig. 2.12: Distribution of microstates in a system *[J97]*. The 16 particles in a confined region (left) have only one possible arrangement (W = 1) and therefore zero entropy ($k_B$lnW $= 0$). When the barrier is removed and the number of possible locations for each particle increases 4x (right), the 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.

[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.

for a low concentration[3]. The energy difference driving ion movement from 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. 2.13). 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 z$e\Delta\phi$ $(J.ion^{-1})$ or z$eN_A\Delta\phi$ $(J.mol^{-1})$. 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$\Delta\phi$ $(J.mol^{-1})$.

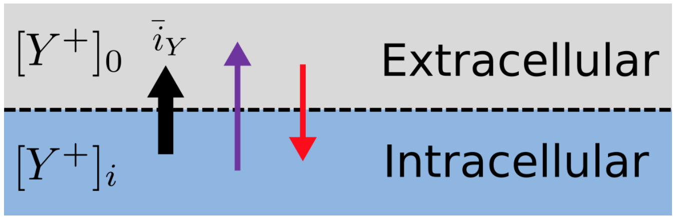


Fig. 2.13: The balance between entropic and electrostatic forces determines the Nernst potential.

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:

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

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

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:

$i_Y = g_Y\left(V - E_Y\right)$.

This defines a linear current-voltage relation ('Ohms law') as shown in Fig. 2.14. The gates to be discussed below modify this open channel conductance.

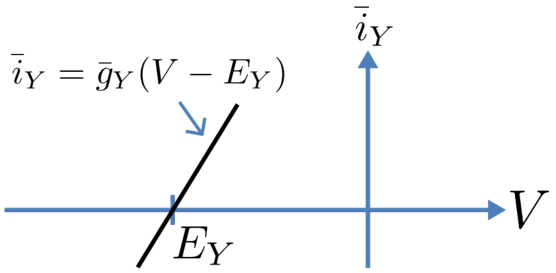


Fig. 2.14: Open channel linear current-voltage relation

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

[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 $F = eN_A$.

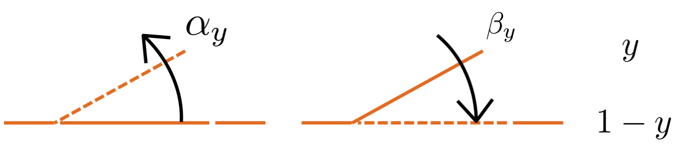


Fig. 2.15: Ion channel gating kinetics. y is the fraction of gates in the open state. α_y and β_y are the rate constants for opening and closing, respectively.

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 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. 2.15):

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

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. 2.16(a).

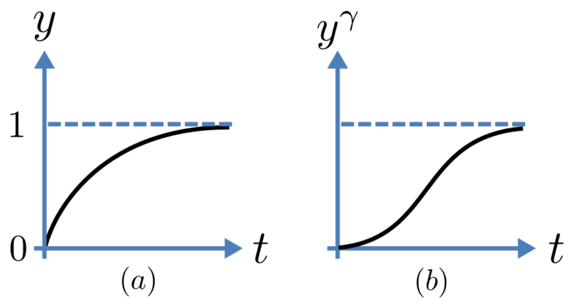


Fig. 2.16: Transient behaviour for one gate (left) and γ gates in series (right). Note that the right hand graph has an initial S-shaped increase, reflecting the multiple gates in series.

The experimental data obtained by Hodgkin and Huxley for the squid axon, however, indicated that the initial current flow began more slowly (Fig. 2.16(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 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)

[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

- **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:

	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}

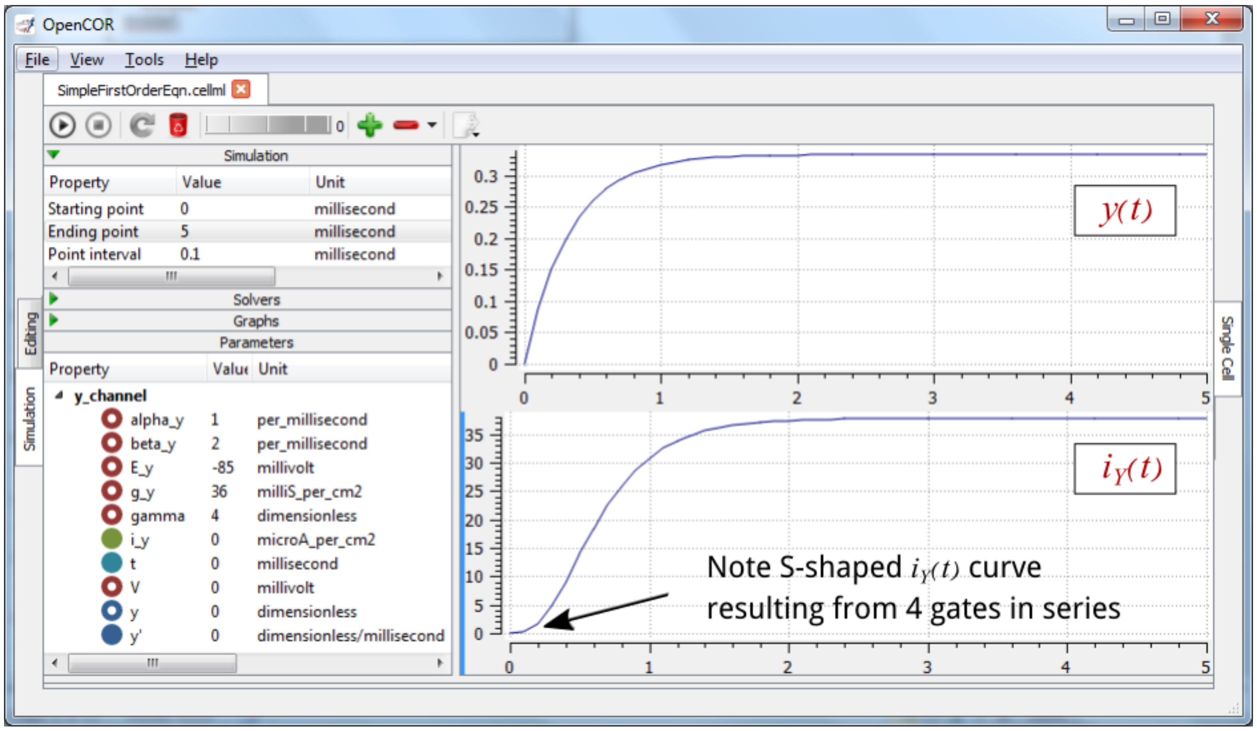
Units for this model, with multiples and fractions, are illustrated in the following *CellML Text* code:

```
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. 2.17.



Simulation

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

Solvers

Graphs

Parameters

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. 2.17: 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.

2.1.8 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

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

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

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

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.

The gating kinetics are described (as before) by

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

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. 2.18) 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 $\frac{\text{dn}}{\text{dt}} \to 0$, $n|_{t=\infty} = n_\infty = \frac{\alpha_n}{\alpha_n + \beta_n}$. 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. 2.18). The steady state current-voltage relation for the channel is illustrated in Fig. 2.19.

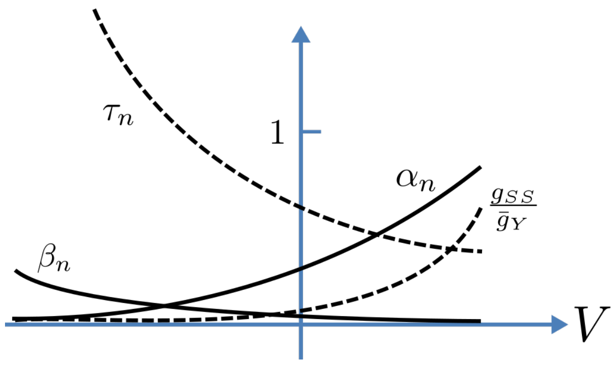


Fig. 2.18: Voltage dependence of rate constants α_n and β_n (ms^{-1}), time constant τ_n (ms) and relative conductance $\frac{g_{SS}}{\bar{g}_Y}$.

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 $v = -(V + 75)$.

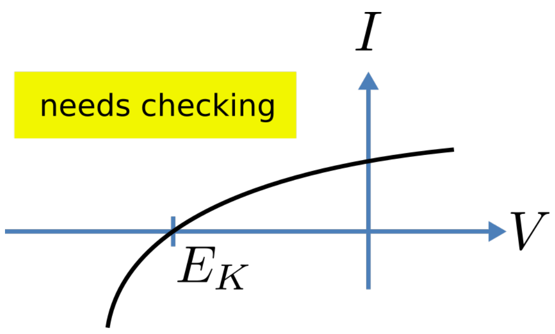
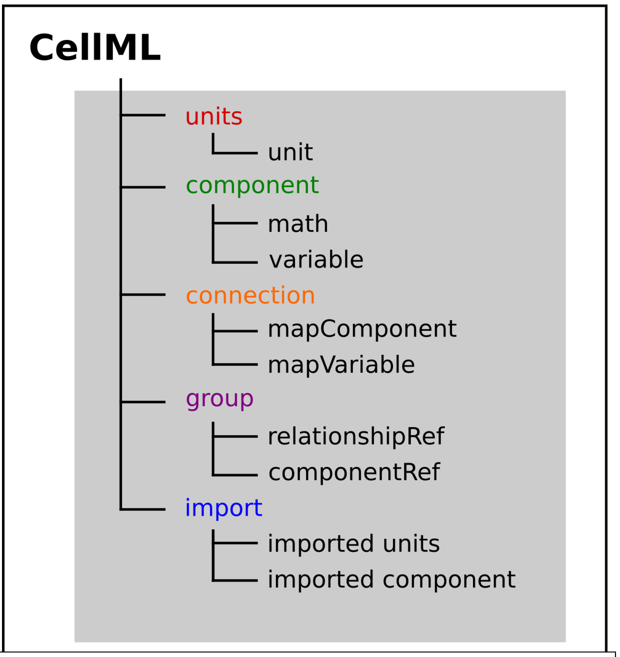


Fig. 2.19: The steady-state current-voltage relation for the potassium channel.

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. 2.20, 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;
   def unit microA_per_cm2 as
```

