**Simple Tutorial**

**for**

**Circulatory Autogen**

**Written by Finbar Argus**

**29/01/2024**

**THIS TUTORIAL IS A WORK IN PROGRESS, IF YOU HAVE ANY CORRECTIONS PLEASE SEND THEM TO finbar.argus@auckland.ac.nz**

1. **Initializing and startup:**

The main project must be run on a Linux operating system, Although you can install a virtual linux machine in windows. One of these virtual machines is VirtualBox Oracle which you can download from the below link:

<https://www.virtualbox.org/>

Also, you can download the latest version of Ubuntu using this link:

<https://ubuntu.com/download/desktop>

Then, you need to install the OpenCOR software from this link:

<https://opencor.ws/downloads/index.html>

And there is a good tutorial PDF that includes many examples, which you can download here:

[Download openCOR tutorial](https://tutorial-on-cellml-opencor-and-pmr.readthedocs.io/en/latest/_downloads/d271cfcef7e288704c61320e64d77e2d/OpenCOR-Tutorial-v17.pdf)

For cloning or pulling the project, the latest version of the code is available on GitHub:

<https://github.com/FinbarArgus/circulatory_autogen>

**Note:**

If you have not worked with git and GitHub, firstly download and install git, and then open the terminal and navigate to a directory where you want the repository to be. Then write these commands to clone the project on your pc:

*git clone https://github.com/FinbarArgus/circulatory\_autogen*

*git remote add origin https://github.com/FinbarArgus/circulatory\_autogen*

If you want to develop the code, then create a fork of the above repo in github, then do the following lines instead of the above:

*git clone https://github.com/YourUsername/circulatory\_autogen*

*git remote add origin https://github.com/YourUsername/circulatory\_autogen*

*git remote add upstream https://github.com/FinbarArgus/circulatory\_autogen*

1. **Directory Definition:**

In this tutorial, we use one particular directory for our project, but it can be different on every computer, So the base directory is defined as “MainDir” in all parts. For example, on our computer, this directory is as below:

MainDir: Home/…/Desktop/

The project directory ([project\_dir]) is the directory we cloned from the GitHub Circulatory\_autogen project to our computer. For example, on the dir may be:

[project\_dir]: Home/…/Desktop/Project/Circulatory\_autogen

Also, OpenCOR files directory is needed for opening the project and installing the python and pythonshell, and we show with OpenCOR\_dir, which is below on our pc:

OpenCOR\_Dir: Home/…/Desktop/OpenCOR

**Note: If running on the ABI HPC, you can use the OpenCOR at the path:**  
***/hpc/farg967/OpenCOR-2022-05-23-Linux/***  
***And ignore the below installation steps.***

1. **Python and libraries installation for this project:**

To run openCOR, you need to use the Python version with openCOR. This required python is available at the below address:

Python installation file : OpenCOR\_dir/python/bin

Then you should add the following path to let the system know where to look for libs when installing python packages:

*Cd [OpenCOR\_dir]/python/bin*

*export LD\_LIBRARY\_PATH=[OpenCOR\_dir]/lib*

Now, you should install the python libraries which are essential to this project: An up to date list of required libraries are detailed in the README.md file. To install packages in OpenCOR’s version of python, navigate to [OpenCOR\_dir]/python/bin directory then do:

*./python -m pip install <packagename>*

**Note:**

For parameter identification, you need to use these libraries:

**mpi4py, scikit-optimize, emcee, corner, schwimmbad, tqdm**

**IMPORTANT:** installing mpi4py requires mpi to be available. Therefore, the following line may be required to install the mpi software on your computer.

***sudo apt install libopenmpi-dev***

**Sometime notify doesn’t exist specific package, at this moment, please follow as steps to re-install packages:**

***cd ~/circulatory\_autogen/user\_run\_files***

[*OpenCOR\_dir]/pythonshell -m pip install package\_name*

1. **Software Outline:**

The Circulatory\_autogen project contains five main folders as presented below:

* **generated\_models: I**ncludes the generated code for the models that have been automatically generated. It also contains the generated models with parameters that have been fit with the parameter identification code. These models can be run in OpenCOR or through OpenCOR’s version of Python.
* **param\_id\_output:** Includes all of the outputs from the parameter identification simulations, including predicted parameter values, minimum costs, standard deviations of parameters (if doing MCMC) and plots of the fitting results and parameter distributions.
* **resources:** Contains the config csv files that the user defines to construct the model that will be generated and to prescrive it’s parameters)
* **src:** Containts the source code for autogeneration, parameter id, and other utilities.)
* **user\_run\_files: I**ncludes essential run files for the user and the user\_inputs.yaml file, which is the main config file for the run settings.

Furthermore, there is a README.md file that has more brief (and possibly up to date) instructions than this tutorial for running the project.

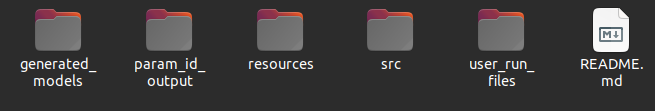


Figure 1

This section shows how to generate and simulate your desired model. Several examples are used to show the generality of the circulatory\_autogen software.

The steps for the autogeneration are:

**1)** Making the **vessel\_array** and **parameters** files in CSV format for the intended model. Those files should be added to the “[project\_dir]/resources” directory, and You must pay attention to the file’s name. If you want to define your own resources\_dir, you can add a “resources\_dir:” entry in “[project\_dir]/user\_run\_files/user\_inputs.yaml” which will override the default “[project\_dir]/resources” directory

Proper names of vessel and parameters files are **“[model name]\_vessel\_array.csv”** and **“[model name]\_parameters.csv”,** respectively. For example, If the model’s name is 3compartment, the mentioned files are:

**3compartment\_vessel\_array.csv**

**3compartment\_parameters.csv**

**Note:** To create a new vessel\_array file, follow the next section, “**creating a new model.**”

**2)** Go to the "[project\_dir]/user\_run\_files" and open the "user\_inputs.yaml" file to edit.

You can use “gedit”, “nano”, or your editor of choice (hopefully vim ;)) to edit the file.

