

---

**basiCO**

**Frank T. Bergmann**

**Apr 22, 2024**



# GETTING STARTED

<b>1</b>	<b>Getting Started</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Use the package . . . . .	3
1.3	Loading a Model . . . . .	3
1.4	Interrogating the Model . . . . .	4
1.5	Analyzing the Model . . . . .	4
1.6	Saving a Model . . . . .	5
<b>2</b>	<b>Accessing Models from the database</b>	<b>7</b>
2.1	BioModels . . . . .	7
2.2	JWS Online . . . . .	10
<b>3</b>	<b>Creating and simulating a simple model</b>	<b>17</b>
<b>4</b>	<b>Getting and setting reaction parameters and reactions</b>	<b>25</b>
4.1	Load model . . . . .	25
4.2	Getting/setting global quantities . . . . .	25
4.3	Getting/setting species (metabolites) . . . . .	26
4.4	Getting/setting reaction parameters . . . . .	27
4.5	Getting/setting reactions . . . . .	28
4.6	Setting scheme . . . . .	28
4.7	Setting reaction name . . . . .	28
<b>5</b>	<b>Editing reaction kinetics</b>	<b>31</b>
<b>6</b>	<b>Parameter Estimation</b>	<b>35</b>
<b>7</b>	<b>Parameter estimation setup</b>	<b>45</b>
7.1	preparing the data . . . . .	45
7.2	setting up the parameter estimation . . . . .	49
7.3	Constraints . . . . .	53
<b>8</b>	<b>Parameter Estimation (import / export)</b>	<b>57</b>
8.1	The Format . . . . .	57
8.2	Example . . . . .	58
<b>9</b>	<b>Optimization</b>	<b>61</b>
9.1	Model . . . . .	61
9.2	The Setup . . . . .	62
9.3	Running the optimization . . . . .	63
9.4	Customn Output . . . . .	63

9.5	Constraints . . . . .	64
<b>10</b>	<b>Simple simulations and plotting</b>	<b>67</b>
10.1	Load a model . . . . .	67
10.2	Run time course . . . . .	67
10.3	Get compartments . . . . .	68
10.4	Get parameters (“global quantities”) . . . . .	69
10.5	Run steady state . . . . .	69
10.6	Use pandas syntax for indexing and plotting . . . . .	70
<b>11</b>	<b>Simulating a model with basico</b>	<b>71</b>
11.1	The run_time_course command . . . . .	72
<b>12</b>	<b>Simulating with custom results</b>	<b>75</b>
12.1	Simulations: . . . . .	77
12.2	Custom Selection list . . . . .	77
12.3	Custom Time Points . . . . .	78
12.4	For expert users . . . . .	79
<b>13</b>	<b>Parameter Scans</b>	<b>81</b>
13.1	Modifying Scan Settings . . . . .	82
13.2	Runnin a scan: . . . . .	84
<b>14</b>	<b>Sensitivity Analysis</b>	<b>85</b>
14.1	Settings . . . . .	85
14.2	Running the Analysis . . . . .	86
<b>15</b>	<b>Working with arrays of Compartments</b>	<b>89</b>
15.1	Linear arrays of compartments . . . . .	90
15.2	Creating 2D array of compartments . . . . .	93
<b>16</b>	<b>Using callbacks</b>	<b>95</b>
16.1	Timeouts . . . . .	96
<b>17</b>	<b>Working with Plots</b>	<b>97</b>
<b>18</b>	<b>Working with Reports</b>	<b>101</b>
18.1	Retrieving Report Definitions . . . . .	101
18.2	Adding Reports . . . . .	102
18.3	Changing values . . . . .	103
18.4	Assigning to task . . . . .	103
<b>19</b>	<b>Working with SBML Ids</b>	<b>105</b>
<b>20</b>	<b>Working with Annotations</b>	<b>111</b>
20.1	Resource Lists / Updating . . . . .	111
20.2	Displaying Annotations: . . . . .	112
20.3	Setting annotations . . . . .	113
<b>21</b>	<b>Working with ipyparallel</b>	<b>117</b>
<b>22</b>	<b>Working with PETab</b>	<b>121</b>
<b>23</b>	<b>Model Selection</b>	<b>123</b>

<b>24 Working with Parameter Sets</b>	<b>125</b>
24.1 Looking at parameter sets . . . . .	125
24.2 Creating / Modifying Parameter sets . . . . .	126
24.3 Removing parameter sets . . . . .	127
24.4 Parameter sets and Parameter Estimation . . . . .	128
<b>25 Profile likelihood</b>	<b>131</b>
25.1 Preparing files . . . . .	131
25.2 Processing the files: . . . . .	133
25.3 Looking at the results . . . . .	133
25.4 Everything together . . . . .	137
<b>26 Metabolic Control Analysis</b>	<b>145</b>
<b>27 Working with Widgets</b>	<b>147</b>
<b>28 basico</b>	<b>149</b>
28.1 basico package . . . . .	149
<b>Python Module Index</b>	<b>217</b>
<b>Index</b>	<b>219</b>



BasiCO is a simplified interface to using COPASI from python. It is a collection of utility python scripts, that hide the complexity of the underlying SWIG generated language bindings.