[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.

```
      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});
      beta_n = 0.125{per_millisec}*exp(V/80{millivolt});
      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;
```

```
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:

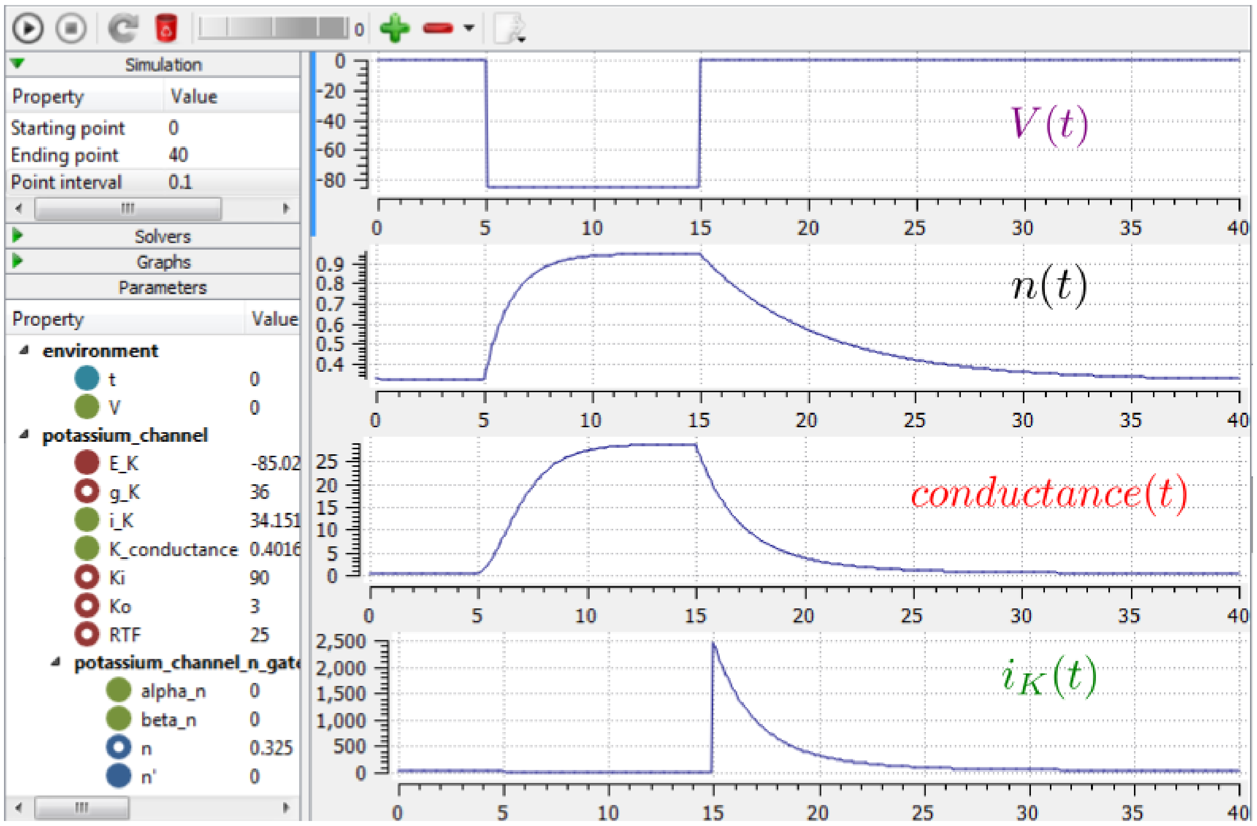
- the event control *select case* which indicates that the voltage is specified to jump from 0 mV to -85 mV at t = 5 ms then back to 0 mV at t = 15 ms. This is only used here in order to test the K channel model; when the potassium_channel component is later imported into a neuron model, **the environment 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 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.

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

[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.



Property	Value
Starting point	0
Ending point	40
Point interval	0.1

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

Fig. 2.21: 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’.

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.

2.1.9 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. 2.22).

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

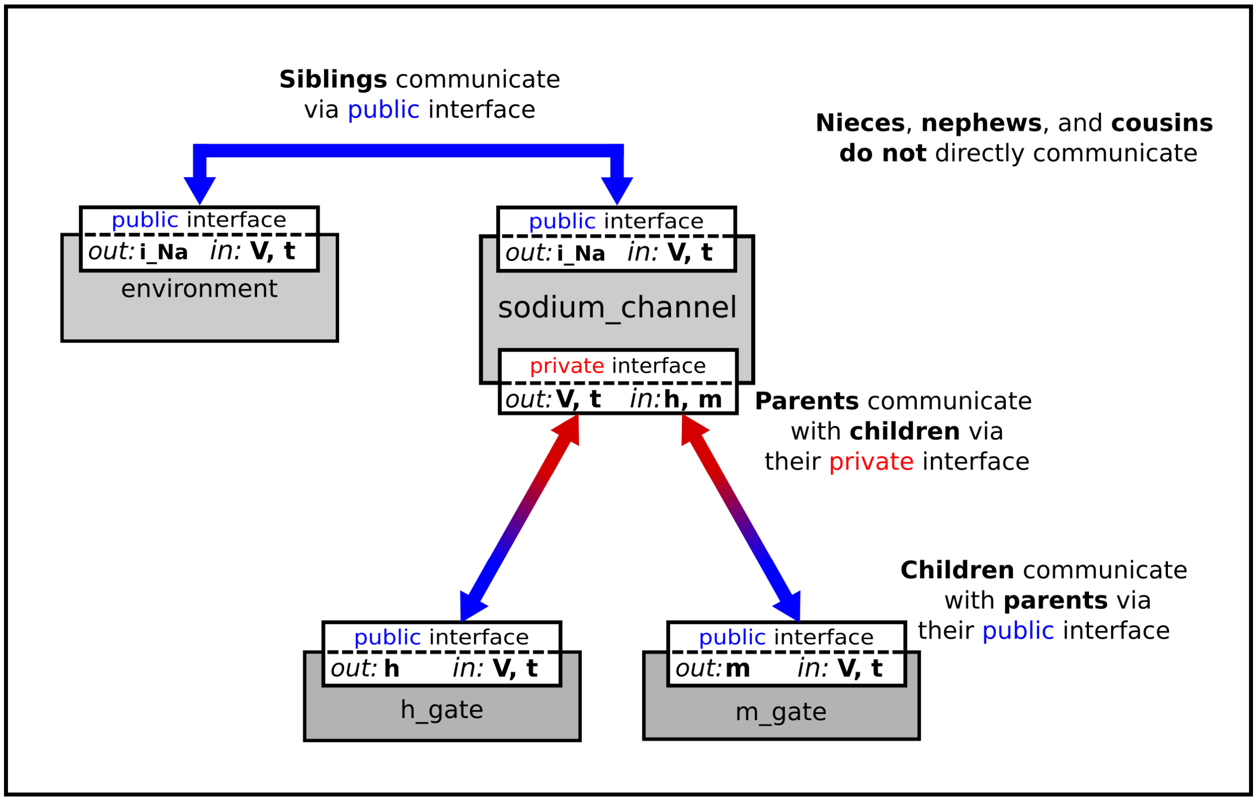

Siblings communicate via public interface
Nieces, nephews, and cousins do not directly communicate
public interface
out: i_Na in: V, t
environment
public interface
out: i_Na in: V, t
sodium_channel
private interface
out: V, t in: h, m
Parents communicate with children via their private interface
Children communicate with parents via their public interface
public interface
out: h in: V, t
h_gate
public interface
out: m in: V, t
m_gate

Fig. 2.22: 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).

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};
   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 E_Na: millivolt {init: 35};
      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 a s
      var V: millivolt {pub: in};
      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. 2.23 with plots $V(t)$, $m(t)$, $h(t)$, $g_{\mathrm{Na}}(t)$ and $i_{\mathrm{Na}}(t)$ for voltage steps from (a) -85mV to -20mV, (b) -85mV to 0mV and (c) -85mV to 20mV. There are several things to note:

1. The kinetics of the m-gate are much faster than the h-gate.
2. 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. 2.18).
3. 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. 2.21 which is only reduced back to zero by stepping the voltage back to its resting value – i.e. *deactivating* it.
4. 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.

5. 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.

2.1.10 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. 2.24), 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

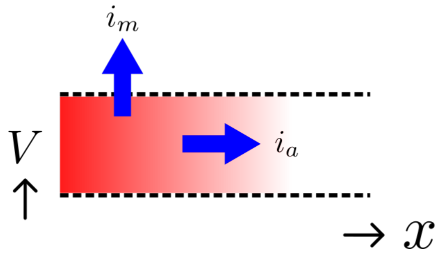


Fig. 2.24: Current flow in a leaky cable.

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

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

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

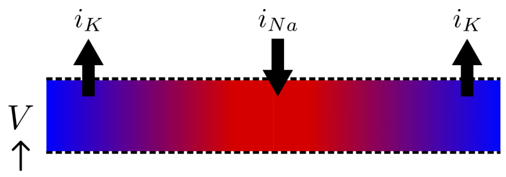
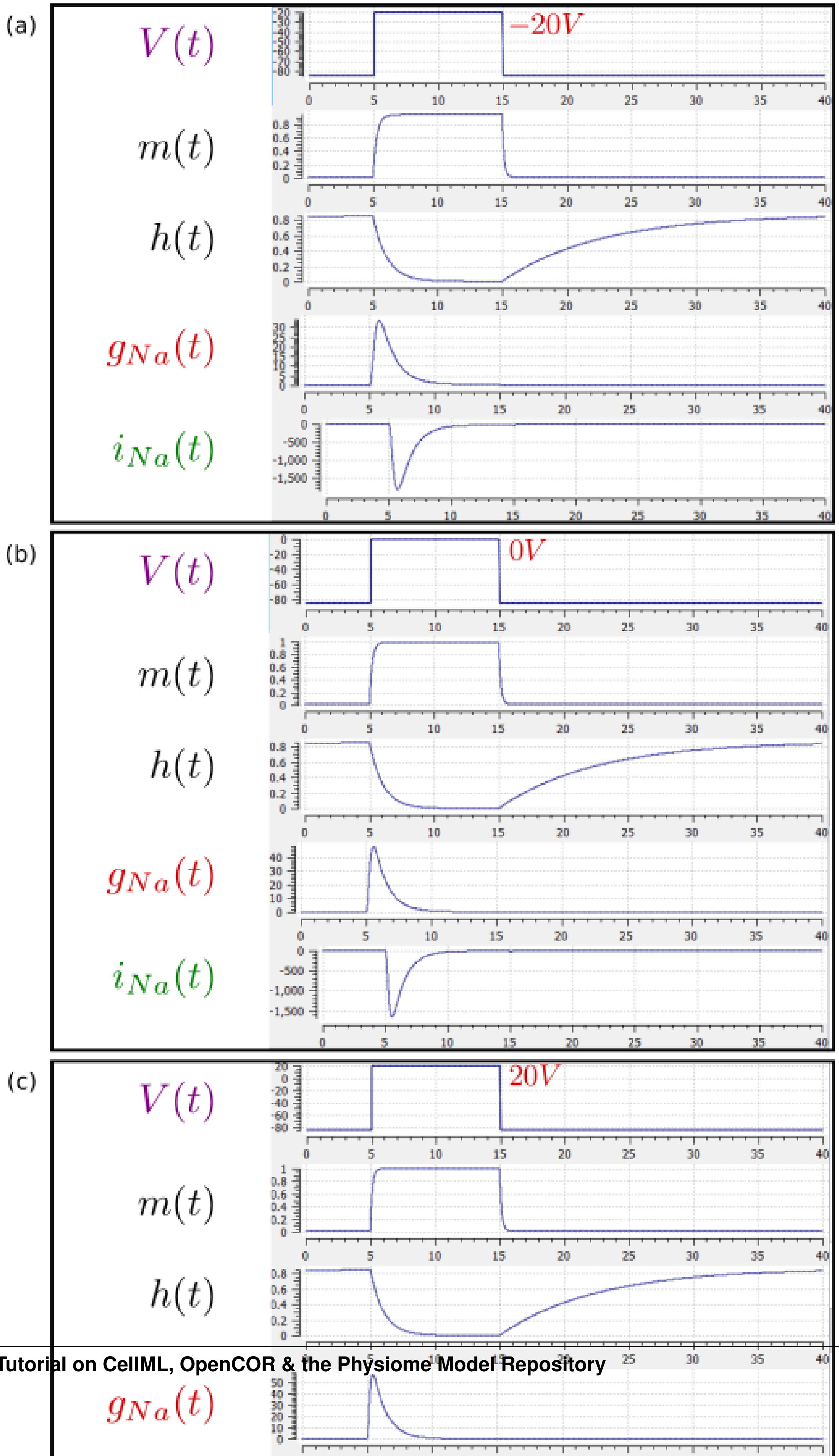


Fig. 2.25: Current flow in a neuron.

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


(a)
$V(t)$
$-20V$
$m(t)$
$h(t)$
$g_{Na}(t)$
$i_{Na}(t)$
(b)
$V(t)$
$0V$
$m(t)$
$h(t)$
$g_{Na}(t)$
$i_{Na}(t)$
(c)
$V(t)$
$20V$
$m(t)$
$h(t)$
$g_{Na}(t)$

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. 2.25). A further small leak current $i_L = g_L\left(V - E_L\right)$ associated with chloride and other ions is also included.

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. 2.26).

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

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

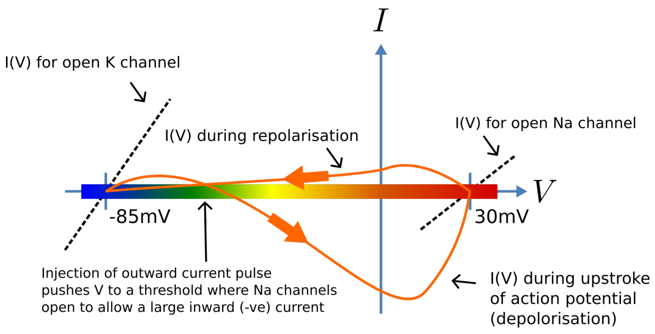


Fig. 2.26: Current-voltage trajectory during an action potential.

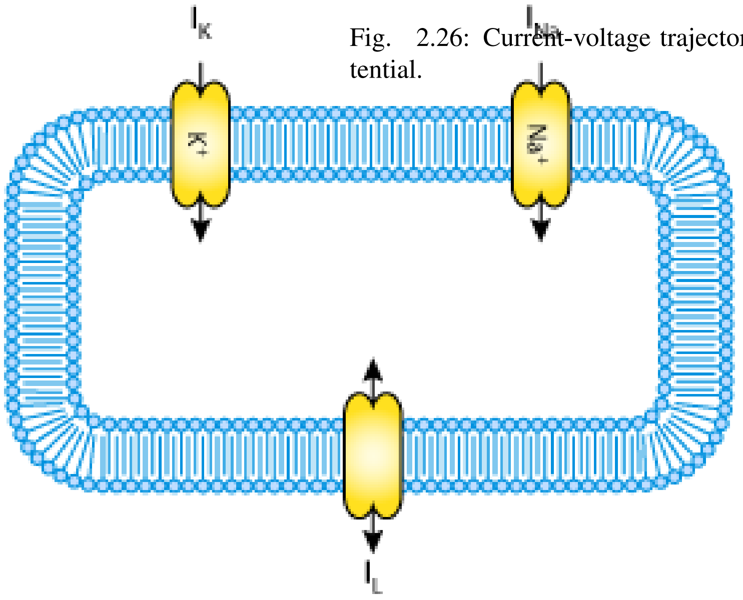


Fig. 2.27: A schematic cell diagram describing the current flows across the cell bilipid membrane that are captured in the Hodgkin-Huxley model. The membrane ion channels are a sodium (Na^+) channel, a potassium (K^+) channel, and a leakage (L) channel (for chloride and other ions) through which the currents I_{Na}, I_K and 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 the new model (in this case the Na and K channel components defined in the previous two sections, together

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

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. 2.28).

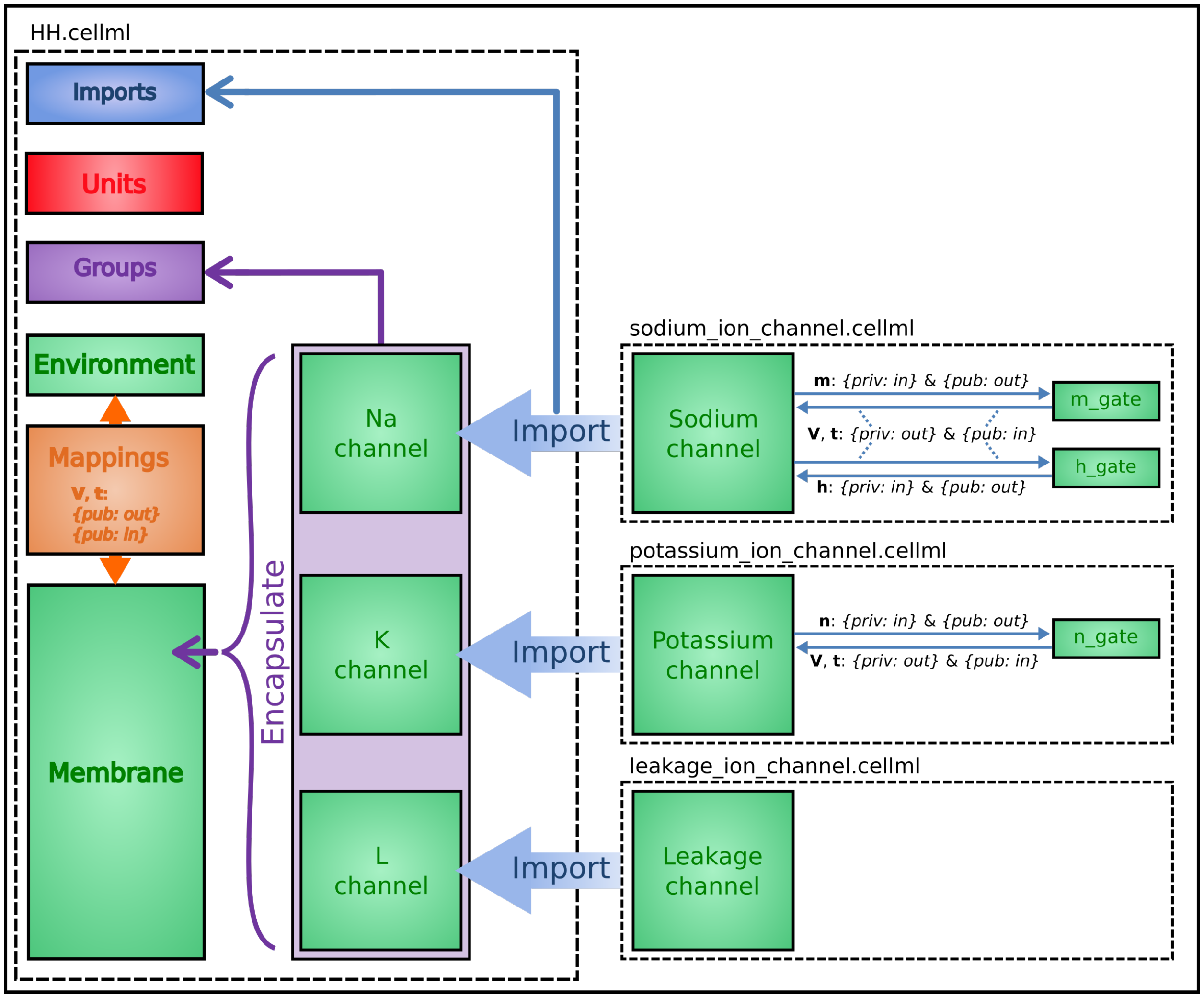

HH.cellml
Imports
Units
Groups
Environment
Mappings
V, t:
{pub: out}
{pub: in}
Membrane
Encapsulate
Na
channel
K
channel
L
channel
Import
Import
Import
sodium_ion_channel.cellml
Sodium
channel
m: {priv: in} & {pub: out}
V, t: {priv: out} & {pub: in}
h: {priv: in} & {pub: out}
m_gate
h_gate
potassium_ion_channel.cellml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

Fig. 2.28: 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
      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;
      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;
```