**3)** you should add the names of the intended vessel and parameters files to the “user\_inputs.yaml”. file\_prefix is the name of your model. subsequently, input\_param\_file should be equal to “[model\_name]\_parameters.csv”. I.e for [model\_name]=physiological

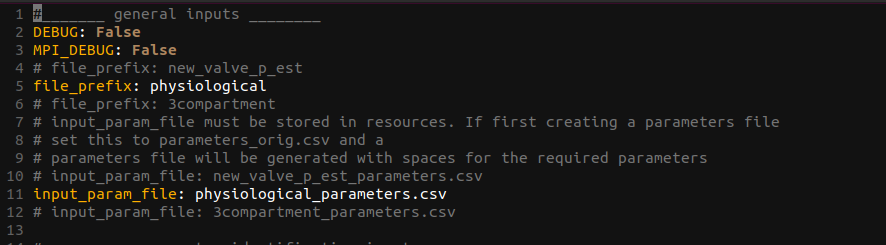
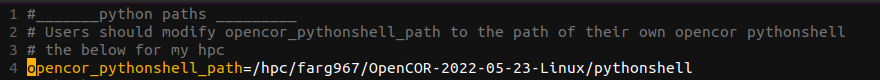


Figure 2

**4)** In the “opencor\_pythonshell\_path.sh” file, change the “opencor\_pythonshell\_path” to the directory of pythonshell in the openCOR folder like this: “<openCOR\_dir>/pythonshell”.



*Figure 3*

**5)** In the user\_run\_files directory do “./run\_autogeneration.sh” to run the autogeneration code. As shown Figure 3, during running, it creates the CellML files, for the generated model, and tests that the simulation runs. Consequently, If there are no errors, it shows the “Model generation has been successful” message at the end.

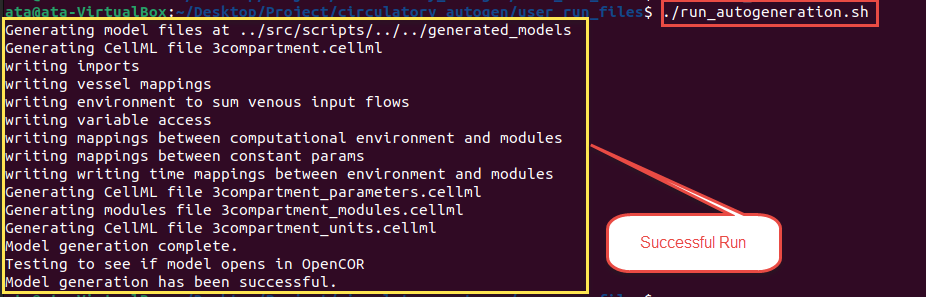


Figure 3

**6)** Generated CellML files are located in the [generated\_models\_dir]/[file\_prefix] directory. (The generated\_models\_dir defaults to [project\_dir]/generated\_models , unless you set generated\_models\_dir in [project\_dir/user\_run\_files/user\_inputs.yaml). As shown in Figure 4, There are five different files in this directory after a successful run. In this example, [file\_prefix] = 3compartment. Four CellML files, and a CSV file were generated.

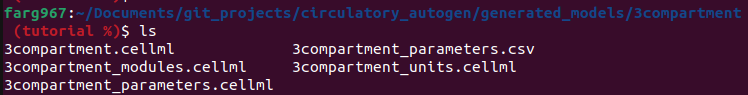


Figure 4

The CSV file includes the model parameters, for typical autogeneration this will be the same as the parameters.csv file in resources/ however when the parameter id is run, it will contain the identified parameter values. The four CellML files contain the modules, parameters, units and constants, and main model.

**7)** Now, you can Open OpenCOR and then open the generated CellML. “[file\_prefix].cellml” is the main CellML file, and it calls to the “[file\_prefix]\_modules.cellml”, “[file\_prefix]\_parameters.cellml”, “[file\_prefix]\_units.cellml” files.

**8)** When [file\_prefix].cellml is opened in OpenCOR, click on the simulation tab on the left side of the screen (as shown in Figure 5, with a yellow box). If there is no error, OpenCOR shows you a new page (Figure 5) (If there is an error specific to the cellml code then it will be shown here). Several individual parts on this page are:

- Simulation settings

- ODE solver settings

- Parameters and variables

- Run control

- Run diagnostics

- Graphs and results

You should set the simulation’s starting, ending, and data output step size. Also, if you have a stiff problem you may need to set the maximum\_time\_step to a small value.

ODE solver settings contains many settings related to the solver, such as maximum step size, iteration method, absolute and relative tolerance, name of solver, etc. It is shown in the blue box in Figure 9.