## GETTING STARTED

### 1.1 Installation

BasiCO is finally available in pypi:

```
pip install copasi-basico
```

however you can pip install it using:

```
pip install git+https://github.com/copasi/basico.git
```

Currently there is one additional package available that provides support for PETab and petab-select, that can be installed using

```
pip install copasi-basico[petab]
```

or

```
pip install git+https://github.com/copasi/basico.git#egg=basico[petab]
```

### 1.2 Use the package

To start using the package, you simply import it:

```
>>> from basico import *
```

### 1.3 Loading a Model

You can load either COPASI, or SBML files directly using the `load_model()` command:

```
>>> load_model('brusselator.cps')
```

You can also load a model by providing a URL:

```
>>> load_model('https://fairdomhub.org/models/287/download?version-1')
```

We also provide a number of example models, that you can directly load with the installation. See `get_examples()` and `load_example()`.

Models from the BioModels Database and JWS can also be directly loaded ( `jws_online`, `biomodels`, `load_biomodel()` )

```
>>> load_biomodel(206)
```

## 1.4 Interrogating the Model

To find out what is in a model, you could use the corresponding functions:

- `get_species()`
- `get_reactions()`
- `get_parameters()`
- `get_compartments()`

Analogous you can also set all of these, by providing the name of the element to modify:

- `set_species()`
- `set_reaction()`
- `set_parameters()`
- `set_compartment()`

New elements are added with:

- `add_species()`
- `add_reaction()`
- `add_parameter()`
- `add_compartment()`

And removed with:

- `remove_species()`
- `remove_reaction()`
- `remove_parameter()`
- `remove_compartment()`

## 1.5 Analyzing the Model

Currently the following analysis tasks have been included:

- `task_timecourse`
- `task_steadystate`
- `task_parameterestimation`

## 1.6 Saving a Model

Saving a model, is done by calling `save_model()`:

```
>>> save_model('model_3.cps')
```

will save the file `model_3.cps` in the current folder. To export the model to SBML use:

```
>>> save_model('model_3.xml', type='sbml')
```



## ACCESSING MODELS FROM THE DATABASE

This example describes how to access models from BioModels and JWS. Further databases could be easily added later on.

### 2.1 BioModels

```
[1]: import sys
     if '../..' not in sys.path:
         sys.path.append('../..')

     import basico.biomodels as biomodels
```

the BioModels REST API allows searching for models, just as one would do in the web interface on <http://biomodels.net>. So one can search by pathway, species or reactions or submitter / author by just entering them.

```
[2]: glycolysis_models = biomodels.search_for_model('glycolysis')
     for model in glycolysis_models:
         print ('Id: %s' % model['id'])
         print ('Name: %s' % model['name'])
         print ('Format: %s' % model['format'])
         print ('')
```

```
Id: BIOMD0000000042
Name: Nielsen1998_Glycolysis
Format: SBML

Id: BIOMD0000000051
Name: Chassagnole2002_Carbon_Metabolism
Format: SBML

Id: BIOMD0000000054
Name: Ataullahkhanov1996_Adenylate
Format: SBML

Id: BIOMD0000000061
Name: Hynne2001_Glycolysis
Format: SBML

Id: BIOMD0000000064
Name: Teusink2000_Glycolysis
```

(continues on next page)

(continued from previous page)

```

Format: SBML

Id: BIOMD0000000071
Name: Bakker2001_Glycolysis
Format: SBML

Id: BIOMD0000000172
Name: Pritchard2002_glycolysis
Format: SBML

Id: BIOMD0000000176
Name: Conant2007_WGD_glycolysis_2A3AB
Format: SBML

Id: BIOMD0000000177
Name: Conant2007_glycolysis_2C
Format: SBML

Id: BIOMD0000000206
Name: Wolf2000_Glycolytic_Oscillations
Format: SBML

```

to get more information about a particular model, the `get_model_info` function provides basic information. As all the functions in this module they take either an integer or string biomodels id as parameter

```
[3]: info = biomodels.get_model_info(206)
```

```
[4]: print (info['name'])
      print (info['description'])
```

```

Wolf2000_Glycolytic_Oscillations
<notes xmlns="http://www.sbml.org/sbml/level2/version3">
  <body xmlns="http://www.w3.org/1999/xhtml">
    <p>Model reproduces the dynamics of ATP and NADH as depicted in Fig 4 of the
    ↪paper. Model successfully tested on Jarnac and MathSBML.</p>
    <br />
    <p>To the extent possible under law, all copyright and related or neighbouring
    ↪rights to this encoded model have been dedicated to the public domain worldwide.
    ↪Please refer to      <a href="http://creativecommons.org/publicdomain/zero/1.0/" title=
    ↪"Creative Commons CC0">CC0 Public Domain Dedication</a>
    ↪for more information.      </p>
    <p>In summary, you are entitled to use this encoded model in absolutely any manner.
    ↪you deem suitable, verbatim, or with modification, alone or embedded it in a larger
    ↪context, redistribute it, commercially or not, in a restricted way or not.</p>
    <br />
    <p>To cite BioModels Database, please use:      <a href="http://www.ncbi.nlm.nih.
    ↪gov/pubmed/20587024" target="_blank">Li C, Donizelli M, Rodriguez N, Dharuri H, Endler
    ↪L, Chelliah V, Li L, He E, Henry A, Stefan MI, Snoep JL, Hucka M, Le Novère N, Laibe C
    ↪(2010) BioModels Database: An enhanced, curated and annotated resource for published
    ↪quantitative kinetic models. BMC Syst Biol., 4:92.</a>
    </p>

```

(continues on next page)