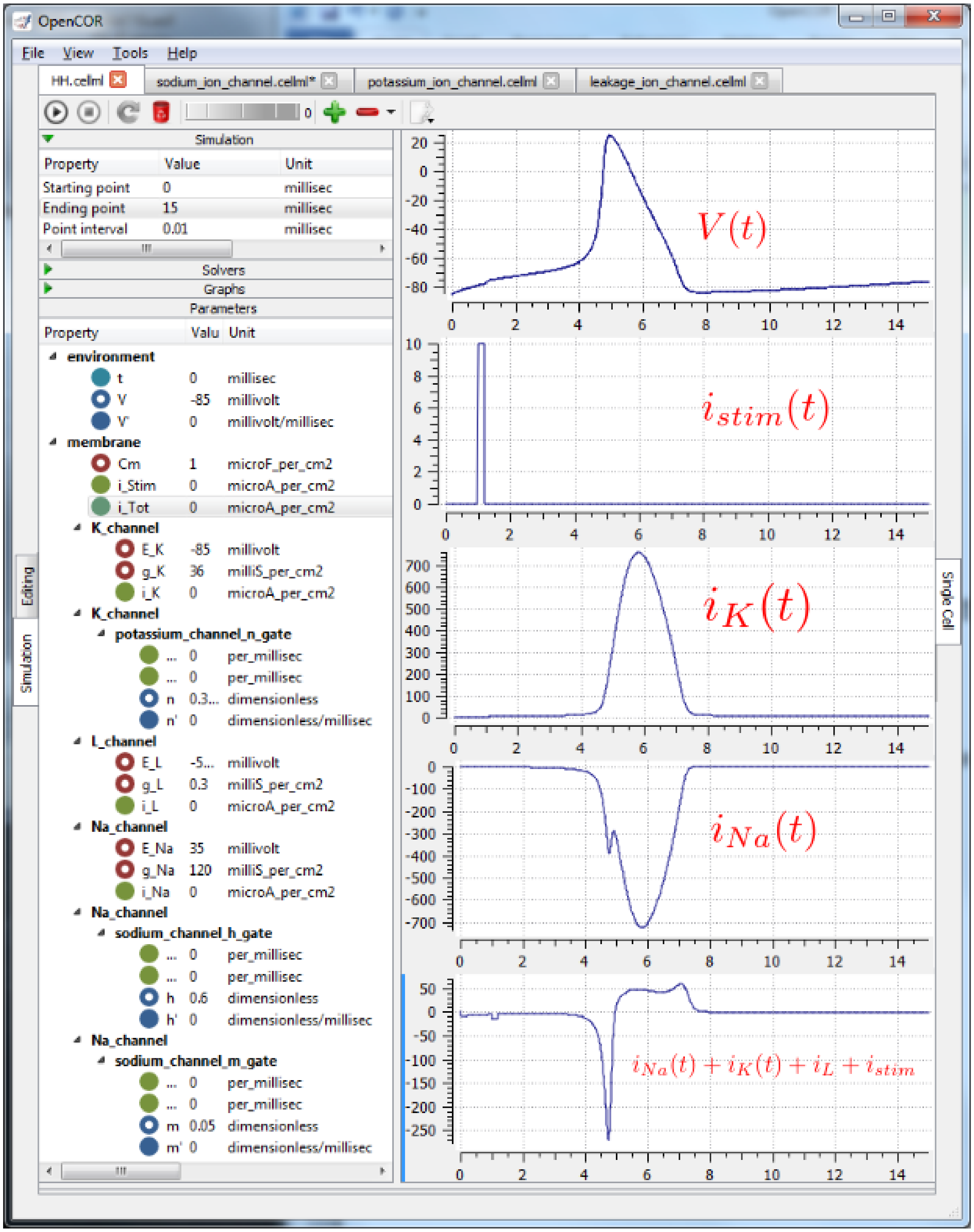
Note that the *CellML Text* code for the potassium channel is *Potassium_ion_channel.cellml* and for the sodium channel is *Sodium_ion_channel.cellml*.

