Modelling stuff

Ciaran Welsh

December 4, 2019

Contents

1 Introduction

This document is essentially a list of practical computational exercises designed to try and distribute some of what I have learnt over the years.

2 Reading material

Here are some suggested papers to read preferably before the tutorial sessions. The better you understand this stuff before the tutorials, the more you'll get out of them.

- ?
- ?
- Python Modules, packages and subpackages

3 Environment Configuration

Configuring your environment can be quite frustrating at times and can eat away a lot of time. For that reason, here are some instructions for getting you set up quickly. Moreover, it is also helpful to know how to configure a modelling project. An unorganised project saps away productivity so I also detail here one way of organising your project specifically for modelling with PyCoTools, COPASI and tellurium. If you run into problems, you can clone or fork an example project from here.

It would be helpful if the configuration was done prior to the sessions as then we can focus on modelling issues, rather than configuration issues.

3.1 Python

Python is a program written in C. It is an executable file (i.e. a binary) that must be compiled and linked from source before you can use it. There are many 'distributions' of Python, which essentially just means different people have compiled it and packaged it in slightly different ways.

My favourite distribution of Python is Miniconda. Miniconda and Anaconda are essentially the same, but Anaconda comes with a whole bunch of additional Python packages. These can be useful, but it is quicker to just use Miniconda.

Task 3.1: Install Miniconda

Google Miniconda and follow the instructions to install Miniconda.

Warning: Make sure the command 'conda' works from terminal or cmd. If it doesn't then you need to add the Miniconda bin directory to your path environment variable.

Anaconda and Miniconda (or just Conda) allow you to create isolated Python environments and switch between them easily. Think of each conda environment you have as a box that is kept separate from the other Python 'boxes'. While the full documentation can be found here the commands to create a conda environment are quite simple.

```
$ conda create --name py36 python=3.6
```

Will create a conda environment.

```
$ conda activate py36
```

will switch to the environment

and

```
$ pip install pycotools3 tellurium
```

will install pycotools3 and tellurium with all their dependencies.

Warning: Neither pycotools3 nor tellurium work on Python 3.7 or 3.8. This is because of a broken dependency. The issue is in the process of being solved (apparently).

3.2 COPASI

Task 3.2: Install COPASI

Install Copasi and configure the environment variables if you need to. You should be able to run ‘CopasiUI‘ from the terminal.

3.3 PyCharm

PyCharm is a significantly better IDE than many of the alternatives. It does a lot for you. You can also get a free Licence for the Pro edition with your university email address. Learning how to use PyCharm is useful for many reasons, one of which is that if and when you migrate to other programming languages, JetBrains will have a an IDE for you. Since all the IDEs are very similar, you only have to learn to use one and the rest fall in place.

Task 3.3: Install PyCharm

Install PyCharm. I prefer to install the JetBrains toolbox and install the IDE from there.

3.4 Configuring a Project

You can configure a project however you like, but with a small amount of effort you can create an organised Python project. Being organised makes things much easier down the line. We will be creating a GitHub repository which contains a Python package that has two subpackages, one for models and the other for data.

```
    ExampleProject/
└── example_project
    ├── data
    │   ├── CopasiFormattedData
    │   ├── data_analysis.py
    │   └── __init__.py
    ├── __init__.py
    ├── models
    │   ├── control_script.py
    │   ├── __init__.py
    │   └── model_strings.py
```

Figure 1: Directory tree for organised modelling project

A package in Python is marked by the special file called ‘__init__.py’. Even if it is a blank file, the presence of this file in a directory marks it as a Python directory. Whenever a Package is imported, the __init__.py (for initialisation) is automatically executed. This makes it a convenient place to store some global variables that can be used anywhere throughout the project.

Throughput the project, we will be using an example project package with the following layout:

The project holds two sub-projects within the main example project (for data and model), each with its own __init__.py file.

- The main __init__.py file (ExampleProject/example_project/__init__.py) holds the global variables, such as path names, to be used throughout the project
- Within the
  - **Note:** The other two __init__.py files will be empty
- The 'model' package to hold all data regarding models such as the code for building, estimating the parameters of and simulating a model
- The 'model_strings.py' file as a storage module to keep model strings isolated from the execution code
- The 'control_script.py' file for containing code for actually running the script
- The 'data' module for holding everything regarding experimental data, such as raw data files and data analysis scripts
- The 'data_analysis.py' script for doing anything data related (normalisation, plotting or automatically formatting for COPASI)
- The 'CopasiFormattedData' folder to hold all experimental data that is already formatted for COPASI (whether this is done manually or automatically).

Task 3.4: Create a Project

In PyCharm, create a new project. Then create a directory tree which looks like Figure 1.