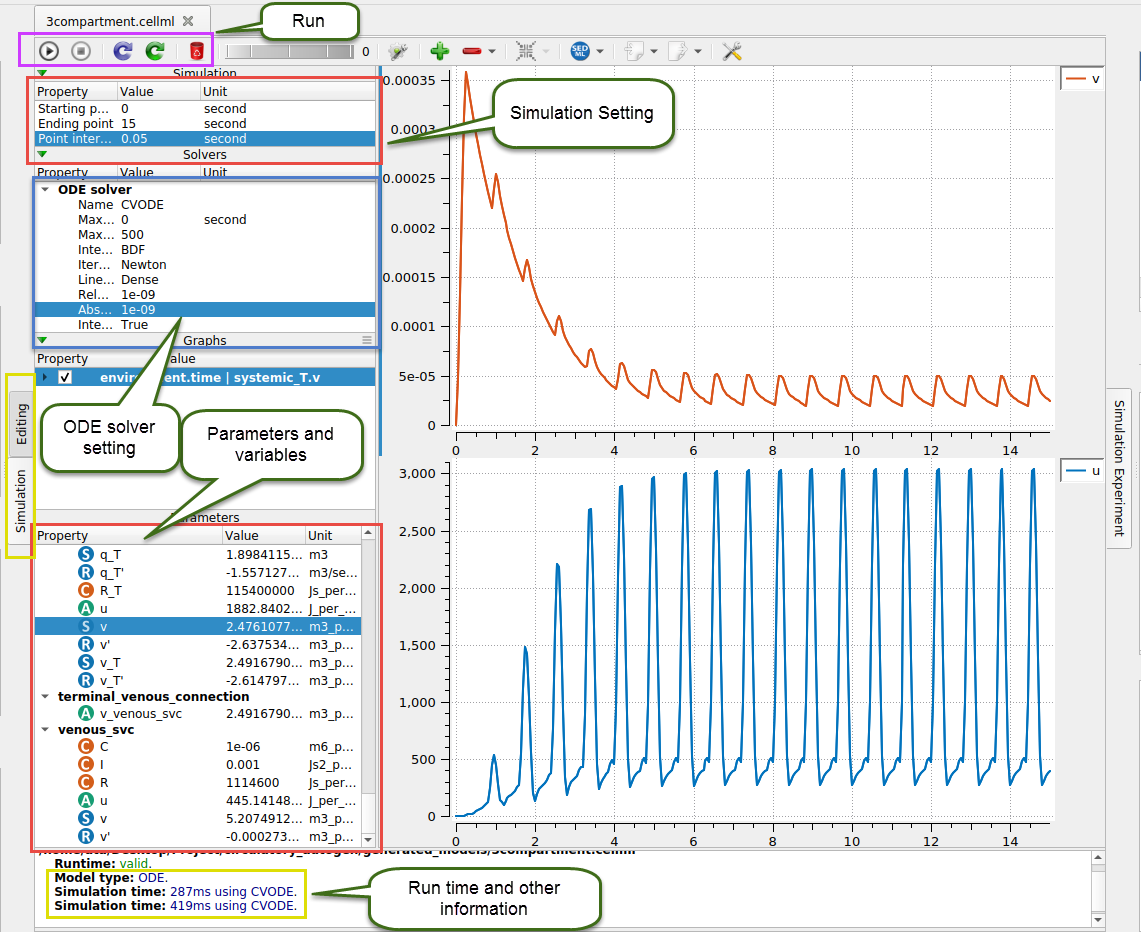


Figure 5

The parameters and variables section shows all constant and variable parameters that are used in the model (all that are made available, see Section 6). You can plot variables by right-clicking each parameter you want for the y-axis and then choosing the x-axis variable. For example, time.

The run control is on the top left section, as shown in purple color in Figure 5. Click on the triangle bottom to run. For further control, see the OpenCOR tutorial.

The results will be shown after running the model. These results include run-time, settings, and other related parameters, as shown in the yellow box at the bottom of Figure 5.

1. **Simple Example: 3Compartment model**

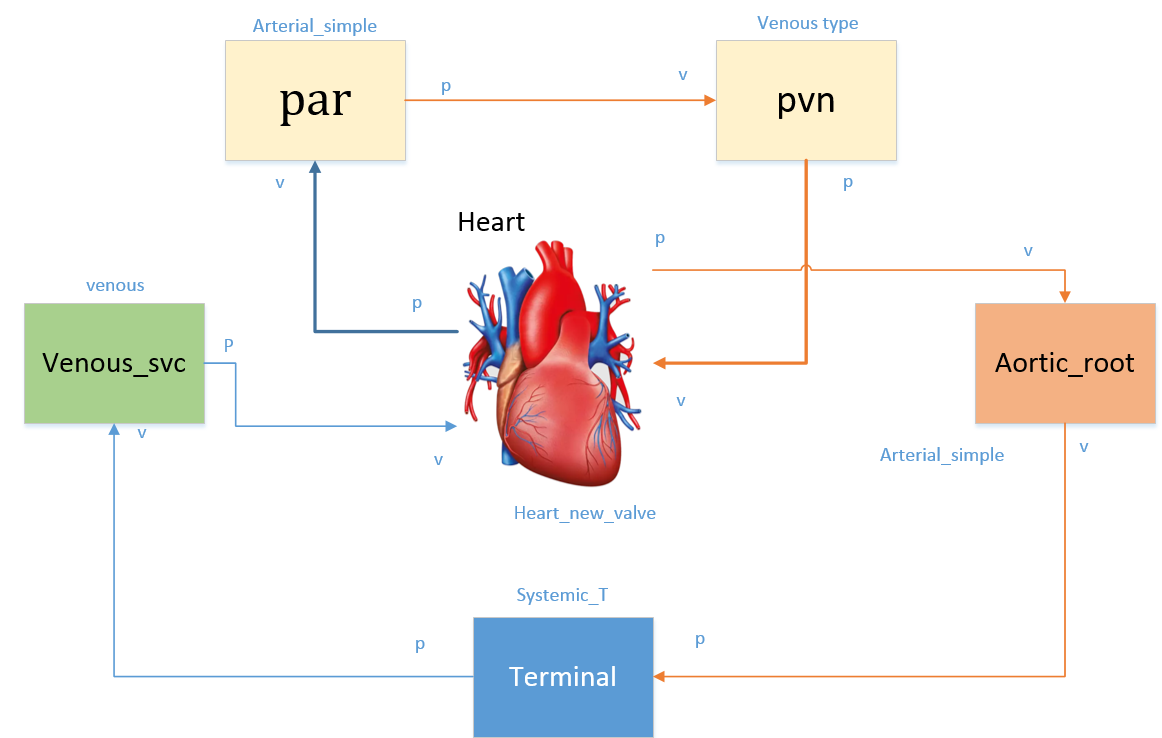


Figure 6 simple model of the cardiovascular system

A schematic view of this model is shown in Figure 6 with details of boundary conditions, parts’ names, vein connection, and direction of blood flow through. The heart is modeled as a simple valve module in this example. According to this schematic, the vessel and parameters files are designed as shown in Figure 7.

