Pycotools Documentation

Release 1

Ciaran Welsh

Dec 10, 2019

Contents

1 Installation **3**

2 Troubleshooting **5**

3 Troubleshooting **7**

4 Documentation **9**
- 4.1 Getting started 9
- 4.2 Tutorials 14
- 4.3 Examples 38
- 4.4 API documentation 73

5 Support **127**

6 People **129**

7 Caveats **131**
- 7.1 Citing PyCoTools 131

Index **133**

PyCoTools is a python package that was developed as an alternative interface into COPASI, simulation software for modelling biochemical systems. The PyCoTools paper can be found here and describes in detail the intentions and functionality of PyCoTools. There are some important differences between the PyCoTools version that is described in the publication and the current version. The first is that PyCoTools is now a python 3 only package. If using Python 2.7 you should create a virtual Python 3.6 environment using conda or virtualenv. My preference is conda. The other major difference is the interface to COPASI's parameter estimation task which is described in the tutorials and examples.

Note: I am in the process of improving this documentation. Although still valid, content in the Tutorials page is quite old and is in the process of being replaced by content in the examples.

CHAPTER 1

Installation

First make sure you use a Python 3.6 environment.

> **Warning:** Using Python 3.7 or 3.8 will not work at this time due to dependency issues (which are unfortunately out of my control).

Then use:

```
$ pip install pycotools3
```

Remember to `source activate` your python 3.6 environment if you need to.

To install from source:

```
$ git clone https://github.com/CiaranWelsh/pycotools3.git
$ cd pycotools3
$ python setup.py install
```

The procedure is the same in linux, mac and windows.

CHAPTER 2

Troubleshooting

Pycotools3 is only supported in Python 3 to Python 3.6. If you are using Python 2.7 or Python 3.7 please create a new conda Python3.7 environment.

```
$ conda create -n py36 python=3.6
$ conda activate py36
$ pip install pycotools3
```

The same commands should work cross platform.

CHAPTER 3

Troubleshooting

If you get errors when trying to build a model using `pycotools3.model.loada()` Make sure you have installed Copasi and added the *Copasi/bin* directory to the path variable.

- On Linux
- On Windows
- On Mac

CHAPTER 4

Documentation

This is a guide to PyCoTools version >2.0.1.

4.1 Getting started

As PyCoTools only provides an alternative interface into some of COPASI's tasks, if you are unfamiliar with COPASI, it is a good idea to become acquainted prior to proceeding. As much as possible, arguments to PyCoTools functions follow the corresponding option in the COPASI user interface.

In addition to COPASI, PyCoTools depends on tellurium which is a Python package for modelling biological systems. While tellurium and COPASI have some of the same features, generally they are complementary and productivity is enhanced by using both together.

More specifically, tellurium uses antimony strings to define a model which is then converted into SBML. PyCoTools provides the `model.BuildAntimony` class which is a wrapper around this tellurium feature, which creates a Copasi model and parses it into a PyCoTools `model.Model`.

Since antimony is described elsewhere we will focus here on using antimony to build a copasi model.

4.1.1 Build a model with antimony

[2]:
```
import site, os
from pycotools3 import model

working_directory = os.path.abspath('') #create a base directory to␣
↪work from
```

(continues on next page)

(continued from previous page)

```
copasi_filename = os.path.join(working_directory,
↪'NegativeFeedbackModel.cps') #create a string to a copasi file on␣
↪system
antimony_string =  '''
    model negative_feedback()
        // define compartments
        compartment cell = 1.0
        //define species
        var A in cell
        var B in cell
        //define some global parameter for use in reactions
        vAProd = 0.1
        kADeg  = 0.2
        kBProd = 0.3
        kBDeg  = 0.4
        //define initial conditions
        A      = 0
        B      = 0
        //define reactions
        AProd:  => A; cell*vAProd
        ADeg: A =>  ; cell*kADeg*A*B
        BProd:  => B; cell*kBProd*A
        BDeg: B =>  ; cell*kBDeg*B
    end
    '''
negative_feedback = model.loada(antimony_string, copasi_filename) #␣
↪load the model
print(negative_feedback)
assert os.path.isfile(copasi_filename) #check that the model exists
```

```
Model(name=negative_feedback, time_unit=s, volume_unit=l, quantity_
↪unit=mol)
```

4.1.2 Create an antmiony string from an existing model

The Copasi user interface is an excellant way of constructing a model and it is easy to convert this model into an antimony string that can be pasted into a document.

[4]:
```
print(negative_feedback.to_antimony())
```

```
// Created by libAntimony v2.9.4
function Constant_flux__irreversible(v)
  v;
end

function Function_for_ADeg(A, B, kADeg)
  kADeg*A*B;
end
```

(continues on next page)

(continued from previous page)

```
function Function_for_BProd(A, kBProd)
  kBProd*A;
end


model *negative_feedback()

  // Compartments and Species:
  compartment cell;
  species A in cell, B in cell;

  // Reactions:
  AProd:  => A; cell*Constant_flux__irreversible(vAProd);
  ADeg: A => ; cell*Function_for_ADeg(A, B, kADeg);
  BProd:  => B; cell*Function_for_BProd(A, kBProd);
  BDeg: B => ; cell*kBDeg*B;

  // Species initializations:
  A = 0;
  B = 0;

  // Compartment initializations:
  cell = 1;

  // Variable initializations:
  vAProd = 0.1;
  kADeg = 0.2;
  kBProd = 0.3;
  kBDeg = 0.4;

  // Other declarations:
  const cell, vAProd, kADeg, kBProd, kBDeg;
end
```

One paradigm of model development is to use antimony to 'hard code' perminent changes to the model and the Copasi user interface for experimental changes. The `Model.open()` method is useful for this paradigm as it opens the model with whatever configurations have been defined.

[5]:
```
negative_feedback.open()
```

4.1.3 Simulate a time course

Since we have used an antimony string, we can simulate this model with either tellurium or Copasi. Simulating with tellurium uses a library called roadrunner which is described in detail elsewhere. To run a simulation with Copasi we need to configure the time course task, make the task executable (i.e. tick the check box in the top right of the time course task) and run the

simulation with CopasiSE. This is all taken care of by the `tasks.TimeCourse` class.

[6]:
```
from pycotools3 import tasks
time_course = tasks.TimeCourse(negative_feedback, end=100, step_
↪size=1, intervals=100)
time_course
```

[6]:
```
<pycotools3.tasks.TimeCourse at 0x1ff411e5588>
```

However a more convenient interface is provided by the `model.simulate` method, which is a wrapper around `tasks.TimeCourse` which additionally parses the resulting data from file and returns a `pandas.DataFrame`

[7]:
```
from pycotools3 import tasks
fname = os.path.join(os.path.abspath(''), 'timcourse.txt')
sim_data = negative_feedback.simulate(0, 100, 1, report_name=fname)
↪##start, end, by
sim_data.head()
```

[7]:

Time	A	B
0	0.000000	0.000000
1	0.099932	0.013181
2	0.199023	0.046643
3	0.295526	0.093275
4	0.387233	0.147810

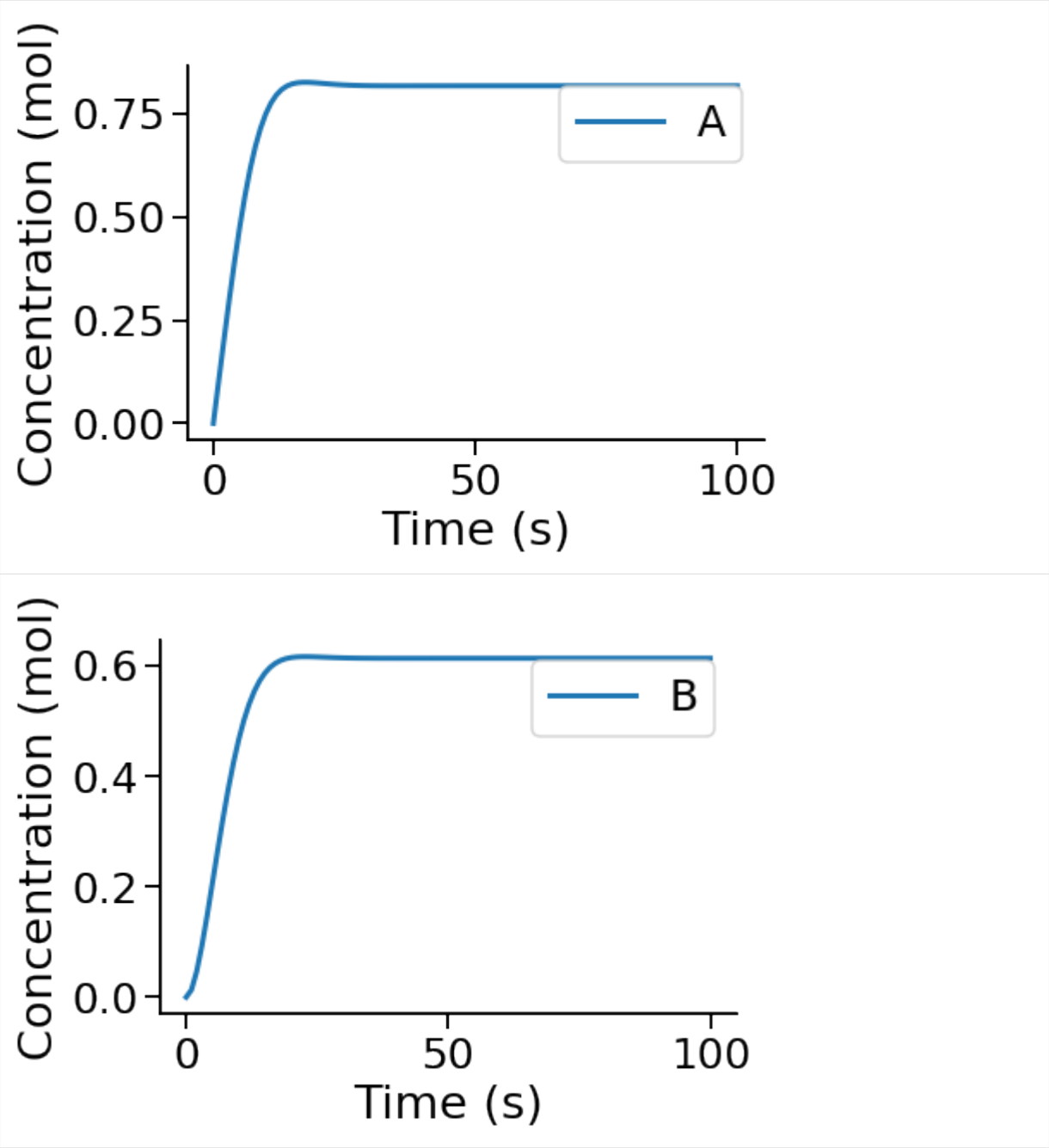
The results are saved in a file defined by the `report_name` option, which defaults to `timecourse.txt` in the same directory as the copasi model.

4.1.4 Visualise a time course

PyCoTools also provides facilities for visualising simulation output. To plot a timecourse, pass the `task.TimeCourse` object to the `viz.PlotTimeCourse` object.

[8]:
```
from pycotools3 import viz
viz.PlotTimeCourse(time_course, savefig=True)
```

[8]:
```
<pycotools3.viz.PlotTimeCourse at 0x1ff411e53c8>
```



More information about running time courses with PyCoTools and Copasi can be found in the *time course tutorial*

4.1.5 Run Parameter Estimation

The following configures a regular copasi parameter estimation (`context='s'`) on all global and initial concentration parameters (`parameters='gm'`) using the genetic algorithm

[15]:
```
from pycotools3 import tasks, viz

with tasks.ParameterEstimation.Context(negative_feedback, fname,␣
↪context='s', parameters='gm') as context:
```

(continues on next page)

(continued from previous page)

```
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 50)
    context.set('number_of_generations', 300)
    context.set('run_mode', True) ##defaults to False
    context.set('pe_number', 2) ## number of repeat items in scan␣
↪task
    context.set('copy_number', 2) ## number of times to copy model
    config = context.get_config()

pe = tasks.ParameterEstimation(config)
data = viz.Parse(pe)
print(data)
```

```
{'NegativeFeedbackModel':           A         B       RSS      kADeg␣
↪     kBDeg    kBProd    vAProd
0  0.000004  0.000321  0.000348  0.198104  0.404187  0.298985  0.
↪099467
1  0.000397  0.000009  0.000358  0.201986  0.396722  0.296337  0.
↪100254
2  0.002442  0.000002  0.000399  0.197253  0.403480  0.299229  0.
↪099694
3  0.110722  0.004461  0.002555  0.198449  0.402032  0.297208  0.
↪097885}
```

pycotools3 supports the configuration of:

- *multiple models at once*
- *multiple parameter estimation repeats at once*
- *profile likelihoods*
- *cross validations*

Also you can checkout the *parameter estimation tutorial*.

4.2 Tutorials

In this section of the documentation I provide detailed explainations of how PyCoTools works, with examples. The tutorials are split into sections which are linked to below.

4.2.1 Running Time-Courses

Copasi enables users to simulate their model with a range of different solvers.

Create a model

Here we do our imports and create the model we use for the tutorial

[1]:
```
import os
import site
from pycotools3 import model, tasks, viz

working_directory = r'/home/ncw135/Documents/pycotools3/docs/source/
↪Tutorials/timecourse_tutorial'
if not os.path.isdir(working_directory):
    os.makedirs(working_directory)

copasi_file = os.path.join(working_directory, 'michaelis_menten.cps
↪')

if os.path.isfile(copasi_file):
    os.remove(copasi_file)

antimony_string = """
model michaelis_menten()
    compartment cell = 1.0
    var E in cell
    var S in cell
    var ES in cell
    var P in cell

    kf = 0.1
    kb = 1
    kcat = 0.3
    E = 75
    S = 1000

    SBindE: S + E => ES; kf*S*E
    ESUnbind: ES => S + E; kb*ES
    ProdForm: ES => P + E; kcat*ES
end
"""

with model.BuildAntimony(copasi_file) as loader:
    mm = loader.load(antimony_string)


mm
```

```
The BuildAntimony context manager is deprecated and will be removed␣
↪in future versions. Please use model.loada instead.
```

[1]:
```
Model(name=michaelis_menten, time_unit=s, volume_unit=l, quantity_
↪unit=mol)
```

Deterministic Time Course

[2]:
```
TC = tasks.TimeCourse(
    mm, report_name='mm_simulation.txt',
    end=1000, intervals=50, step_size=20
)

## check its worked
os.path.isfile(TC.report_name)

import pandas
df = pandas.read_csv(TC.report_name, sep='\t')
df.head()
```

[2]:
```
   Time       [E]          [S]       [ES]       [P]  Values[kf]  ␣
↪Values[kb]  \
0     0  75.00000  1000.000000   1.000000     1.000         0.1  ␣
↪       1
1    20   2.00306   479.797000  73.996900   448.206         0.1  ␣
↪       1
2    40  12.80010    62.104800  63.199900   876.695         0.1  ␣
↪       1
3    60  75.13160     0.119830   0.868371  1001.010         0.1  ␣
↪       1
4    80  75.99560     0.000604   0.004434  1001.990         0.1  ␣
↪       1

   Values[kcat]
0           0.3
1           0.3
2           0.3
3           0.3
4           0.3
```

When running a time course, you should ensure that the number of intervals times the step size equals the end time, i.e.:

```
- $$intervals \cdot step\_size = end$$
```

The default behaviour is to output all model variables as they can easily be filtered later in the Python environment. However, the `metabolites`, `global_quantities` and `local_parameters` arguments exist to filter the variables that are simulated prior to running the time course.

[3]:
```
TC=tasks.TimeCourse(
    mm,
    report_name='mm_timecourse.txt',
    end=100,
    intervals=50,
    step_size=2,
```

(continues on next page)

(continued from previous page)

```
    global_quantities = ['kf'], ##recall that antimony puts all␣
↪parameters as global quantities
)

##check that we only kf as a global variables
pandas.read_csv(TC.report_name,sep='\t').head()
```

[3]:

	Time	[E]	[S]	[ES]	[P]	Values[kf]
0	0	75.00000	1000.000	1.0000	1.0000	0.1
1	2	1.10437	881.378	74.8956	45.7263	0.1
2	4	1.16265	836.516	74.8373	90.6465	0.1
3	6	1.22737	791.698	74.7726	135.5300	0.1
4	8	1.29962	746.928	74.7004	180.3720	0.1

An alternative and more convenient interface into the `tasks.TimeCourse` class is using the `model.Model.simulate` method. This is simply a wrapper and is used like so.

[4]:
```
data = mm.simulate(start=0, stop=100, by=0.1)
data.head()
```

[4]:

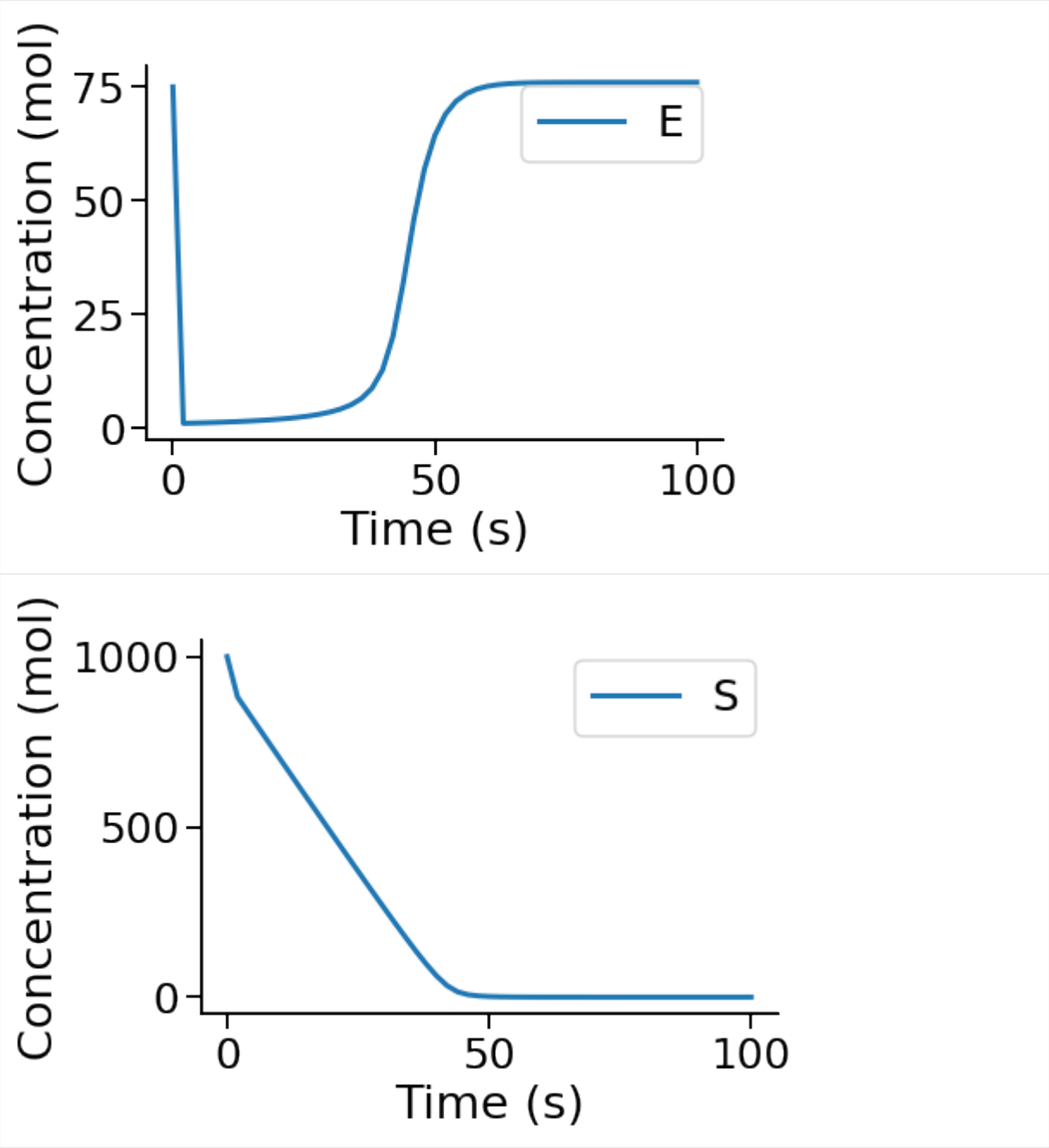
Time	E	ES	P	S
0.0	75.00000	1.0000	1.00000	1000.000
0.1	1.05984	74.9402	3.02124	924.039
0.2	1.05665	74.9433	5.26956	921.787
0.3	1.05920	74.9408	7.51783	919.541
0.4	1.06175	74.9382	9.76601	917.296

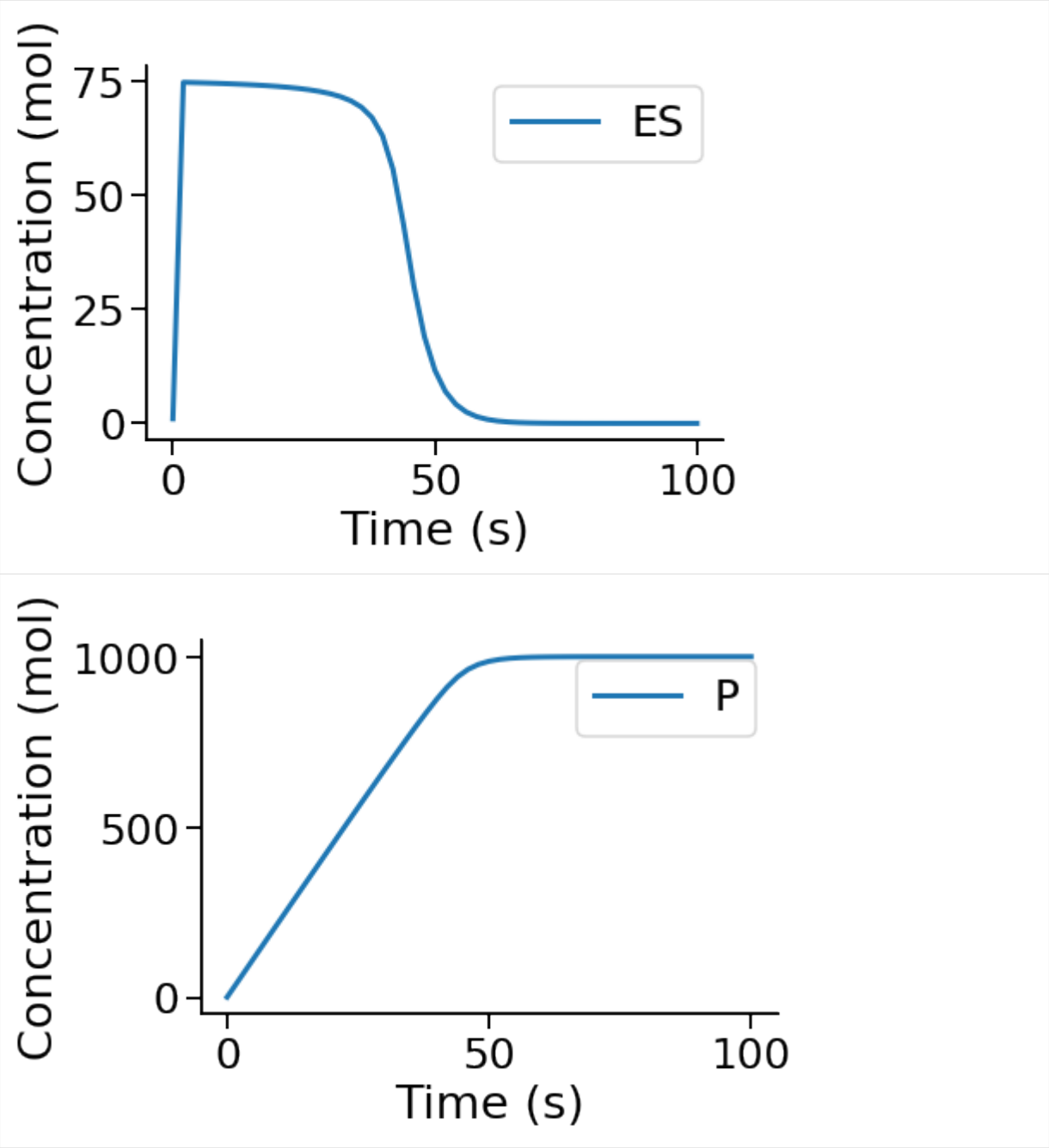
This mechanism of running a time course has the advantage that 1) pycotools parses the data back into python in the form of a `pandas.DataFrame` and 2) the column names are automatically pruned to remove the copasi reference information.

