Modelling stuff

Ciaran Welsh

December 11, 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. Importantly, what is described here is only one way of organising a modelling project, but it has been refined through experience and is what works best for me. Hopefully it will also work for you.

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.

- Raue et al. [2013]
- Alon [2007]
- Python Modules, packages and subpackages
- Introduction to GitHub

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 detailed 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.

It would be helpful the configuration steps below were done prior to the workshop as then we can focus on modelling issues, rather than configuration issues.

3.1 Python

Python is a program written in C. 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 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 (instructions). 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 an IDE for you tailered to that language. 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.

Task 3.4: Install CellDesigner

Install Cell Designer

Task 3.5: Install Github

Install git and make a free GitHub account.

3.4 Configuring a Project

With a small amount of effort, we can build an organised modelling project which will dramatically enhanced productivity down the line when you need to extend the model and make changes. We will be creating a project which contains a single Python package that itself has two subpackages (see Figure 1). We will also turn this project into a GitHub repository.

Task 3.6: Python pacakges and modules

If you are not familiar with the concept of Python packages, subpackages and modules, now might be a good time to do a quick google to get a broad understanding (no need to go overboard). This will help you understand why the project is configured the way it is.

Task 3.7: Downloading the example project

Since some aspects of configuring our project may initially be a bit confusing, clone, fork or download the example project. This project contains a skeleton that you can use as a template. The quickest way would be for go the link and press the `fork` button. This will make a copy in your own GitHub. Then go into the settings section of the respository on your own GitHub and change the name to whatever you like. Now, clone your own repository using `git clone https://github.com/<your name>/<your project name>`.

Note: If you want to configure this yourself instead, you can omit this step and follow the instructions below. If you download it, it's still a good idea to understand the prroject structure, by reading below.

.

Task 3.8: Create a Project

We will be using a project structure that looks like that shown in Figure 1. If you haven't already downloaded the example project, create this structure somewhere in your file system. In PyCharm, create a new project. Then navigate to the root of the tree you have just created (or downloaded) and hit open.

```
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 package. Whenever a package is imported, the `__init__.py` (for initialisation) is automatically executed. This makes it a convenient place to store some variables that can be used anywhere throughout the project.

Task 3.9: Configure Python Interpreter in PyCharm and Running Code

We have created a Conda environment called `py36`. Now we need to tell PyCharm to use this environment. At first, PyCharm can feel unintuitive, but stick with it and it will enhance your productivity (trust me).

- Open the settings in PyCharm from the file menu
- Select `Project: <project name>` from the left, then project interpreter
- We want to select the Python3.6 conda environment called `py36`. If this is already in the dropdown at the top of the screen, select it, press okay and then you're done. If not, continue.
- Click on the gears icon, top right and press `add` to tell PyCharm about a new Python interpreter.
- Click on `existing environment` and check the box that says `make available to other environments`
- Click the three dots to open up a file system and nagivate to the Python executable that you want to use.
- On linux (or Mac) you can open a terminal, switch to your `py36` environment using `$ conda activate py36` and type `$ which python` to locate the Python executable. Copy and paste this into the box in PyCharm and okay everything.
- On Windows, you can do the same thing by opening a `cmd` prompt or `PowerShell`, switching to your `py36` environment using `conda activate py36` then using the command `> where python`. Copy and paste the location of the Python executable into the box in PyCharm and then okay everything.
- Now that you have configured PyCharm, you can create any Python script, right click somewhere and press `Run <script_name`, to run some Python code.

Here is some information describing the components of the project.

- The main `__init__.py` file (`ExampleProject/example_project/__init__.py`) holds variables that are used throughout the project. By keeping them in one place we never loose or forget about these variables and we don't get conflicts resulting from duplicates. Examples of what variables we will be defining in here include path names and flags for modifying the behaviour of the program.

  > **Note:** The other two `__init__.py` files will be empty. This is only necessary to make your Python modules importable.

- The `model` package holds everything regarding your model such as the code for building and simulating a model
- The `model_strings.py` file functions as a storage module to keep model strings isolated from the execution code
- The `control_script.py` holds all code regarding actually loading and running simulations. It responds to the variables defined in `__init__.py` to modify what we actually want to do. For example, you could have a variable called `OPEN_WITH_COPASI` or `CONFIGURE_PARAMETER_ESTIMATION`.
- The `data` module holds everything regarding experimental data, such as raw data files and data analysis scripts.
- The `data_analysis.py` script does anything data related (normalisation, plotting or automatically formatting for COPASI). For now, this will be empty but it will likely be needed in the future.

- The 'CopasiFormattedData' folder holds all experimental data that is formatted for COPASI. I usually do this programatically to save time in the long run but it can also be done in excel.

If you are not using the example project (which is already configured), a few pieces of code should be added to this project before we begin modelling.

Firstly, if we want to import code from a Python module in your project (such as `data_analysis.py`, `__init__.py` or `model_strings.py`) to where we want to use the code (`control_script.py`) then we have to tell Python where the project is. To do this we point Python towards the directory containing your package (i.e. the `example_project` folder). Python has a special variable called the PYTHONPATH variable that exists exactly for this reason. The template code below in `control_script.py` does this using the `site.addsitedir` function. If this doesn't make much sense to you, don't get hung-up on it. This is a Python issue that a better understanding of the Python module hierarchy will sort out. For now, just insert the code shown below and forget about it.

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 3.10: Configure the control script