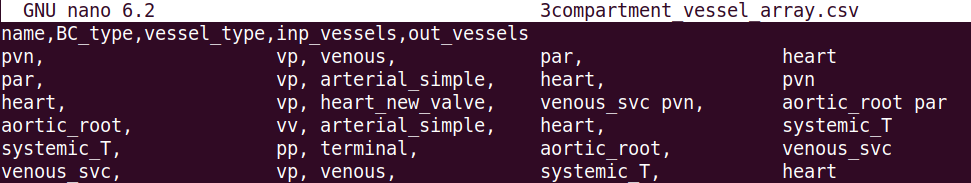


Figure 7 Example of vessel array file

There are five columns in this vessel\_array file: name, BC\_type, vessel\_type, inp\_vessels, and out\_vessels. Name is the user-chosen module’s name, for example, heart or aortic\_root. BC\_type is the boundary condition type for circulatory system vessels. The BC of vessel modules should be one of the vv, vp, pv, and pp types. The first letter defines input pressure (p) or flow (v) boundary condition, the second letter defines output pressure (p) or flow (v) boundary condition. For example the pulmonary vein (pvn) in the above example has a vp BC type, meaning it has input flow and output pressure boundary conditions.

Note: A vessel with output flow BC must be connected to a vessel with input pressure BC and vice versa. E.g a vp can only have either a vv or vp vessel as ouputs. This is checked in the software, and will throw an error if there are incorrect BC connections.

The vessel\_type and BC type must match a module with the same vessel\_type and BC\_type in the modules\_config JSON file, whose directory is:

**“[project\_dir]/src/generators/resources/module\_config.json”**

This file is the config file which defines the variables of a module and the connection ports that allows linking between modules, each module links to a cellml module. This will be discussed more in Section 6.

Inp\_vessels and out\_vessels are the inputs and outputs of each module, respectively. Some modules have multiple inputs and outputs, like split\_type vessels that split from one to two vessels, and the heart...

*IMPORTANT: The order of input and output vessels is important for the heart module. The order must be  
inp\_vessels: 1:inferior vena cava, 2:superior vena cava, 3:pulmonary vein  
out\_vessels: 1:aorta, 2:pulmonary artery.  
A simple two-vessel pulmonary system will be used if the pulmonary vessels are not included.*

*NOTE: Currently, the terminal vessels should only have a BC starting with ‘pp’*

**Results:**

In Figure 8, three different variables are plotted versus time. The step size was adjusted to 0.01 seconds. These variables are the Aortic\_root flow rate, pvn flow rate, and the volume of the heart, respectively. This is the model before doing model calibration. See section 7 for info on the parameter identification.

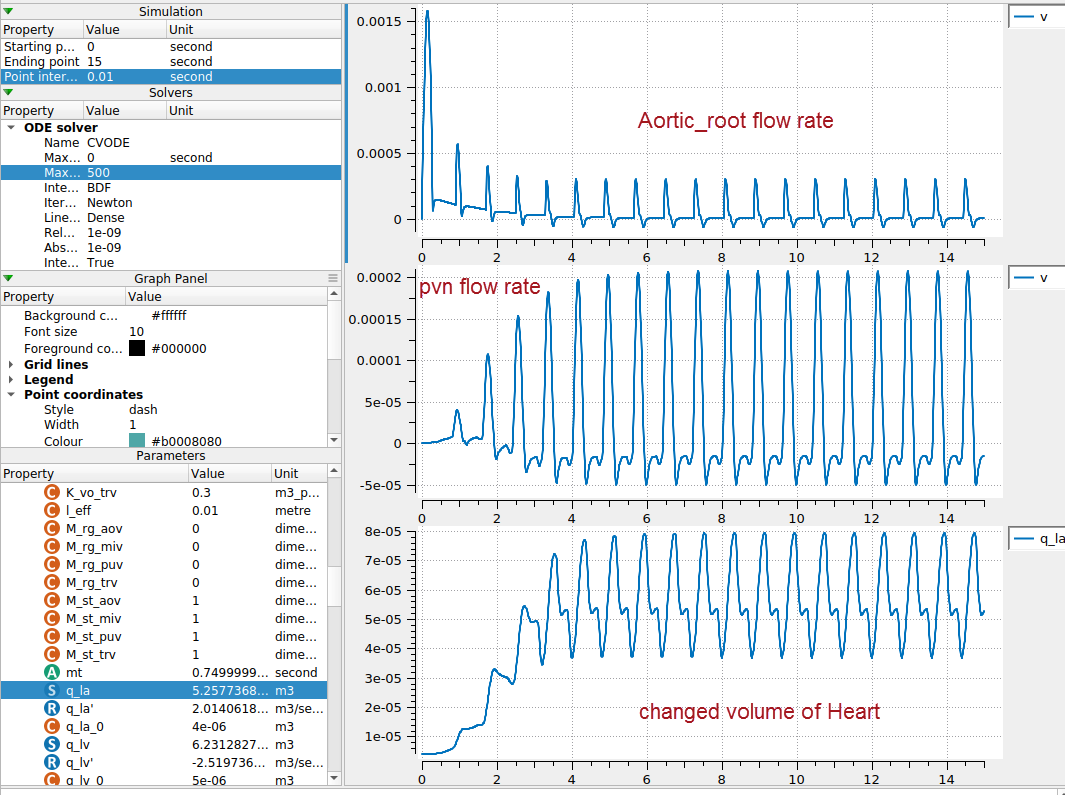


Figure 8 Output results for the 3compartment model

1. **Creating a new model**

This section discusses creating a vessel array and parameters files to build a new desired model.