(continued from previous page)

```
</body>
</notes>
```

the model information object also contains the list of file names associated with the entry. To just get that list the convenience function `get_files_for_model` exists. The main document is contained in a sublist called `main`. So the main entry can be retrieved using

```
[5]: first_entry = info['files']['main'][0]
```

```
[6]: print ("Main FileName is: '{0}' and has size {1} kb".format(first_entry['name'], first_
    ↪entry['fileSize']))
```

```
Main FileName is: 'BIOMD0000000206_url.xml' and has size 27693 kb
```

to actually get hold of the model itself, you can use the `get_content_for_model` function, that takes a model id, as well as an optional filename. If the filename is not given, the first main content will be chosen automatically. So to download the model of biomodel #206, one could simply call:

```
[7]: sbml = biomodels.get_content_for_model(206)
```

```
[8]: print(sbml[:1000]) # just printing the first couple of lines
```

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<sbml xmlns="http://www.sbml.org/sbml/level2/version3" level="2" metaid="metaid_0000001"
    ↪version="3">
  <model id="Wolf2000_Glycolytic_Oscillations" metaid="metaid_0000002" name="Wolf2000_
    ↪Glycolytic_Oscillations">
    <notes>
      <body xmlns="http://www.w3.org/1999/xhtml">
        <p>Model reproduces the dynamics of ATP and NADH as depicted in Fig 4 of the
    ↪paper. Model successfully tested on Jarnac and MathSBML.</p>
        <br/>
        <p>To the extent possible under law, all copyright and related or neighbouring
    ↪rights to this encoded model have been dedicated to the public domain worldwide.
    ↪Please refer to      <a href="http://creativecommons.org/publicdomain/zero/1.0/" title=
    ↪"Creative Commons CC0">CC0 Public Domain Dedication</a>
          for more information.      </p>
        <p>In summary, you are entitled to use this encoded model in absolutely any manner
    ↪you deem suitable, verbatim, or with modification, alone or embedd
```

of course you can simply call `load_biomodel(206)` to load a biomodel into `basico`.

```
[9]: biomodels.search_for_model('kummer2000')
```

```
[9]: [{'format': 'SBML',
    'id': 'BIOMD0000000329',
    'lastModified': '2014-12-11T00:00:00Z',
    'name': 'Kummer2000 - Oscillations in Calcium Signalling',
    'submissionDate': '2014-12-11T00:00:00Z',
    'submitter': 'Vijayalakshmi Chelliah',
    'url': 'https://www.ebi.ac.uk/biomodels/BIOMD0000000329'}]
```

## 2.2 JWS Online

we also provide access to models from JWS online.

```
[10]: import basico.jws_online as jws
```

with `get_all_models` you would get a list of all the models in JWS online. To search for models, you have 2 options. One is to search for models by species. For example to search for all models involving 'atp'

```
[11]: atp_models = jws.get_models_for_species('atp')
      for model in atp_models:
          print(model['slug'])
```

```
achcar1
achcar10
achcar11
achcar12
achcar13
achcar14
achcar2
achcar3
achcar4
achcar5
achcar6
achcar7
achcar8
achcar9
albert1
arnold10
arnold10-2
arnold11
arnold6
arnold7
arnold8
arnold9
assmus
bali
bier2
bradshaw
bray2
bruggeman1
bulik1
bulik2
bulik3
chance1
chassagnole1
chassagnole3
conant1
conant2
curto1
dano1
dano2
dano3
```

(continues on next page)



(continued from previous page)

dupreez1  
dupreez2  
dupreez3  
dupreez4  
dupreez5  
dupreez6  
dupreez7  
fribourg1  
fribourg2  
fridlyand1  
galazzo1  
gustavsson1  
gustavsson2  
gustavsson3  
gustavsson4  
gustavsson5  
hald  
heinrich  
hoefnagel1  
hoefnagel2  
hoefnagel\_mixedacid  
holzhutter  
hynne  
jamshidi  
jiang1  
kerkhovena  
kerkhovenc  
kolmeisky1  
kongas1  
kouril2  
kouril3  
kouril4  
kouril6  
kouril7  
kouril8  
kouril9  
lambeth  
levering1  
levering2  
maher1  
marinhernandez1  
marinhernandez2  
marinhernandez3  
mayya1  
mosca1  
mulquiney  
mulquiney1  
mulquiney2  
nazaret1  
neves1  
nielsen  
nishio1

(continues on next page)

(continued from previous page)

```

olah
orth1
penkler1
penkler2
penkler2aa
poolman
pritchard1
proctor1
rohwer1
rovers1
saavedra
sengupta1
smallbone0
smallbone1
smallbone10
smallbone11
smallbone12
smallbone13
smallbone14
smallbone15
smallbone16
smallbone17
smallbone18
smallbone19
smallbone2
smallbone3
smallbone4
smallbone5
smallbone6
smallbone7
smallbone8
smallbone9
tang1
teusink2
uys
valero
vaneunen1
vanheerden1
vanheerden2
vanniekerk1
whillier4
wolf
wolf1

```

or you could get the models for a specific reaction for example, all models involving ‘pfk’ would be:

```

[12]: pfk_models = jws.get_models_for_reaction('pfk')
      for model in pfk_models:
          print(model['slug'])

```