Visualization

[5]:
```
viz.PlotTimeCourse(TC)
```

[5]:
```
<pycotools3.viz.PlotTimeCourse at 0x16c80294358>
```

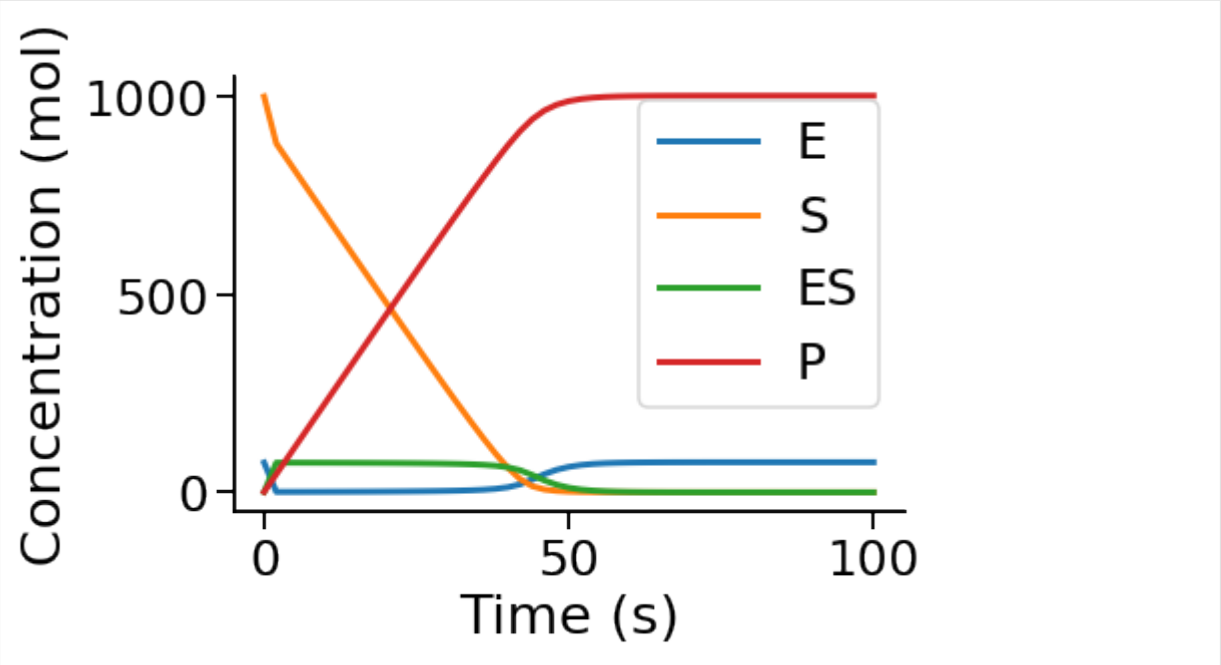

Concentration (mol)
75
50
25
0
0
50
100
Time (s)
E
Concentration (mol)
1000
500
0
0
50
100
Time (s)
S



It is also possible to plot these on the same axis by specifying `separate=False`

[6]:
```
viz.PlotTimeCourse(TC, separate=False)
```

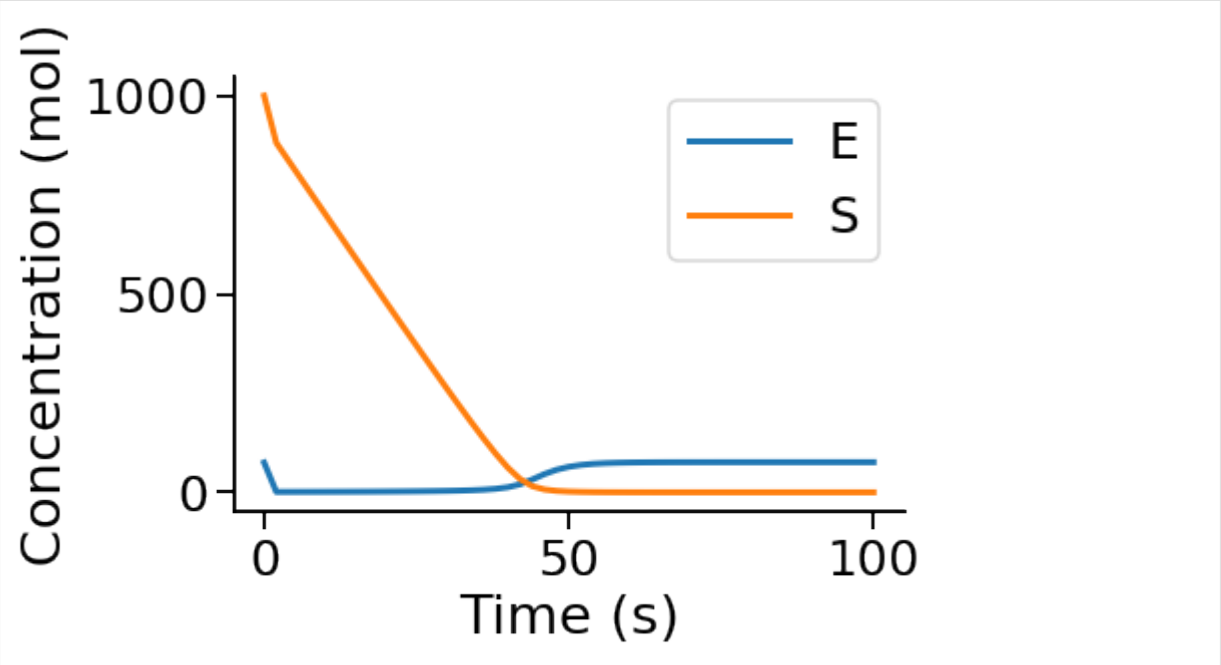
[6]:
```
<pycotools3.viz.PlotTimeCourse at 0x16ca06f91d0>
```

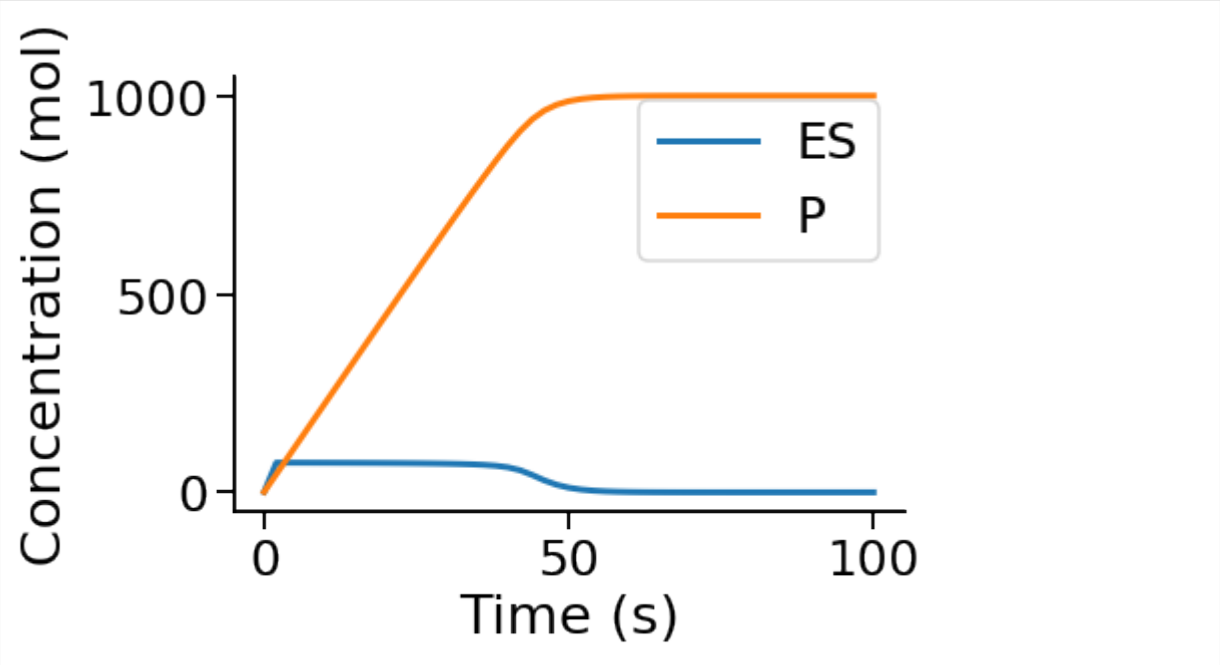


or to choose the y variables,

[7]:
```
viz.PlotTimeCourse(TC, y=['E', 'S'], separate=False)
viz.PlotTimeCourse(TC, y=['ES', 'P'], separate=False)
```

[7]:
```
<pycotools3.viz.PlotTimeCourse at 0x16ca0760748>
```



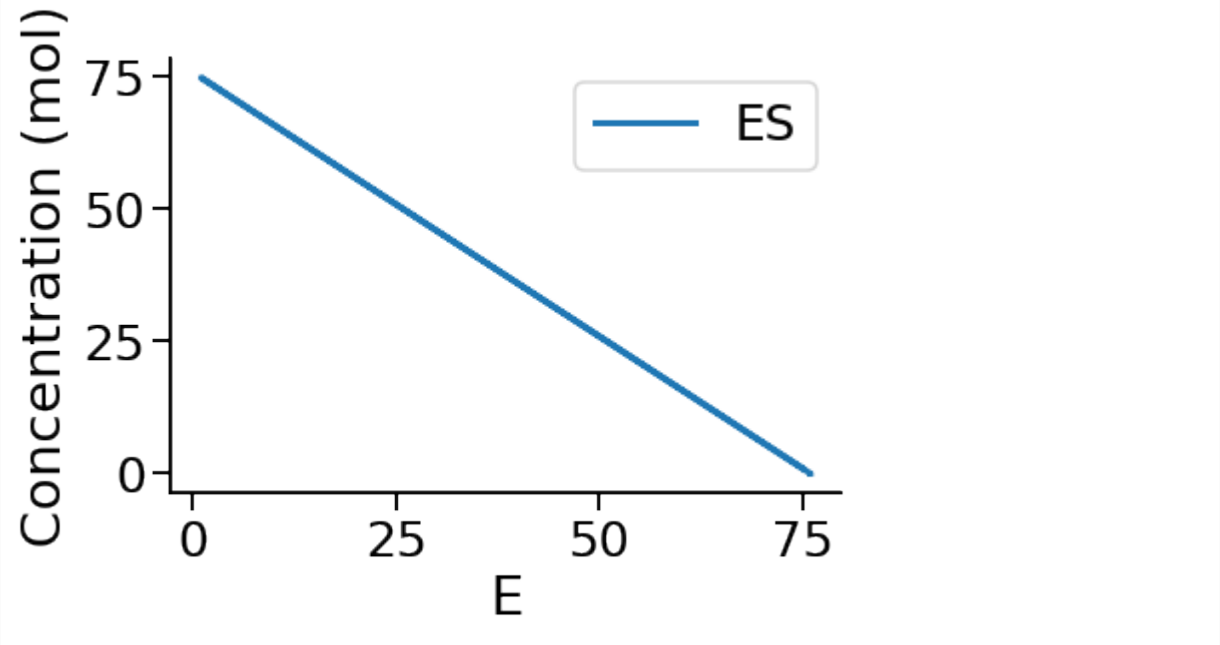


Plot in Phase Space

Choose the x variable to plot phase space. Same arguments apply as above.

[8]:
```
viz.PlotTimeCourse(TC, x='E', y='ES', separate=True)
```

[8]:
```
<pycotools3.viz.PlotTimeCourse at 0x16ca07c1b38>
```

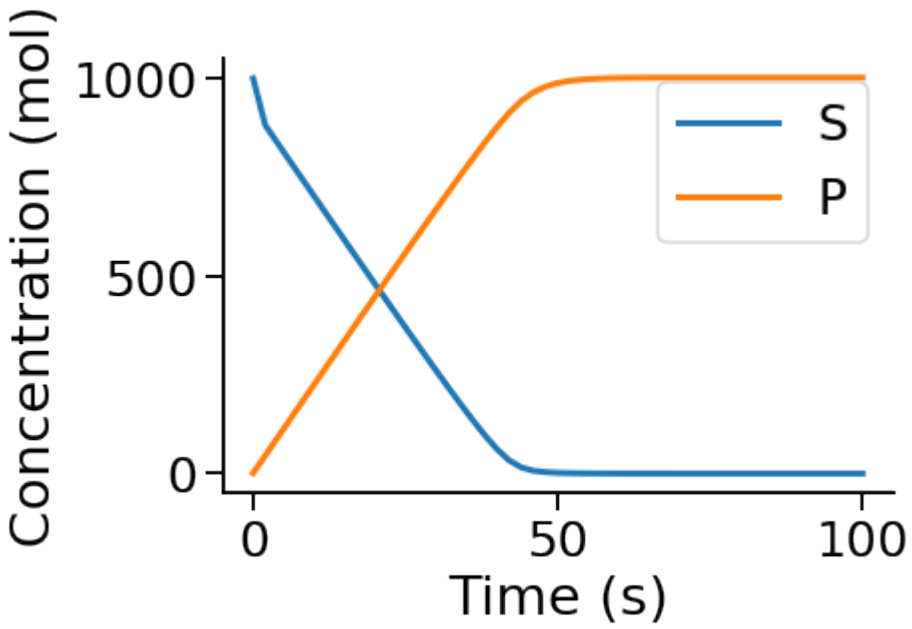


Save to file

Use the `savefig=True` option to save the figure to file and give an argument to the filename option to choose the filename.

```
[9]: viz.PlotTimeCourse(TC, y=['S', 'P'], separate=False, savefig=True,␣
     ↪filename='MyTimeCourse.png')
```

```
[9]: <pycotools3.viz.PlotTimeCourse at 0x16ca0847cf8>
```



Alternative Solvers

Valid arguments for the `method` argument of `TimeCourse` are:

- deterministic
- direct
- gibson_bruck
- tau_leap
- adaptive_tau_leap
- hybrid_runge_kutta
- hybrid_lsoda

Copasi also includes a `hybrid_rk45` solver but this is not yet supported by Pycotools. To use an alternative solver, pass the name of the solver to the `method` argument.

Stochastic MM

For demonstrating simulation of stochastic time courses we build another michaelis-menten type reaction schema. We need to do this so we can set `unit substance = item`, or in other words, change the model to particle numbers - otherwise there are too may molecules in the system to simulate a stochastic model

[10]:
```
copasi_file = os.path.join(working_directory, 'michaelis_menten_
↪stochastic.cps')

antimony_string = """
model michaelis_menten()
    compartment cell = 1.0;
    var E in cell;
    var S in cell;
    var ES in cell;
    var P in cell;

    kf = 0.1;
    kb = 1;
    kcat = 0.3;
    E = 75;
    S = 1000;

    SBindE: S + E => ES; kf*S*E;
    ESUnbind: ES => S + E; kb*ES;
    ProdForm: ES => P + E; kcat*ES;

    unit substance = item;

end
"""

with model.BuildAntimony(copasi_file) as loader:
    mm = loader.load(antimony_string)
```

```
The BuildAntimony context manager is deprecated and will be removed␣
↪in future versions. Please use model.loada instead.
```

Run a Time Course Using Direct Method

[11]:
```
data = mm.simulate(0, 100, 1, method='direct')
data.head(n=10)
```

[11]:

Time	E	ES	P	S
0	75	1	1	1000
1	0	76	18	908
2	2	74	36	892

(continues on next page)

(continued from previous page)

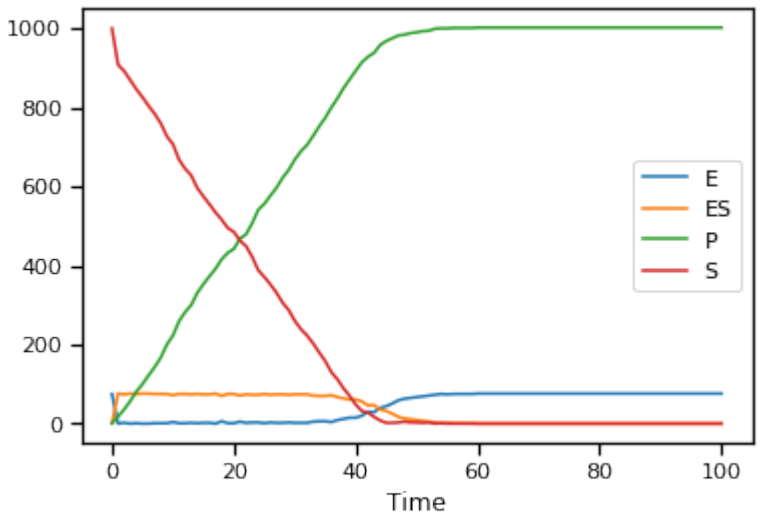
```
3       0   76    57    869
4       1   75    81    846
5       0   76   100    826
6       0   76   122    804
7       1   75   143    784
8       1   75   167    760
9       1   75   200    727
```

Plot stochastic time course

Note that we can also use the `pandas`, `matplotlib` and `seaborn` libraries for plotting

[12]:
```
import matplotlib
import seaborn
seaborn.set_context('notebook')
data.plot()
```

[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x16ca0810c88>
```



Notice how similar the stochastic simulation is to the deterministic. As the number of molecules being simulated increases, the stochastic simulation converges to the deterministic solution. Another way to put it, stochastic effects are greatest when simulating small molecule numbers

4.2.2 Insert Parameters

Parameters can be inserted automatically into a Copasi model.

Build a demonstration model

While antimony or the COPASI user interface are the preferred ways to build a model, PyCoTools does have a mechanism for constructing COPASI models. For variation and demonstra-

tion, this method is used here.

[1]:
```
import os
import site
from pycotools3 import model, tasks, viz
## Choose a working directory for model
working_directory = os.path.abspath('')
copasi_file = os.path.join(working_directory, 'MichaelisMenten.cps')

if os.path.isfile(copasi_file):
    os.remove(copasi_file)


kf = 0.01
kb = 0.1
kcat = 0.05
with model.Build(copasi_file) as m:
    m.name = 'Michaelis-Menten'
    m.add('compartment', name='Cell')

    m.add('metabolite', name='P', concentration=0)
    m.add('metabolite', name='S', concentration=30)
    m.add('metabolite', name='E', concentration=10)
    m.add('metabolite', name='ES', concentration=0)

    m.add('reaction', name='S bind E', expression='S + E -> ES',␣
↪rate_law='kf*S*E',
          parameter_values={'kf': kf})

    m.add('reaction', name='S unbind E', expression='ES -> S + E',␣
↪rate_law='kb*ES',
         parameter_values={'kb': kb})

    m.add('reaction', name='ES produce P', expression='ES -> P + E',
↪ rate_law='kcat*ES',
          parameter_values={'kcat': kcat})

mm = model.Model(copasi_file)
mm
```

[1]:
```
Model(name=Michaelis-Menten, time_unit=s, volume_unit=ml, quantity_
↪unit=mmol)
```

Insert Parameters from Python Dictionary

[2]:
```
params = {'E': 100,
          'P': 150}

## Insert into model
```

(continues on next page)

(continued from previous page)

```
I = model.InsertParameters(mm, parameter_dict=params)
##format the parameters for displaying nicely
I.parameters.index = ['Parameter Value']
I.parameters.transpose()
```

[2]:
```
   Parameter Value
E              100
P              150
```

Alternatively use `inplace=True` argument (analogous to the `pandas` library) to modify the object inplace, rather than needing to assign

[3]:
```
model.InsertParameters(mm, parameter_dict=params, inplace=True)
```

[3]:
```
<pycotools3.model.InsertParameters at 0x15c51a255c0>
```

Insert Parameters from Pandas DataFrame

[4]:
```
import pandas
params = {'(S bind E).kf': 50,
          '(S unbind E).kb': 96}
df = pandas.DataFrame(params, index=[0])
df
```

[4]:
```
   (S bind E).kf  (S unbind E).kb
0             50               96
```

[ ]:
```

```

[5]:
```
model.InsertParameters(mm, df=df, inplace=True)
```

[5]:
```
<pycotools3.model.InsertParameters at 0x15c519f8dd8>
```

Insert Parameters from Parameter Estimation Output

First we'll get some parameter estimation data by *fitting* a model to simulated data.

[6]:
```
fname = os.path.join(os.path.abspath(''), 'timecourse.txt')
print(fname)
data = mm.simulate(0, 50, 1, report_name=fname)
assert os.path.isfile(fname)
```

```
D:\pycotools3\docs\source\Tutorials\timecourse.txt
```

[7]:
```
with  tasks.ParameterEstimation.Context(copasi_file, fname, context=
↪'s', parameters='l') as context:
    context.set('randomize_start_values', True)
    context.set('lower_bound', 0.01)
```

(continues on next page)

(continued from previous page)

```
    context.set('upper_bound', 100)
    context.set('run_mode', True)
    config = context.get_config()
print(config)
PE = tasks.ParameterEstimation(config)
```

```
datasets:
    experiments:
        timecourse:
            affected_models:
            - MichaelisMenten
            filename: D:
↪\pycotools3\docs\source\Tutorials\timecourse.txt
            mappings:
                E:
                    model_object: E
                    object_type: Metabolite
                    role: dependent
                ES:
                    model_object: ES
                    object_type: Metabolite
                    role: dependent
                P:
                    model_object: P
                    object_type: Metabolite
                    role: dependent
                S:
                    model_object: S
                    object_type: Metabolite
                    role: dependent
                Time:
                    model_object: Time
                    role: time
            normalize_weights_per_experiment: true
            separator: "\t"
    validations: {}
items:
    fit_items:
        (ES produce P).kcat:
            affected_experiments:
            - timecourse
            affected_models:
            - MichaelisMenten
            affected_validation_experiments: []
            lower_bound: 1.0e-06
            start_value: model_value
            upper_bound: 1000000.0
        (S bind E).kf:
            affected_experiments:
            - timecourse
```

(continues on next page)

(continued from previous page)