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. 2.29.

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. 2.30 . If you make changes to a model in the *CellML Text* view,



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

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. 2.29: 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.

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.

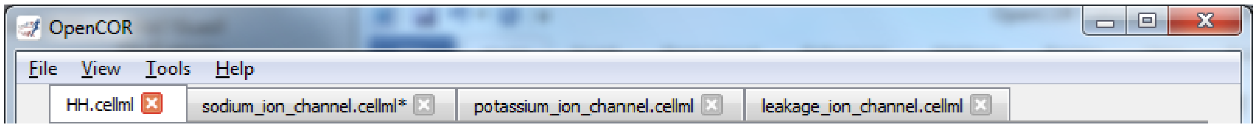

OpenCOR
File
View
Tools
Help
HH.cellml
sodium_ion_channel.cellml*
potassium_ion_channel.cellml
leakage_ion_channel.cellml

Fig. 2.30: The HH.cellml model and its three sub-models are available under separate tabs in OpenCOR.

2.1.11 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. 2.31(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. 2.31(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)

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

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

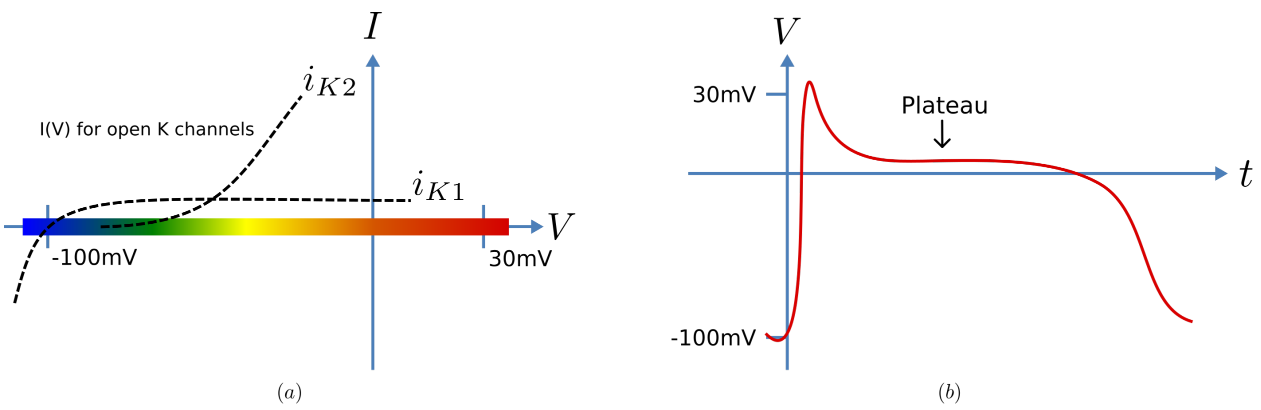


Fig. 2.31: Current-voltage relations (a) around the equilibrium potentials for the potassium and sodium channels in cardiac cells. The sodium channel is similar to the one in neurons but the two potassium channels have an inward rectifying property that stops leakage of potassium ions out of the cell 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. 2.32 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).

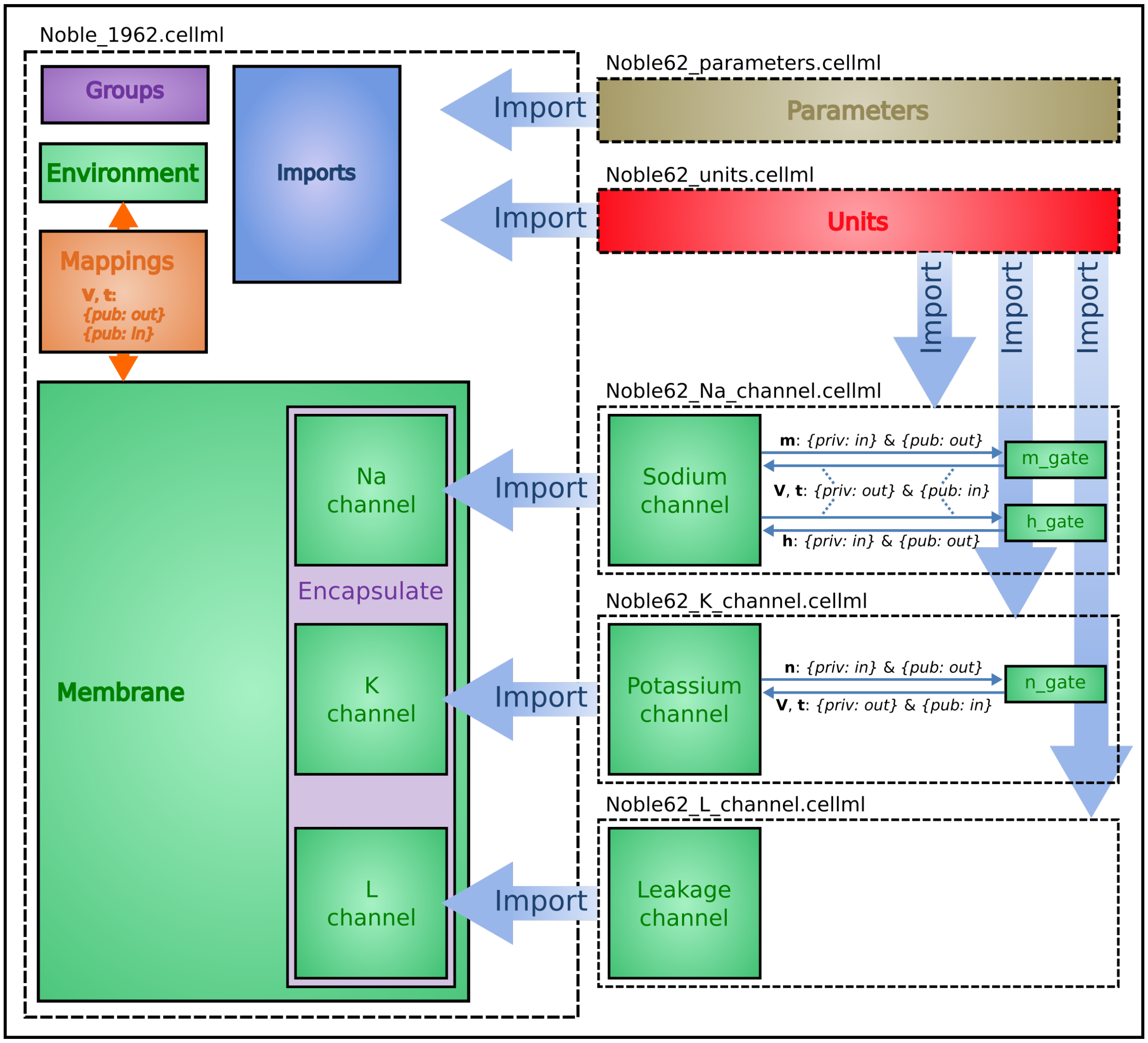


Fig. 2.32: Overall structure of the Noble62 CellML model showing the encapsulation hierarchy (purple), the CellML model imports (blue) and the other key parts (units, components & mappings) of the top level CellML model. Note that the overall structure of the Noble62 model differs from that of the earlier 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. 2.32. Interpretation of the model outputs is given in the Fig. 2.32 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};
      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};
   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};
      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};
      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 units, parameters and model components are defined in separate files that are imported into a composite model.