```

achcar1
achcar10
achcar11

```

(continues on next page)

(continued from previous page)

achcar12  
achcar13  
achcar14  
achcar2  
achcar3  
achcar4  
achcar5  
achcar6  
achcar7  
achcar8  
achcar9  
albert1  
chassagnole2  
conant1  
conant2  
dupreez1  
dupreez2  
dupreez3  
dupreez4  
dupreez5  
dupreez6  
dupreez7  
gustavsson1  
gustavsson2  
gustavsson3  
gustavsson4  
gustavsson5  
hald  
kerkhovena  
kerkhovenc  
levering1  
levering2  
machado1  
marinhernandez1  
marinhernandez2  
marinhernandez3  
mosca1  
mulquiney  
mulquiney1  
mulquiney2  
orth1  
penkler1  
penkler2  
penkler2aa  
pritchard1  
ralser1  
smallbone0  
smallbone1  
smallbone10  
smallbone11  
smallbone12  
smallbone13

(continues on next page)

(continued from previous page)

```

smallbone14
smallbone15
smallbone16
smallbone17
smallbone18
smallbone2
smallbone3
smallbone4
smallbone5
smallbone6
smallbone7
smallbone8
smallbone9
vanniekerk1

```

for each of these ids, you can get the manuscript, to find out what the model is all about, or the sbml model itself

```
[13]: manuscript = jws.get_manuscript('wolf')
```

```
[14]: print(manuscript['title'])
```

```

Transduction of intracellular and intercellular dynamics in yeast glycolytic
↪oscillations.

```

```
[15]: print(manuscript['abstract'])
```

```

Under certain well-defined conditions, a population of yeast cells exhibits glycolytic
↪oscillations that synchronize through intercellular acetaldehyde. This implies that
↪the dynamic phenomenon of the oscillation propagates within and between cells. We here
↪develop a method to establish by which route dynamics propagate through a biological
↪reaction network. Application of the method to yeast demonstrates how the oscillations
↪and the synchronization signal can be transduced. That transduction is not so much
↪through the backbone of glycolysis, as via the Gibbs energy and redox coenzyme couples
↪(ATP/ADP, and NADH/NAD), and via both intra- and intercellular acetaldehyde.

```

```
[16]: sbml = jws.get_sbml_model('wolf')
```

```
[17]: print(sbml[:1000])
```

```

<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1">
  <model id="wolf" name="wolf">
    <listOfCompartments>
      <compartment metaid="metaid_0" sboTerm="SBO:0000410" id="default_compartment"
↪spatialDimensions="3" size="1" constant="true"/>
    </listOfCompartments>
    <listOfSpecies>
      <species metaid="metaid_1" id="at" name="ATP" compartment="default_compartment"
↪initialConcentration="2" hasOnlySubstanceUnits="false" boundaryCondition="false"
↪constant="false">
        <annotation>
          <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:dc=
↪"http://purl.org/dc/elements/1.1/" xmlns:dcterms="http://purl.org/dc/terms/" xmlns:

```

(continues on next page)

(continued from previous page)

```
↪vCard="http://www.w3.org/2001/vcard-rdf/3.0#" xmlns:bqbiol="http://biomodels.net/  
↪biology-qualifiers/" xmlns:bqmodel="http://biomodels.net/model-qualifiers/">  
  <rdf:Description rdf:about="#metaid_1">  
    <bqbiol:is>  
      <rdf:Bag>
```

[ ]:

[ ]:



## CREATING AND SIMULATING A SIMPLE MODEL

Here we show how to create a basic model using basiCO, and simulating it. We start as usual by importing basiCO.

```
[1]: import sys
if '../..' not in sys.path:
    sys.path.append('../..')
try:
    from basico import *
except ImportError:
    !pip install -q copasi-basico
    from basico import *
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Now lets create a new model, passing along the name that we want to give it. Additional supported parameters for the model consist of:

- `quantity_unit`: which sets the unit to use for species concentrations (defaults to mol)
- `volume_unit`: the unit to use for three dimensional compartments (defaults to litre (l))
- `time_unit`: the unit to use for time (defaults to second (s))
- `area_unit`: the unit to use for two dimensional compartments
- `length_unit`: the unit to use for one dimensional compartments

```
[2]: new_model(name='Simple Model');
```

now we add a basic reaction that converts a chemical species A irreversibly into B. We can do that by just calling `addReaction` with the chemical formula to use. In this case this would be: `A -> B`. The reaction will be automatically created using mass action kinetics.

```
[3]: add_reaction('R1', 'A -> B');
```

Since we had a new model, this created the Species A and B as well as a compartment `compartment`, in which those chemicals reside. The species have an initial concentration of 1. To verify we can call `get_species`, which returns a dataframe with all information about the species (either all species, or the one filtered to):

```
[4]: get_species().initial_concentration
```

```
[4]: name
A      1.0
```

(continues on next page)

(continued from previous page)

```
B      1.0
Name: initial_concentration, dtype: float64
```

to change the initial concentration, we use `set_species`, and specify which property we want to change:

```
[5]: set_species('B', initial_concentration=0)
     set_species('A', initial_concentration=10)
     get_species().initial_concentration

[5]: name
     A      10.0
     B       0.0
     Name: initial_concentration, dtype: float64
```

to see the kinetic parameters of our reaction we can use `get_reaction_parameters`, and we see that the parameter has been created by default with a value of 0.1

```
[6]: get_reaction_parameters()

[6]:      value reaction  type mapped_to
     name
(R1).k1    0.1        R1    local
```