```
            affected_models:
            - MichaelisMenten
            affected_validation_experiments: []
            lower_bound: 1.0e-06
            start_value: model_value
            upper_bound: 1000000.0
        (S unbind E).kb:
            affected_experiments:
            - timecourse
            affected_models:
            - MichaelisMenten
            affected_validation_experiments: []
            lower_bound: 1.0e-06
            start_value: model_value
            upper_bound: 1000000.0
models:
    MichaelisMenten:
        copasi_file: D:
↪\pycotools3\docs\source\Tutorials\MichaelisMenten.cps
        model: Model(name=Michaelis-Menten, time_unit=s, volume_
↪unit=ml, quantity_unit=mmol)
settings:
    calculate_statistics: false
    config_filename: config.yml
    context: s
    cooling_factor: 0.85
    copy_number: 1
    create_parameter_sets: false
    cross_validation_depth: 1
    fit: 1
    iteration_limit: 50
    lower_bound: 0.01
    max_active: 3
    method: genetic_algorithm
    number_of_generations: 200
    number_of_iterations: 100000
    overwrite_config_file: false
    pe_number: 1
    pf: 0.475
    pl_lower_bound: 1000
    pl_upper_bound: 1000
    population_size: 50
    prefix: null
    problem: Problem1
    quantity_type: concentration
    random_number_generator: 1
    randomize_start_values: true
    report_name: PEData.txt
    results_directory: ParameterEstimationData
```

(continues on next page)

(continued from previous page)

```
    rho: 0.2
    run_mode: true
    save: false
    scale: 10
    seed: 0
    start_temperature: 1
    start_value: 0.1
    std_deviation: 1.0e-06
    swarm_size: 50
    tolerance: 1.0e-05
    update_model: false
    upper_bound: 100
    use_config_start_values: false
    validation_threshold: 5
    validation_weight: 1
    weight_method: mean_squared
    working_directory: D:\pycotools3\docs\source\Tutorials
```

Now we can insert the estimated parameters using:

[8]:
```
##index=0 for best parameter set (i.e. lowest RSS)
model.InsertParameters(mm, parameter_path=PE.results_directory[
↪'MichaelisMenten'], index=0, inplace=True)
```

[8]:
```
<pycotools3.model.InsertParameters at 0x15c6fb8c2e8>
```

Insert Parameters using the `model.Model().insert_parameters` method

The same means of inserting parameters can be used from the model object itself

[9]:
```
mm.insert_parameters(parameter_dict=params, inplace=True)
```

Change parameters using `model.Model().set`

Individual parameters can also be changed using the `set` method. For example, we could set the `metabolite` with name `S concentration` or `particle numbers` to `55`

[10]:
```
mm.set('metabolite', 'S', 55, 'name', 'concentration')

## or

mm.set('metabolite', 'S', 55, 'name', 'particle_numbers')
```

[10]:
```
Model(name=Michaelis-Menten, time_unit=s, volume_unit=ml, quantity_
↪unit=mmol)
```

4.2.3 Parameter Scan

Copasi supports three types of scan, a regular parameter scan, a repeat scan and sampling from a parametric distributions.

We first build a model to work with throughout the tutorial.

[1]:
```
import os
import site
from pycotools3 import model, tasks, viz
import pandas

working_directory = r'/home/ncw135/Documents/pycotools3/docs/source/
↪Tutorials/timecourse_tutorial'
if not os.path.isdir(working_directory):
    os.makedirs(working_directory)

copasi_file = os.path.join(working_directory, 'michaelis_menten.cps
↪')

if os.path.isfile(copasi_file):
    os.remove(copasi_file)

antimony_string = """
model michaelis_menten()
    compartment cell = 1.0
    var E in cell
    var S in cell
    var ES in cell
    var P in cell

    kf = 0.1
    kb = 1
    kcat = 0.3
    E = 75
    S = 1000

    SBindE: S + E => ES; kf*S*E
    ESUnbind: ES => S + E; kb*ES
    ProdForm: ES => P + E; kcat*ES
end
"""

with model.BuildAntimony(copasi_file) as loader:
    mm = loader.load(antimony_string)


mm
```

```
The BuildAntimony context manager is deprecated and will be removed␣
↪in future versions. Please use model.loada instead.
```

[1]:
```
Model(name=michaelis_menten, time_unit=s, volume_unit=l, quantity_
↪unit=mol)
```

[2]:
```
S = tasks.Scan(
    mm, scan_type='scan', subtask='time_course', report_type='time_
↪course',
    report_name = 'ParameterScanOfTimeCourse.txt', variable='S',
    minimum=1, maximum=20, number_of_steps=8, run=True,
)

## Now check parameter scan data exists
os.path.isfile(S.report_name)
```

[2]:
```
True
```

Two Way Parameter Scan

By default, scan tasks are removed before setting up a new scan. To set up dual scans, set clear_scans to False in a second call to `Scan` so that the first is not removed prior to adding the second.

[3]:
```
## Clear scans for setting up first scan
tasks.Scan(
    mm, scan_type='scan', subtask='time_course', report_type='time_
↪course',
    variable='E', minimum=1, maximum=20, number_of_steps=8,␣
↪run=False, clear_scan=True,
)

## do not clear tasks when setting up the second
S = tasks.Scan(
    mm, scan_type='scan', subtask='time_course', report_type='time_
↪course',
    report_name = 'TwoWayParameterScanOfTimeCourse.csv', variable='S
↪',
    minimum=1, maximum=20, number_of_steps=8, run=True, clear_
↪scan=False,
)

## check the output exists
os.path.isfile(S.report_name)
```

[3]:
```
True
```

An arbitrary number of scans can be setup this way. Further, its possible to chain together scans with repeat or random distribution scans.

Repeat Scan Items

Repeat scans are very useful for running multiple parameter estimations and for running stochastic time courses.

[4]:
```
## Assume Parameter Estimation task already configured
tasks.Scan(
    mm, scan_type='repeat', subtask='parameter_estimation', report_
↪type='parameter_estimation',
    number_of_steps=6, run=False, ##set run to True to run via␣
↪CopasiSE
)


## Assume model runs stochastically and time course settings are␣
↪already configured
tasks.Scan(
    mm, scan_type='repeat', subtask='time_course', report_type=
↪'time_course',
    number_of_steps=100, run=False,  ##set run to True to run via␣
↪CopasiSE
)
```

[4]:
```
<pycotools3.tasks.Scan at 0x1cb859370b8>
```

4.2.4 Parameter Estimation

[4]:
```
import os, glob
import site
from pycotools3 import viz, model, misc, tasks
from io import StringIO
import pandas
%matplotlib inline
```

Build a Model

[8]:
```
working_directory = os.path.abspath('')

copasi_file = os.path.join(working_directory, 'negative_feedback.cps
↪')
ant = """
        model negative_feedback
            compartment cell = 1.0
            var A in cell
            var B in cell

            vAProd = 0.1
```

(continues on next page)

(continued from previous page)

```
            kADeg = 0.2
            kBProd = 0.3
            kBDeg = 0.4
            A = 0
            B = 0

            AProd: => A; cell*vAProd
            ADeg: A =>; cell*kADeg*A*B
            BProd: => B; cell*kBProd*A
            BDeg: B => ; cell*kBDeg*B
        end
        """
mod = model.loada(ant, copasi_file)

## open model in copasi
#mod.open()
mod
```

[8]:
```
Model(name=negative_feedback, time_unit=s, volume_unit=l, quantity_
↪unit=mol)
```

Collect some experimental data

Organise your experimental data into delimited text files

[9]:
```
experimental_data = StringIO(
    """
Time,A,B
 0, 0.000000, 0.000000
 1, 0.099932, 0.013181
 2, 0.199023, 0.046643
 3, 0.295526, 0.093275
 4, 0.387233, 0.147810
 5, 0.471935, 0.206160
 6, 0.547789, 0.265083
 7, 0.613554, 0.322023
 8, 0.668702, 0.375056
 9, 0.713393, 0.422852
10, 0.748359, 0.464639
    """.strip()
)

df = pandas.read_csv(experimental_data, index_col=0)

fname = os.path.join(os.path.abspath(''), 'experimental_data.csv')
df.to_csv(fname)

assert os.path.isfile(fname)
```

The Config Object

The interface to COPASI's parameter estimation using `pycotools3` revolves around the `ParameterEstimation.Config` object. `ParameterEstimation.Config` is a dictionary-like object which allows the user to define their parameter estimation problem. All features of COPASI's parameter estimations task are supported, including configuration of `validation experiments`, `affected experiments`, `affected validation experiments` and `constraints` as well additional features such as the configuration of multiple models simultaneously via the `affected_models` keyword.

The `ParameterEstimation.Config` object expects at the bare minimum some information about the models being configured, some experimental data, some fit items and a working directory. The remaining options are automatically filled in with defaults.

[10]:
```
config = tasks.ParameterEstimation.Config(
    models=dict(
        negative_feedback=dict(
            copasi_file=copasi_file
        )
    ),
    datasets=dict(
        experiments=dict(
            first_dataset=dict(
                filename=fname,
                separator=','
            )
        )
    ),
    items=dict(
        fit_items=dict(
            A={},
            B={},
        )
    ),
    settings=dict(
        working_directory=working_directory
    )
)
config
```

[10]:
```
datasets:
    experiments:
        first_dataset:
            affected_models:
            - negative_feedback
            filename: D:
↪\pycotools3\docs\source\Tutorials\experimental_data.csv
            mappings:
                A:
                    model_object: A
                    object_type: Metabolite
```

(continues on next page)

(continued from previous page)

```
                    role: dependent
                B:
                    model_object: B
                    object_type: Metabolite
                    role: dependent
                Time:
                    model_object: Time
                    role: time
            normalize_weights_per_experiment: true
            separator: ','
    validations: {}
items:
    fit_items:
        A:
            affected_experiments:
            - first_dataset
            affected_models:
            - negative_feedback
            affected_validation_experiments: []
            lower_bound: 1.0e-06
            start_value: model_value
            upper_bound: 1000000
        B:
            affected_experiments:
            - first_dataset
            affected_models:
            - negative_feedback
            affected_validation_experiments: []
            lower_bound: 1.0e-06
            start_value: model_value
            upper_bound: 1000000
models:
    negative_feedback:
        copasi_file: D:\pycotools3\docs\source\Tutorials\negative_
↪feedback.cps
        model: Model(name=negative_feedback, time_unit=s, volume_
↪unit=l, quantity_unit=mol)
settings:
    calculate_statistics: false
    config_filename: config.yml
    context: s
    cooling_factor: 0.85
    copy_number: 1
    create_parameter_sets: false
    cross_validation_depth: 1
    fit: 1
    iteration_limit: 50
    lower_bound: 1.0e-06
    max_active: 3
```

(continues on next page)

(continued from previous page)

```
    method: genetic_algorithm
    number_of_generations: 200
    number_of_iterations: 100000
    overwrite_config_file: false
    pe_number: 1
    pf: 0.475
    pl_lower_bound: 1000
    pl_upper_bound: 1000
    population_size: 50
    prefix: null
    problem: Problem1
    quantity_type: concentration
    random_number_generator: 1
    randomize_start_values: false
    report_name: PEData.txt
    results_directory: ParameterEstimationData
    rho: 0.2
    run_mode: false
    save: false
    scale: 10
    seed: 0
    start_temperature: 1
    start_value: 0.1
    std_deviation: 1.0e-06
    swarm_size: 50
    tolerance: 1.0e-05
    update_model: false
    upper_bound: 1000000
    use_config_start_values: false
    validation_threshold: 5
    validation_weight: 1
    weight_method: mean_squared
    working_directory: D:\pycotools3\docs\source\Tutorials
```

The COPASI user will be familiar with most of these settings, though there are also a few additional options.

Once built, a `ParameterEstimation.Config` object can be passed to `ParameterEstimation` object.

[11]:
```
PE = tasks.ParameterEstimation(config)
```

By default, the `run_mode` setting is set to False. To run the parameter estimation in background processes using CopasiSE, set `run_mode` to `True` or `parallel`.

[12]:
```
config.settings.run_mode = True
PE = tasks.ParameterEstimation(config)
viz.Parse(PE)['negative_feedback']
# config
```

[12]:
```
          A         B           RSS
0  0.000001  0.000001  7.955450e-12
```

Running multiple parameter estimations

With pycotools, parameter estimations are run via the scan task interface so that we have the option of running the same problem `pe_number` times. Additionally, pycotools provides a way of copying a model `copy_number` times so that the final number of parameter estimations that get executed is pe_numbercdot‘copy_number‘‘.

[13]:
```
config.settings.copy_number = 4
config.settings.pe_number = 2
config.settings.run_mode = True
PE = tasks.ParameterEstimation(config)
```

And sure enough we have ran the problem 8 times.

[14]:
```
viz.Parse(PE)['negative_feedback']
```

[14]:
```
          A         B           RSS
0  0.000001  0.000001  7.955430e-12
1  0.000001  0.000001  7.955450e-12
2  0.000001  0.000001  7.955450e-12
3  0.000001  0.000001  7.955450e-12
4  0.000001  0.000001  7.955450e-12
5  0.000001  0.000001  7.955450e-12
6  0.000001  0.000001  7.955450e-12
7  0.000001  0.000001  7.955450e-12
```

4.2.5 A shortcut for configuring the `ParameterEstimation.Config` object

Manually configuring the `ParameterEstimation.Config` object can take some time as it is bulky, but necessarily so in order to enable users to configure any type of parameter estimation. The `ParameterEstimation.Config` class should be used directly when a lower level interface into COPASI configurations are required. For instance, if you want to configure different boundaries for each parameter, choose which parameters are affected by which experiment, mix timecourse and steasy state experiments, define independent variables, add constraints or choose which models are affected by which experiments, you can use the `ParameterEstimation.Config` class directly.

However, if you want a more standard configuration such as all parameters estimated between the same boundaries, all experiments affecting all parameters and models etc.. then you can use the `ParameterEstimation.Context` class to build the `ParameterEstimation.Config` class for you. The `ParameterEstimation.Context` class has a `context` argument that defaults to `'s'` for `simple`. While not yet implemented, everntually, alternative options for `context` will be provided to support other common patters, such as

`cross_validation` or `chaser_estimations` (global followed by local algorithm). Note that an option is no longer required for `model_selection` since it is innately incorporated via the `affected_models` argument.

To use the `ParameterEstimation.Context` object

[17]:
```
with tasks.ParameterEstimation.Context(mod, fname, context='s',␣
↪parameters='g') as context:
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 10)
    context.set('copy_number', 4)
    context.set('pe_number', 2)
    context.set('run_mode', True)
    context.set('separator', ',')
    config = context.get_config()

pe = tasks.ParameterEstimation(config)
```

[18]:
```
viz.Parse(pe)['negative_feedback']
```

[18]:

	RSS	kADeg	kBDeg	kBProd	vAProd
0	8.851340e-13	0.2	0.4	0.3	0.1
1	8.851340e-13	0.2	0.4	0.3	0.1
2	8.851340e-13	0.2	0.4	0.3	0.1
3	8.851340e-13	0.2	0.4	0.3	0.1
4	8.851340e-13	0.2	0.4	0.3	0.1
5	8.851340e-13	0.2	0.4	0.3	0.1
6	8.851340e-13	0.2	0.4	0.3	0.1
7	8.851340e-13	0.2	0.4	0.3	0.1

The `parameters` keyword provides an easy interface for parameter selection. Here are the available options: - `g` specifies that all global variables are to be estimated - `l` specifies that all local parameters are to be estimated - `m` specifies that all metabolites are to be estimated - `c` specifies that all compartment volumes are to be estimated - `a` specifies that all of the above will be estimated

These options can also be combined. For example, `parameters='cgm'` means that compartment volumes, global quantities and metabolite concentrations (or particle numbers) will be estimated.

[ ]:
```

```

4.3 Examples

4.3.1 Working with an existing copasi model

You can create a new model using the antimony language. Sometimes however, you have already built a model using the CoapsiUI and want to use pycotools3 with this model. This example shows you how to use the *pycotools3.model.Model* class directly.

Create a PyCoTools model from an existing model

```
from pycotools3 import model

# remember to input the string to your own model here
copasi_file = <'string/to/model.cps'>

mod = model.Model(copasi_file)

assert isinstance(mod, model.Model)
```

Extracting the antimony string associated with a copasi model

It is often useful to have an antimony string generated directly from a copasi model.

```
ant_string = mod.to_antimony()
print(ant_string)
```

4.3.2 Running Time Series

First generate a model

```
import os
from pycotools3 import model, tasks


model_string = """
model model1()

    R1:   => A ; k1*S;
    R2: A =>   ; k2*A;
    R3:   => B ; k3*A;
    R4: B =>   ; k4*B*C; //feedback term
    R5:   => C ; k5*B;
    R6: C =>   ; k6*C;

    S = 1;
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
    k5 = 0.1;
    k6 = 0.1;

    A = 0;
    B = 0;
    C = 0;
```

(continues on next page)

(continued from previous page)

```
    APlusB := A + B;
end
"""

# when running from a script you can use the special __file__ variable
if os.path.isfile(__file__):
    copasi_file = os.path.join(os.path.dirname(__file__), 'copasi_file.cps')
else:
    copasi_file = os.path.join(os.path.abspath(''), 'copasi_file.cps')

# create a copasi model
mod = model.loada(model_string, copasi_file)
assert isinstance(mod, model.Model)
```

To run a time course simulation use:

```
>>> # from 0 to 100 by step size of 0.1
>>> mod.simulate(0, 100, 0.1, variables='m')
          A         B         C
Time
0      0.000000  0.000000  0.000000
1      0.095163  0.004837  0.000159
2      0.181269  0.018730  0.001208
3      0.259182  0.040810  0.003882
4      0.329680  0.070277  0.008766
5      0.393469  0.106377  0.016316
6      0.451188  0.148384  0.026874
7      0.503415  0.195577  0.040682
8      0.550671  0.247230  0.057889
9      0.593430  0.302596  0.078559
10     0.632121  0.360899  0.102679
```

The output is a *:py:class:'pandas.DataFrame* <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>'_.

Note: The *variables* argument is *'m'* by default.

This gives you the time series of all the metabolites. If you also want global quantities (or even local parameters - though they are constants anyway) you can use *'g'* or *'l'*.

For instance:

Return time series for global variables

```
>>> mod.simulate(0, 100, 0.1, variables='g')
        APlusB   S    k1    k2    k3    k4    k5    k6
```

(continues on next page)

(continued from previous page)

```
Time
0      0.000000  1  0.1  0.1  0.1  0.1  0.1  0.1
1      0.100000  1  0.1  0.1  0.1  0.1  0.1  0.1
2      0.199999  1  0.1  0.1  0.1  0.1  0.1  0.1
3      0.299992  1  0.1  0.1  0.1  0.1  0.1  0.1
4      0.399957  1  0.1  0.1  0.1  0.1  0.1  0.1
5      0.499847  1  0.1  0.1  0.1  0.1  0.1  0.1
6      0.599572  1  0.1  0.1  0.1  0.1  0.1  0.1
7      0.698992  1  0.1  0.1  0.1  0.1  0.1  0.1
8      0.797901  1  0.1  0.1  0.1  0.1  0.1  0.1
9      0.896026  1  0.1  0.1  0.1  0.1  0.1  0.1
10     0.993020  1  0.1  0.1  0.1  0.1  0.1  0.1
```

Return time series for metabolites and global variables

```
>>> mod.simulate(0, 100, 0.1, variables='mg')
```

A APlusB B C S ... k2 k3 k4 k5 k6

Time ... 0 0.000000 0.000000 0.000000 0.000000 1 ... 0.1 0.1 0.1 0.1 0.1 1 0.095163 0.100000 0.004837 0.000159 1 ... 0.1 0.1 0.1 0.1 0.1 2 0.181269 0.199999 0.018730 0.001208 1 ... 0.1 0.1 0.1 0.1 0.1 3 0.259182 0.299992 0.040810 0.003882 1 ... 0.1 0.1 0.1 0.1 0.1 4 0.329680 0.399957 0.070277 0.008766 1 ... 0.1 0.1 0.1 0.1 0.1 5 0.393469 0.499847 0.106377 0.016316 1 ... 0.1 0.1 0.1 0.1 0.1 6 0.451188 0.599572 0.148384 0.026874 1 ... 0.1 0.1 0.1 0.1 0.1 7 0.503415 0.698992 0.195577 0.040682 1 ... 0.1 0.1 0.1 0.1 0.1 8 0.550671 0.797901 0.247230 0.057889 1 ... 0.1 0.1 0.1 0.1 0.1 9 0.593430 0.896026 0.302596 0.078559 1 ... 0.1 0.1 0.1 0.1 0.1 10 0.632121 0.993020 0.360899 0.102679 1 ... 0.1 0.1 0.1 0.1 0.1

Return time series for metabolites, global variables and local parameters (though remember there are none in this current topology)

```
>>> mod.simulate(0, 100, 0.1, variables='mgl')
          A         B        C
Time
0      1.000000  1.000000  1.00000
1      0.082146  0.072788  2.81673
2      0.069317  0.062367  2.83276
3      0.068980  0.062080  2.82647
4      0.068817  0.061934  2.81989
5      0.068657  0.061789  2.81332
6      0.068497  0.061645  2.80677
7      0.068337  0.061502  2.80023
8      0.068178  0.061358  2.79371
9      0.068019  0.061215  2.78720
10     0.067861  0.061073  2.78071
```

Alternative simulation methods

Copasi supports several model simulation algorithms. PyCoTools supports most of these, including:

- deterministic (the default)
- direct
- gibson_bruck
- tau_leap
- adaptive_tau_leap
- hybrid_runge_kutta
- hybrid_lsoda
- hybrid_rk45

To use one of these alternative methods, ensure your model is adequate for the simulation you are performing (i.e. no reversible reactions and low enough copy numbers for stochastic simulation) and use the *method* argument to `pycotools3.model.Model.simulate()`

Note: The example model above is not suitable to stochastic simulation

---

4.3.3 Plotting

If you have used `pycotools3.model.Model.simulate()` then you will have a `pandas.DataFrame`. In this case, you might as well use either *pandas* plotting facilities or *matplotlib* with *seaborn*. You could also use *plotly*, *plotly* with *dash* or *bokeh*.

There are also some inherent plotting facilities in pycotools.

With PyCoTools

Visualisation in pycotools works by passing a plotter class an instance of a task. In this case we need a handle to the TimeCourse task.

```
>>> from pycotools3 import tasks, viz
>>> tc = tasks.TimeCourse(model=mod, start=0, end=10, step_size=1)
>>> viz.PlotTimeCourse(tc, savefig=True, show=True)
```

Since the inherent plotting module is basically just a wrapper around matplotlib and seaborn, it might be a good idea to use these tools instead.

Here's an example.

With matplotlib

Here's a very simple example using matplotlib.

4.3.4 Integration with Tellurium

Tellurium is a Python package built on top of libRoadRunner, a C++ package for simulation and analysis of models in systems biology. Antimony is a model definition language for SBML models that was written by the authors of *Tellurium* and *libRoadRunner*.

Since PyCoTools has adopted *Antimony*, the same model can be simulated and analysed with both COPASI and tellurium.

Here's a short example of how using the tellurium back end.