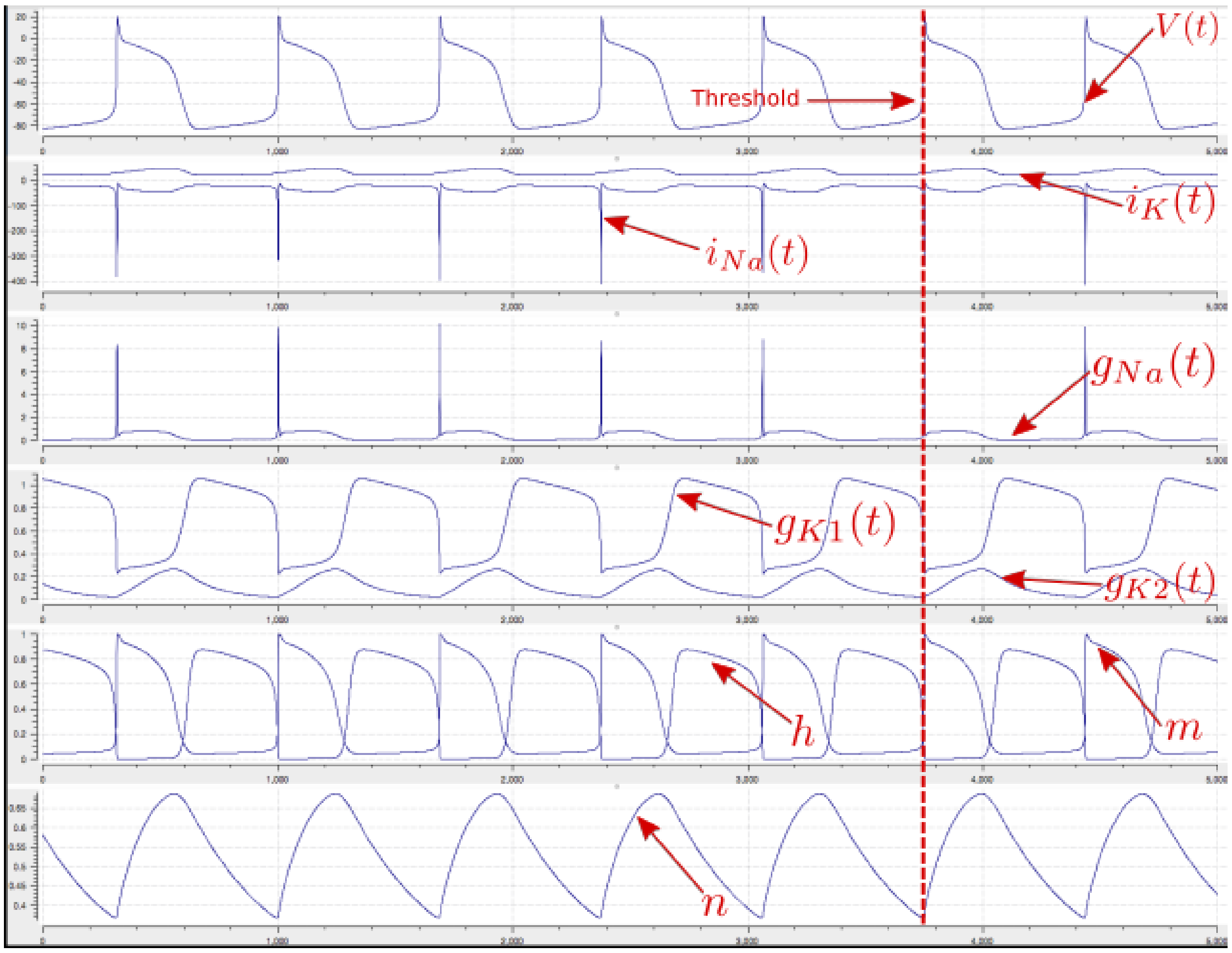


Fig. 2.33: Output from the Noble62 model (OpenCOR link). Top panel is $V(t)$, the cardiac action potential. The next panel has the two membrane ion channel currents $i_{\text{Na}}(t)$ and $i_K(t)$. Note that $i_{\text{Na}}(t)$ has a very brief downward (i.e. inward current) spike that is triggered when the membrane voltage reaches about -70mV. This is caused by the huge increase in sodium channel conductance $g_{\text{Na}}(t)$ 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.

2.1.12 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.

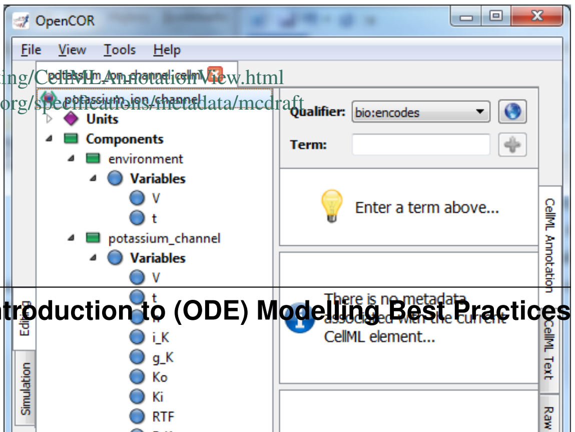
CellML models therefore contain two parts, one dealing with **syntax** (the MathML definition of the models together with the structure of components, connections, groups, units, etc) as discussed in previous sections, and one dealing with **semantics** (the meanings of the terms used in the models) 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 (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.



[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

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

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. 2.34). 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. 2.35 shows the information displayed about this qualifier.

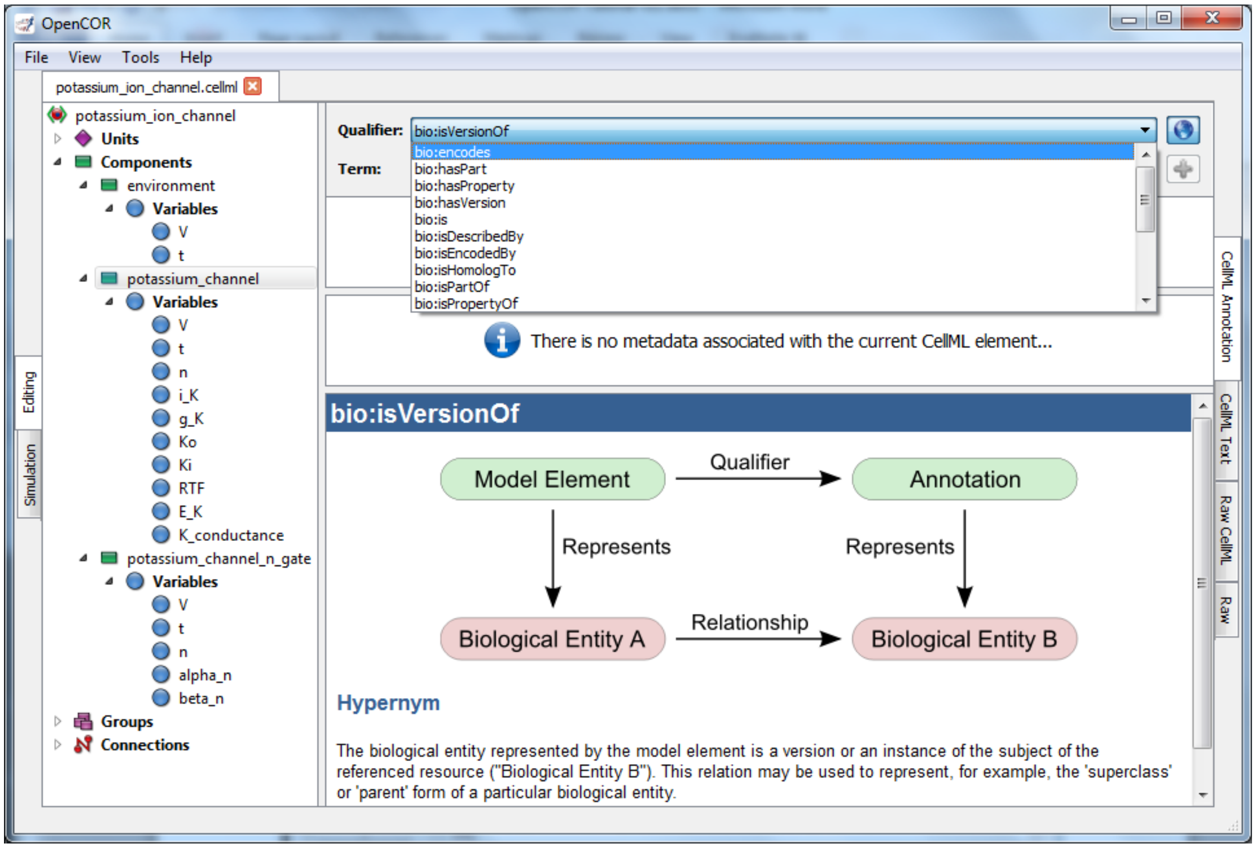
Now you need to retrieve some possible ontological terms to describe the *potassium_channel* component. For this you must enter a search term, which in our case is 'potassium channel' (note that regular expressions are supported [7]). This returns 24 possible ontological terms as shown in Fig. 2.36. 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. 2.37).

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. 2.36. 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. 2.38, 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;
```

[7] http://en.wikipedia.org/wiki/Regular_expression



Qualifier: bio:isVersionOf

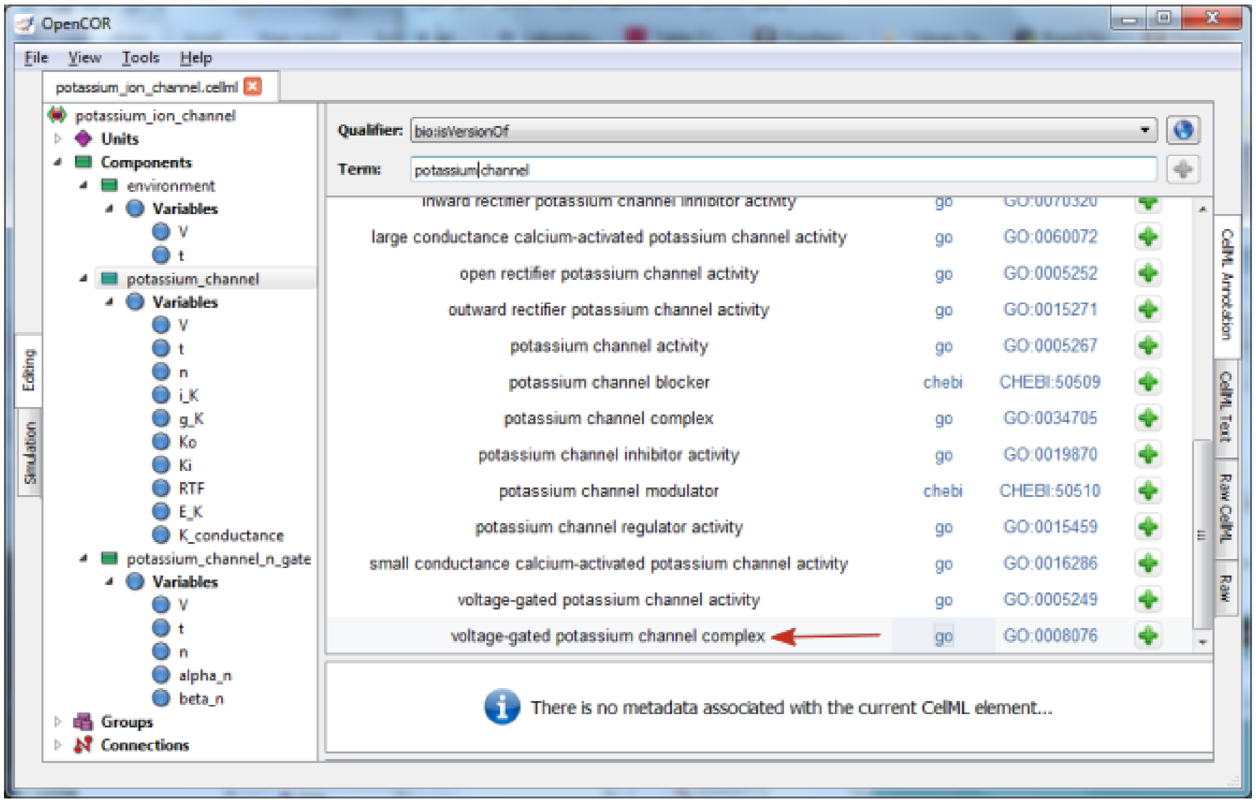
There is no metadata associated with the current CellML element...

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. 2.35: 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.



potassium_ion_channel
- Units
- Components
  - environment
    - Variables
      - V
      - t
  - potassium_channel
    - Variables
      - V
      - t
      - n
      - i_K
      - g_K
      - Ko
      - Ki
      - RTF
      - E_K
      - K_conductance
  - potassium_channel_n_gate
    - Variables
      - V
      - t
      - n
      - alpha_n
      - beta_n
- Groups
- Connections

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. 2.36: The ontological terms listed when ‘potassium channel’ is entered into the search box next to *Term*.