```python
# ExampleProject/example_project/models/control_script.py
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 3.11: Configure the the projects `__init__.py`

```
# 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
```

We can now 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

.

3.5 GitHub

Git and GitHub are one of your most important tools. When used correctly you can version your code, that if you make some changes that made your model worse, you can always revert to an earlier verison. Moreover, it makes sharing code, models, simulations and working together much, much easier.

Task 3.12: 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.

Note: This task can be safely omitted and not impact the rest of the project. However I recommend that you at least figure out the basics.

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.

Task 5.13: Toy Wiring Diagram

Draw a wiring diagram of a toy (made up) network. You will later simulate this network so make it interesting but don't over complicate it. You should include at least one Michaelis-Menten and one Competitive Inhibition reaction.

Task 5.14: 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.

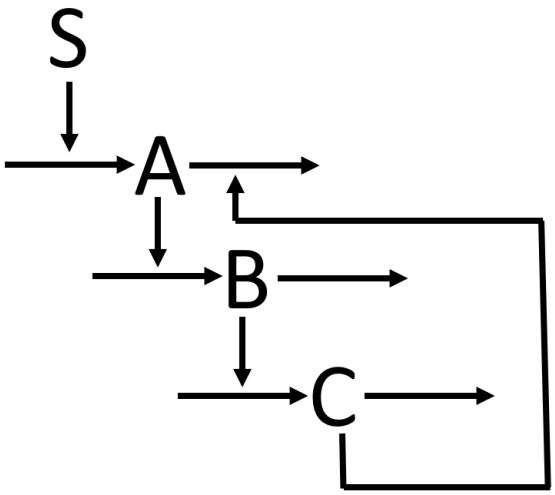

Figure 2: Example wiring diagram of a network where A is produced in response to stimulus S. Then a signal cascade begins, starting with A, flowing through B to C. C causes the degradation of A and therefore completes a negative feedback loop.

Task 5.15: 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'.

It is not always obvious how to use Antimony and the docs are not always helpful for every question you might have. However, there is an alternative option. The tellurium package has fairly comprehensive conversion facilities that can directly convert SBML to antimony. This means that it is possible to build a SBML model using whichever tool you like and then convert it into an antimony string. Therefore, if you cannot figure out how to implement a model feature using antimony, build the model in Copasi, export the sbml and then convert it into an antimony string to see the antimony syntax for that feature. The auto-generated antimony string is somewhat bloated so I always prefer to write the string myself. However, if I ever get stuck with a feature, I'll write a toy model in COPASI which implements the feature and autogenerate the antimony to see how its done.

Task 5.16: Reverse engineer an antimony string

Download an sbml model from BioModels. Don't spend too long picking but you may as well try to find one that is relevant to your own research. Scour the tellurium documentation for the function to convert sbml into antimony and then use it.

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 5.17: 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 the GUI. 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 add 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 a model in SBML format.