```
import tellurium as te

antimony_string = """
model simple_parameter_estimation()
    compartment Cell = 1;

    A in Cell;
    B in Cell;
    C in Cell;

    // reactions
    R1: A => B ; Cell * k1 * A;
    R2: B => A ; Cell * k2 * B * C;
    R3: B => C ; Cell * k3 * B;
    R4: C => B ; Cell * k4 * C;

    // initial concentrations
    A = 100;
    B = 1;
    C = 1;

    // reaction parameters
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
end
"""

mod = te.loada(antimony_string)

# simulate a time series with 11 time points between 0 and 10
timeseries_data = mod.simulate(0, 10, 11)

# compute the steady state
```

(continues on next page)

(continued from previous page)

```
mod.conservedMoietyAnalysis = True
mod.setSteadyStateSolver('nleq')
mod.steadyState()
mod.steadyStateSelections = ['A', 'B', 'C']
print(dict(zip(*mod.steadyStateSelections,
                mod.getSteadyStateValues())))
```

4.3.5 Simple Parameter Estimation

This is an example of how to configure a simple parameter estimation using pycotools. We first create a toy model for demonstration, then simulate some experimental data from it and fit it back to the model, using pycotools for configuration.

```
import os, glob
import pandas, numpy
import matplotlib.pyplot as plt
import seaborn
from pycotools3 import model, tasks, viz
seaborn.set_context(context='talk')         # set seaborn context␣
↪for formatting output of plots

## Choose a directory for our model and analysis. Note this can be␣
↪anywhere.
working_directory = os.path.abspath('')

## In this model, A gets reversibly converted to B but the␣
↪backwards reaction is additionally regulated by C.
## B is reversibly converted into C.
antimony_string = """
model simple_parameter_estimation()
    compartment Cell = 1;

    A in Cell;
    B in Cell;
    C in Cell;

    // reactions
    R1: A => B ; Cell * k1 * A;
    R2: B => A ; Cell * k2 * B * C;
    R3: B => C ; Cell * k3 * B;
    R4: C => B ; Cell * k4 * C;

    // initial concentrations
    A = 100;
    B = 1;
    C = 1;
```

(continues on next page)

(continued from previous page)

```
    // reaction parameters
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
end
"""

# Create a path to a copasi file
copasi_file = os.path.join(working_directory, 'example_model.cps')

## build model
mod = model.loada(antimony_string, copasi_file)
assert isinstance(mod, model.Model)

## simulate some data, returns a pandas.DataFrame
data = mod.simulate(0, 20, 1)

## write data to file
experiment_filename = os.path.join(working_directory, 'experiment_
↪data.txt')
data.to_csv(experiment_filename)

## We now have a model and some experimental data and can
## configure a parameter estimation
```

Parameter estimation configuration in pycotools3 revolves around the `tasks.ParameterEstimation.Config` object which is the input to the parameter estimation task. The object necessarily takes a lot of manual configuration to ensure it is flexible enough for any parameter estimation configuration. However, the `ParameterEstimation.Context` class is a tool for simplifying the construction of a Config object.

```
with tasks.ParameterEstimation.Context(mod, experiment_filename,␣
↪context='s', parameters='g') as context:
    context.set('separator', ',')
    context.set('run_mode', True)
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 100)
    context.set('lower_bound', 1e-1)
    context.set('upper_bound', 1e1)

    config = context.get_config()

pe = tasks.ParameterEstimation(config)

data = viz.Parse(pe).data
print(data)
```

4.3.6 Simple Parameter with Steady State Data

The short story here is that PyCoTools distinguishes time series and steady state data automatically, using the presence or absence of the *time* column.

Here's an example.

```
import os, glob
import pandas, numpy
import matplotlib.pyplot as plt
import seaborn
from pycotools3 import model, tasks, viz
seaborn.set_context(context='talk')          # set seaborn context␣
↪for formatting output of plots

## Choose a directory for our model and analysis. Note this can be␣
↪anywhere.
working_directory = os.path.abspath('')

## In this model, A gets reversibly converted to B but the␣
↪backwards reaction is additionally regulated by C.
## B is reversibly converted into C.
antimony_string = """
model simple_parameter_estimation()
    compartment Cell = 1;

    A in Cell;
    B in Cell;
    C in Cell;

    // reactions
    R1: A => B ; Cell * k1 * A;
    R2: B => A ; Cell * k2 * B * C;
    R3: B => C ; Cell * k3 * B;
    R4: C => B ; Cell * k4 * C;

    // initial concentrations
    A = 100;
    B = 1;
    C = 1;

    // reaction parameters
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
end
"""

# Create a path to a copasi file
copasi_file = os.path.join(working_directory, 'example_model.cps')
```

(continues on next page)

(continued from previous page)

```
## build model
mod = model.loada(antimony_string, copasi_file)
assert isinstance(mod, model.Model)

## create some made up data
data = pandas.DataFrame({'A': 30, 'B': 10, 'C': 10}, index=[0])

## write data to file
experiment_filename = os.path.join(working_directory, 'experiment_
↪data.txt')
data.to_csv(experiment_filename, index=False)
```

We now have a model and some experimental data and can configure a parameter estimation. Configuring steady state data is semantically identical to configuring time series data. The difference is that our *data* no longer has a *time* column and so PyCoTools assumes that it is steady state data.

Now, as usual, we configure the parameter estimation with the *Context* manager.

```
with tasks.ParameterEstimation.Context(mod, experiment_filename,␣
↪context='s', parameters='g') as context:
    context.set('separator', ',')
    context.set('run_mode', True)
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 100)
    context.set('lower_bound', 1e-1)
    context.set('upper_bound', 1e1)

    config = context.get_config()

pe = tasks.ParameterEstimation(config)

data = viz.Parse(pe).data
print(data)
```

4.3.7 Parameter Estimation using Prefix Argument

Sometimes we would like to select a set of parameters to estimate and leave the rest fixed. One way to do this is to compile a list of parameters you would like to estimate and enter them into the `ParameterEstimation.Config` class. A quicker alternative is to use the *prefix* setting.

```
import os, glob
import pandas, numpy
import matplotlib.pyplot as plt
import seaborn
```

(continues on next page)

(continued from previous page)

```
from pycotools3 import model, tasks, viz
seaborn.set_context(context='talk')

## Choose a directory for our model and analysis
working_directory = os.path.abspath('')
```

The *prefix* argument will setup the configuration of a parameter estimation containing only parameters that start with *prefix*. While this can be anything, its quite useful to use the _ character and then add an _ to any parameters that you want estimated. In this way you can keep your estimated parameters marked.

```
antimony_string = """
model simple_parameter_estimation()
    compartment Cell = 1;

    A in Cell;
    B in Cell;
    _C in Cell;

    // reactions
    R1: A => B  ; Cell * _k1 * A;
    R2: B => A  ; Cell * k2 * B * _C;
    R3: B => _C ; Cell * _k3 * B;
    R4: _C => B ; Cell * k4 * _C;

    // initial concentrations
    A = 100;
    B = 1;
    _C = 1;

    // reaction parameters
    _k1 = 0.1;
    k2 = 0.1;
    _k3 = 0.1;
    k4 = 0.1;
end
"""

copasi_file = os.path.join(working_directory, 'example_model.cps')

## build model
mod = model.loada(antimony_string, copasi_file)

assert isinstance(mod, model.Model)

fname = os.path.join(working_directory, 'experiment_data.txt')
data = mod.simulate(0, 20, 1, report_name=fname)
## write data to file
data.to_csv(fname)
```

And now we configure a parameter estimation like normal but set *prefix* to '_'.

```
with tasks.ParameterEstimation.Context(mod, fname, context='s',␣
↪parameters='a') as context:
    context.set('separator', ',')
    context.set('run_mode', True)
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 100)
    context.set('lower_bound', 1e-1)
    context.set('upper_bound', 1e1)
    context.set('prefix', '_')
    config = context.get_config()

pe = tasks.ParameterEstimation(config)

data = viz.Parse(pe).data
print(data)
```

4.3.8 Parameter Estimation with Independent Variables

The concept of independent variables is important for parameter fitting because it enables us to simultaneously fit multiple datasets to a single model. The is achieved by iterating over all the datasets in your objective function and changing variables (such as initial concentration parameters) to what ever they should be for that dataset. These variables are independent variables and they basically define the initial conditions for fitting the dataset.

Independent variables are handled in PyCoTools by appending the string '*_indep*' after a variable in the data file itself. PyCoTools will then recognize the variable and set it as independent rather than dependent.

Here's an example:

```
import os, glob
import pandas, numpy, seaborn
from pycotools3 import model, tasks, viz
seaborn.set_context(context='talk')         # set seaborn context␣
↪for formatting output of plots

## Choose a directory for our model and analysis. Note this can be␣
↪anywhere.
working_directory = os.path.abspath('')

## In this model, A gets reversibly converted to B but the␣
↪backwards reaction is additionally regulated by C.
## B is reversibly converted into C.
antimony_string = """
model simple_parameter_estimation()
    compartment Cell = 1;
```

(continues on next page)

(continued from previous page)

```
    A in Cell;
    B in Cell;
    C in Cell;

    // reactions
    R1: A => B ; Cell * k1 * A;
    R2: B => A ; Cell * k2 * B * C;
    R3: B => C ; Cell * k3 * B;
    R4: C => B ; Cell * k4 * C;

    // initial concentrations
    A = 100;
    B = 1;
    C = 1;

    // reaction parameters
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;

end
"""

# Create a path to a copasi file
copasi_file = os.path.join(working_directory, 'example_model.cps')

# build model
mod = model.loada(antimony_string, copasi_file)
assert isinstance(mod, model.Model)

# simulate some data, returns a pandas.DataFrame
data = mod.simulate(0, 20, 1)

# creates a new column in the dataset called A_indep
data['A_indep'] = 50
# the initial abundance of A will now be set to 50 prior to␣
↪estimation

# write data to file
experiment_filename = os.path.join(working_directory, 'experiment_
↪data.txt')
data.to_csv(experiment_filename)
```

We now configure a parameter estimation like normal.

```
with tasks.ParameterEstimation.Context(mod, experiment_filename,␣
↪context='s', parameters='g') as context:
```

(continues on next page)

(continued from previous page)

```
    context.set('separator', ',')
    context.set('run_mode', True)
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 100)
    context.set('lower_bound', 1e-1)
    context.set('upper_bound', 1e1)

    config = context.get_config()

pe = tasks.ParameterEstimation(config)

data = viz.Parse(pe).data
print(data)
```

4.3.9 Advanced Parameter Estimation

Parameter estimation configuration in PyCoTools revolves around the `ParameterEstimation.Config` object, which is merely a nested data storage class. When you use the `ParameterEstimation.Context` object, you are using a high level interface to create a `ParameterEstimation.Config` object. However, it is also possible to build it manually. It should be stressed however, that the Context method should be preferred when possible. The low level interface can be tedious but gives you more flexibility for more fine tuned parameter estimation configurations. While there is no getting around the fact that building the `ParameterEstimation.Config` object can take time, I have built in a 'default system' whereby if a (non-compulsory) section is left blank, then PyCoTools tries to use the default values.

The `ParameterEstimation.Config` Object is a bit like a Struct from other languages or a tree (see `bunch.Bunch`). You can access attributes using either dot notation or dictionary like access.

There are four main sections to the Config object:

- *Models*
- *Datasets*
- *Items*
- *Settings*

Each of these are described in detail below. Note that all branches in the config object described here are themselves nodes in the tree with children unless they are labelled with *leaf*. These nodes expect the usual key: value pairs.

Models

This is a nested structure containing an arbitrary number of models for simultaneous configuration. It is the users responsibility that it makes sense to configure multiple models for parameter estimation in the same way.

- models
  - model1_name (leaf): full string to model 1 copasi file
  - model2_name (leaf): full string to model 2 copasi file
  - ...

```
models = dict(
    model1=dict(copasi_file='</full/path/to/copasi_file1.cps'),
    model2=dict(copasi_file='</full/path/to/copasi_file2.cps')
)
```

Note: The nested structure of the models section remains a deprecated version of PyCoTools. Since it does not need to be nested, this will be changing to non-nested in the near future.

Datasets

This is where we tell PyCoTools about the data were using for parameter estimation. The datasets attribute is itself nested with the following structure:

- datasets
  - experiments
  - validations

Both *experiments* and *validations* are nested structures and they are pretty much identical. The difference is that those configured via the *experiments* section are used for calibrating/training the model while those in the validation section are only used for validation (or testing). For simplicity, the structure of both is described in one section.

- experiments (or validation)
  - experiment_name. An arbitrary string representing the name of the experiment.
    - filename (leaf). The full path to the dataset
    - affected_models. Analogous to affected_experiments or affected_validation_experiments, you can have an experiment target only one (or more) model. This feature is a superser of COPASI. Defaults to the string 'all' which is translated to all models.

* mappings. Another nested structure for mapping arguments. If left blank, PyCoTools will assume 1:1 mappings between experimental data file headers and model variables. Independent variables are assumed to contain a trailing *_indep*, i.e. *PI3K_indep*. This should have as many elements as there are columns in the data file.
  - Experimental variable name (or time). These should be the same as used for data column headers.
  - model_object (leaf node). The object that corresponds to the experimental variable name.
  - role (leaf node). Either *time*, *ignored* (default), *dependent* or *independent*
* separator (leaf). Overrides the separator in the settings section, for when they are different. However, good practice is to always use the same separator and set the separator in the settings section.
* normalize_weights_per_experiment (leaf): boolean, default=True.

Here's an example of the datasets section.

```
datasets=dict(
    experiments= dict(
        report1 = dict(
            filename='full/path/to_datafile1.csv',
            affected_models='all',
            mappings=dict(
                Time=dict(
                    model_object='Time',
                    role='time'
                ),
                A=dict(
                    model_object='A'),
                    role='dependent'),
                ),
            )
        ),
        # note the absence of the mappings field. This tells
        # PyCoTools that you have used the suggested convention
        # of matching model variables with data file headers and␣
↪using
        # '_indep' suffix for independent variables.
        report2=dict(
            filename='full/path/to_datafile2.csv',
            separator='\t'  #overrides separator from main settings␣
↪menu
        )
    ),
    # This data will not be used for parameter estimation. Only␣
↪validation.
```

(continues on next page)

(continued from previous page)

```
    validations=dict(
        report3=dict(
            filename='full/path/to_datafile3.csv',
            affected_models='model1',  # this validation experiment␣
↪only affects model1
            # were excepting default mapping convention
        ),
    )
)
```

Items

This is where we configure the parameters to be estimated, their boundaries, start values and affected experiments. The *items* structure is composed of *fit_items* and *constraint_items*.

- items
  - fit_items
  - constraint_items

Similarly to the experiment section, fit_items and constraint_items are nearly identical. The difference is that whilst fit items are used to define the parameter space constraints are used to restrict the parameter space to a subset of the full parameter space. An estimation with constraints can explore beyond the restrictions imposed by the constraints but solutions that violate the constraints will not be excepts. In contrast, the solution cannot go beyond the boundaries of the boundaries set by the fit_items.

```
items = dict(
    fit_items=dict(
        A=dict(
            affected_experiments=['report1'],
            affected_models=['model1'],
            affected_validation_experiments=['report3'],
            lower_bound=15,
            start_value=0.1,
            upper_bound=35
        ),
        B=dict(
            affected_experiments=['report1', 'report2'],
            affected_models=['model1'],
            affected_validation_experiments=['report3'],
            lower_bound=0.05,
            start_value=1.05,
            upper_bound=36
        ),
        C=dict(
            affected_experiments=['report1', 'report2'],
            affected_models=['model1'],
```

(continues on next page)

(continued from previous page)

```
            affected_validation_experiments=['report3'],
            lower_bound=0.05,
            start_value=1.0,
            upper_bound=36
        )
    ),
    constraint_items=dict(
        C=dict(
            affected_experiments=['report1', 'report2'],
            affected_models=['model1'],
            affected_validation_experiments=['report3'],
            lower_bound=16,
            start_value=1.05,
            upper_bound=26
            )
    )
)
```

Note: *affected_experiments*, *affected_models*, and *affected_validation_experiments* all accept the special string *all* which resolves to all of your data files. This is default behaviour for both *affected_experiments* and *affected_models* whereas the default behaviour for *affected_validation_experiments* is None.

Settings

These are global settings for the parameter estimation.

```
settings = dict(
    calculate_statistics=False,      # Corresponds to the `calculate_
↪statistics` flag in copasi
    config_filename=config.yml       # Filename for saving config to␣
↪file
    context=s,                       # Alters the behaviour of the␣
↪configuration. See :py:class:`ParameterEstimation.Context`.
    cooling_factor=0.85              # Parameter estimation␣
↪algorithm setting
    copy_number=1,                   # How many times to copy the␣
↪copasi file for simultaneous runs
    create_parameter_sets=False,     # Corresponds to the create_
↪parameter_sets flag in copasi
    cross_validation_depth=1,        # depth of cross validation.␣
↪Corresponds to COPASI, (though this feature was buggy during␣
↪development)
    fit=1,                           # This is an index of parameter␣
↪estimation. Increment by 1 to repeat a similar parameter␣
↪estimation to test alternative configurations
```

(continues on next page)

(continued from previous page)

```
    iteration_limit=50,                 # Parameter estimation␣
↪algorithm setting
    lower_bound=0.05                    # Default lower boundary for␣
↪all parameters in the estimation. Can be overwritten under the␣
↪fit_items section to have different boundaries for every fit item.
    max_active=3,                       # When running
    method=genetic_algorithm_sr,        # which algorithm to use
    number_of_generations=100,          # Parameter estimation␣
↪algorithm setting
    number_of_iterations=100000,        # Parameter estimation␣
↪algorithm setting
    overwrite_config_file=False,        # Set to True to explicitely␣
↪overwrite existing configuration file.
    pe_number=1,                        # How many parameter estimations
    pf=0.475                            # Parameter estimation␣
↪algorithm settings
    pl_lower_bound=1000,                # When context is set to 'pl'␣
↪for profile likelihood configurations, this defines the upper␣
↪boundary of the analysis. The upper boundary is the best␣
↪estimated parameter multiplied by this value.
    pl_upper_bound=1000,                # When context is set to 'pl'␣
↪for profile likelihood configurations, this defines the lower␣
↪boundary of the analysis. The lower boundary is the best␣
↪estimated parameter divided by this value.
    population_size=38,                 # Parameter estimation␣
↪algorithm setting
    prefix=None,                        # Prefix used to automatically␣
↪locate parameters to be estimated. For instance, you can 'tag'␣
↪each parameter you want to include in the estimation with an␣
↪underscore at the begining (i.e. _kAktPhosphorylation) to filter␣
↪the parameters for estimation.
    problem=Problem1,                   # This is the name of the␣
↪folder that will be created to contain the results.
    quantity_type=concentration,        # either 'concentration' or
↪'particle_numbers' to switch between the two.
    random_number_generator=1,          # Parameter estimation␣
↪algorithm setting.
    randomize_start_values=False,       # Corresponds to the 'randomize_
↪start_values' flag in copasi
    report_name=PEData.txt              # The base report name for the␣
↪parameter estimation output. This is automatically modified when␣
↪copy_number is > 1. The results have as many rows as `pe_number`.
    results_directory=ParameterEstimationData,  # This folder␣
↪stores the actual parameter estimation results, within the fit␣
↪directory (which is within the Problem directory)
    rho=0.2                             # Parameter estimation␣
↪algorithm setting
    run_mode=False,                     # Switch between False
    save=False,                         # Save the model to file after␣
↪configuration or not.
```

(continues on next page)

(continued from previous page)

```
    scale=10,                          # Parameter estimation␣
↪algorithm setting
    seed=0,                            # Parameter estimation␣
↪algorithm setting
    start_temperature=1,               # Parameter estimation␣
↪algorithm setting
    start_value=0.1                    # Parameter estimation␣
↪algorithm setting
    starting_parameter_sets=None,      # Experimental feature.
    std_deviation=1.0e-06              # Parameter estimation␣
↪algorithm setting
    swarm_size=50,                     # Parameter estimation␣
↪algorithm setting
    tolerance=1.0e-05                  # Parameter estimation␣
↪algorithm setting
    update_model=False,                # Corresponds to the update␣
↪model flag in copasi
    upper_bound=36,                    # Default upper boundary for␣
↪all parameters in the estimation. Can be overwritten under the␣
↪fit_items section to have different boundaries for every fit item.
    use_config_start_values=False,   # If True, parameter estimation␣
↪will start from the start values specified under the `fit_items`␣
↪section.
    validation_threshold=8.5           # Corresponds to the validation␣
↪threshold in COPASI. This is the default value that can be␣
↪overwritten by giving this argument to the validation dataset␣
↪section.
    validation_weight=4,               # Corresponds to the validation␣
↪weight in COPASI.  This is the default value that can be␣
↪overwritten by giving this argument to the validation dataset␣
↪section.
    weight_method=value_scaling,     # Which weight method to use.␣
↪Default='mean_squared'. Other options: mean, standard_deviation␣
↪or value_scaling
    working_directory=/home/ncw135/Documents/pycotools3/Tests    #␣
↪The overall directory for the whole analysis. Defaults to the␣
↪same directory containing the first copasi file found for␣
↪configuration.
)
```

Building a `ParameterEstimation.Config` object

When you have configured the relevant sections, you can simply call the `ParameterEstimation.Config` constructor to create your object.

Assuming you have nested dictionaries containing the apprioriate information detailed above:

```
config = tasks.ParameterEstimation.Config(
            models=models,
            datasets=datasets,
            items=items,
            settings=settings
        )
```

The config is formatted using yaml for ease of inspection.

Note: It is possible to load from yaml file on disk. Documentation to come.

Using a `ParameterEstimation.Context` as a template

The most effective way to use the low level interface is to let the `ParameterEstimation.Context` do most of the work and then retrieve the mostly configured config string and then make your desired ammendments.

Saving and loading configurations from and to yaml

To save an existing configuration to file to yaml, assuming *PE* is an instantiated `pycotools3.class.ParameterEstimation`

```
fname = os.path.join(os.path.dirname(__file__), 'config_file.yml')
PE.config.to_yaml(fname)
assert os.path.isfile(fname)
```

To load the configuration from the yaml again:

```
# assuming our yaml configuration lives here:
fname = os.path.join(os.path.dirname(__file__), 'config_file.yml')
config = ParameterEstimation.Config.from_yaml(fname)
```

The *config* is an instance of `ParameterEstimation.Config` which can be passed to the `ParameterEstimation`

```
pe = ParameterEstimation(config)
```

4.3.10 Simple Parameter Estimation

This is an example of how to configure a simple parameter estimation using pycotools. We first create a toy model for demonstration, then simulate some experimental data from it and fit it back to the model, using pycotools for configuration.

Configuring a model for parameter estimation

```
import os, glob
import pandas, numpy
import matplotlib.pyplot as plt
import seaborn
from pycotools3 import model, tasks, viz
seaborn.set_context(context='talk')         # set seaborn context␣
↪for formatting output of plots

## Choose a directory for our model and analysis. Note this can be␣
↪anywhere.
working_directory = os.path.abspath('')

## In this model, A gets reversibly converted to B but the␣
↪backwards reaction is additionally regulated by C.
## B is reversibly converted into C.
antimony_string = """
model simple_parameter_estimation()
    compartment Cell = 1;

    A in Cell;
    B in Cell;
    C in Cell;

    // reactions
    R1: A => B ; Cell * k1 * A;
    R2: B => A ; Cell * k2 * B * C;
    R3: B => C ; Cell * k3 * B;
    R4: C => B ; Cell * k4 * C;

    // initial concentrations
    A = 100;
    B = 1;
    C = 1;

    // reaction parameters
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
end
"""

# Create a path to a copasi file
copasi_file = os.path.join(working_directory, 'example_model.cps')

## build model
mod = model.loada(antimony_string, copasi_file)
assert isinstance(mod, model.Model)
```

(continues on next page)

(continued from previous page)

```
## simulate some data, returns a pandas.DataFrame
data = mod.simulate(0, 20, 1)

## write data to file
experiment_filename = os.path.join(working_directory, 'experiment_
↪data.txt')
data.to_csv(experiment_filename)

## We now have a model and some experimental data and can
## configure a parameter estimation
```

Serial Execution

As shown in other examples, you can run a single parameter estimation using PyCoTools as a controller by setting *run_mode=True*.