```
    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;
```

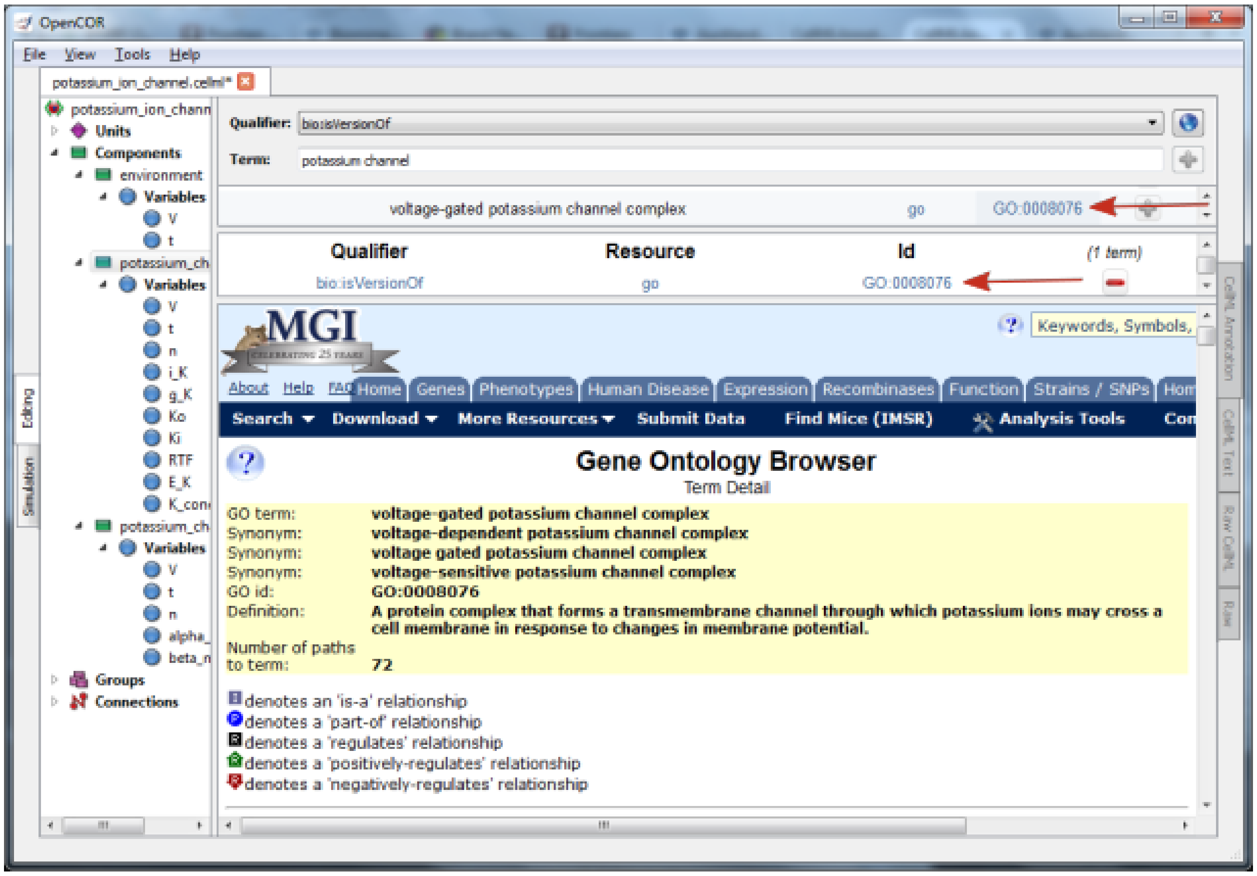
When saved (the *CellML Annotation* tag will appear un-grayed), the result of these annotations is to add metadata to the CellML file. If you switch to the *CellML Text* view you will see that the elements that have been annotated appear with ID numbers, as shown above. These point to the corresponding 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.

2.1.13 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

[8] This is a project being carried out at the University of Washington, Seattle, using an annotation tool called SEMGEN (...).



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. 2.37: 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.

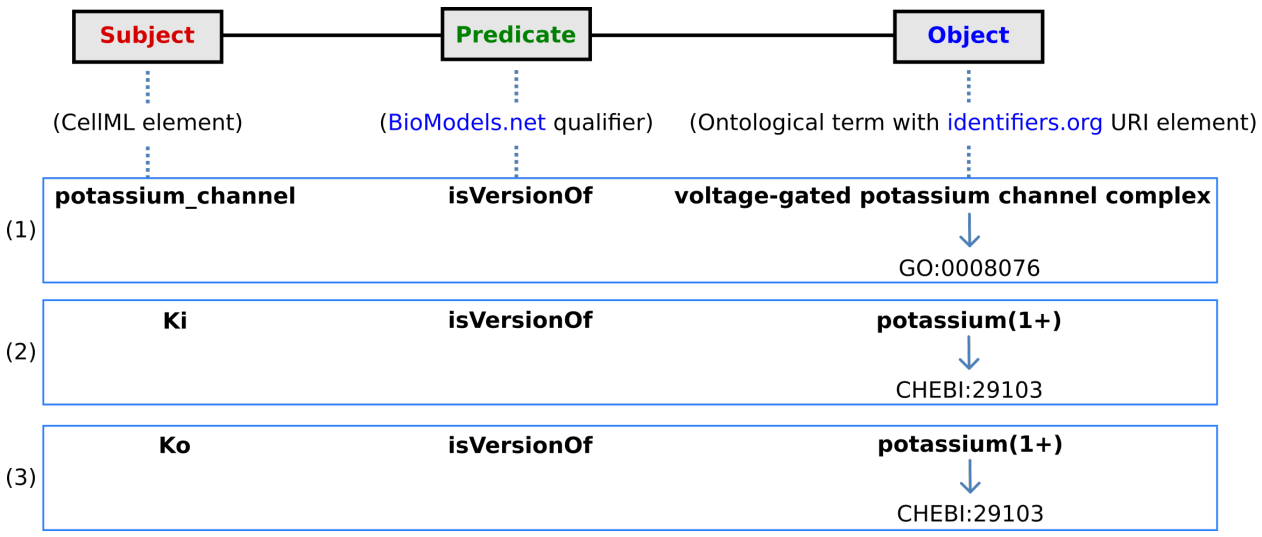


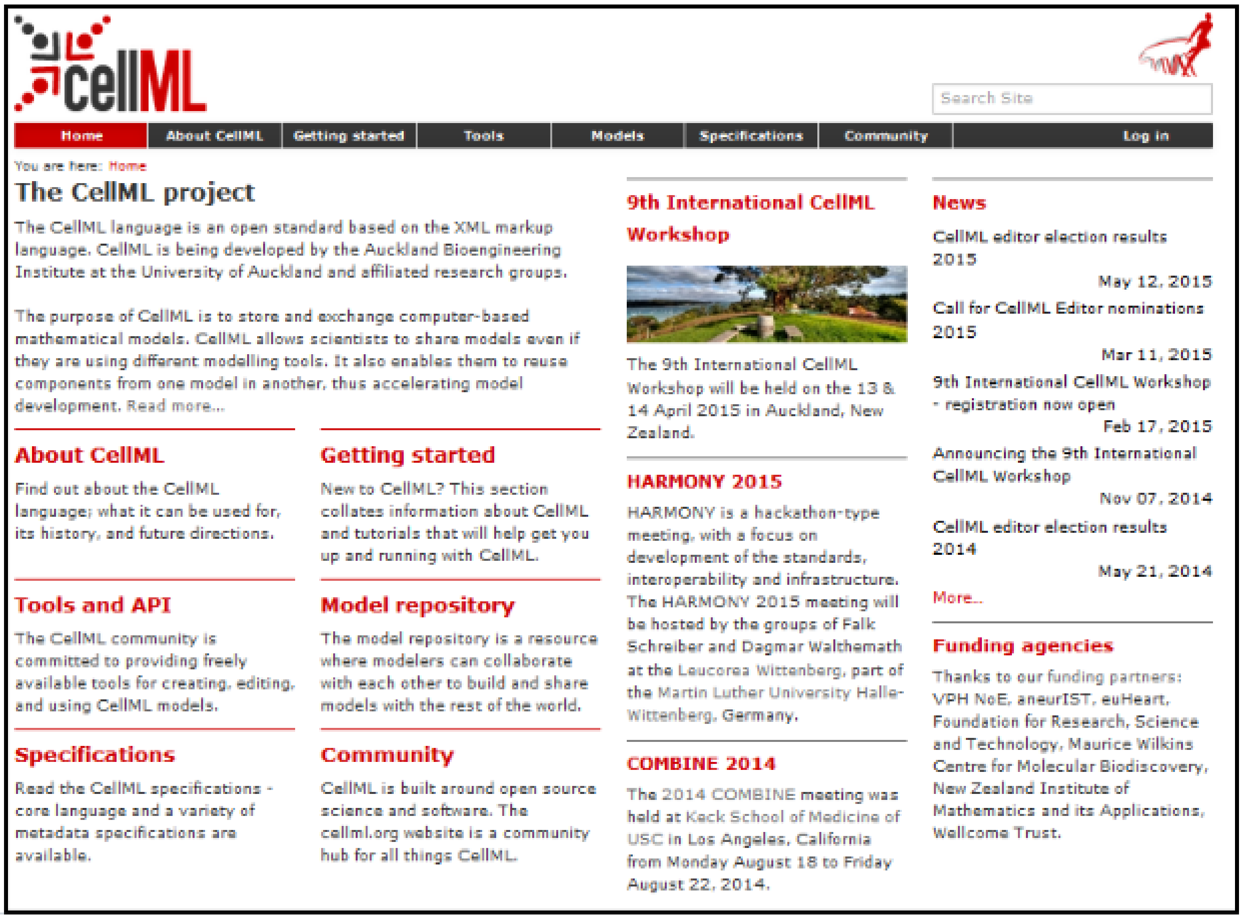
Fig. 2.38: The RDF triple used in CellML metadata to link a CellML element (component or variable) with an ontological term from one of the five ontologies accessed via identifiers.org, using a predicate qualifier from BioModels.net. The three examples of annotated CellML model elements shown are for (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).

are approximately 640 public workspaces and another 200 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.

The CellML project website and the CellML Physiome Model Repository are shown in Fig. 2.39 and Fig. 2.40.

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)

Browse by category



The CellML project

The CellML language is an open standard based on the XML markup language. CellML is being developed by the Auckland Bioengineering Institute at the University of Auckland and affiliated research groups.

The purpose of CellML is to store and exchange computer-based mathematical models. CellML allows scientists to share models even if they are using different modelling tools. It also enables them to reuse components from one model in another, thus accelerating model development. Read more...

About CellML

Find out about the CellML language; what it can be used for, its history, and future directions.

Getting started

New to CellML? This section collates information about CellML and tutorials that will help get you up and running with CellML.

Tools and API

The CellML community is committed to providing freely available tools for creating, editing, and using CellML models.

Model repository

The model repository is a resource where modelers can collaborate with each other to build and share models with the rest of the world.

Specifications

Read the CellML specifications - core language and a variety of metadata specifications are available.

Community

CellML is built around open source science and software. The cellml.org website is a community hub for all things CellML.

9th International CellML Workshop

The 9th International CellML Workshop will be held on the 13 & 14 April 2015 in Auckland, New Zealand.

HARMONY 2015

HARMONY is a hackathon-type meeting, with a focus on development of the standards, interoperability and infrastructure. The HARMONY 2015 meeting will be hosted by the groups of Falk Schreiber and Dagmar Walthemath at the Leucorea Wittenberg, part of the Martin Luther University Halle-Wittenberg, Germany.

COMBINE 2014

The 2014 COMBINE meeting was held at Keck School of Medicine of USC in Los Angeles, California from Monday August 18 to Friday August 22, 2014.

News

CellML editor election results 2015
May 12, 2015

Call for CellML Editor nominations 2015
Mar 11, 2015

9th International CellML Workshop - registration now open
Feb 17, 2015

Announcing the 9th International CellML Workshop
Nov 07, 2014

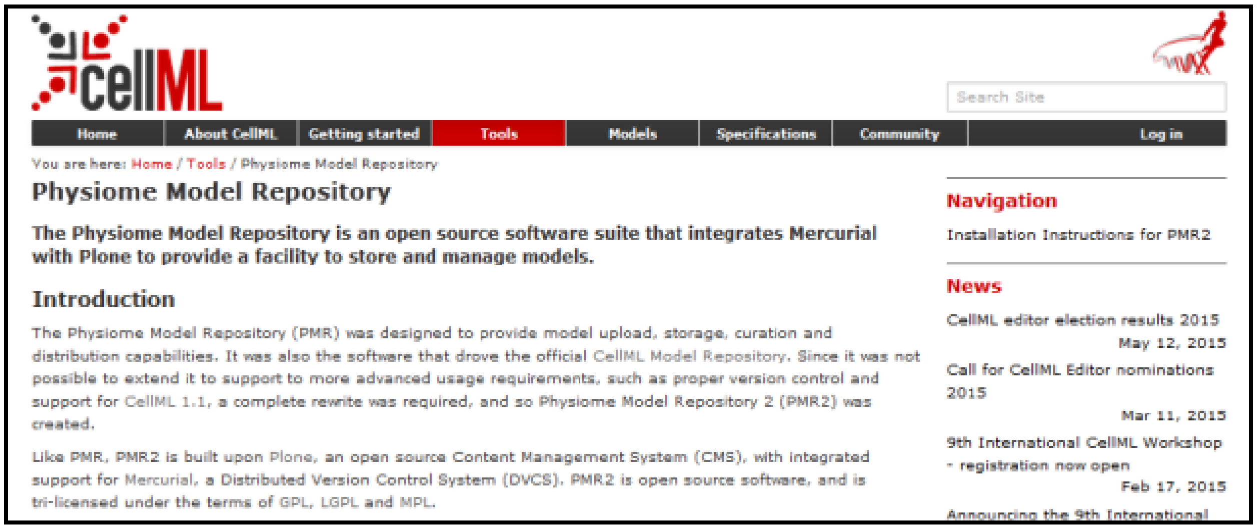
CellML editor election results 2014
May 21, 2014

More...

Funding agencies

Thanks to our funding partners: VPH NoE, aneurIST, euHeart, Foundation for Research, Science and Technology, Maurice Wilkins Centre for Molecular Biodiscovery, New Zealand Institute of Mathematics and its Applications, Wellcome Trust.

Fig. 2.39: The website for the CellML project at www.cellml.org.



Physiome Model Repository

The Physiome Model Repository is an open source software suite that integrates Mercurial with Plone to provide a facility to store and manage models.

Introduction

The Physiome Model Repository (PMR) was designed to provide model upload, storage, curation and distribution capabilities. It was also the software that drove the official CellML Model Repository. Since it was not possible to extend it to support to more advanced usage requirements, such as proper version control and support for CellML 1.1, a complete rewrite was required, and so Physiome Model Repository 2 (PMR2) was created.

Like PMR, PMR2 is built upon Plone, an open source Content Management System (CMS), with integrated support for Mercurial, a Distributed Version Control System (DVCS). PMR2 is open source software, and is tri-licensed under the terms of GPL, LGPL and MPL.

Navigation

Installation Instructions for PMR2

News

CellML editor election results 2015
May 12, 2015

Call for CellML Editor nominations 2015
Mar 11, 2015

9th International CellML Workshop - registration now open
Feb 17, 2015

Fig. 2.40: The website for the Physiome Model Repository project at www.cellml.org/tools/pmr.

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 *Electrophysiology*. This opens a web page (Fig. 2.41) using a 32 character string that has been randomly generated as the ID for the exposure page for that model.

Note that the string is still unique even with only 5 characters:

e.g. https://models.physiomeproject.org/exposure/f4b71/hund_rudy_2004_a.cellml/view

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. 2.42. 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 CellMLmodel on the lower 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.

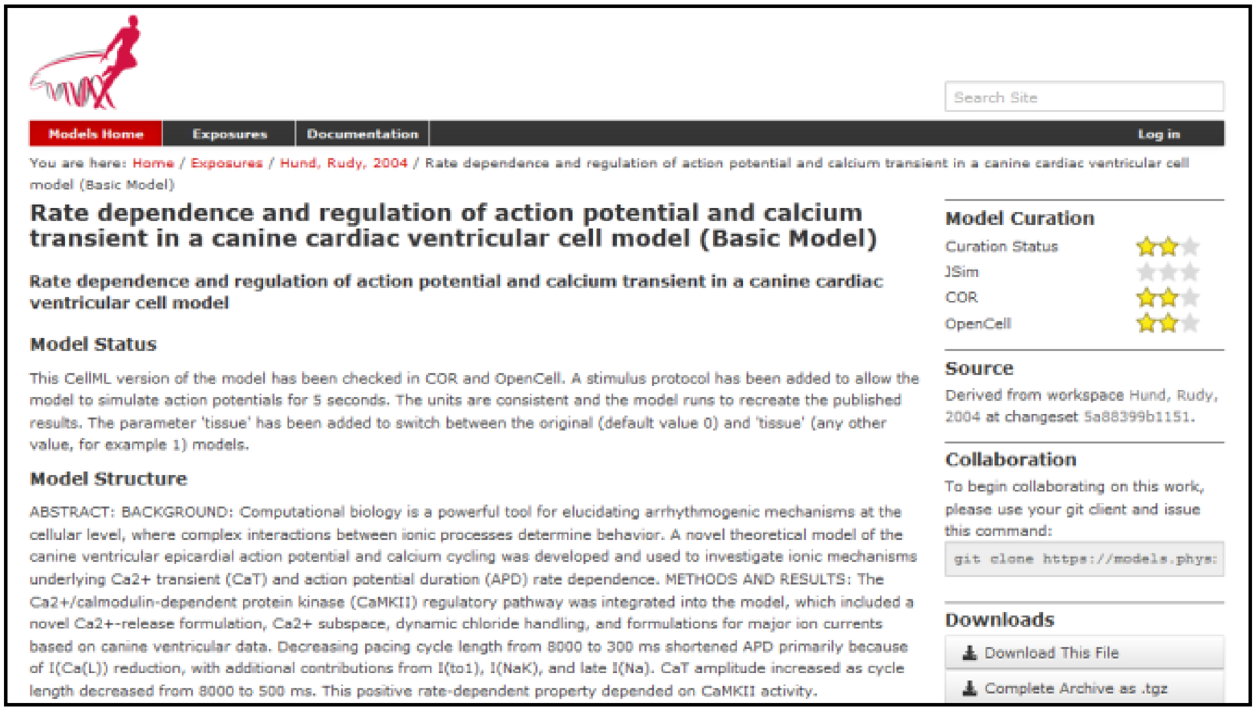
Model Curation - Provides the curation status of the model. Note: this is soon to be updated.

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

Generated Code - Creates code (C, C-IDA, F77, MATLAB or Python) for the model.

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

[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.



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.

Model Curation

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

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.phys:
```