Task 5.18: Antimony Combinations Example

Go to the AntimonyCombinations docs and run the example in a new script alongside 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 have a set 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.

Task 6.19: 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 pythonmatplotlib).

Task 6.20: Run A steady state simulation

Run a steady state simulation with tellurium. Note that the steady state task is not supported by PyCoTools, so you'll have to open the model with the user interface if you want to run a steady state with COPASI.

Task 6.21: 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 python$_{init.py)toholdthefullpathtoa(yetnon-existent)datafileunderthepythonCOPASI_FORMATTED_}$
- 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 pythondf.columns)) and easily save the data to file.
- Save the data to the newly created global variable (using pythonpandas.csv)).

Task 6.22: 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 ashrandom initial conditions) button is checked. Play around with the algorithms hyperparameters and notice how they affect convergance.

Task 6.23: 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 pythonproblem) or pythonfit) settings to isolate new results from previous runs.

- Use the pythonproblem) and pythonfit) settings to organise your parameter estimations into folders
- Estimate all global quantities
- Estimate all metabolites
- Estimate all global quantities and metabolites
- Change the algorithm you are using to something different (e.g. particle swarm). Also, change the hyperparameters of the algorithm. What impact do they have the results and convergance?
- 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 6.24: Visualising Parameter Estimation Results

Now that we have some parameter estimation data we can produce some diagnostic plots to evaluate the estimations performance and compare simulated versus experimental data.

- Plot a waterfall plot
- Plot a boxplot of you parameter distributions
- Plot a violin of you parameter distributions
- Plot the solution of the best fitting parameters against the experimental data. Note that there are features in the pythonpycotools.viz) module for this.
- Read the parameter estimation data into Python using the pythonviz.Parse) class.

- Explore the parameter estimation data using pandas.
- Insert best parameters into the model.
- Insert best parameters into the model and open with the COPASI UI. Simulate the parameter estimations using the current solution statics method to visualise the experiment versus simulated data within copasi.
- Insert the second best parameter set into the model and repeat the above.

Task 6.25: Profile Likelihoods

Perform an identifiability analysis using the Profile likelihood method using PyCoTools. Due to a unfortunate mishap, I forgot to rewrite the code for visualising profile likelihoods after jazzing up PyCoTools. Optionally plot the profile likelihoods yourself using matplotlib and seaborn. You can parse the data into Python using the usual pythonviz.Parse) 'class.

Task 6.26: Using the low level interface

The ParameterEstimation.Context takes care of all of the detailed arguments required to configure a parameter estimation. However, sometimes it is necessary to have a more fine grained level of control over the configuration.
The ParameterEstimation.Context class exists in order to construct a ParameterEstimation.Config object. However it is also possible to just create the Config object yourself. See the docs for more information on this. We'll be using a yaml file to hold our configuration options and then we'll read those options into PyCoTools.

Note: It is not necessary to actually run parameter estimations here. The point of this task is to show you how flexible the configuration options are using PyCoTools and COPASI. Therefore, set `run_mode=False` and open the model (using `model.open()`) each time you run your program. What changes do you see in the parameter estimation task?

- Start by using the Context class to create a template for your configuration.
- When printed, the config object in yaml format. Print the config object and copy and paste it into a new file with a .yaml extension.
- Load the configuration into PyCoTools using the `tasks.ParameterEstimation.Config.from_yaml` static method.
- Setup different boundary conditions for several of your estimated parameters.
- Modify the `affected_experiment` option
- Add a constraint item
- Play around with the other various options.

graphicx

7 Practice Modelling Project

The purpose of this section is to try to give you some modelling experience in an environment that is as close to real world as you can get without actually working on a real modelling problem.

In the system we are working with there are three input variables (or drugs): `S`, `I1` and `I2`. The

stimulus, `S`, induces phosphorylation and activation of N while the inhibitors `I1` and `I2` inhibit G and D respectively.

I have conducted 4 comprehensive time series experiments which measure the phosphorylated, amounts of four key system players: `N`, `G`, `K` and `D`. Each experiment was conducted over 2 hours and the system was sampled at 8 time points. The data have also been normalised for background noise by subtracting the 0 time point from each of the observables. In all experiments, you can assume that the treatments are present in saturating concentrations and that this does not change over the duration of the experiment.