```
with tasks.ParameterEstimation.Context(mod, experiment_filename,␣
↪context='s', parameters='g') as context:
    context.set('separator', ',')
    context.set('run_mode', True)
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 100)
    context.set('lower_bound', 1e-1)
    context.set('upper_bound', 1e1)
    config = context.get_config()
pe = tasks.ParameterEstimation(config)
```

Parallel Execution

PyCoTools also enables you to increase parameter estimation throughput by automating the parallel execution of multiple model copies. To do this, we simply set *run_mode='parallel'*. With *'parallel'* mode, there are a few additional settings that you need to know about.

- *copy_number* is the number of model copies that you want to run
- *pe_number* is the number of parameter estimations each of *copy_number* models will run in serial
- *nproc* is the number of processes to use for getting through all *copy_number* models.

The configuration below will run 20 parameter estimations using a pool of 6 processes.

```
with tasks.ParameterEstimation.Context(mod, experiment_filename,␣
↪context='s', parameters='g') as context:
    context.set('separator', ',')
    context.set('run_mode', 'parallel')
```

(continues on next page)

(continued from previous page)

```
    context.set('copy_number', 20)
    context.set('pe_number', 1)
    context.set('max_active', 6)
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 100)
    context.set('lower_bound', 1e-1)
    context.set('upper_bound', 1e1)
    config = context.get_config()
pe = tasks.ParameterEstimation(config)
```

On a Computer Cluster

If you have access to a computer cluster, then PyCoTools already supports *Slurm* and *SunGridEngine* sheduling systems. If you are using *slurm*, set *run_mode='slurm'* and PyCoTools will submit *copy_number* jobs using *sbatch*. If you are using *SunGridEngine* then set *run_mode='sge'*.

```
with tasks.ParameterEstimation.Context(mod, experiment_filename,␣
↪context='s', parameters='g') as context:
    context.set('separator', ',')
    context.set('run_mode', 'slurm') # or sge
    context.set('copy_number', 300)
    context.set('pe_number', 1)
    context.set('max_active', 6)
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 100)
    context.set('lower_bound', 1e-1)
    context.set('upper_bound', 1e1)
    config = context.get_config()
pe = tasks.ParameterEstimation(config)
```

Note: PyCoTools will expect Copasi to already be available in the environment so that the command *CopasiSE* will produce Copasi's help message and not an error.

If you are using a scheduling system different to *SGE* or *Slurm* then you'll have to write your own wrapper, analogous to `pycotools3.tasks.Run.submit_copasi_job_SGE()` and `pycotools3.tasks.Run.submit_copasi_job_slurm()`. This shouldn't be too difficult. If you run into trouble please submit an issue on GitHub.

4.3.11 Multiple Parameter Estimations

Configuring multiple parameter estimations is easy with COPASI because you can configure a parameter estimation task, configure a scan repeat item with the *subtask* set to *parameter*

estimation and hit run. This is what pycotools does under the hood to configure a parameter estimation, even if the desired number of parameter estimations is 1.

Moreover, pycotools additionally supports the running of multiple copies of your copasi file in separate processes, or on a cluster.

```
from pycotools3 import model, tasks

working_directory = os.path.abspath('')

antimony_string =  '''
    model negative_feedback()
        // define compartments
        compartment cell = 1.0
        //define species
        var A in cell
        var B in cell
        //define some global parameter for use in reactions
        vAProd = 0.1
        kADeg  = 0.2
        kBProd = 0.3
        kBDeg  = 0.4
        //define initial conditions
        A      = 0
        B      = 0
        //define reactions
        AProd:  => A; cell*vAProd
        ADeg: A =>  ; cell*kADeg*A*B
        BProd:  => B; cell*kBProd*A
        BDeg: B =>  ; cell*kBDeg*B
    end
    '''

 copasi_file = os.path.join(working_directory, 'negative_fb.cps')
 mod = model.loada(antimony_string, copasi_file )

 data_fname = os.path.join(working_directory, 'timecourse.txt')
 mod.simulate(0, 10, 1, report_name=data_fname)

 assert os.path.isfile(data_fname)
```

Increasing Parameter Estimation Throughput

The *pe_number* argument specifies the number that gets entered into the copasi scan repeat item while the *copy_number* argument specifies the number of identical model copies to make.

```
with tasks.ParameterEstimation.Context(mod, data_fname, context='s',
↪ parameters='g') as context:
    context.set('copy_number', 2)
```

(continues on next page)

(continued from previous page)

```
    context.set('pe_number', 2)
    context.set('randomize_start_values', True)
    context.set('run_mode', True)
    config = context.get_config()

pe = tasks.ParameterEstimation(config)
data = viz.Parse(pe)
```

Note: The *copy_number* argument here doesn't really do anything useful because *run_mode=True*. This tells pycotools to run the parameter estimations in series (i.e. back to back) and therefore the *copy_number* argument here does nothing.

However, it is also possible to give *run_mode='parallel'* and in this case, each of the model copies will be run simultaneously.

```
with tasks.ParameterEstimation.Context(mod, data_fname, context='s',
↪ parameters='g') as context:
    context.set('copy_number', 2)
    context.set('pe_number', 2)
    context.set('randomize_start_values', True)
    context.set('run_mode', 'parallel)
    config = context.get_config()

pe = tasks.ParameterEstimation(config)
data = viz.Parse(pe)
```

Warning: Users should not use *run_mode='parallel'* in combination with a high *copy_number* as it will slow your system.

Your system has a limited amount of resources and can only handle a number of parameter estimations being run at once. For this reason, be careful when choosing the *copy_number*. For reference, my computer can run approximately 8 parameter estimations in different processes before slowing.

If you have access to a cluster running either SunGrid Engine or Slurm then each of the *copy_number* models will be submitted as separate jobs. To do this set *run_mode='slurm* or *run_mode='sge'* (see `tasks.Run`).

Warning: The cluster functions are fully operational on the Newcastle University clusters but untested on other clusters. If you run into trouble, contact me for help.

It is easy to support other cluster systems by adding a method to `tasks.Run` using `tasks.Run.run_sge()` and `tasks.Run.run_slurm()` as examples.

4.3.12 Parameter estimation with multiple models

This is an example of how to configure a parameter estimation for multiple COPASI models using pycotools. We first create two similar but different toy models for demonstration, then simulate some experimental data from one of them and and fit it back to both models.

```
import os, glob
import pandas, numpy
import matplotlib.pyplot as plt
import seaborn
from pycotools3 import model, tasks, viz

# set seaborn context
seaborn.set_context(context='talk')

## Choose a directory for our model and analysis
working_directory = os.path.abspath('')

model1_string = """
model model1()

    R1:    => A ; k1*S;
    R2: A =>   ; k2*A;
    R3:    => B ; k3*A;
    R4: B =>   ; k4*B*C; //feedback term
    R5:    => C ; k5*B;
    R6: C =>   ; k6*C;

    S = 1;
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
    k5 = 0.1;
    k6 = 0.1;
end
"""

model2_string = """
model model2()
    R1:    => A ; k1*S;
    R2: A =>   ; k2*A*C; //feedback term
    R3:    => B ; k3*A;
    R4: B =>   ; k4*B;
    R5:    => C ; k5*B;
    R6: C =>   ; k6*C;

    S = 1;
    k1 = 0.1;
    k2 = 0.1;
```

(continues on next page)

(continued from previous page)

```
    k3 = 0.1;
    k4 = 0.1;
    k5 = 0.1;
    k6 = 0.1;
end
"""
# create paths to where we want the two models
copasi_file1 = os.path.join(working_directory, 'model1.cps')
copasi_file2 = os.path.join(working_directory, 'model2.cps')

# Assemble into lists
antimony_strings = [model1_string, model2_string]
copasi_files = [copasi_file1, copasi_file2]

# create models
model_list = []
for i in range(len(copasi_files)):
    model_list.append(model.loada(antimony_strings[i], copasi_
↪files[i]))

## simulate some data, returns a pandas.DataFrame
data = model_list[0].simulate(0, 20, 1)

## write data to file
experiment_filename = os.path.join(working_directory, 'data_from_
↪model1.txt')
data.to_csv(experiment_filename)

# Create the context, passing the model list rather than the Model␣
↪object
with tasks.ParameterEstimation.Context(model_list, experiment_
↪filename, context='s', parameters='g') as context:
    context.set('separator', ',')
    context.set('run_mode', True)
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 25)
    context.set('lower_bound', 1e-1)
    context.set('upper_bound', 1e1)

    config = context.get_config()

# Do the parameter estimation
pe = tasks.ParameterEstimation(config)

# Parse the resulting data
data = viz.Parse(pe).data
print(data)
```

4.3.13 The `pycotools.model.Model` class

The pycotools3 tasks module contains classes for a bunch of copasi tasks that can be configured from python using pycotools. To simplify some of these tasks, wrappers have been build around these task classes in the `model.Model` class so that they can be used like a regular method. Here I demonstrate some of these.

We first configure a model for the demonstration

```
import os, glob
import pandas, numpy
import matplotlib.pyplot as plt
import seaborn
from pycotools3 import model, tasks, viz
seaborn.set_context(context='talk')

## Choose a directory for our model and analysis
working_directory = os.path.abspath('')

## In this model, A gets reversibly converted to B but the␣
↪backwards reaction is additionally regulated by C.
## B is reversibly converted into C.
antimony_string = """
model simple_parameter_estimation()
    compartment Cell = 1;

    A in Cell;
    B in Cell;
    C in Cell;

    // reactions
    R1: A => B ; Cell * k1 * A;
    R2: B => A ; Cell * k2 * B * C;
    R3: B => C ; Cell * k3 * B;
    R4: C => B ; Cell * k4 * C;

    // initial concentrations
    A = 100;
    B = 1;
    C = 1;

    // reaction parameters
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
end
"""

copasi_file = os.path.join(working_directory, 'example_model.cps')
```

(continues on next page)

(continued from previous page)

```
## build model
mod = model.loada(antimony_string, copasi_file)

assert isinstance(mod, model.Model)
```

Inserting parameters

```
dct = {
  'k1': 55,
  'k2': 36
}
mod.insert_parameters(parameter_dict=dct, inplace=True)
```

or

```
mod = mod.insert_parameters(parameter_dict=dct)
```

or

```
import pandas
df = pandas.DataFrame(dct, index=[0])
mod.insert_parameters(df=df, inplace=True)
```

or if the dataframe *df* has more than one parameter set we can specify the rank using the *index* argument.

```
import pandas
##insert second best parameter set
mod.insert_parameters(df=df, inplace=True, index=1)
```

Note: This is most useful when using `viz.Parse` output dataframes, which are `pandas.DataFrame` objects containing parameters in the columns and parameter sets in the rows, sorted by best RSS

---

or, assuming the variable *results_directory* is a directory to a folder containing parameter estimation results.

```
mod.insert_parameters(parameter_path=results_directory,␣
↪inplace=True)
```

Simulating a time course

```
data = mod.simulate(0, 10, 11)
```

Simulates a deterministic time course, 11 time points between 0 and 10. *data* contains a `pandas.DataFrame` object with variables along the columns and time points down the rows.

```
fname = os.path.join(os.path.dirname(__file__), 'simulation_data.csv
↪')
## write data to file named fname
data = mod.simulate(0, 10, 11, report_name=fname)
```

Like with the other shortcuts, arguments for the `tasks.TimeCourse` class are pass on.

```
data = mod.simulate(0, 10, 11, method='direct')
```

```
fname = ps.path.join(os.path.dirname(__file__), 'scan_results.csv')
mod.scan(variable='A', minimum=5, maximum=10, report_name=fname)
```

By default the scan type is set to 'scan'. We can change this

```
fname = ps.path.join(os.path.dirname(__file__), 'scan_results.csv')
mod.simulate(0, 10, 11, method='direct', run_mode=False)
mod.scan(variable='A', scan_type='repeat',
         number_of_steps=10, report_name=fname,
         subtask='timecourse')
```

Note: In the *mod.simulate* we configure copasi to run a stochastic time course but do not execute. We then configure the repeat scan task to run the stochastic time course 10 times.

Sensitivities

```
sens = mod.sensitivities(
            subtask='steady_state', cause='all_parameters',
            effect='all_variables'
       )
```

4.3.14 Profile Likelihoods

Since a profile likelihood is just a parameter scan of parameter estimations, all we need to do to configure a profile likelihood analysis is to setup an appropriate `ParameterEstimation.Config` object and feed it into the `ParameterEstimation` class. This would be tedious to do manually but is easy with `ParameterEstimation.Context`

```
import os, glob
import pandas, numpy
import matplotlib.pyplot as plt
import seaborn
from pycotools3 import model, tasks, viz

working_directory = os.path.abspath('')

antimony_string = """
model simple_parameter_estimation()
    compartment Cell = 1;

    A in Cell;
    B in Cell;
    C in Cell;

    // reactions
    R1: A => B ; Cell * k1 * A;
    R2: B => A ; Cell * k2 * B * C;
    R3: B => C ; Cell * k3 * B;
    R4: C => B ; Cell * k4 * C;

    // initial concentrations
    A = 100;
    B = 1;
    C = 1;

    // reaction parameters
    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
end
"""

copasi_file = os.path.join(working_directory, 'example_model.cps')

## build model
mod = model.loada(antimony_string, copasi_file)

assert isinstance(mod, model.Model)

## simulate some data, returns a pandas.DataFrame
data = mod.simulate(0, 20, 1)

## write data to file
experiment_filename = os.path.join(working_directory, 'experiment_
↪data.txt')
data.to_csv(experiment_filename)
```

The profile likelihood is calculated around the current parameter set in the model. If you want to change the current parameter set, maybe to the best fitting parameter set from a parameter estimation you can use the `InsertParameters` class. For now, we'll assume the best parameter set is already in the model.

```
with tasks.ParameterEstimation.Context(
    mod, experiment_filename,
    context='pl', parameters='gm'
) as context:
    context.set('method', 'hooke_jeeves')
    context.set('pl_lower_bound', 1000)
    context.set('pl_upper_bound', 1000)
    context.set('pe_number', 25) # number of steps in each profile␣
↪likelihood
    context.set('run_mode', True)
    config = context.get_config()
```

We set the method to hooke and jeeves, a local optimiser which does well with profile likelihoods. We also set the *pl_lower_bound* and *pl_upper_bound* arguments to 1000 (which are defaults anyway). These are multipliers, not boundaries, of the profile likelihood. For instance, if the best estimated parameter for *A* was 1, then the profile likelihood would stretch from 1-e3 to 1e3.

Now, like with other parameter estimations we can simply do

```
tasks.ParameterEstimation(config)
```

Because the *context=pl* was used, pycotools knows to copy the model for each parameter, remove the parameter of interest from the parameter estimation task and create a scan of the parameter of interest.

Note: You will see a bunch of warnings about "model "X" is not in the affected model list". You will get one of these for each model. These warnings can be ignored.

4.3.15 Cross validation

Validation experiments are not used in model calibration. Instead the objective function is evaluated on validation data to see if the model can predict data it has not already seen. This can then be used as stopping criteria for the algorithm as we give a threshold for the closeness of the validation fits to simulations. This idea is common practice in machine learning and is used to prevent overfitting.

Cross validation is a new feature of pycotools3 but has been supported by COPASI for some years. The idea is to rotate calibration and validation datasets until you have tried all the combinations.

Cross validation can help identify datasets which do and don't fit well together. Here we create a model, simulate 3 datasets, make a data set up and use cross validation to infer the dataset

that is made up.

```
# imports and create our antimony model string
from pycotools3 import model, tasks

working_directory = os.path.abspath('')
antimony_string =  '''
    model negative_feedback()
        // define compartments
        compartment cell = 1.0
        //define species
        var A in cell
        var B in cell
        //define some global parameter for use in reactions
        vAProd = 0.1
        kADeg  = 0.2
        kBProd = 0.3
        kBDeg  = 0.4
        //define initial conditions
        A      = 0
        B      = 0
        //define reactions
        AProd:  => A; cell*vAProd
        ADeg: A =>  ; cell*kADeg*A*B
        BProd:  => B; cell*kBProd*A
        BDeg: B =>  ; cell*kBDeg*B
    end
    '''

# Create string to where we want to copasi model to go
copasi_file = os.path.join(os.path.dirname(__file__), 'negative_fb.
↪cps')
mod = model.loada(antimony_string, copasi_file )  # create the␣
↪pycotools model

# create some filenames for experimental data
tc_fname1 = os.path.join(working_directory, 'timecourse1.txt')
tc_fname2 = os.path.join(working_directory, 'timecourse2.txt')
ss_fname1 = os.path.join(working_directory, 'steady_state1.txt')
ss_fname2 = os.path.join(working_directory, 'steady_state2.txt')

# simulate/create some experimental data
model.simulate(0, 5, 0.1, report_name=self.tc_fname1)
model.simulate(0, 10, 0.5, report_name=self.tc_fname2)
dct1 = {
    'A': 0.07,
    'B': 0.06,
    'C': 2.8
}
dct2 = {
```

(continues on next page)

(continued from previous page)

```
    'A': 846,
    'B': 697,
    'C': 739
}
ss1 = pandas.DataFrame(dct1, index=[0])
ss1.to_csv(self.ss_fname1, sep='\t', index=False)
ss2 = pandas.DataFrame(dct2, index=[0])
ss2.to_csv(self.ss_fname2, sep='\t', index=False)

experiments = [ss1, ss2]
```

Configuring a cross validation experiment is similar to running parameter estimation or profile likelihoods: the difference is that you use *context='cv'* as argument to `ParameterEstimation.Context`.

```
with tasks.ParameterEstimation.Context(
    model, experiments, context='cv', parameters='gm'
) as context:
    context.set('randomize_start_values', True)
    context.set('method', 'genetic_algorithm')
    context.set('population_size', 20)
    context.set('number_of_generations', 50)
    context.set('validation_threshold', 500)
    context.set('cross_validation_depth', 1) ## 3/4 datasets␣
↪calibration; 1/4 for validation.
    context.set('copy_number', 3) #3 per model (5 models here)
    context.set('run_mode', True)
    context.set('lower_bound', 1e-3)
    context.set('upper_bound', 1e2)
    config = context.get_config()

pe = ParameterEstimation(config)
data = pycotools3.viz.Parse(pe).concat()
```

Note: The *cross_validation_depth* argument specifies how far to go combinatorially. For instance, when *cross_validation_depth=2* and there are 4 datasets, all combinations of 2 datasets for experiments and 2 for validation will be applied.

Warning: While validation experiments are correctly configured with pycotools, there seems to be some instability in the current release of Copasi regarging multiple experiments in the *validation datasets* feature. Validation experiments work well when only one validation experiment is specified, but can crash when more than one is given.

Note: The *copy_number* applies per model here. So 4 datasets, *cross_validation_depth=1*

means four models are configured for validation. Also configured is the model without any validation experiments for convenience.

The *validation_weight* and *validation_threshold* arguments are specific for validations. The copasi docs are vague on precisely what these mean but the higher the threshold, the more rigerous the validation.

4.4 API documentation

Here you will find detailed information about every module class and method in the pycotools3 package.

4.4.1 The model module

The pycotools3 model is of central importance in pycotools.

Model	Construct a pycotools3 model from a copasi file
ImportSBML	Import from sbml file
InsertParameters	Parse parameters into a copasi model
BuildAntimony	Build a copasi model using antimony
Build	Build a copasi model.

pycotools3.model.Model

class pycotools3.model.**Model**(*copasi_file*, *quantity_type='concentration'*, *new=False*, ***kwargs*)

Construct a pycotools3 model from a copasi file

The Model object is of central importance in pycotools as it extracts relevant information from a copasi file file into python.

These are *Model* attributes and properties:

Examples