One standard vessel array file contains five important columns elaborated in Table 1. Vessel\_name is the name of a common organ or part of the cardiovascular system, BC\_type is the type of the boundary condition for the vessel’s input and output, and vessel\_type can be defined as the desired module which exists in the modules file and JSON file; However, the readily usable vessel\_types are possible as below, inp\_vessel, and out\_vessel, are the input and output of each part, respectively.

Table 1 vessel array file columns and input data types

|  |  |
| --- | --- |
| **Column name** | **Possible inputs** |
| Vessel\_name | User defined, but it is better to use common names like heart, pvn, par, etc. |
| BC\_type | 'vv', 'vp', 'pv', 'pp', pp\_wCont, pp\_wLocal, nn, (linked to BC\_type in the module\_config.json file) |
| Vessel\_type | 'heart', 'arterial', 'arterial\_simple', 'venous', 'terminal', 'split\_junction', 'merge\_junction', 2in2out\_junction, gas\_transport\_simple, pulomonary\_GE, baroreceptor, chemoreceptor, (linked to vessel\_type in the module\_config.json file). |
| Inp\_vessels | name of the input vessels, which is one (or more) of the vessel\_name entries in the other rows |
| Out\_vessel | name of the output vessels, which is one (or more) of the vessel\_name entries in other rows |

Figure 7 shows an example of a vessel\_array file.

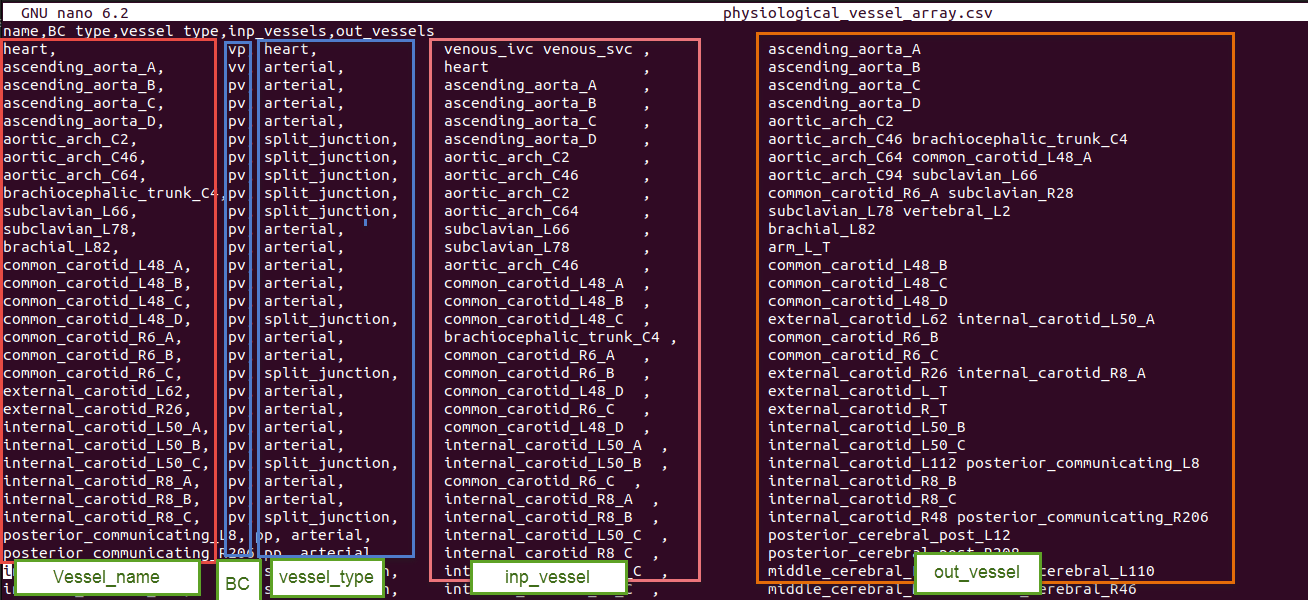


Figure 9 Example of vessel\_array file

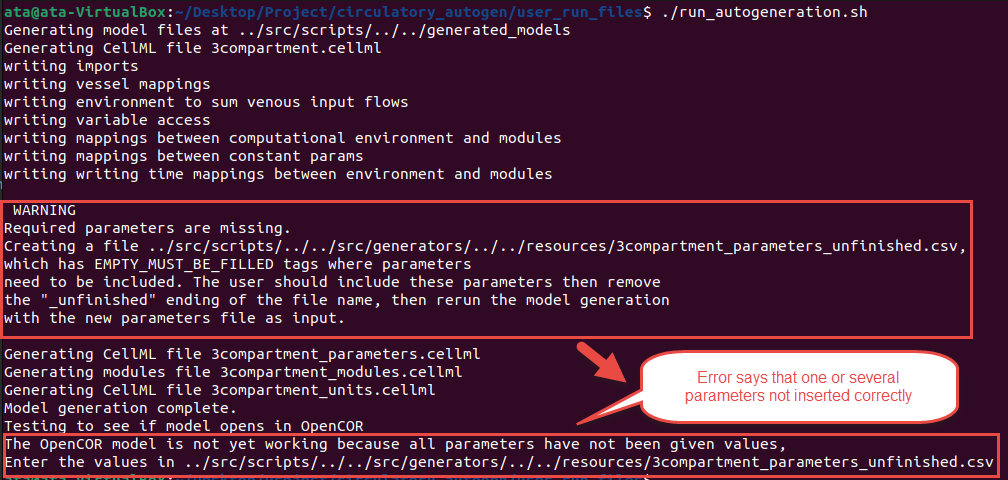
Every row of the vessel array file represents a specific part or module in the defined system. Therefore, each module needs several parameters for modeling and generating a CellML file.