Here are the experiments:

1. Stimulation of a cell line with S alone (`S.csv`).
2. Stimulation of a cell line with S and I1 together (`SI1.csv`).
3. Stimulation of a cell line with S and I2 together (`SI2.csv`).
4. Stimulation of a cell line with S with I1 and I2 together (`SI.csv`).

Task 7.27: Building a model from data

Use the experimental data you have been given to figure out the topology of the network that this data was simulated from.
However, that does not mean that there are not other topologies that will fit the data. Here are some points to consider:

- Firstly, look at the `csv` files and plot your graphs. Use Python to write a function to visualise your results. This should go in `data_analysis.py`. You should read your data into Python using `pandas.read_csv` then iterate over your column names to plot the data using `matplotlib` and `seaborn`. The most effective way to visualise would be a to tile the plots from a single experiments in a grid, which you can do using `plt.subplots`.
- Read the experimental protocol. This is essential for determining how to set up your simulations.
- Draw a wiring diagram(s) for candidate topologies that might fit the data. You're welcome to use CellDesigner but remember it can take more time than just drawing a rough diagram by hand.
- Build the model(s) using antimony. Simulate and plot the data with whichever tool you like.
- Use parameter estimation and model selection methods to calibrate your model to the experimental data.
- If you have a core hypothesis and some other ideas that you are less certain of, you could try using AntimonyCombinations to build combinations of model before configuring them for parameter estimation using PyCoTools.

References

U. Alon. Network motifs: theory and experimental approaches. *Nature Reviews Genetics*, 8(6):450, 2007.

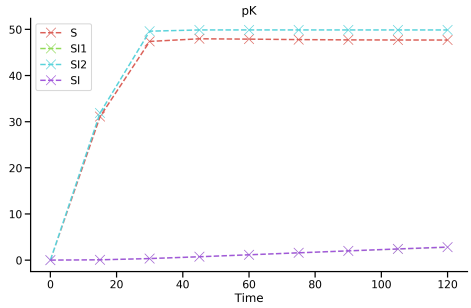

Figure 3: Measurements of phosphorylated K

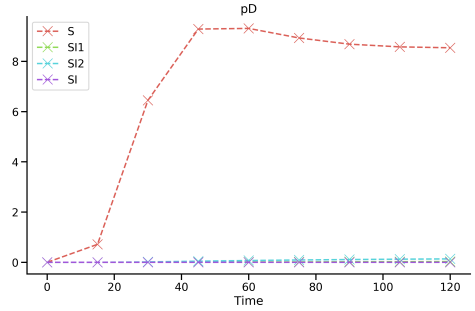

Figure 4: Measurements of phosphorylated D

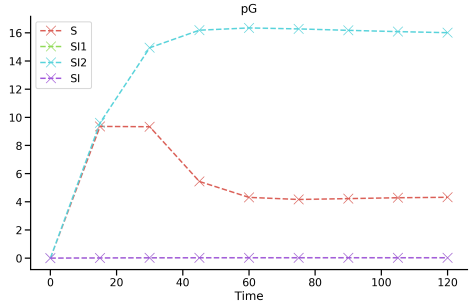

Figure 5: Measurements of phosphorylated G

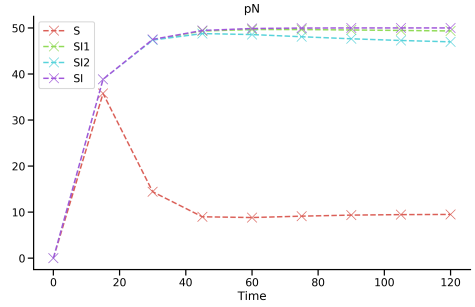

Figure 6: Measurements of phosphorylated N

A. Raue, M. Schilling, J. Bachmann, A. Matteson, M. Schelke, D. Kaschek, S. Hug, C. Kreutz, B. D. Harms, F. J. Theis, et al. Lessons learned from quantitative dynamical modeling in systems biology. *PloS one*, 8(9):e74335, 2013.