A few pieces of code should be added to this project before we begin modelling. Firstly, we are essentially building a Python package which contains the code to create a model. Therefore, we will be able to import this package (or modules, functions and classes from within it) into any python scripts. However, for Python to be able to find the package it must be added to the Python path where it stores the available packages. To do this, we can simply tell Python where our project is using a command from another package called 'site' to add the directory containing 'example_project' to the Python PATH variable (see task 3.5).

Note: PyCharm has options in 'run configuration' for adding your project directories to the PYTHON_PATH variable. These are usually set to True by default, so if you use Pycharm you actually don't need this step. However, if the code is ported to another IDE (such as Spyder) you may run into problems

Task 3.5: Configure the PYTHON_PATH variable

Inside 'ExampleProject/example_project/models/control_script.py' put

```
import os
import site
# Add path to sources root to Python's PATH variable
site.addsitedir(os.path.dirname(os.path.dirname(os.path.abspath(''))))
# note: Pycharm already does this for you.
# But doesn't hurt to add it here anyway

# Get the model string by importing it from your models_strings module
from example_project.models.model_strings import model_string

# imports all the global variables (notice that we
#  can print out the WORKING_DIRECTORY variable)
from example_project import *


# Any functions or classes you write will go here

if __name__ == '__main__':

# Any code that uses the functions or classes your have created above,
# will go here. We will be using flags defined in our __init__.py
# to modify the behaviour of this script. Since the Flags are boolean,
# we just us a simple if statement for each of them

if PRINT_WORKING_DIRECTORY:
# prints /home/ncw135/Documents/ExampleProject/example_project
print(WORKING_DIRECTORY)
```

Task 3.6: Configure the your projects global variables

Add the following to 'ExampleProject/example_project/__init__.py'.

```
import os, glob
# Global variables are always in caps, to distinguish them from local variables
WORKING_DIRECTORY = os.path.dirname(__file__)
DATA_DIR = os.path.join(WORKING_DIRECTORY, 'data')
COPASI_FORMATTED_DATA_DIR = os.path.join(DATA_DIR, 'CopasiFormattedData')

# Flags that change the behaviour of the control_script

# flag to demonstrate the principle of flags
PRINT_WORKING_DIRECTORY = True
```

Now, because of our configuration we can import the various parts of the project within the 'control_script' and begin modelling.

Note: If something isn't working and you've spent too much time trying to fix, you can clone or fork my example project from Github

.

Task 3.7: Create a GitHub Repository

Sign up for a GitHub account if you do not already have one. Turn your version of this boiler plate project into a GitHub repository. One set of instructions can be found here.

4 Ordinary Differential Equation (ODE) Models

This section is dedicated to building, parameterising and simulating ODE models using the tools commonly used in systems biology. ODEs are widespread in science and you can build them yourselves using pretty much any programming environment. In systems biology however, we tend to let software abstract away the actual equation building in favour of a 'biochemical centric' perspective. Regardless of how the equations are built, they are still all ODE models so you should be aware (though not necessarily an expert) of the maths involved in ODE modelling. This is beyond the scope of these tutorials, but I'd advise you to make sure you understand the concept of numerical integration and optimization (though again, not necessarily how to actually code them).

5 Model Construction

ODE models in systems biology are best built from a visual representation of the network, otherwise it is easy to get get confused and make mistakes. The networks should be as tidy as possible. There is no use building a messy network with wires crossing wires everywhere because that detracts from the purpose of having a network: to guide your thinking about how the network is going to behave.

Biochemical networks are often drawn in a cartoon format at the end of a mechanistic biology paper. These are often ambiguous and should be avoided. Instead, a good approach is to draw a wiring diagram (Figure 2) on paper to get a network that seems reasonable and then use CellDesigner to make the representation more formal.

Note: The arrows in these networks define the general relationship between model components, but not the exact mathematical relationship. For instance, an arrow between A and B might represent a mass action relationship or some other kind of rate law (i.e. Michaelis-Menten)

.

Remember: A neat and tidy network is not just for aesthetics but will help you think about your network.

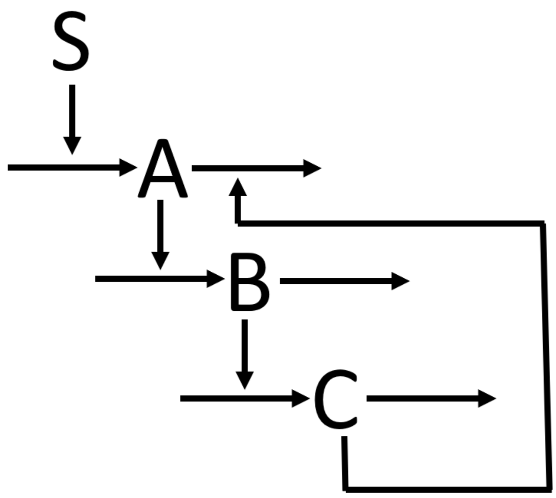

Figure 2: Example wiring diagram of a network where A is produced in response to stimulus S. Then a signal cascade begins, starting with A, flowing through B to C. C causes the degradation of A and therefore completes a negative feedback loop.