```
>>> from pycotools3.model import Model
>>> model_path = r'/full/path/to/model.cps'
>>> model = Model(model_path) ##work in concentration units
>>> model = Model(model_path, quantity_type='particle_numbers')
 ↪## work in particle numbers
```

Property	Description
copasi_file	Full path to model
root	Full path directory containing model
reference	Copasi model reference
time_unit	Time unit
name	Model name
volume_unit	Volume unit
quantity_unit	Quantity unit
area_unit	Area Unit
length_unit	Length unit
avagadro	Avagadro's number
key	Model key
states	List of states in correct order defined by copasi StateTemplate element.
fit_item_order	Order in which fit items appear
all_variable_names	List of reactions, metabolites, global_quantities local_parameters, compartment names as string
number_of_reactions	Number of reactions in `model.Model`

__init__(*copasi_file*, *quantity_type='concentration'*, *new=False*, ***kwargs*)

Parameters

- **copasi_file** (*str*) – full path to a copasi file
- **quantity_type** (*str*) – either 'concentration' (default) or 'particle_numbers'
- **new** (*bool*) – True when constructing a new model

Methods

__init__(copasi_file[, quantity_type, new])	**param copasi_file** full path to a copasi file
add(component_name, **kwargs)	add a model component to the model
add_compartment(compartment)	Add compartment to model
add_component(component_name, component[, ...])	add a model component to the model
add_function(function)	Add function to model
add_global_quantity(global_quantity)	Add global quantity to model
add_local_parameter(local_parameter)	Add a local parameter to the model, specifically into the String='kinetic Parameters' section of parameter sets

Continued on next page

Table 2 – continued from previous page

add_metabolite(metab)	Add a metabolite to the model xml
add_reaction(reaction[, expression, rate_law])	**param reaction** py:class:*Reaction* or *str*. If *str* then
add_state(state, value)	Append state on to end of state template.
convert_molar_to_particles(mole… mol_unit, …)	Convert molarity to particle numbers
convert_particles_to_molar(particles, …)	Converts particle numbers to Molarity.
get(component, value[, by])	Factory method for getting a model component by a value of a certain type
get_variable_names([which, …])	Get the names of variables in the model.
insert_parameters(**kwargs)	Wrapper around the InsetParameters class
open([copasi_file, as_temp])	Open model with the gui.
refresh()	Save the file then reload the Model.
remove(component, name)	General factor method for removing model components
remove_compartment(value[, by])	Remove a compartment with the attribute given as the 'by' and value arguments
remove_function(value[, by])	remove a function from model
remove_global_quantity(value[, by])	Remove a global quantity from your model
remove_metabolite(value[, by])	Remove metabolite from model.
remove_reaction(value[, by])	Remove reaction
remove_state(state)	Remove state from StateTemplate and InitialState fields.
reset_cache(prop)	Delete property from cache then reset it
save([copasi_file])	Save copasiML to copasi_filename.
scan([inplace])	Perform a parameter scan on model
set(component, match_value, new_value[, …])	Set a model components attribute to a new value
simulate(start, stop, by[, variables])	
to_antimony()	Args:
to_df()	Convert kwargs to 1D df :return: pandas.DataFrame
to_dict()	get kwargs as dictionary :return: dict
to_sbml([sbml_file])	convert model to sbml
to_string()	Produce kwargs as string format for using in __str__ methods in subclasses.

Attributes

active_parameter_set	get active parameter set
all_variable_names	The names of all compartments, metabolites, global quantities, reactions and local parameters in the model.
area_unit	*str*. The currently defined area unit.
avagadro	Not really needed but good to check the consistancy of avagadros number.
compartments	Get list of model compartments
constants	Get list of constants from xml attribute 'cn="String=Kinetic Parameters"
copasi_file	*Model*
fit_item_order	Get names of parameters being fitted in the order they appear
functions	get model functions
global_quantities	*list* each element is `GlobalQuantity`
key	Get the model reference - the 'key' from self.get_model_units
length_unit	*str*
local_parameters	Get local parameters in model.
metabolites	*list*. Each element is `Metabolite`
name	*str*. The model name
number_of_reactions	*int* number of reactions
parameter_descriptions	*list*. Each element a `ParameterDescription`
parameter_sets	Here for potential future implementation of easy switching between parameter sets :return:
parameters	get all locals, globals and metabs as pandas dataframe
quantity_unit	*str*. The currently defined quantity unit
reactions	assemble a list of reactions
reference	Get model reference from xml
root	Root directory for model.
states	The states (metabolites, globals, compartments) in the order they are read by Copasi from the StateTemplate element.
time_unit	*str* current time unit defined by copasi
volume_unit	*str*. The currently defined volume unit

active_parameter_set

get active parameter set

Not really in use

Returns `etree.Element`

Args:

Returns:

add(*component_name*, ***kwargs*)
add a model component to the model

Parameters

- **component_name** – str‘. i.e. ‘reaction’, ‘function’, ‘metabolite
- **component** – py:class:*model.<component>*. The component class to add i.e. Metabolite
- **reaction_expression** – When adding reaction using string as first arg, this argument takes the reaction expression (i.e. A -> B)
- **reaction_rate_law** – When adding reaction using string as first argument this argument takes the reaction rate law (i.e. k*A)
- ****kwargs** –

Returns py:class:‘Model

add_compartment(*compartment*)
Add compartment to model

Parameters **compartment** – py:class:*Compartment*

Returns py:class:*Model*

add_component(*component_name*, *component*, *reaction_expression=None*, *reaction_rate_law=None*)
add a model component to the model

Parameters

- **component_name** – str‘. i.e. ‘reaction’, ‘function’, ‘metabolite
- **component** – py:class:*model.<component>*. The component class to add i.e. Metabolite
- **reaction_expression** – When adding reaction using string as first arg, this argument takes the reaction expression (i.e. A -> B) (Default value = None)
- **reaction_rate_law** – When adding reaction using string as first argument this argument takes the reaction rate law (i.e. k*A) (Default value = None)

Returns class:‘Model

Return type py

add_function(*function*)
Add function to model

Parameters **function** – py:class:*Function*.

Returns py:class:*Model*

add_global_quantity(*global_quantity*)
Add global quantity to model

Parameters **global_quantity** – str‘ or `GlobalQuantity`. If str

is the name of global_quantity to add and default `GlobalQuantity` properties are adopted. If `GlobalQuantity`, a `GlobalQuantity` instance must be prebuilt and passes as arg.

Returns py:class:*Model*

add_local_parameter(*local_parameter*)
Add a local parameter to the model, specifically into the String='kinetic Parameters' section of parameter sets

Parameters **local_parameter** – py:class:*LocalParameter*

Returns py:class:*Model*

add_metabolite(*metab*)
Add a metabolite to the model xml

Parameters **metab** – str‘ or `Metabolite`. If str

is the name of metabolite to add and default `Metabolite` properties are adopted. If `Metabolite`, a `Metabolite` instance must be prebuilt and passes as arg.

Returns py:class:*Model*

add_reaction(*reaction*, *expression=None*, *rate_law=None*)

Parameters **reaction** – py:class:*Reaction* or *str*. If *str* then

must be the name of the reaction. expression: (Default value = None) rate_law: (Default value = None)

Returns py:class:*Model*

add_state(*state*, *value*)
Append state on to end of state template. Used within add_metabolite and add_global_quantity. Shouldn't need to use manually

Parameters

- **state** – str‘. A valid key
- **value** – int‘, *float*. Value for state

Returns:

all_variable_names
The names of all compartments, metabolites, global quantities, reactions and local parameters in the model.

Returns *list*. Each element is *str*

Args:

Returns:

area_unit

str. The currently defined area unit.

Args:

Returns:

Type return

avagadro
Not really needed but good to check the consistancy of avagadros number.

This number was changed between version 16 and 19 and caused a bug

Args:

Returns:

compartments
Get list of model compartments

Returns *list*. Each element is `Compartment`

Args:

Returns:

constants
Get list of constants from xml attribute ‘cn=”String=Kinetic Parameters”

Returns *list* each element `LocalParameter`

static convert_molar_to_particles(*moles*, *mol_unit*, *compartment_volume*)
Convert molarity to particle numbers

Parameters

- **moles** – int‘ or *float*. Number of moles in mol_unit to convert
- **mol_unit** – str‘. Mole unit to convert from.

suppoerted: fmol, pmol, nmol, umol, mmol or mol compartment_volume: int‘ or *float*. Volume of compartment containing specie to convert

Returns int‘. number of particles

static convert_particles_to_molar(*particles*, *mol_unit*, *compartment_volume*)

Converts particle numbers to Molarity.

##TODO build support for copasi's newest units

Parameters

- **particles** – int‘ Number of particles to convert
- **mol_unit** – str‘. The quantity unit, i.e:

fmol, pmol, nmol, umol, mmol or mol compartment_volume: int‘, *float*. Volume of compartment containing specie to convert

Returns float‘. Molarity

copasi_file

Model

Type Returns

fit_item_order

Get names of parameters being fitted in the order they appear

Returns *list*

Args:

Returns:

functions

get model functions

Returns *list* each element a *py:class:‘Function*

get(*component*, *value*, *by='name'*)

Factory method for getting a model component by a value of a certain type

Parameters

- **component** – str‘. The component i.e. *metabolite* or *local_parameter*
- **value** – str‘. Value of the attribute to match by i.e. metabolite called A
- **by** – str‘. Which attribute to search by. i.e. name or key or value (Default value = ‘name’)

Returns

py:class:Model.<component>‘

Get reaction called A2B:

Get metabolite called A:

Get all reactions which have a fixed simulation_type:

Get all compartments with an initial value of 15 (concentration or particles depending on quantity_type):

Get metabolites in the nucleus compartment:

Return type Instance of ‘

```
>>> model.get('reaction', 'A2B', by='name')
```

```
>>> model.get('metabolite', 'A', by='name')
```

```
>>> model.get('global_quantity', 'fixed', by='simulation_
↪type')
```

```
>>> model.get('compartment', 15, by='initial_value')
```

```
>>> model.get('metabolite', 'nuc', by='compartment')
```

get_model_object(*string*)

Retrieve a model object, such as a parameter or metabolite :param string: name of parameter :type string: str

Returns correct model object

get_parameters_as_antimony()

get parametes as antimony string Returns (string):

get_parameters_as_dict()

Uses :py:meth:‘model.Model.get_parameters_as_antimony then converts into a dict. Returns:

get_variable_names(*which='a'*, *include_assignments=True*, *prefix=None*)

Get the names of variables in the model. If include_assignments is off these are ommited from the results (this is useful for ParameterEstimation) as they are not generally estimated. Prefix provides a way of filtering the returned list

Parameters

- **which** – string. Default='a'. A string containing any or all of characters ‘a’, ‘m’, ‘g’, ‘l’, ‘c’ for all, metabolites, global_quantities, local_parameters and compartments respectively
- **include_assignments** – Boolean. Default=True. If True, return global variables with assignments
- **prefix** – str. Default=None. If given, returned parameter names are filtered to only include parameter with *prerfix* at the begining.

Returns ‘list’ of variable names

global_quantities

list each element is `GlobalQuantity`

Args:

Returns:

Type return

insert_parameters (***kwargs*)

Wrapper around the InsetParameters class

Parameters

- **kwargs** – Arguments for InsertParameters
- ****kwargs** –

Returns py:class:*Model*

key

Get the model reference - the 'key' from self.get_model_units

Returns *str*

Args:

Returns:

length_unit

str

Args:

Returns:

Type return

local_parameters

Get local parameters in model. local_parameters are those which are actively used in reactions and do not have a global variable assigned to them. The constant property returns all local parameters regardless of simulation type (fixed or assignment)

Returns *list*. Each element is `LocalParameter`

Args:

Returns:

metabolites

list. Each element is `Metabolite`

Args:

Returns:

Type return

name

str. The model name

Args:

Returns:

Type return

number_of_reactions

int number of reactions

Args:

Returns:

Type return

open(*copasi_file=None*, *as_temp=False*)

Open model with the gui. In order to work the environment variables must be properly set so that the command *CopasiUI* in the terminal or command prompt opens the model.

First *Model.save()* the model to copasi_file then open with CopasiUI. Optionally open with a temporary filename.

Parameters

- **copasi_file** – str‘ or *None*. Same as `model.Save()` (Default value = None)
- **as_temp** – bool‘. Use temp file to open the model and remove

afterwards (Default value = False)

Returns None‘

parameter_descriptions

list. Each element a `ParameterDescription`

Args:

Returns:

Type return

parameter_sets

Here for potential future implementation of easy switching between parameter sets :return:

Args:

Returns:

parameters

get all locals, globals and metabs as pandas dataframe

Returns `pandas.DataFrame`

Args:

Returns:

quantity_unit

str. The currently defined quantity unit

Args:

Returns:

Type return

reactions

assemble a list of reactions

Returns *list* each element a `Reaction`

reference

Get model reference from xml

Returns *str*

Args:

Returns:

refresh()

Save the file then reload the Model. Can't use the save method though because the save method uses the refresh method. :return:

Args:

Returns:

remove(*component*, *name*)

General factor method for removing model components

Parameters

- **component** – str‘ which component to remove (i.e. metabolite)
- **name** – str‘ name of component to remove

Returns py:class:*Model*

remove_compartment(*value*, *by='name'*)

Remove a compartment with the attribute given as the 'by' and value arguments

Parameters

- **value** – str‘. Value of attribute to match i.e. 'Nucleus'
- **by** – str‘ which attribute to match i.e. 'name' or 'key' (Default value = 'name')

Returns py:class:*Model*

remove_function(*value*, *by='name'*)

remove a function from model

Parameters

- **value** – str‘ value of attribute to match (i.e the functions name)
- **by** – str‘ which attribute to match by. default=’name’

Returns py:class:*model.Model*

remove_global_quantity (*value*, *by='name'*)
Remove a global quantity from your model

Parameters

- **value** – value to match by (i.e. ProteinA or ProteinB)
- **by** – attribute to match (i.e. name or key) (Default value = ‘name’)

Returns py:class:*model.Model*

remove_metabolite (*value*, *by='name'*)
Remove metabolite from model.

Parameters

- **value** – str‘. Attribute value to remove
- **by** – str‘ Any metabolite attribute type to match (Default value = ‘name’)

Returns

py:class:*Model*

Usage: ## Remove attribute called ‘A’

Remove metabolites with initial concentration of 0

```
>>> model.remove_metabolite('A', by='name')
```

```
>>> model.remove_metabolite(0, by='concentration')
```

remove_reaction (*value*, *by='name'*)
Remove reaction

Parameters

- **value** – str‘. Value of attibute
- **by** – attribute of reaction to match default=’name

str which :py:class‘Reaction‘ atrribute to match

Returns py:class:*Model*

remove_state (*state*)
Remove state from StateTemplate and InitialState fields. USed for deleting metabolites and global quantities.

Parameters **state** – str‘. key of state to remove (i.e. Metabolite_1)

Returns py:class:*Model*

reset_cache(*prop*)
Delete property from cache then reset it

Parameters **prop** – str‘. property to reset

Returns py:class:*Model*

root
Root directory for model. The directory where copasi_file is saved.

Does not need a setter since root is derived from copasi_file property

Returns *str*

Args:

Returns:

save(*copasi_file=None*)
Save copasiML to copasi_filename.

Parameters **copasi_filename** – str‘ or *None*. Deafult is *None*. When *None* defaults to same filepath the model came from. If another path, saves to that path. copasi_file: (Default value = None)

Returns py:class:*Model*

scan(*inplace=False*, ***kwargs*)
Perform a parameter scan on model

This is a wrapper around `tasks.Scan` and accepts all of the same arguments, except the model which is already provided.

Parameters ****kwargs** –

Returns:

sensitivities(*inplace=False*, ***kwargs*)
Perform a sensitivity analysis on model

This is a wrapper around `tasks.Sensitivities` and accepts all of the same arguments, except the model which is already provided.

Parameters ****kwargs** –

Returns:

set(*component*, *match_value*, *new_value*, *match_field='name'*, *change_field='name'*)
Set a model components attribute to a new value

Parameters

- **component** – str‘ type of component to change (i.e. metbaolite)
- **match_value** – str‘, *int*, *float* depending on value of *match_field*.

The value to match. new_value: str‘, *int* or *float* depending on value of *match_field*

new value for component attribute match_field: str‘. The attribute of component to match by. (Default value = ‘name’) change_field: str‘ The attribute of the component matched that you want to change? (Default value = ‘name’)

Returns

py:class:*Model*

Set initial concentration of metabolite called ‘X’ to 50:

Set name of global quantity called ‘G’ to ‘H’:

```
>>> model.set('metabolite', 'X', 50, match_field='name',␣
↪change_field='concentration')
```

```
>>> model.set('global_quantity', 'G', 'H', match_field='name
↪', change_field='name')
```

states

The states (metabolites, globals, compartments) in the order they are read by Copasi from the StateTemplate element.

Returns *OrderedDict*

Args:

Returns:

time_unit

str current time unit defined by copasi

Args:

Returns:

Type return

to_antimony()

Args:

Returns return:

to_sbml(*sbml_file=None*)

convert model to sbml

Parameters **sbml_file** – str‘. Path for SBML. Defaults to same as copasi filename

Returns str‘. Path to smbl file

to_tellurium()

return a roadrunner model via the tellurium package Returns:

volume_unit

str. The currently defined volume unit

Args:

Returns:

Type return

pycotools3.model.ImportSBML

class pycotools3.model.**ImportSBML**(*sbml_file*, *copasi_file=None*)

Import from sbml file

Accepts an SBML file, converts it to copasi format and reads it into a Model object

__init__(*sbml_file*, *copasi_file=None*)

Parameters

- **sbml_file** (*str*) – path to sbml
- **copasi_file** (*None*, *str*) – Default is None and pycotools automatically creates a copasi model with the same name as the sbml file. Otherwise, a path to copasi_file.

Methods

__init__(sbml_file[, copasi_file])	**param sbml_file** path to sbml
convert()	Perform conversion using CopasiSE :return:
copasi_filename()	
load_model()	

convert()

Perform conversion using CopasiSE :return:

Args:

Returns:

copasi_filename()

load_model()

pycotools3.model.InsertParameters

class pycotools3.model.**InsertParameters**(*model*, *parameter_dict=None*, *df=None*, *parameter_path=None*, *index=0*, *quantity_type='concentration'*, *inplace=False*)

Parse parameters into a copasi model

Insert parameters from a file, dictionary or a pandas dataframe into a copasi file.

__init__(*model*, *parameter_dict=None*, *df=None*, *parameter_path=None*, *index=0*, *quantity_type='concentration'*, *inplace=False*)

Parameters

- **model** (*Model*) – The model to parse parameters into
- **parameter_dict** (*dict*) – Default None. If not None, dict[parameter_name] = parameter_value
- **df** (*pandas.DataFrame*) – Default None. If not None, a dataframe containing parameters to insert
- **parameter_path** (*str*) – Default None. If not None a path to parameter estimation output file
- **index** (*int*) – Default 0 (best RSS). When multiple parameter sets available, rank of best fit you want to insert
- **quantity_type** (*str*) – concentration (default) or particle_numbers
- **inplace** (*bool*) – Whether to operate inplace or return a new model

Methods

__init__(model[, parameter_dict, df, ...])	**param model** The model to parse parameters into
insert()	User other methods defined in this class to insert parameters into the model :return:
insert_compartments()	insert new parameters into compartment :return:
insert_global_quantities()	insert new parameters into compartment :return:

Continued on next page

Table 5 – continued from previous page

insert_locals()	**return**
insert_metabolites()	insert new parameters into compartment :return:
read_model(m)	**param m**
to_dict()	Args:

Attributes

parameters	Get parameters depending on the type of input.

insert()
User other methods defined in this class to insert parameters into the model :return:

Args:

Returns:

insert_compartments()
insert new parameters into compartment :return:

Args:

Returns:

insert_global_quantities()
insert new parameters into compartment :return:

Args:

Returns:

insert_locals()

Returns

insert_metabolites()
insert new parameters into compartment :return:

Args:

Returns:

parameters
Get parameters depending on the type of input. Converge on a pandas dataframe. Columns = parameters, rows = parameter sets

Use check parameter consistency to see whether headers have been pruned or not. If not try pruning them

Args:

Returns:

to_dict()
Args:

Returns return:

pycotools3.model.BuildAntimony

class pycotools3.model.**BuildAntimony**(*copasi_file: str*)
Build a copasi model using antimony

A context manager to create a copasi model *copasi_file* using the antimony language

Examples

```
working_directory = os.path.dirname(__file__)
copasi_filename = os.path.join(working_directory,
↪'NegativeFeedbackModel.cps')
with model.BuildAntimony(copasi_filename) as loader:
    negative_feedback = loader.load(
        '''
        model negative_feedback()
            // define compartments
            compartment cell = 1.0
            //define species
            var A in cell
            var B in cell
            //define some global parameter for use in reactions
            vAProd = 0.1
            kADeg = 0.2
            kBProd = 0.3
            kBDeg = 0.4
            //define initial conditions
            A = 0
            B = 0
            //define reactions
            AProd: => A; cell*vAProd
            ADeg: A =>; cell*kADeg*A*B
            BProd: => B; cell*kBProd*A
            BDeg: B => ; cell*kBDeg*B
        end
        '''
    )
print(negative_feedback)
```

__init__(*copasi_file: str*)

Parameters **copasi_file** (*str*) – Path to a valid location on disk to store the copasi file

Methods

__init__(copasi_file)	**param copasi_file** Path to a valid location on disk to store the copasi file
load(antimony_str)	Load the antimony string `antimony_str` into a `copasi_file` and *Model*.

load(*antimony_str*)

Load the antimony string `antimony_str` into a `copasi_file` and *Model*.

Args antimony_str (str): A valid antimony string encoding a model

return

model (*Model*) A PyCoTools model containing the model defined in the **:parameter:'antiomny_str'**.

Parameters **antimony_str** –

Returns:

pycotools3.model.Build

class pycotools3.model.**Build**(*copasi_file*)

Build a copasi model.

Context manager for building a copasi model with only PyCoTools Functions.

Users should also see *BuildAntimony*

__init__(*copasi_file*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(copasi_file)	Initialize self.

4.4.2 The tasks module

TimeCourse	Simulate a time course
ParameterEstimation.Config	A parameter estimation configuration class
ParameterEstimation	Interface to COPASI's parameter estimation task
ParameterEstimation.Context	A high level interface to create a *ParameterEstimation.Config* object.
Sensitivities	Interface to COPASI sensitivity task
Scan	Interface to COPASI scan task
Reports	Creates reports in copasi output specification section.

pycotools3.tasks.TimeCourse

class pycotools3.tasks.**TimeCourse**(*model*, ***kwargs*)

Simulate a time course

A class for running a time course from python using a copasi model. All but one of copasi's solvers are supported and available via the *method* kwarg.

TimeCourse Kwargs	Description
intervals	Default: 100
step_size	Default: 0.01
end	Default: 1
start	Default: 0
update_model	Default: False
method	Default: deterministic
output_event	Default: False
scheduled	Default: True
automatic_step_size	Default: False
start_in_steady_state	Default: False
integrate_reduced_model	Default: False
relative_tolerance	Default: 1e-6
absolute_tolerance	Default: 1e-12
max_internal_steps	Default: 10000
max_internal_step_size	Default: 0
subtype	Default: 2
use_random_seed	Default: True
random_seed	Default: 1
epsilon	Default: 0.001
lower_limit	Default: 800
upper_limit	Default: 1000
partitioning_interval	Default: 1
runge_kutta_step_size	Default: 0.001
run	Default: True
correct_headers	Default: True
save	Default: False
<report_kwargs>	Arguments for :ref:‘ report_kwargs <report_kwargs>‘ are also accepted here

Args:

Returns:

__init__(*model*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(model, **kwargs)	Initialize self.
adaptive_tau_leap()	**return**

Continued on next page

Table 10 – continued from previous page

check_integrity(allowed, given)	Method to raise an error when a wrong kwarg is passed to a subclass
convert_bool_to_numeric(dct)	CopasiML uses 1's and 0's for True or False in some but not all places.
convert_bool_to_numeric2()	CopasiML uses 1's and 0's for True or False in some but not all places.
create_task()	Begin creating the segment of xml needed for a time course.
deterministic()	:return:lxml.etree._Element
direct()	**return**
get_report_key()	cross reference the timecourse task with the newly created time course reort to get the key
get_variable_from_string(m, v[, glob])	Use model entity name to get the pycotools3 variable
gibson_bruck()	**return**
hybrid_lsoda()	**return**
hybrid_rk45()	**return**
hybrid_runge_kutta()	**return**
read_model(m)	**param m**
set_report()	ser a time course report containing time and all species or global quantities defined by the user.
set_timecourse()	Set method specific sections of xml.
simulate()	
tau_leap()	**return**
update_properties(kwargs)	method for updating properties from kwargs

Attributes

schema

adaptive_tau_leap()

Returns

create_task()
Begin creating the segment of xml needed for a time course. Define task and problem definition. This section of xml is common to all methods :return: lxml.etree._Element

Args:

Returns:

deterministic()
:return:lxml.etree._Element

direct()

Returns

get_report_key()
cross reference the timecourse task with the newly created time course reort to get the key

Args:

Returns:

gibson_bruck()

Returns

hybrid_lsoda()

Returns

hybrid_rk45()

Returns

hybrid_runge_kutta()

Returns

set_report()
ser a time course report containing time and all species or global quantities defined by the user.

Returns pycotools3.model.Model

Args:

Returns:

set_timecourse()
Set method specific sections of xml. This is a method element after the problem element that looks like this:

Returns lxml.etree._Element

Args:

Returns:

simulate()

tau_leap()

Returns

pycotools3.tasks.ParameterEstimation.Config

class pycotools3.tasks.ParameterEstimation.**Config**(*models*, *datasets*, *items*, *settings={}*, *defaults=None*)

A parameter estimation configuration class

Stores as attibutes all the settings needed for configuration of a parameter estimation using COPASI. Base class is a `Bunch`

Structure of a `ParameterEstimation.Config` Object has four main sections:

- models
- datasets
- items
- settings

Examples