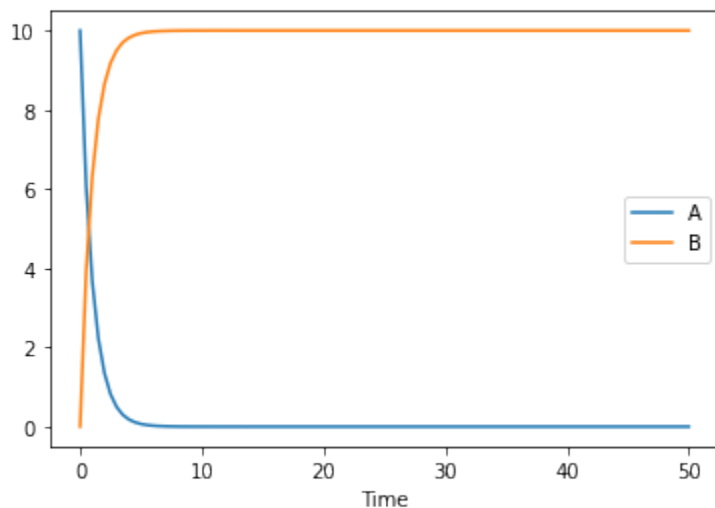
to change that parameter, we use `set_reaction_parameters`, specifying the value to be changed:

```
[7]: set_reaction_parameters('(R1).k1', value=1)
     get_reaction_parameters('k1')

[7]:      value reaction  type mapped_to
     name
(R1).k1    1.0        R1    local
```

now lets simulate our model for 10 seconds:

```
[8]: result = run_time_course(duration=50)
     result.plot();
```

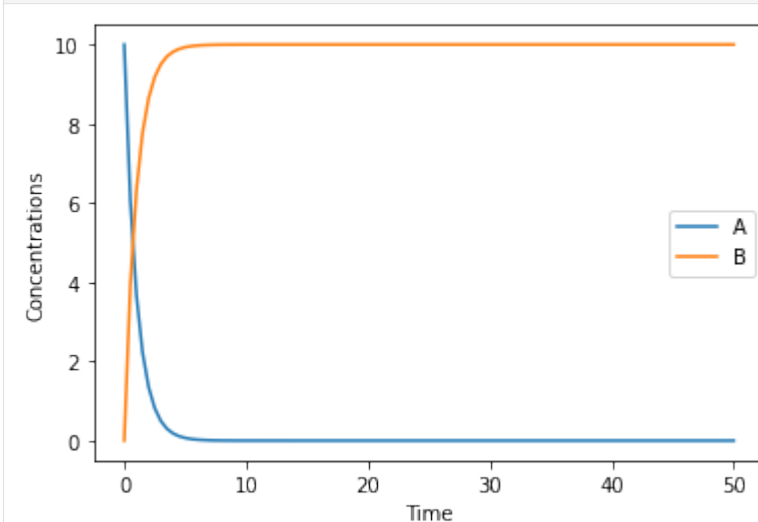


this basic plot is done by pandas, and returns a `matplotlib.axes.AxesSubplot` object, that can be used to modify



it further (full documentation is in the [pandas documentation](#)), if you wanted to change the Y label for example you could run:

```
[9]: ax = result.plot();
     ax.set_ylabel('Concentrations');
```



to simulate the model stochastically, you can specify the simulation method. COPASI supports many different simulations methods:

- **deterministic**: using the COPASI LSODA implementation
- **stochastic**: using the Gibson Bruck algorithm
- **directMethod**: using the Gillespie direct method

others are:

- **tauleap**, **adaptivesa**, **radau5**, **hybridlsoda**, **hybridode45**

So lets try and simulate the model stochastically:

```
[10]: result = run_time_course(duration=50, method='stochastic')
```

```
Error while initializing the simulation: >ERROR 2023-08-10T13:20:07<
At least one particle number in the initial state is too big.
```

simulation failed in this time because the particle numbers, that the stochastic simulation is based upon is too high!  
Lets check:

```
[11]: get_species().initial_particle_number
```

```
[11]: name
A      6.022141e+24
B      0.000000e+00
Name: initial_particle_number, dtype: float64
```

so we just set the initial particle number of a to be smaller, and run the simulation again, this time returning particle numbers rather than concentrations for the resulting dataframe

```
[12]: set_species('A', initial_particle_number=100)
```

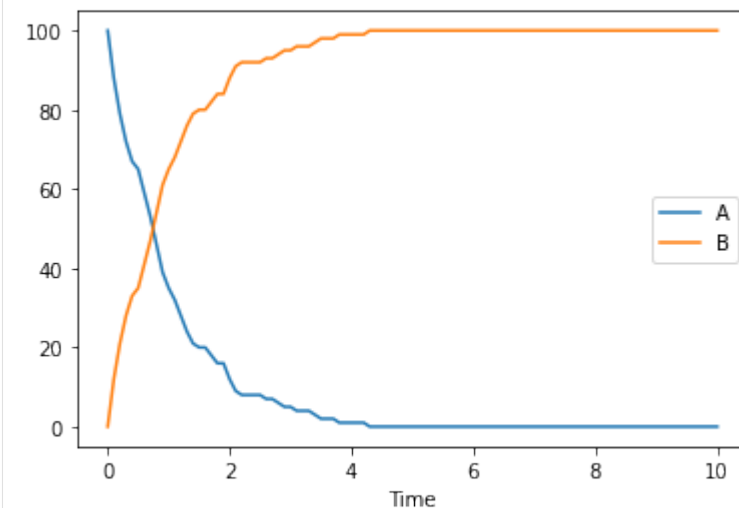
Alternatively we could have modified the models quantity unit, which currently was set to:

```
[13]: get_model_units()
[13]: {'time_unit': 's',
      'quantity_unit': 'mol',
      'length_unit': 'm',
      'area_unit': 'm²',
      'volume_unit': 'l'}
```

So initially we had a concentration of 10 mol/l, which does not lend itself for stochastic simulation. Using the `set_model_unit` command with a more appropriate `quantity_unit` and `volume_unit` would be the proper solution.