These parameters should be inserted in the parameters file: **“[resources\_dir]/[file\_prefix]\_parameters.csv”**

This file has the structure as shown in Table 2. Also, An example of the file is shown below.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Table 2 Structure of parameter file   |  |  | | --- | --- | | **Column Name** | **Description** | | Variable\_name | Parameter name | | Units | Unit in the defined units in CellML’s unit file | | Value | Value of parameter | | Data\_reference | Reference of the parameter value. Typically in [last\_name][date][first\_word\_of\_paper] format for papers. | |

**Note:** If you forget to add or insert any needed parameter in the file when you run the code, It shows you this message at the end:



At this time, you should go to the below file that will have been created:

**"[resources\_dir]/[file\_prefix]\_parameters\_unfinished.csv"**

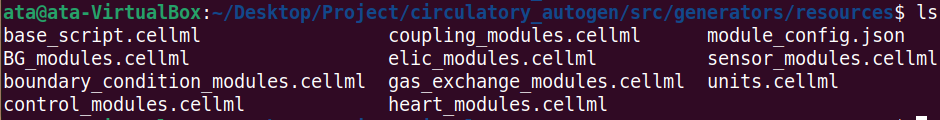
This file will include the parameters which were not inserted in the file with EMPTY\_MUST\_BE\_FILLED value and Data\_reference entries. So, you should add the parameters’ value in the file and a reference. Subsequently, delete the last part of the file’s name (“unfinished”) and rerun the code to solve the issue.

**Modules and definition of a new module**

There are many modules in the below directory, and they can be used in the model. The main directory is:

**“[project\_dir]/src/generators/resources”**

In this directory, there are several CellML files which contain the modules that can be coupled together, and a module\_config.json file defines the connection ports and variables of each cellml module.



The “base\_script.cellml” is the template of the main cellml file that gets generated. The base\_script.cellml file is shown in Figure 10. It uses the “units.cellml” in the main generated code to add all types of units. Also, “module\_config.json” is the main file used in autogeneration to know how to couple the cellml files in the arrangement defined by the vessel\_array file.

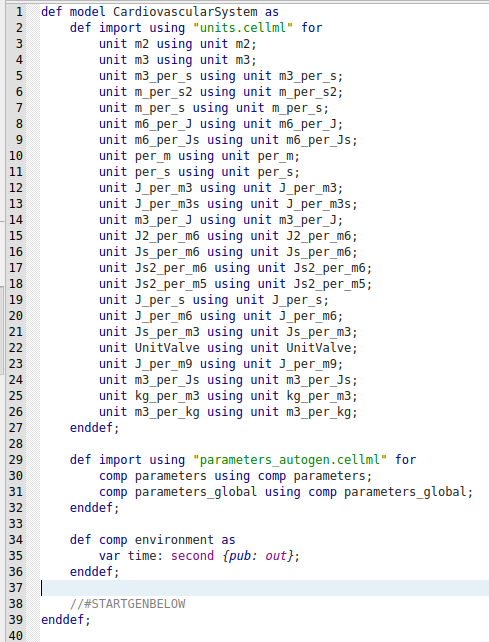


Figure 10 “base\_script.cellml”

If you want to create a new module, you must add the related JSON data in the “module\_config.json” and link to the file where the cellml file is contained, e.g. “BG\_modules.cellml” or another cellml file in [project\_dir]/src/generators/resources/.

As shown in Figure 11, there are three different parts for each module. The primary specification includes vessel\_type, boundary condition type, and module\_file. The ports and their types, and finally, variables and constants.



Figure 11 one of the “module\_config.json” datasets

One of the modules is shown in Figure 12. The main body of a specific module contains variables declaration, constitutive parameters, and state variables. Then, you should write the constitutive relations and, eventually, ODE equations.

In the next, we show a simple example to create a new module you can find in the JSON file.

|  |  |
| --- | --- |
|  |  |

Figure 12 one of the modules in the BG\_modules file

**Example:**

We want to define a new vessel type with the name of “arterial” with boundary condition type “vp”. Additionally, we want to use the “vp\_type” module, whose cellml code is shown in Figure 12. Also, the module is located in the “BG\_module” file.

Vessel\_type, BC\_type, module\_format, the module\_file location, module\_type, and other related information are added to this “module\_config.json” file, as shown in Figure 13. We can now use this vessel\_type in the vessel\_array file in the resources directory to add the module with specified inputs, outputs, and parameters. Consider that In the ports, you should add the “vessel\_port” type for connecting to the other parts. Additionally, each module can be used in many vessel\_types.

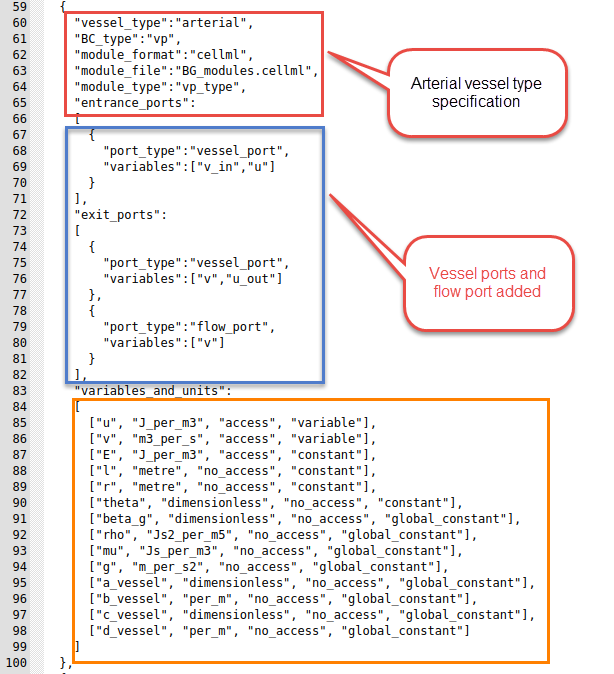


Figure 13 Example of a module linked to the vp\_type cellml module in Fig 12.

The entries in the module\_config.json file are detailed as follows:

*vessel\_type:* This will be the “vessel\_type” entry in the vessel\_array file

*BC\_type:* This will be the “BC\_type” entry in the vessel\_array file

*module\_format:* Currently only cellml but in the future will allow for cpp modules and others.