```
>>> ## create a model
>>> antimony_string = '''
...                 model TestModel1()
...                     R1: A => B; k1*A;
...                     R2: B => A; k2*B
...                     A = 1
...                     B = 0
...                     k1 = 4;
...                     k2 = 9;
...                 end
...                 '''
>>> copasi_filename = os.path.join(os.path.dirname(__file__),
↪'example_model.cps')
>>> mod = moddel.loada(antimony_string, copasi_filename)
>>> ## Simulate some data from the model and write to file
```

(continues on next page)

(continued from previous page)

```
>>> fname = os.path.join(os.path.dirname(__file__), 'timeseries.
↪txt')
>>> data = self.model.simulate(0, 10, 11)
>>> data.to_csv(fname)
>>>
>>> ## create nested dict containing all the relevant arguments␣
↪for your configuration
>>> config_dict = dict(
...         models=dict(
...              ## model name is the users choice here
...             example1=dict(
...                 copasi_file=copasi_filename
...             )
...         ),
...         datasets=dict(
...             experiments=dict(
...                  ## experiment names are the users choice
...                 report1=dict(
...                     filename=self.TC1.report_name,
...                 ),
...             ),
...             ## our validations entry is empty here
...             ## but if you have validation data this should
...             ## be the same as the experiments section
...             validations=dict(),
...         ),
...         items=dict(
...             fit_items=dict(
...                 A=dict(
...                     affected_experiments='report1'
...                 ),
...                 B=dict(
...                     affected_validation_experiments=['report2
↪']
...                 ),
...             k1={},
...             k2={},
...             ),
...             constraint_items=dict(
...                 k1=dict(
...                     lower_bound=1e-2,
...                     upper_bound=10
...                 )
...             )
...         ),
...         settings=dict(
...             method='genetic_algorithm_sr',
...             population_size=2,
...             number_of_generations=2,
```

(continues on next page)

(continued from previous page)

```
...             working_directory=os.path.dirname(__file__),
...             copy_number=4,
...             pe_number=2,
...             weight_method='value_scaling',
...             validation_weight=2.5,
...             validation_threshold=9,
...             randomize_start_values=True,
...             calculate_statistics=False,
...             create_parameter_sets=False
...         )
...     )
>>> config = ParameterEstimation.Config(**config_dict)
```

__init__(*models*, *datasets*, *items*, *settings={}*, *defaults=None*)

Initialisation method for Config class

Parameters

- **models** (*dict*) – Dict containing model names and paths to copasi files
- **datasets** (*dict*) – Dict containing experiments and validation experiments
- **items** (*dict*) – Dict containing fit items and constraint items
- **settings** (*dict*) – Dict containing all other settings for parameter estimation
- **defaults** (*ParameterEstimation._Defaults*) – Custom set of Defaults to use for unspecified arguments

Returns:

Methods

__init__(models, datasets, items[, ...])	Initialisation method for Config class
check_integrity(allowed, given)	Method to raise an error when a wrong kwarg is passed to a subclass
clear()	
configure()	Configure the class for production of parameter estimation config
convert_bool_to_numeric(dct)	CopasiML uses 1's and 0's for True or False in some but not all places.
convert_bool_to_numeric2()	CopasiML uses 1's and 0's for True or False in some but not all places.
copy()	

Continued on next page

Table 12 – continued from previous page

fromDict(d)	Recursively transforms a dictionary into a Munch via copy.
fromYAML(*args, **kwargs)	
from_json(string)	Create config object from json format :param string: a valid json string :type string: Str
from_yaml(yml)	Read config object from yaml file :param yml: full path to text file containing configuration arguments in yaml format :type yml: str
fromkeys	Returns a new dict with keys from iterable and values equal to value.
get(k[,d])	
get_variable_from_string(m, v[, glob])	Use model entity name to get the pycotools3 variable
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.
set_default_fit_items_dct()	Configure missing entries for items.fit_items when they are in nested dict format
set_default_fit_items_str()	Configure missing entries for items.fit_items when they are strings pointing towards model variables
setdefault(k[,d])	
toDict()	Recursively converts a munch back into a dictionary.
toJSON(**options)	Serializes this Munch to JSON.
toYAML(**options)	Serializes this Munch to YAML, using *yaml.safe_dump()* if no *Dumper* is provided.
to_json()	Output arguments as json
to_yaml([filename])	Output arguments as yaml
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
update_properties(kwargs)	method for updating properties from kwargs
values()	

Attributes

constraint_items	The constraint items as nested dict
experiment_filenames	A list of experiment filesnames
experiment_names	A list of experiment names
experiments	The experiments property :returns: datasets.experiments as dict
fit_items	The fit items as nested dict
model_objects	A list of model objects for mapping
schema	
validation_filenames	a list of validation filenames
validation_names	A list of validation names
validations	The validations property :returns: datasets.validations as dict

configure()
Configure the class for production of parameter estimation config

Like a main method for this class. Uses the other methods in the class to configure a `ParameterEstimation.Config object`

Returns Operates inplace and returns None

constraint_items
The constraint items as nested dict

Type Returns

experiment_filenames
A list of experiment filesnames

Type Returns

experiment_names
A list of experiment names

Type Returns

experiments
The experiments property :returns: datasets.experiments as dict

fit_items
The fit items as nested dict

Type Returns

from_json(*string*)
Create config object from json format :param string: a valid json string :type string: Str

Returns ParameterEstimation.Config

static from_yaml(*yml*)
Read config object from yaml file :param yml: full path to text file containing configuration arguments in yaml format :type yml: str

Returns ParameterEstimation.Config

model_objects
A list of model objects for mapping

Type Returns

models_affected_experiments
Get which experiment datasets affect which models

Returns dict. Keys are model names, value are list of affected models

models_affected_validation_experiments
Get which experiment datasets affect which models

Returns dict. Keys are model names, value are list of affected models

multi_experiments
List of experiment names that have more than one experiment separated by blank line Returns (tuple): (list, int)

set_default_fit_items_dct()
Configure missing entries for items.fit_items when they are in nested dict format

Returns None. Method operates inplace on class attributes

set_default_fit_items_str()
Configure missing entries for items.fit_items when they are strings pointing towards model variables

Returns None. Method operates inplace on class attributes

to_json()
Output arguments as json

Returns: str All arguments in json format

to_yaml(*filename=None*)
Output arguments as yaml

Parameters **filename** (*str, None*) – If not None (default), path to write yaml configuration to

Returns Config object as string in yaml format

validation_filenames
a list of validation filenames

Type Returns

validation_names
A list of validation names

Type Returns

validations
The validations property :returns: datasets.validations as dict

pycotools3.tasks.ParameterEstimation

class pycotools3.tasks.**ParameterEstimation**(*config*)
Interface to COPASI's parameter estimation task

Examples

Assuming a *ParameterEstimation.Config* object has been configured and is called *config* >>> pe = ParameterEstimation(config)

__init__(*config*)
Configure a the parameter estimation task in copasi

Pycotools supports all the features of parameter estimation configuration as copasi, plus a few additional ones (such as the affected models setting).

Parameters **config** (ParameterEstimation.Config) – An appropriately configured *ParameterEstimation.Config* class

Examples

See *ParameterEstimation.Config* or *ParameterEstimation.Context* for detailed information on how to produce a *ParameterEstimation.Config* object. Note that the *ParameterEstimation.Context* class is higher level and should be the preferred way of constructing a *ParameterEstimation.Config* object while the *ParameterEstimation.Config* class gives you the same level of control as copasi but is bulkier to write.

Assuming the *ParameterEstimation.Config* class has already been created >>> pe = ParameterEstimation(config)

Methods

__init__(config)	Configure a the parameter estimation task in copasi
check_integrity(allowed, given)	Method to raise an error when a wrong kwarg is passed to a subclass
convert_bool_to_numeric(dct)	CopasiML uses 1's and 0's for True or False in some but not all places.
convert_bool_to_numeric2()	CopasiML uses 1's and 0's for True or False in some but not all places.

Continued on next page

Table 14 – continued from previous page

do_checks()	validate integrity of user input
get_component(model, component, string)	Get component called string from model
get_model_objects_from_strings()	Get model objects from the strings provided by the user in the Config class :return: list of *model.Model* objects
get_variable_from_string(m, v[, glob])	Use model entity name to get the pycotools3 variable
read_model(m)	**param m**
run(models)	Run a parameter estimation using command line copasi.
update_properties(kwargs)	method for updating properties from kwargs

Attributes

fit_dir	Property holding the directory where the parameter estimation fitting occurs.
global_quantities	list of strings of global quantities present in the models
local_parameters	list of strings of local parameters in the model
metabolites	list of strings of metabolites in the model
models	Get models
models_dir	A directory containing models
problem_dir	Property holding the directory where the parameter estimation problem is stored :returns: str.
results_directory	A directory containing results, parameter estimation report files from copasi
schema	
valid_methods	

class Config(*models*, *datasets*, *items*, *settings={}*, *defaults=None*)

A parameter estimation configuration class

Stores as attibutes all the settings needed for configuration of a parameter estimation using COPASI. Base class is a `Bunch`

Structure of a *ParameterEstimation.Config* Object has four main sections:

- models
- datasets

- items
- settings

Examples

```
>>> ## create a model
>>> antimony_string = '''
...                 model TestModel1()
...                     R1: A => B; k1*A;
...                     R2: B => A; k2*B
...                     A = 1
...                     B = 0
...                     k1 = 4;
...                     k2 = 9;
...                 end
...                 '''
>>> copasi_filename = os.path.join(os.path.dirname(__file__
↪), 'example_model.cps')
>>> mod = moddel.loada(antimony_string, copasi_filename)
>>> ## Simulate some data from the model and write to file
>>> fname = os.path.join(os.path.dirname(__file__),
↪'timeseries.txt')
>>> data = self.model.simulate(0, 10, 11)
>>> data.to_csv(fname)
>>>
>>> ## create nested dict containing all the relevant␣
↪arguments for your configuration
>>> config_dict = dict(
...         models=dict(
...              ## model name is the users choice here
...             example1=dict(
...                 copasi_file=copasi_filename
...             )
...         ),
...         datasets=dict(
...             experiments=dict(
...                  ## experiment names are the users choice
...                 report1=dict(
...                     filename=self.TC1.report_name,
...                 ),
...             ),
...             ## our validations entry is empty here
...             ## but if you have validation data this␣
↪should
...             ## be the same as the experiments section
...             validations=dict(),
...         ),
...         items=dict(
...             fit_items=dict(
```

(continues on next page)

(continued from previous page)

```
...                 A=dict(
...                     affected_experiments='report1'
...                 ),
...                 B=dict(
...                     affected_validation_experiments=[
↪'report2']
...                 ),
...             k1={},
...             k2={},
...             ),
...             constraint_items=dict(
...                 k1=dict(
...                     lower_bound=1e-2,
...                     upper_bound=10
...                 )
...             )
...         ),
...         settings=dict(
...             method='genetic_algorithm_sr',
...             population_size=2,
...             number_of_generations=2,
...             working_directory=os.path.dirname(__file__),
...             copy_number=4,
...             pe_number=2,
...             weight_method='value_scaling',
...             validation_weight=2.5,
...             validation_threshold=9,
...             randomize_start_values=True,
...             calculate_statistics=False,
...             create_parameter_sets=False
...         )
...     )
>>> config = ParameterEstimation.Config(**config_dict)
```

configure()

Configure the class for production of parameter estimation config

Like a main method for this class. Uses the other methods in the class to configure a `ParameterEstimation.Config` object

Returns Operates inplace and returns None

constraint_items

The constraint items as nested dict

Type Returns

experiment_filenames

A list of experiment filesnames

Type Returns

experiment_names
A list of experiment names

Type Returns

experiments
The experiments property :returns: datasets.experiments as dict

fit_items
The fit items as nested dict

Type Returns

from_json(*string*)
Create config object from json format :param string: a valid json string :type string: Str

Returns ParameterEstimation.Config

static from_yaml(*yml*)
Read config object from yaml file :param yml: full path to text file containing configuration arguments in yaml format :type yml: str

Returns ParameterEstimation.Config

model_objects
A list of model objects for mapping

Type Returns

models_affected_experiments
Get which experiment datasets affect which models

Returns dict. Keys are model names, value are list of affected models

models_affected_validation_experiments
Get which experiment datasets affect which models

Returns dict. Keys are model names, value are list of affected models

multi_experiments
List of experiment names that have more than one experiment separated by blank line Returns (tuple): (list, int)

set_default_fit_items_dct()
Configure missing entries for items.fit_items when they are in nested dict format

Returns None. Method operates inplace on class attributes

set_default_fit_items_str()
Configure missing entries for items.fit_items when they are strings pointing towards model variables

Returns None. Method operates inplace on class attributes

to_json()
Output arguments as json

Returns: str All arguments in json format

to_yaml(*filename=None*)
Output arguments as yaml

Parameters **filename** (*str, None*) – If not None (default), path to write yaml configuration to

Returns Config object as string in yaml format

validation_filenames
a list of validation filenames

Type Returns

validation_names
A list of validation names

Type Returns

validations
The validations property :returns: datasets.validations as dict

class Context(*models*, *experiments*, *working_directory=None*, *context='s'*, *parameters='mg'*, *filename=None*, *validation_experiments={}*, *settings={}*)
A high level interface to create a *ParameterEstimation.Config* object.

Enables the construction of a *ParameterEstimation.Config* object assuming one of several common patterns of usage.

Examples

Assuming that we have two copasi models (*mod1* and *mod2*) and two experimental data files (*fname1*, *fname2*), correctly formatted according to the copasi specification. We can generate a config object that specifies the fitting of both experiments to both models and to fit all global and local parameters *parameters='gl'* in each.

```
with ParameterEstimation.Context(
    [mod1, mod2], [fname1, fname2],
    context='s', parameters='gl') as context:
    context.set('method', 'genetic_algorithm_sr')
    context.set('number_of_generations', 25)
    context.set('population_size', 10)
    config = context.get_config()

pe = ParameterEstimation(config)
```

Or for profile likelihoods on the first model *mod1*

get_config_cv()
configure for cross validation Returns:

get_config_pl()
configure for profile likelihoods Returns:

set(*parameter*, *value*)
Set the value of `parameter` to `value`.

Looks for the first instance of `parameter` and sets its value to `value`.

Parameters

- **parameter** – A key somewhere in the nested structure of the config object
- **value** – A value to replace the current value with

Returns None

setd(*dct*)
Set the value of multiple settings using a dict[setting] = value.

Iterates over *ParameterEstimation.Context.set()* with key value pairs

Parameters **dct** (*dict*) – a settings dict where keys are settings and values are setting values

Returns None

do_checks()
validate integrity of user input

duplicate_for_every_experiment(*model*, *fit_items*, *lower_bounds*, *start_values*, *upper_bounds*)
Replicate Copasi's duplicate for every experiment button.

Parameters

- **model** –
- **fit_items** –
- **lower_bounds** –
- **start_values** –
- **upper_bounds** –

Returns:

fit_dir
Property holding the directory where the parameter estimation fitting occurs. This can be enumerated under a single problem directory to group similar parameter estimations :returns: str. A directory.

get_model_objects_from_strings()
Get model objects from the strings provided by the user in the Config class :return: list of *model.Model* objects

Returns list of model objects

global_quantities
list of strings of global quantities present in the models

Type Returns

local_parameters
list of strings of local parameters in the model

Type Returns

metabolites
list of strings of metabolites in the model

Type Returns

models
Get models

Returns the models entry of the *ParameterEstimation.Config* object

models_dir
A directory containing models

Each model will be configured in a different directory when multiple models are being configured simultaneously :returns: dct. Location of models directories

problem_dir
Property holding the directory where the parameter estimation problem is stored :returns: str. A directory.

results_directory
A directory containing results, parameter estimation report files from copasi

Each model configured will have their own results directory :returns: dict[model] = results_directory

run(*models*)
Run a parameter estimation using command line copasi.

Parameters **models** – dict of models. Output from _setup()

Returns dict of models. Output from _setup()

Return type param models

pycotools3.tasks.ParameterEstimation.Context

class pycotools3.tasks.ParameterEstimation.**Context**(*models*, *experiments*, *working_directory=None*, *context='s'*, *parameters='mg'*, *filename=None*, *validation_experiments={}*, *settings={}*)

A high level interface to create a `ParameterEstimation.Config` object.

Enables the construction of a `ParameterEstimation.Config` object assuming one of several common patterns of usage.

Examples

Assuming that we have two copasi models (*mod1* and *mod2*) and two experimental data files (*fname1*, *fname2*), correctly formatted according to the copasi specification. We can generate a config object that specifies the fitting of both experiments to both models and to fit all global and local parameters *parameters='gl'* in each.

```
with ParameterEstimation.Context(
    [mod1, mod2], [fname1, fname2],
    context='s', parameters='gl') as context:
    context.set('method', 'genetic_algorithm_sr')
    context.set('number_of_generations', 25)
    context.set('population_size', 10)
    config = context.get_config()