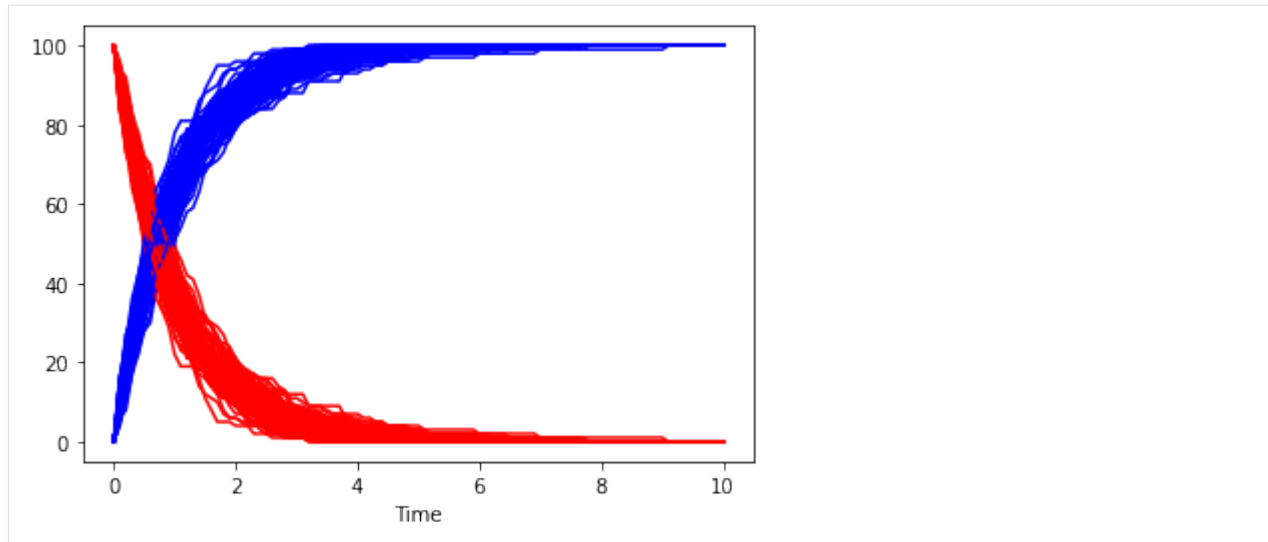
When running stochastic simulations, you might want to specify the seed to be used, so that traces become reproducible. In COPASI you have two parameters for that seed, the actual seed, and `use_seed` a boolean indicating whether that seed is to be used for the next simulation. For a single trace we use both:

```
[14]: result = run_time_course(duration=10, method='stochastic', use_numbers=True, seed=1234,
      ↪ use_seed=True)
      result.plot();
```



of course one stochastic trace will not be enough, so let's run many of them. This time the species will be plotted separately, so that it is easy to reuse the same color. This time we also don't use the seed specified before.

```
[15]: fig, ax = plt.subplots()
      for i in range(100):
          result = run_time_course(duration=10, method='stochastic', use_numbers=True, use_
          ↪ seed=False)
          result.plot(y='A', color='r', ax=ax, legend=None);
          result.plot(y='B', color='b', ax=ax, legend=None);
```



so far, we were only using mass action kinetics, but of course we could use any other kinetic as well. COPASI comes with a large number of functions already inbuilt. You can see those, running the `get_functions` command. It is filterable by name, and whether or not the formula is reversible, or general (general reactions can be used for either reversibility). Since we modelled our reaction as irreversible, let's look at the irreversible functions we have:

```
[16]: get_functions(reversible=False)
```

```
[16]:
```

name	reversible	formula
Allosteric inhibition (MWC)	False	$V \cdot \frac{\text{substrate}/K_s \cdot (1 + (\text{substrate}/K_s))^{(n-1)}}{L \cdot (\dots)}$
Catalytic activation (irrev)	False	$V \cdot \text{substrate} \cdot \text{Activator} / ((K_m + \text{substrate}) \cdot (K_a + \text{Act} \dots))$
Competitive inhibition (irr)	False	$V \cdot \text{substrate} / (K_m + \text{substrate} + K_m \cdot \text{Inhibitor} / K_i)$
Constant flux (irreversible)	False	$v$
Henri-Michaelis-Menten (irreversible)	False	$V \cdot \text{substrate} / (K_m + \text{substrate})$

(continues on next page)

(continued from previous page)