*module\_file:* The file within [project\_dir]/src/generators/resources/ that contains the cellml module that this config entry links to.

*module\_type:* The name of the module/computational\_environment within the [module\_file] cellml file.

*entrance\_ports:* specification of the port types that this module can take if it is connected as an “out\_vessel” to another module.

*Exit\_ports:* specification of the port types that this module cal take if it is connected as an “inp\_vessel” to another module.

*port\_types:* The name of the type of port. If two vessels are connected vessel\_a to vessel\_b, and vessel\_a has an exit\_port with the same port\_type as an entrance\_port of vessel\_b, then a connection will be made.

TODO Finbar show example diagram of port\_types

*variables:* These are the variables within the module that will be connected to the variables in the corresponding port of the connected vessel/module.

*variables\_and\_units:* This specifies all of the constants and the accesible variables of the cellml module. The entries are:

* [0] *variable name:* corresponding to the name in the cellml file
* [1] *variable unit:* corresponsing to the unit specification in units.cellml
* [2] *access or no\_access:* Whether the variable can be accessed within the cellml simulation. Should always be access for accessibility, unless you want to decrease memory usage.
* [3] *parameter type:* can be constant, global\_constant, or variable.
  + All constants are required to be entered in the [resources\_dir]/[file\_prefix]\_parameters.csv file with the following naming convention: [variable\_name]\_[vessel\_name]
  + All global\_constants are requred to be entered in the [resources\_dir]/[file\_prefix]\_parameters.csv file as just [variable\_name]

1. **Parameter Identification**

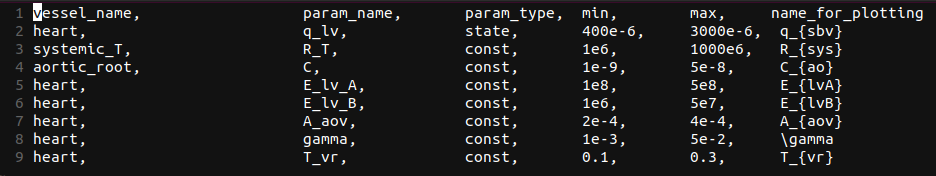
The parameter identification part of Circulatory\_Autogen is designed to allow calibration of a model to experimental or clinical data. It implements an optimisation method to find the best fit parameters that give a minimal (local minima) error difference between the model output and the ground truth observables (experimental or clinical data or user specified.). The creation of two more configuration files is necessary:

**params\_for\_id file:**

**[resources\_dir]/[file\_prefix]\_params\_for\_id.csv**

This file defines which parameters (constants and initial\_states) within your model that you will vary in the parameter id process and their allowed ranges (prior distribution). Figure 14 shows an example of the params\_for\_id.csv file for the 3compartment model shown earlier.

The entries in the file are detailed as follows:

Figure 14: 3compartment\_params\_for\_id.csv

*vessel\_name:* the name of the vessel/module the parameter is in

*param\_name:* the name of the parameter in the cellml module (not including the vessel\_name suffix that is included in the [file\_prefix]\_parameters.csv file).

*param\_type:* “state” or “const”: whether the parameter is the initial value of a state or a const. THIS WILL BE DEPRACATED. ALL SHOULD BE CONST. Initial values that need to identified should be defined as constants within the cellml module.

*min:* The minimimum of the range of possible values (min of the uniform distribution).

*min:* The maximum of the range of possible values (max of the uniform distribution).

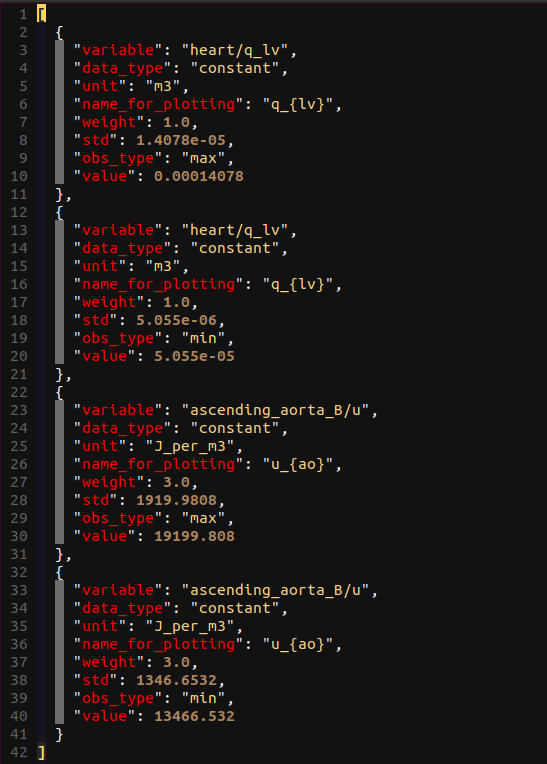
*name\_for\_plotting:* The name (latex format) that will be used when automatically potting comparisons with observables and predictions.

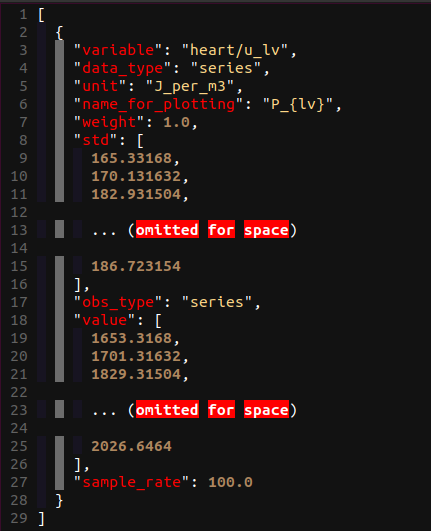
Note: In the future we plan on including other types of priors rather than just uniform.