pe = ParameterEstimation(config)
```

Or for profile likelihoods on the first model *mod1*

__init__(*models*, *experiments*, *working_directory=None*, *context='s'*, *parameters='mg'*, *filename=None*, *validation_experiments={}*, *settings={}*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(models, experiments[, ...])	Initialize self.
get_config()	
set(parameter, value)	Set the value of parameter to value.

Attributes

acceptable_context_args
acceptable_parameters_args
experiment_filetypes

get_config_cv()
configure for cross validation Returns:

get_config_pl()
configure for profile likelihoods Returns:

set(*parameter*, *value*)
Set the value of parameter to value.

Looks for the first instance of parameter and sets its value to value.

Parameters

- **parameter** – A key somewhere in the nested structure of the config object
- **value** – A value to replace the current value with

Returns None

setd(*dct*)
Set the value of multiple settings using a dict[setting] = value.

Iterates over ParameterEstimation.Context.set() with key value pairs

Parameters **dct** (*dict*) – a settings dict where keys are settings and values are setting values

Returns None

pycotools3.tasks.Sensitivities

class pycotools3.tasks.**Sensitivities**(*model*, ***kwargs*)
Interface to COPASI sensitivity task

__init__(*model*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(model, **kwargs)	Initialize self.
add_list_of_variables_element()	
check_integrity(allowed, given)	Method to raise an error when a wrong kwarg is passed to a subclass
convert_bool_to_numeric(dct)	CopasiML uses 1's and 0's for True or False in some but not all places.
convert_bool_to_numeric2()	CopasiML uses 1's and 0's for True or False in some but not all places.
create_new_report()	
create_problem()	
create_sensitivity_task()	
get_component(model, component, string)	Get component called string from model
get_report_key()	
get_single_object_references()	
get_variable_from_string(m, v[, glob])	Use model entity name to get the pycotools3 variable
process_data()	
read_model(m)	**param m**
replace_sensitivities_task()	
run_task()	
sensitivity_task_key()	Get the sensitivity task as it currently is in the model as etree.Element :return:
set_cause()	
set_effect()	
set_method()	
set_report()	
set_secondary_cause()	
set_subtask()	
update_properties(kwargs)	method for updating properties from kwargs

Attributes

cross_section_cause
cross_section_effect
evaluation_cause
evaluation_effect
optimization_cause
optimization_effect
parameter_estimation_cause

Continued on next page

Table 19 – continued from previous page

parameter_estimation_effect
schema
sensitivitity_number_map
steady_state_cause
steady_state_effect
subtasks
time_series_cause
time_series_effect
update_model

add_list_of_variables_element()

create_new_report()

create_problem()

create_sensitivity_task()

get_report_key()

get_single_object_references()

process_data()

replace_sensitivities_task()

run_task()

sensitivity_task_key()
Get the sensitivity task as it currently is in the model as etree.Element :return:

Args:

Returns:

set_cause()

set_effect()

set_method()

set_report()

set_secondary_cause()

set_subtask()

pycotools3.tasks.Scan

class pycotools3.tasks.**Scan**(*model*, ***kwargs*)
Interface to COPASI scan task

Scan Kwargs	Description
update_model	Default: False
subtask	Default: parameter_estimation
report_type	Default: profile_likelihood. Name of report from *Reports* class
output_in_subtask	Default: False
adjust_initial_conditions	Default: False
number_of_steps	Default: 10
maximum	Default: 100
minimum	Default: 0.01
log10	Default: False
distribution_type	Default: normal
scan_type	Default: scan
scheduled	Default: True
save	Default: False
clear_scans	Default: True. If true, will remove all scans present then add new scan
run	Default: False
<report_kwargs>	Arguments for report_kwargs are also accepted here

Args:

Returns:

__init__(*model*, ***kwargs*)

Parameters

- **model** – Model
- **kwargs** –

Methods

__init__(model, **kwargs)	**param model**
check_integrity(allowed, given)	Method to raise an error when a wrong kwarg is passed to a subclass
convert_bool_to_numeric(dct)	CopasiML uses 1's and 0's for True or False in some but not all places.
convert_bool_to_numeric2()	CopasiML uses 1's and 0's for True or False in some but not all places.
create_scan()	metabolite cn:
define_report()	Use Report class to create report :return:
execute()	

Continued on next page

Table 20 – continued from previous page

get_report_key()	
get_variable_from_string(m, v[, glob])	Use model entity name to get the pycotools3 variable
read_model(m)	**param m**
remove_scans()	Remove all scans that have been defined.
set_scan_options()	
update_properties(kwargs)	method for updating properties from kwargs

Attributes

schema

create_scan()

metabolite cn: CN=Root,Model=New Model,Vector=Compartments[nuc],Vector=Metabolites[A

Returns

Args:

Returns:

define_report()
Use Report class to create report :return:

Args:

Returns:

execute()

get_report_key()

remove_scans()
Remove all scans that have been defined.

Returns

Args:

Returns:

set_scan_options()

pycotools3.tasks.Reports

class pycotools3.tasks.**Reports**(*model*, ***kwargs*)
Creates reports in copasi output specification section. Which report is controlled by the

report_type key word. The following are valid types of report:

Report Types	Description
time_course	Report definition for collection of time course data.
parameter_estimation	Collect parameter estimates from parameter estimations run from the parameter estimation task
multi_parameter_estimation	Collect parameter estimation data from parameter estimations run from the scan task with copasi's repeat feature
profile_likelihood	Collect both the parameter being scanned value and the parameter estimates

Here are the keyword arguments accepted by the Reports class.

Args:

Returns:

__init__(*model*, ***kwargs*)

A class for configuring reports :param model: Model to add report configuration to :type model: `model.Model` :param **kwargs: Arguments for report

Methods

__init__(model, **kwargs)	A class for configuring reports :param model: Model to add report configuration to :type model: `model.Model` :param **kwargs: Arguments for report
`check_integrity`(allowed, given)	Method to raise an error when a wrong kwarg is passed to a subclass
clear_all_reports()	Having multile reports defined at once can be really annoying and give you unexpected results.
`convert_bool_to_numeric`(dct)	CopasiML uses 1's and 0's for True or False in some but not all places.
`convert_bool_to_numeric2`()	CopasiML uses 1's and 0's for True or False in some but not all places.
`get_component`(model, component, string)	Get component called string from model
`get_variable_from_string`(m, v[, glob])	Use model entity name to get the pycotools3 variable
multi_parameter_estimation()	Define a parameter estimation report and include the progression of the parameter estimation (function evaluations).
parameter_estimation()	Define a parameter estimation report and include the progression of the parameter estimation (function evaluations).

Continued on next page

Table 22 – continued from previous page

profile_likelihood()	Create report of a parameter and best value for a parameter estimation for profile likelihoods
read_model(m) **param m**	
remove_report(report_name)	remove report called report_name
run()	Execute code that builds the report defined by the kwargs
scan()	creates a report to collect scan time course results.
sensitivity()	
timecourse()	creates a report to collect time course results.
update_properties(kwargs)	method for updating properties from kwargs

Attributes

schema

clear_all_reports()
Having multile reports defined at once can be really annoying and give you unexpected results. Use this function to remove all reports before defining a new one to ensure you only have one active report any once. :return:

Args:

Returns:

multi_parameter_estimation()
Define a parameter estimation report and include the progression of the parameter estimation (function evaluations). Defaults to including all metabolites, global variables and local variables with the RSS best value These can be over-ridden with the global_quantities, LocalParameters and metabolites keywords.

Args:

Returns:

parameter_estimation()
Define a parameter estimation report and include the progression of the parameter estimation (function evaluations). Defaults to including all metabolites, global variables and local variables with the RSS best value These can be over-ridden with the global_quantities, LocalParameters and metabolites keywords.

Args:

Returns:

profile_likelihood()
Create report of a parameter and best value for a parameter estimation for profile likelihoods

Args:

Returns:

remove_report(*report_name*)
remove report called report_name

Parameters **report_name** – return: pycotools3.model.Model

Returns pycotools3.model.Model

run()
Execute code that builds the report defined by the kwargs

scan()
creates a report to collect scan time course results.

By default all species and all global quantities are used with Time on the left most column. This behavior can be overwritten by passing lists of metabolites to the metabolites keyword or global quantities to the global quantities keyword

Args:

Returns:

sensitivity()

timecourse()
creates a report to collect time course results.

By default all species and all global quantities are used with Time on the left most column. This behavior can be overwritten by passing lists of metabolites to the metabolites keyword or global quantities to the global quantities keyword

Args:

Returns:

4.4.3 The viz module

The viz module exists to make visualising simulation output quick and easy for common patterns, such as plotting time courses or comparing parameter estimation output to experimental data. However it should be emphasised that the *matplotlib* and *seaborn* libraries are always close to hand in Python.

The viz module is currently in a state of rebuilding and so I only describe here the features which currently work.

Parse	General class for parsing copasi output into Python.
PlotTimeCourse	Plot time course data
Boxplots	Plot a boxplot for multi parameter estimation data.

pycotools3.viz.Parse

class pycotools3.viz.**Parse**(*cls_instance*, *log10=False*, *copasi_file=None*, *alpha=0.95*, *rss_value=None*, *num_data_points=None*)

General class for parsing copasi output into Python.

First argument is an instance of a pycotools3 class.

instance	Description
tasks.TimeCourse	Parse time course data from TC.report_name into pandas.df
tasks.ParameterEstimation	Parse parameter estimation data from PE.report_name into pandas.df
tasks.Scan	Parse scan data from scan.report_name
Parse	enable parsing from a parse instance. Just returns itself
str	Parse data from folder of parameter estimation data into pandas.df. Requires the copasi file argument.

Args:

Returns:

__init__(*cls_instance*, *log10=False*, *copasi_file=None*, *alpha=0.95*, *rss_value=None*, *num_data_points=None*)

Parameters

- **cls_instance** – A instance of pycotools3 class
- **log10** – *bool*. Whether to work on log10 scale
- **copasi_file** – *str*. Optional but necessary when cls_instance is string. Must be the copasi_file which produced the parameter estimation data as Parse extracts data headers from the copasi file
- **rss_value** – float When cls is a profile likelihood with the current_parameters setting,

  rss_value may not be empty. It is not automatically inferable from the COPASI model and must be specified separetly.

- **num_data_points** – int When cls is a profile likelihood with current paraemters setting, the number of data points can-

not be automatically inferred for the calculation of likelihood ratio based confidence intervals. Therefore, this must be specified by the user.

Methods

__init__(cls_instance[, log10, copasi_file, ...])	**param cls_instance**
from_chaser_estimations(cls_instance[, folder])	**return**
from_folder()	**param folder** return:
from_multi_parameter_estimation(cls_instance)	Results without headers - parse the results give them the proper headers then overwrite the file again
from_profile_likelihood()	Parse data from `tasks.ProfileLikelihood` :return: `pandas.DataFrame`
from_timecourse()	read time course data into pandas dataframe.
parse()	determine class type of self.cls_instance and call the appropirate method for parsing the data type :return:
parse_scan()	read scan data into pandas Dataframe.

Attributes

from_parameter_estimation	Parse parameter estimation data.

from_chaser_estimations(*cls_instance*, *folder=None*)

Returns

Parameters

- **cls_instance** –
- **folder** – (Default value = None)

Returns:

from_folder()

Parameters **folder** – return:

Returns:

static from_multi_parameter_estimation(*cls_instance*)
Results come without headers - parse the results give them the proper headers then overwrite the file again

Parameters

- **cls_instance** – instance of MultiParameterEstiamtion
- **folder** – afternative folder to parse from. Useful for tests (Default value = None)

Returns:

from_parameter_estimation
Parse parameter estimation data. Store the data in a cache. :return:

Args:

Returns:

from_profile_likelihood()
Parse data from `tasks.ProfileLikelihood` :return:

`pandas.DataFrame`

Args:

Returns:

from_timecourse()
read time course data into pandas dataframe. Remove copasi generated square brackets around the variables :return: pandas.DataFrame

Args:

Returns:

parse()
determine class type of self.cls_instance and call the appropirate method for parsing the data type :return:

Args:

Returns:

parse_scan()
read scan data into pandas Dataframe. :return: pandas.DataFrame

Args:

Returns:

pycotools3.viz.PlotTimeCourse

class pycotools3.viz.**PlotTimeCourse**(*cls*, ***kwargs*)
Plot time course data

Time course kwargs:

kwarg	Description
x	*str*. Parameter to go on x axis. defaults to 'Time'. If not 'Time' then plot is a phase space plot. Required.
y	*str* or *list* of *str*. Parameters for the y axis. Required.
log10	*bool* plot on log10 scale
separate	bool' separate time courses onto different axes. Default: True
**kwargs	See kwargs for more options

Args:

Returns:

__init__(*cls*, ***kwargs*)

Parameters

- **cls** – Instance of tasks.TimeCourse class
- **kwargs** –

Methods

__init__(cls, **kwargs)	**param cls**
context([font_scale, rc])	**param context** (Default value = 'talk')
create_directory(results_directory)	create directory for results and switch to it
parse(cls, log10[, copasi_file])	Mixin method interface to parse class :return:
plot()	**return**
plot_kwargs()	
save_figure(directory, filename[, dpi])	**param directory**
truncate(data, mode, theta)	mixin method interface to truncate data
update_properties(kwargs)	method for updating properties from kwargs

plot()

Returns

pycotools3.viz.Boxplots

class pycotools3.viz.**Boxplots**(*cls*, ***kwargs*)

Plot a boxplot for multi parameter estimation data.

kwarg	Description
num_per_plot	Number of parameter per plot. Remainder fills up another plot.
**kwargs	see kwargs options

Args:

Returns:

__init__(*cls*, ***kwargs*)

Parameters

- **cls** – instance of tasks.MultiParameterEstimation or string . Same as PlotTimeCourseEnsemble
- **kwargs** –

Methods

__init__(cls, **kwargs)	**param cls**
context([font_scale, rc])	**param context** (Default value = 'talk')
create_directory()	**return**
divide_data()	split data into multi plot :return:
parse(cls, log10[, copasi_file])	Mixin method interface to parse class :return:
plot()	Plot multiple parameter estimation data as boxplot :return:
plot_kwargs()	
save_figure(directory, filename[, dpi])	**param directory**
truncate(data, mode, theta)	mixin method interface to truncate data
update_properties(kwargs)	method for updating properties from kwargs

create_directory()

Returns

divide_data()
split data into multi plot :return:

Args:

Returns:

plot()
Plot multiple parameter estimation data as boxplot :return:

Args:

Returns:

CHAPTER 5

Support

Users are encouraged to post an issue on GitHub for help or to report bugs.

CHAPTER 6

People

PyCoTools has been developed by Ciaran Welsh in Daryl Shanley's lab at Newcastle University.

CHAPTER 7

Caveats

- Non-ascii characters are minimally supported and can break PyCoTools
- Do not use unusual characters or naming systems (i.e. A reaction name called "A -> B" will break pycotools)
- In COPASI we can have (say) a global quantity and a metaboltie with the same name because they are different entities. This is not supported in Pycotools and you must use unique names for every model component

7.1 Citing PyCoTools

If you made use of PyCoTools, please cite this article using:

- Welsh, C.M., Fullard, N., Proctor, C.J., Martinez-Guimera, A., Isfort, R.J., Bascom, C.C., Tasseff, R., Przyborski, S.A. and Shanley, D.P., 2018. PyCoTools: a Python toolbox for COPASI. Bioinformatics, 34(21), pp.3702-3710.

And also please remember to cite COPASI:

- Hoops, S., Sahle, S., Gauges, R., Lee, C., Pahle, J., Simus, N., Singhal, M., Xu, L., Mendes, P. and Kummer, U., 2006. COPASI—a complex pathway simulator. Bioinformatics, 22(24), pp.3067-3074.

and tellurium:

- Medley, J.K., Choi, K., König, M., Smith, L., Gu, S., Hellerstein, J., Sealfon, S.C. and Sauro, H.M., 2018. Tellurium notebooks—An environment for reproducible dynamical modeling in systems biology. PLoS computational biology, 14(6), p.e1006220.

Index

Symbols

__init__() (*pycotools3.model.Build method*), 92
__init__() (*pycotools3.model.BuildAntimony method*), 91
__init__() (*pycotools3.model.ImportSBML method*), 88
__init__() (*pycotools3.model.InsertParameters method*), 89
__init__() (*pycotools3.model.Model method*), 74
__init__() (*pycotools3.tasks.ParameterEstimation method*), 103
__init__() (*pycotools3.tasks.ParameterEstimation.Config method*), 99
__init__() (*pycotools3.tasks.ParameterEstimation.Context method*), 111
__init__() (*pycotools3.tasks.Reports method*), 117
__init__() (*pycotools3.tasks.Scan method*), 115
__init__() (*pycotools3.tasks.Sensitivities method*), 112
__init__() (*pycotools3.tasks.TimeCourse method*), 94
__init__() (*pycotools3.viz.Boxplots method*), 124
__init__() (*pycotools3.viz.Parse method*), 120
__init__() (*pycotools3.viz.PlotTimeCourse method*), 123

A

active_parameter_set (*pycotools3.model.Model attribute*), 76
adaptive_tau_leap() (*pycotools3.tasks.TimeCourse method*), 96
add() (*pycotools3.model.Model method*), 77
add_compartment() (*pycotools3.model.Model method*), 77
add_component() (*pycotools3.model.Model method*), 77
add_function() (*pycotools3.model.Model method*), 77
add_global_quantity() (*pycotools3.model.Model method*), 78
add_list_of_variables_element() (*pycotools3.tasks.Sensitivities method*), 114
add_local_parameter() (*pycotools3.model.Model method*), 78
add_metabolite() (*pycotools3.model.Model method*), 78
add_reaction() (*pycotools3.model.Model method*), 78
add_state() (*pycotools3.model.Model method*), 78
all_variable_names (*pycotools3.model.Model attribute*), 78
area_unit (*pycotools3.model.Model attribute*), 79

avagadro (*pycotools3.model.Model attribute*), 79

B

Boxplots (*class in pycotools3.viz*), 124
Build (*class in pycotools3.model*), 92
BuildAntimony (*class in pycotools3.model*), 91

C

clear_all_reports() (*pycotools3.tasks.Reports method*), 118
compartments (*pycotools3.model.Model attribute*), 79
Config (*class in pycotools3.tasks.ParameterEstimation*), 97
configure() (*pycotools3.tasks.ParameterEstimation.Config method*), 101, 106
constants (*pycotools3.model.Model attribute*), 79
constraint_items (*pycotools3.tasks.ParameterEstimation.Config attribute*), 101, 106
Context (*class in pycotools3.tasks.ParameterEstimation*), 111
convert() (*pycotools3.model.ImportSBML method*), 88
convert_molar_to_particles() (*pycotools3.model.Model static method*), 79
convert_particles_to_molar() (*pycotools3.model.Model static method*), 79
copasi_file (*pycotools3.model.Model attribute*), 80
copasi_filename() (*pycotools3.model.ImportSBML method*), 88
create_directory() (*pycotools3.viz.Boxplots method*), 125
create_new_report() (*pycotools3.tasks.Sensitivities method*), 114
create_problem() (*pycotools3.tasks.Sensitivities method*), 114
create_scan() (*pycotools3.tasks.Scan method*), 116
create_sensitivity_task() (*pycotools3.tasks.Sensitivities method*), 114
create_task() (*pycotools3.tasks.TimeCourse method*), 96

D

define_report() (*pycotools3.tasks.Scan method*), 116
deterministic() (*pycotools3.tasks.TimeCourse method*), 96
direct() (*pycotools3.tasks.TimeCourse method*), 96
divide_data() (*pycotools3.viz.Boxplots method*), 125
do_checks() (*pycotools3.tasks.ParameterEstimation method*), 109
duplicate_for_every_experiment() (*pycotools3.tasks.ParameterEstimation method*), 109

E

execute() (*pycotools3.tasks.Scan method*), 116
experiment_filenames (*pycotools3.tasks.ParameterEstimation.Config attribute*), 101, 106
experiment_names (*pycotools3.tasks.ParameterEstimation.Config attribute*), 101, 106
experiments (*pycotools3.tasks.ParameterEstimation.Config attribute*), 101, 107

F

fit_dir (*pycotools3.tasks.ParameterEstimation attribute*), 109
fit_item_order (*pycotools3.model.Model attribute*), 80
fit_items (*pycotools3.tasks.ParameterEstimation.Config*

attribute), 101, 107
from_chaser_estimations() (*pycotools3.viz.Parse method*), 121
from_folder() (*pycotools3.viz.Parse method*), 121
from_json() (*pycotools3.tasks.ParameterEstimation.Config method*), 101, 107
from_multi_parameter_estimation() (*pycotools3.viz.Parse static method*), 121
from_parameter_estimation (*pycotools3.viz.Parse attribute*), 122
from_profile_likelihood() (*pycotools3.viz.Parse method*), 122
from_timecourse() (*pycotools3.viz.Parse method*), 122
from_yaml() (*pycotools3.tasks.ParameterEstimation.Config static method*), 101, 107
functions (*pycotools3.model.Model attribute*), 80

G

get() (*pycotools3.model.Model method*), 80
get_config_cv() (*pycotools3.tasks.ParameterEstimation.Context method*), 108, 112
get_config_pl() (*pycotools3.tasks.ParameterEstimation.Context method*), 109, 112
get_model_object() (*pycotools3.model.Model method*), 81
get_model_objects_from_strings() (*pycotools3.tasks.ParameterEstimation method*), 109
get_parameters_as_antimony() (*pycotools3.model.Model method*), 81
get_parameters_as_dict() (*pycotools3.model.Model method*), 81
get_report_key() (*pycotools3.tasks.Scan method*), 116
get_report_key() (*pycotools3.tasks.Sensitivities method*), 114
get_report_key() (*pycotools3.tasks.TimeCourse method*), 96
get_single_object_references() (*pycotools3.tasks.Sensitivities method*), 114
get_variable_names() (*pycotools3.model.Model method*), 81
gibson_bruck() (*pycotools3.tasks.TimeCourse method*), 96
global_quantities (*pycotools3.model.Model attribute*), 81
global_quantities (*pycotools3.tasks.ParameterEstimation attribute*), 110

H

hybrid_lsoda() (*pycotools3.tasks.TimeCourse method*), 96
hybrid_rk45() (*pycotools3.tasks.TimeCourse method*), 96
hybrid_runge_kutta() (*pycotools3.tasks.TimeCourse method*), 96

I

ImportSBML (*class in pycotools3.model*), 88
insert() (*pycotools3.model.InsertParameters method*), 90
insert_compartments() (*pycotools3.model.InsertParameters method*), 90
insert_global_quantities() (*pycotools3.model.InsertParameters method*), 90
insert_locals() (*pycotools3.model.InsertParameters method*), 90
insert_metabolites() (*pycotools3.model.InsertParameters method*), 90
insert_parameters() (*pycotools3.model.Model method*), 82

InsertParameters (*class in pycotools3.model*), 89

K

key (*pycotools3.model.Model attribute*), 82

L

length_unit (*pycotools3.model.Model attribute*), 82
load() (*pycotools3.model.BuildAntimony method*), 92
load_model() (*pycotools3.model.ImportSBML method*), 88
local_parameters (*pycotools3.model.Model attribute*), 82
local_parameters (*pycotools3.tasks.ParameterEstimation attribute*), 110

M

metabolites (*pycotools3.model.Model attribute*), 82
metabolites (*pycotools3.tasks.ParameterEstimation attribute*), 110
Model (*class in pycotools3.model*), 73
model_objects (*pycotools3.tasks.ParameterEstimation.Config attribute*), 102, 107
models (*pycotools3.tasks.ParameterEstimation attribute*), 110
models_affected_experiments (*pycotools3.tasks.ParameterEstimation.Config attribute*), 102, 107
models_affected_validation_experiments (*pycotools3.tasks.ParameterEstimation.Config attribute*), 102, 107
models_dir (*pycotools3.tasks.ParameterEstimation attribute*), 110
multi_experiments (*pycotools3.tasks.ParameterEstimation.Config attribute*), 102, 107
multi_parameter_estimation() (*pycotools3.tasks.Reports method*), 118

N

name (*pycotools3.model.Model attribute*), 82
number_of_reactions (*pycotools3.model.Model attribute*), 83

O

open() (*pycotools3.model.Model method*), 83

P

parameter_descriptions (*pycotools3.model.Model attribute*), 83
parameter_estimation() (*pycotools3.tasks.Reports method*), 118
parameter_sets (*pycotools3.model.Model attribute*), 83
ParameterEstimation (*class in pycotools3.tasks*), 103
ParameterEstimation.Config (*class in pycotools3.tasks*), 104
ParameterEstimation.Context (*class in pycotools3.tasks*), 108
parameters (*pycotools3.model.InsertParameters attribute*), 90
parameters (*pycotools3.model.Model attribute*), 83
Parse (*class in pycotools3.viz*), 120
parse() (*pycotools3.viz.Parse method*), 122
parse_scan() (*pycotools3.viz.Parse method*), 122
plot() (*pycotools3.viz.Boxplots method*), 125
plot() (*pycotools3.viz.PlotTimeCourse method*), 123
PlotTimeCourse (*class in pycotools3.viz*), 122
problem_dir (*pycotools3.tasks.ParameterEstimation attribute*), 110
process_data() (*pycotools3.tasks.Sensitivities method*), 114

profile_likelihood() (*pycotools3.tasks.Reports method*), 118

Q

quantity_unit (*pycotools3.model.Model attribute*), 83

R

reactions (*pycotools3.model.Model attribute*), 84
reference (*pycotools3.model.Model attribute*), 84
refresh() (*pycotools3.model.Model method*), 84
remove() (*pycotools3.model.Model method*), 84
remove_compartment() (*pycotools3.model.Model method*), 84
remove_function() (*pycotools3.model.Model method*), 84
remove_global_quantity() (*pycotools3.model.Model method*), 85
remove_metabolite() (*pycotools3.model.Model method*), 85
remove_reaction() (*pycotools3.model.Model method*), 85
remove_report() (*pycotools3.tasks.Reports method*), 119
remove_scans() (*pycotools3.tasks.Scan method*), 116
remove_state() (*pycotools3.model.Model method*), 85
replace_sensitivities_task() (*pycotools3.tasks.Sensitivities method*), 114
Reports (*class in pycotools3.tasks*), 116
reset_cache() (*pycotools3.model.Model method*), 86
results_directory (*pycotools3.tasks.ParameterEstimation attribute*), 110
root (*pycotools3.model.Model attribute*), 86
run() (*pycotools3.tasks.ParameterEstimation method*), 110
run() (*pycotools3.tasks.Reports method*), 119
run_task() (*pycotools3.tasks.Sensitivities method*), 114

S

save() (*pycotools3.model.Model method*), 86
Scan (*class in pycotools3.tasks*), 114
scan() (*pycotools3.model.Model method*), 86
scan() (*pycotools3.tasks.Reports method*), 119
Sensitivities (*class in pycotools3.tasks*), 112
sensitivities() (*pycotools3.model.Model method*), 86
sensitivity() (*pycotools3.tasks.Reports method*), 119
sensitivity_task_key() (*pycotools3.tasks.Sensitivities method*), 114
set() (*pycotools3.model.Model method*), 86
set() (*pycotools3.tasks.ParameterEstimation.Context method*), 109, 112
set_cause() (*pycotools3.tasks.Sensitivities method*), 114
set_default_fit_items_dct() (*pycotools3.tasks.ParameterEstimation.Config method*), 102, 107
set_default_fit_items_str() (*pycotools3.tasks.ParameterEstimation.Config method*), 102, 107
set_effect() (*pycotools3.tasks.Sensitivities method*), 114
set_method() (*pycotools3.tasks.Sensitivities method*), 114
set_report() (*pycotools3.tasks.Sensitivities method*), 114
set_report() (*pycotools3.tasks.TimeCourse method*), 96
set_scan_options() (*pycotools3.tasks.Scan method*), 116
set_secondary_cause() (*pycotools3.tasks.Sensitivities method*),

114
set_subtask() (pycotools3.tasks.Sensitivities method), 114
set_timecourse() (pycotools3.tasks.TimeCourse method), 96
setd() (pycotools3.tasks.ParameterEstimation.Context method), 109, 112
simulate() (pycotools3.tasks.TimeCourse method), 97
states (pycotools3.model.Model attribute), 87

T

tau_leap() (pycotools3.tasks.TimeCourse method), 97
time_unit (pycotools3.model.Model attribute), 87
TimeCourse (class in pycotools3.tasks), 93
timecourse() (pycotools3.tasks.Reports method), 119
to_antimony() (pycotools3.model.Model method), 87
to_dict() (pycotools3.model.InsertParameters method), 91
to_json() (pycotools3.tasks.ParameterEstimation.Config method), 102, 107
to_sbml() (pycotools3.model.Model method), 87
to_tellurium() (pycotools3.model.Model method), 87
to_yaml() (pycotools3.tasks.ParameterEstimation.Config method), 102, 108

V

validation_filenames (pycotools3.tasks.ParameterEstimation.Config attribute), 102, 108
validation_names (pycotools3.tasks.ParameterEstimation.Config attribute), 102, 108
validations (pycotools3.tasks.ParameterEstimation.Config attribute), 102, 108
volume_unit (pycotools3.model.Model attribute), 87