Hill Cooperativity	$V * (\text{substrate} / \text{Shalve})^h / (1 + (\text{substrate} / \text{Shalve})^h)$
Hyperbolic modifier (irrev)	$V * \text{substrate} * (1 + b * \text{Modifier} / (a * Kd)) / (K_m * (1 + \text{Modif}...$
Mass action (irreversible)	$k1 * \text{PRODUCT} < \text{substrate}_i >$
Mixed activation (irrev)	$V * \text{substrate} * \text{Activator} / (K_m * (K_a + \text{Activator}) + \text{sub}...$
Mixed inhibition (irr)	$V * \text{substrate} / (K_m * (1 + \text{Inhibitor} / K_i) + \text{substrate} * (1...$
Noncompetitive inhibition (irr)	$V * \text{substrate} / ((K_m + \text{substrate}) * (1 + \text{Inhibitor} / K_i))$
Specific activation (irrev)	$V * \text{substrate} * \text{Activator} / (K_m * K_a + (K_m + \text{substrate}) * ...$
Substrate activation (irr)	$V * (\text{substrate} / K_s a)^2 / (1 + \text{substrate} / K_s c + \text{substrate}...$
Substrate inhibition (irr)	$V * \text{substrate} / (K_m + \text{substrate} + K_m * (\text{substrate} / K_i)^2)$
Uncompetitive inhibition (irr)	$V * \text{substrate} / (K_m + \text{substrate} * (1 + \text{Inhibitor} / K_i))$
Bi (irreversible)	$v_{\max} * A * B / (K_m a * K_m b + A * K_m b + B * K_m a + A * B)$
Arrhenius Bimolecular (irr)	$A * \exp(-E_a / (R * T)) * S1 * S2$
Arrhenius Unimolecular (irr)	$A * \exp(-E_a / (R * T)) * S$

general \

name	
Allosteric inhibition (MWC)	False
Catalytic activation (irrev)	False
Competitive inhibition (irr)	False
Constant flux (irreversible)	False
Henri-Michaelis-Menten (irreversible)	False
Hill Cooperativity	False
Hyperbolic modifier (irrev)	False
Mass action (irreversible)	False
Mixed activation (irrev)	False
Mixed inhibition (irr)	False
Noncompetitive inhibition (irr)	False
Specific activation (irrev)	False
Substrate activation (irr)	False
Substrate inhibition (irr)	False
Uncompetitive inhibition (irr)	False
Bi (irreversible)	False
Arrhenius Bimolecular (irr)	False
Arrhenius Unimolecular (irr)	False

mapping

name	
Allosteric inhibition (MWC)	<code>{'substrate': 'substrate', 'Inhibitor': 'modif...</code>
Catalytic activation (irrev)	<code>{'substrate': 'substrate', 'Activator': 'modif...</code>
Competitive inhibition (irr)	<code>{'substrate': 'substrate', 'Inhibitor': 'modif...</code>
Constant flux (irreversible)	<code>{'v': 'parameter'}</code>
Henri-Michaelis-Menten (irreversible)	<code>{'substrate': 'substrate', 'Km': 'parameter', ...</code>
Hill Cooperativity	<code>{'substrate': 'substrate', 'Shalve': 'paramete...</code>
Hyperbolic modifier (irrev)	<code>{'substrate': 'substrate', 'Modifier': 'modifi...</code>
Mass action (irreversible)	<code>{'k1': 'parameter', 'substrate': 'substrate'}</code>
Mixed activation (irrev)	<code>{'substrate': 'substrate', 'Activator': 'modif...</code>
Mixed inhibition (irr)	<code>{'substrate': 'substrate', 'Inhibitor': 'modif...</code>
Noncompetitive inhibition (irr)	<code>{'substrate': 'substrate', 'Inhibitor': 'modif...</code>
Specific activation (irrev)	<code>{'substrate': 'substrate', 'Activator': 'modif...</code>
Substrate activation (irr)	<code>{'substrate': 'substrate', 'V': 'parameter', '...</code>
Substrate inhibition (irr)	<code>{'substrate': 'substrate', 'Km': 'parameter', ...</code>
Uncompetitive inhibition (irr)	<code>{'substrate': 'substrate', 'Inhibitor': 'modif...</code>

(continues on next page)

(continued from previous page)

```

Bi (irreversible)                {'vmax': 'parameter', 'A': 'substrate', 'B': '...
Arrhenius Bimolecular (irr)      {'A': 'parameter', 'Ea': 'parameter', 'R': 'pa...
Arrhenius Unimolecular (irr)    {'A': 'parameter', 'Ea': 'parameter', 'R': 'pa...

```

So lets change the kinetic function that our reaction should use. Here we simply specify the function name that we got from the call before. This will introduce new local parameters, for Km and Vmax to the model. (We of course could have used the function parameter already add the add\_reaction command above.

```
[17]: set_reaction('R1', function='Henri-Michaelis-Menten (irreversible)')
```

```
[18]: get_reactions()
```

```

[18]:      scheme flux particle_flux      function \
name
R1    A -> B   0.0              0.0  Henri-Michaelis-Menten (irreversible)

      key sbml_id      mapping
name
R1    Reaction_0      {'substrate': 'A', 'Km': 0.1, 'V': 0.1}