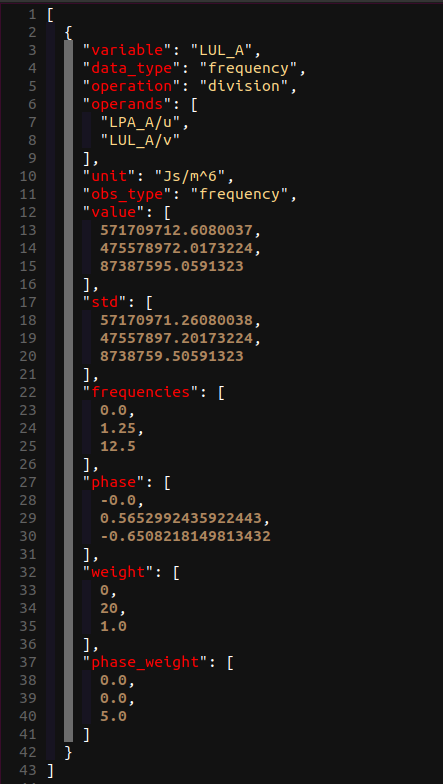
**param id observables file:**

**[param\_id\_obs\_path] (this is defined by the user in [project\_dir]/user\_run\_files/user\_inputs.yaml)**

This file defines the ground truth observables that will be used in the cost function for the parameter id optimisation algorithm. It also defines the measurement standard deviation, and weighting for each observable. Examples of obs\_data.json file are shown in Figures 15, 16, and 17 for constant, series, and frequency data, respectively.

Figure 15: obs\_data.json file

Figure 16: obs\_data.json file showing data\_type: “series” entry

Figure 17: obs\_data.json entry showing data\_type: “frequency” entry

The entries in the obs\_data.json files are:

*variable:* [vessel\_name]/[variable\_name] , where variable name is the variable name in the cellml module and module\_config.json file.

*data\_type:* The format of the data. This can be “constant”, “series”, or “frequency”, as shown above.

*unit:* The unit, which should be the same as the unit in the variable in the model.

*name\_for\_plotting:* The name that will be in the automated plots comparing observable data to model output. (latex format)

*weight:* The weighting to put on this observables entry in the cost function. Default should be 1.0

*std:* The standard deviation, which is used in the cost function. The cost function is the relative absolute error (AE) or mean squared error (MRE), each normalised by the std.

*obs\_type:* The operation that will be done on the model output series before it being applied to calculate the cost function, (min, max, mean, etc). **THIS HAS BEEN DEPRACATED IN FAVOR OF THE “operation” entry.**

*value:* The value of the ground truth, either a scalar for constant data\_type, or a list of values for series or frequency data\_types.

*Sample\_rate:* not needed or set to “null” for constant and frequency data\_types. It defines the sample rate of the observable series values.

*operation:* **This will make obs\_type obsolete.** This defines the operation that will be done on the operands/variable. The possible operations to be done on model outputs are defined in [project\_dir]/src/param\_id/operation\_funcs.py and in [project\_dir]/operation\_funcs\_user/operation\_funcs\_user.py for user defined operations.

*operands:* The above defined “operation” can take in multiple variables. If operands is defined, then the “variable” entry will be a placeholder name for the calculated variable and the operands will define the model variables that are used to calculate the final variable that will be compared to the observable value entry/s.

**Parameter Identification Settings:**

To run the parameter identification we need to set a few entries in the [project\_dir]/user\_run\_files/user\_inputs.yaml file:

*param\_id\_method:* What optimisation method we use. Currently this can only be genetic\_algorithm, but more methods are being implemented. Eventually we aim to use CVODES to allow for gradient based optimisation methods.

*pre\_time:* this is the amount of time the simulation is run to get to steady state before comparing to the observables from obs\_data.json

*sim\_time:* The amount of time used to compare simulation output and observable data. This should be equal to the length of a series observable entry divided by the “sample\_rate”. If not, only up to the minimum length of observable data and modelled data will be compared.

*maximum\_step:* The maximum time step for the CVODE solver

*dt:* The output time step (**This hasn’t been tested well for anything but 0.01 s currently)**

*param\_id\_obs\_path:* the path to the obs\_data.json file described above.

*ga\_options:*

*cost\_type:* AE or MSE for absolute error or mean squared error.

*num\_calls\_to\_function:* How many forward simulations of pre\_time+sim\_time will be run in the optimisation algorithm.

**Features that are implemented but are yet to be included in this tutorial**

* Frequency domain fitting, amplitude and phase
* Using operations and operands in the obs\_data.json file
  + multiple (>2) operands can be used for addition
  + Operations can be defined by the user in the operation\_funcs\_user file.
* Cpp generation for coupling with 1D models
* Cpp generation for implementing time delays with 1D models

**Acknowledgements**

Ata Moradi for writing the first version of this tutorial

The Auckland Bioengineering Institute and the Animus laboratory for testing, using, and helping to develop this software.

**Appendix A: Cardiovascular system and control models**

The article below is a good source for getting more information about the human cardiopulmonary system and the control models. You can download the article from the link.

*Albanese, A., Cheng, L., Ursino, M., & Chbat, N. W. (2016). An integrated mathematical model of the human cardiopulmonary system: model development. American Journal of Physiology-Heart and Circulatory Physiology, 310(7), H899-H921.*

[**https://journals.physiology.org/doi/epdf/10.1152/ajpheart.00230.2014**](https://journals.physiology.org/doi/epdf/10.1152/ajpheart.00230.2014)

**Appendix B: Other Notes**

**Boundary conditions type:**

|  |  |  |
| --- | --- | --- |
| **BC Type** | **Input** | **Output** |
| **PV** | Pressure | Flow rate |
| **PP** | Pressure | Pressure |
| **VV** | Flow rate | Flow rate |
| **VP** | Flow rate | pressure |

To be extended to available types: for now **to see available modules, look at [project\_dir]/src/generator/resources/module\_config.json**