Downloads

Download This File

Complete Archive as .tgz

Fig. 2.41: The Physiome Model Repository exposure page for the basic Hund-Rudy 2004 model.

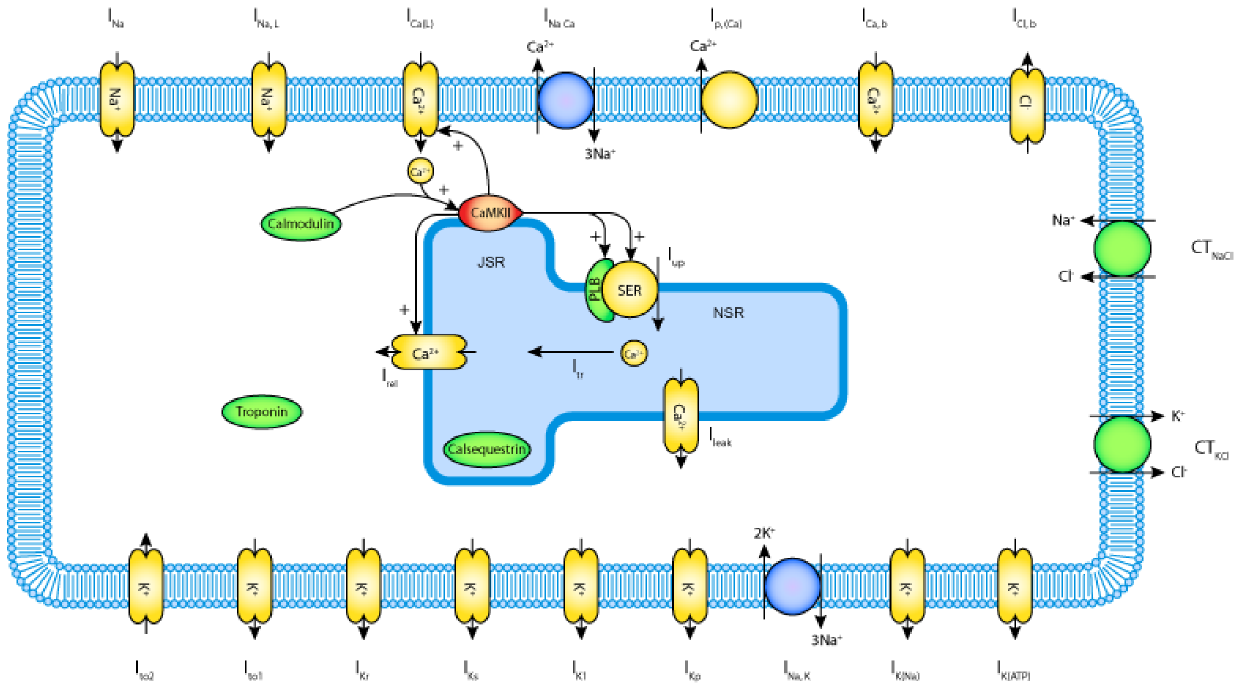


Fig. 2.42: A diagrammatic representation of the Hund-Rudy 2004 model.