Task 5.8: Toy Wiring Diagram

Draw a wiring diagram of a toy (made up) network. You will later simulate this network so make it interesting but dont over complicate it. You should include at least one Michaelis-Menten and one Competitive Inhibition reaction.

Task 5.9: Cell Designer

Draw the same network you drew for the wiring exercise using CellDesigner.

5.1 Antimony

Antimony is the name of a piece of software that has defined its own language for creating SBML models. The idea is that you build the model in this human-friendly format and then use the conversion tools provided for converting the antimony string into a simulatable model. The docs for antimony can be found here and here.

Task 5.10: Build an Antimony String

Build an antimony version of your model using the documentation to help you. Assign your antimony to a variable inside 'model_strings.py' so that you can import the string inside the 'control_script.py'.

5.2 Model loading

Both tellurium and PyCoTools work from antimony strings. Tellurium is a Python wrapper around a C++ solver called roadrunner. Therefore the executable model used within tellurium is actually an extended Roadrunner model.

Task 5.11: Model Loading

Load your antimony string into a roadrunner model using tellurium. Load your model into a PyCoTools model. Open the PyCoTools model in copasi without. Do this in the control script, at the bottom under the

```
if __name__ == '__main__'
```

Since we will probably want to use one of these model for every additional task we add to the project (i.e time series simulation), we might as well at this at the top of the main block, above all the 'if' statements.

5.3 Antimony Combinations

AntimonyCombinations is a Python package developed to take a core hypothesis along with some extension hypotheses and automate the construction of all combinations of model in SBML format.

Task 5.12: Antimony Combinations Example

Go to the AntimonyCombinations docs and run the example in a new script along side your 'control_script.py'.

Note: In a later task, this section becomes more relevant so feel free to move on and come back later.

6 Simulation

In this section, you'll be bombarded with a bunch of tasks designed to get you acquainted with the tellurium PyCoTools. Since both tools work with Antimony strings, you can use both tools seemlessly with your model.

You should use the tellurium and PyCoTools documentation to work out how to do the

Task 6.13: Run A time series

Run a time series with both tellurium and PyCoTools on your model. Plot the time series using both tools. Plot the data using 'matplotlib'.

Task 6.14: Create some fake data for fitting to your model

To demonstrate parameter estimation, the quickest way is to simulate some data from the model you have built and configure a parameter estimation using this simulated data.

- Create a new variable in your '__init__.py' to hold the full path to a (yet non-existent) data file under the 'COPASI_FORMATTED_DATA_DIR'. As this is a global variable (not to be confused with a 'global quantity' inside the model) used elsewhere in the project, make the name all capital letters.
- Run a time series using either tellurium or pycotools.
- Ensure your variable names are exactly the same as those used in your model. If you have used tellurium, you should convert your results object to a 'pandas.DataFrame so you can easily modify the names of the columns (accessed with 'df.columns') and easily save the data to file.
- Save the data to the newly created global variable (using 'pandas.csv').

Task 6.15: Parameter Estimation in Copasi

Open your model in copasi and set up a parameter estimation with your simulated data. Pick the parameters you want to estimate, define some plots (bottom right corner) and try out a few algorithms. Make sure the 'random initial conditions' button is checked. Play around with the algorithms hyperparameters and notice how they affect convergance.

Task 6.16: Parameter Estimation Using PyCoTools

Configure and run a parameter estimation using PyCoTools. PyCoTools essentially automates the procedure from the previous task. Play around with the options in PyCoTools by configuring the a parameter estimation to setup all the following parameter estimations. When changing from one configuration to another, change the 'problem' or 'fit' settings to isolate new results from previous runs.

- Use the 'problem' and 'fit' settings to organise your
- Estimate all global quantities
- Estimate all metabolites
- Estimate all global quantities and metabolites
- Change the algorithm you are using to something different. Also change the hyperparameters (like iteration limit or population size)
- Run a parameter estimation in 'parallel' mode
- Use the 'prefix' option to be selective about parameters you are estimating.
- **Note:** PyCharms find and replace feature is particularly useful when renaming parameters to include the prefix

Task 6.17: Visualising Parameter Estimation Results

That that we have

Plot time series plot parameter estimations run steady state with roadrunner and with copasi Insert parameters intoa model plotting the best fit. accessing parameter estimation data. using pycotools on the cluster using the advanced interface profile likelihoods