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

Simulate using OpenCell - This will be OpenCOR once the SED-ML API is included in OpenCOR.

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. 2.43) 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]*.

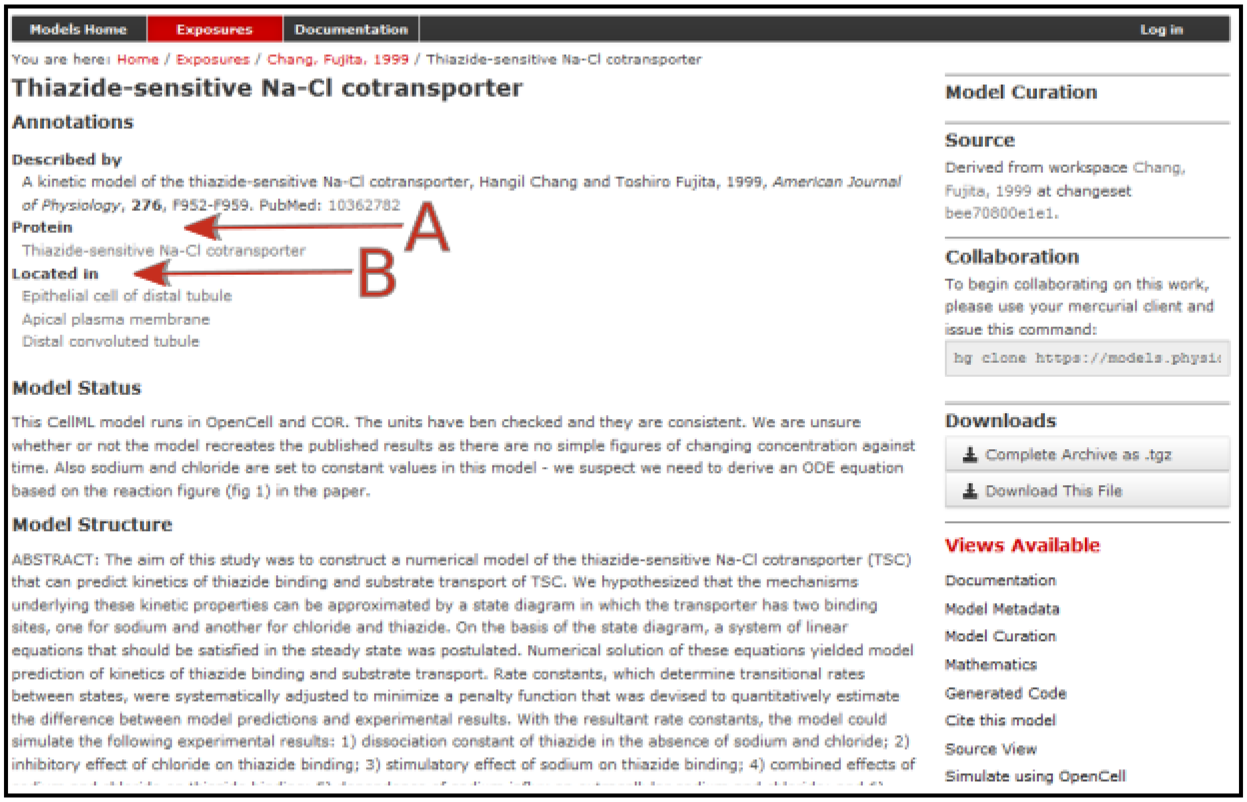
2.1.14 SED-ML, functional curation and Web Lab

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. 2.33).

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. 2.44.

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. 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

[2] https://creativecommons.org/licenses/by/3.0/
[1] The ‘Simulation Experiment Description Markup Language’: sed-ml.org
[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.



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

Views Available

Documentation

Model Metadata

Model Curation

Mathematics

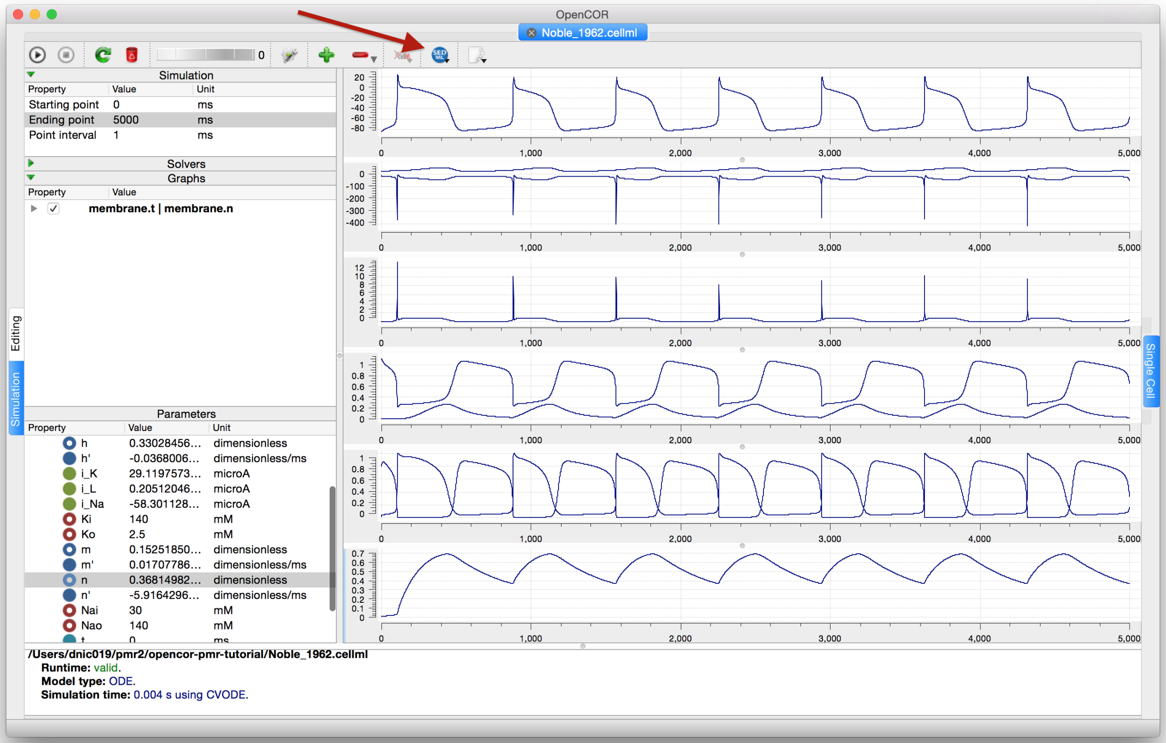
Generated Code

Cite this model

Source View

Simulate using OpenCell

Fig. 2.43: 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.



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. 2.44: 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.

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. 2.45 adapted from *[CJ15]*).

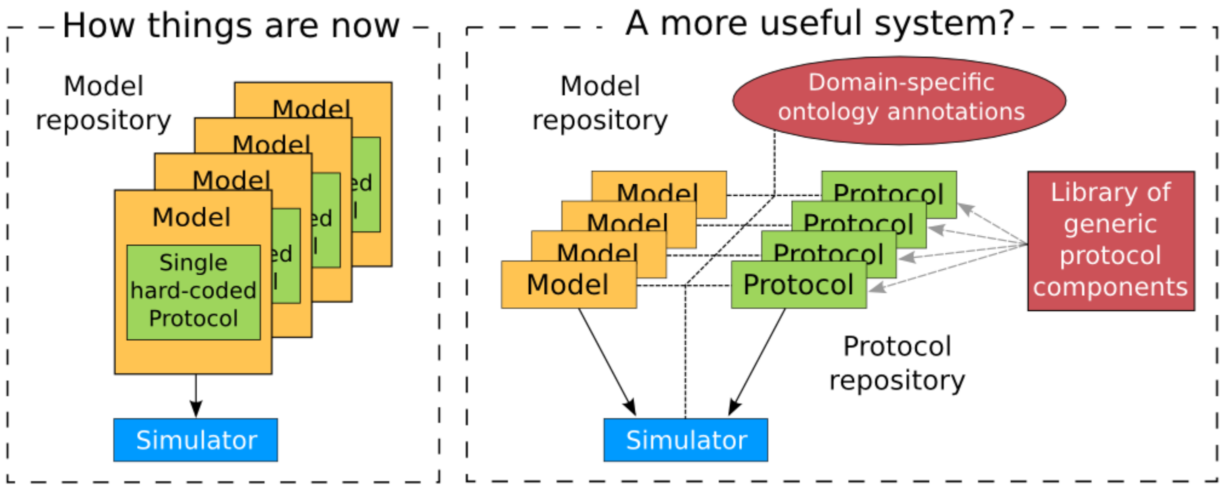


Fig. 2.45: A schematic of the way we organise model and protocol descriptions. Web Lab provides an interface to a Model/Protocol Simulator, storing and displaying the results for cardiac electrophysiology models.

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.

2.1.15 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.
- Operating system: OS X Yosemite 10.10.3.

[1] www.mathworks.com/products/matlab

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.

2.1.16 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

2.1.17 References

See https://www.cellml.org/getting-started/tutorials.

Latest version rendered at: http://tutorial-on-cellml-opencor-and-pmr.readthedocs.org/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.

CHAPTER 3

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

3.1 MAP Client

Todo

- Fill out a description of how to get MAP Client (preferably linking to MAP Client docs) and the required plugins for our tasks.
- possibly also a summary of the downloads required for the tasks, but they're probably better off in the actual documentation.

3.2 OpenCOR

Instructions for obtaining and using OpenCOR are covered in the *Tutorial on CellML, OpenCOR & the Physiome Model Repository*.

3.3 OpenCMISS-Neon

Todo

Description of how to get Neon - preferably linking to Neon docs, but they probably don't exist for this instance of the DTP CP module.

Todo

What other tools should be mentioned?

CHAPTER 4

DTP Computational Physiology to do list

4.1 General

Todo

- any general todo's go here.
- make sure the todo's are turned off in the config?

4.2 Within sections

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/v2016.03/source/opencor-tutorial/source/index.rst, line 73.)

Todo

- Fill out a description of how to get MAP Client (preferably linking to MAP Client docs) and the required plugins for our tasks.

- possibly also a summary of the downloads required for the tasks, but they're probably better off in the actual documentation.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dtp-compphys/checkouts/v2016.03/source/software.rst, line 13.)

Todo

Description of how to get Neon - preferably linking to Neon docs, but they probably don't exist for this instance of the DTP CP module.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dtp-compphys/checkouts/v2016.03/source/software.rst, line 26.)

Todo

What other tools should be mentioned?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dtp-compphys/checkouts/v2016.03/source/software.rst, line 31.)

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/v2016.03/source/todolist.rst, line 9.)

Bibliography

[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 pace-maker 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.