```

```
[19]: get_reaction_parameters()
```

```

[19]:      value reaction  type mapped_to
name
(R1).Km    0.1        R1  local
(R1).V     0.1        R1  local

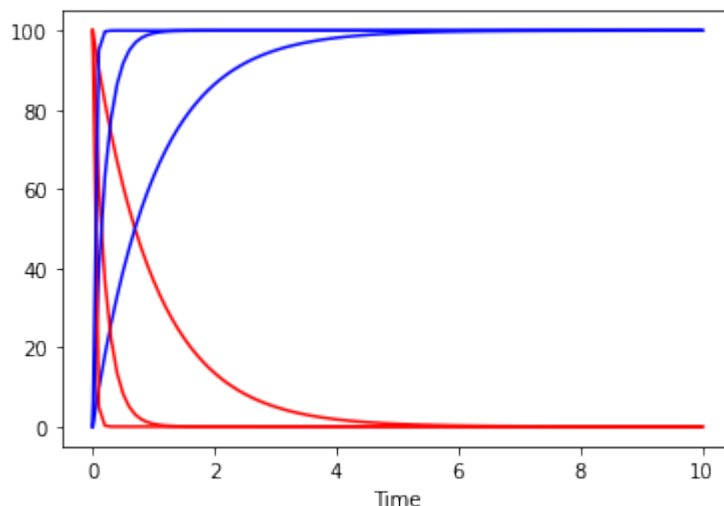
```

and now we can look at how the plot would look at repeating the simulation for several vmax values:

```

[20]: fig, ax = plt.subplots()
      for vm in [0.1, 0.5, 3]:
          set_reaction_parameters('(R1).V', value=vm)
          result = run_time_course(duration=10, method='deterministic', use_numbers=True)
          result.plot(y='A', color='r', ax=ax, legend=None);
          result.plot(y='B', color='b', ax=ax, legend=None);

```





## GETTING AND SETTING REACTION PARAMETERS AND REACTIONS

```
[1]: import sys
     sys.path.append('../..')
     %matplotlib inline
```

```
[2]: from basico import *
```

### 4.1 Load model

```
[3]: biomod = load_example("LM-test1")
```

### 4.2 Getting/setting global quantities

It is possible to set:

- 'initial\_value'
- 'initial\_expression'
- 'expression'
- 'status'
- 'type'

```
[4]: get_parameters()
```

```
[4]:      type unit  initial_value initial_expression \
name
epsilon      fixed           0.78
offset       fixed           0.10
signal  assignment           0.10

      expression  value  rate      key
name
epsilon              NaN   0.0  ModelValue_0
offset              NaN   0.0  ModelValue_1
signal  [P] * Values[epsilon] + Values[offset]  NaN   NaN  ModelValue_2
```

```
[5]: set_parameters(name= 'offset', initial_value = 50)
```

```
[6]: set_parameters('signal', expression='[P] * Values[epsilon] + Values[offset]')
```

```
[7]: get_parameters()
```

```
[7]:
```

	name	type	unit	initial_value	initial_expression	\
	epsilon	fixed		0.78		
	offset	fixed		50.00		
	signal	assignment		50.00		

	name	expression	value	rate	key
	epsilon		NaN	0.0	ModelValue_0
	offset		NaN	0.0	ModelValue_1
	signal	[P] * Values[epsilon] + Values[offset]	NaN	NaN	ModelValue_2

### 4.3 Getting/setting species (metabolites)

```
[8]: get_species()
```

```
[8]:
```

	name	compartment	type	unit	initial_concentration	\
	E	compartment	reactions	mmol/ml	0.010000	
	S	compartment	reactions	mmol/ml	10.000001	
	ES	compartment	reactions	mmol/ml	0.000000	
	P	compartment	reactions	mmol/ml	0.000000	

	name	initial_particle_number	initial_expression	expression	concentration	\
	E	6.022142e+18			NaN	
	S	6.022142e+21			NaN	
	ES	0.000000e+00			NaN	
	P	0.000000e+00			NaN	

	name	particle_number	rate	particle_number_rate	key
	E	NaN	NaN	NaN	Metabolite_1
	S	NaN	NaN	NaN	Metabolite_0
	ES	NaN	NaN	NaN	Metabolite_2
	P	NaN	NaN	NaN	Metabolite_3

```
[9]: get_species(name = 'E')['initial_concentration']
```

```
[9]: name
E      0.01
ES     0.00
Name: initial_concentration, dtype: float64
```



### 4.3.1 Setting

```

if 'name' in kwargs:
    metab.setObjectName(kwargs['name'])

if 'unit' in kwargs:
    metab.setUnitExpression(kwargs['unit'])

if 'initial_concentration' in kwargs:
    metab.setInitialConcentration(kwargs['initial_concentration']),

if 'initial_particle_number' in kwargs:
    metab.setInitialValue(kwargs['initial_particle_number']),

if 'initial_expression' in kwargs:
    metab.setInitialExpression(kwargs['initial_expression'])

if 'expression' in kwargs:
    metab.setExpression(kwargs['expression'])

```

```
[10]: set_species(name = 'E', new_name = 'Lilija')
```

```
[11]: set_species(name = 'Lilija', initial_concentration = 123456)
```

```
[12]: get_species(name = 'Lilija')
```

```
[12]:
```

	compartment	type	unit	initial_concentration	\
name					
Lilija	compartment	reactions	mmol/ml	123456.0	

```

initial_particle_number initial_expression expression concentration \
name
Lilija 7.434694e+25 NaN

```

```

particle_number rate particle_number_rate key
name
Lilija NaN NaN NaN Metabolite_1

```

```
[13]: # change it back
set_species(new_name = 'E', name = 'Lilija')
```

## 4.4 Getting/setting reaction parameters

```
[14]: get_reaction_parameters()
```

```
[14]:
```

	value	reaction	type	mapped_to
name				
(R1).k1	130.0	R1	local	
(R1).k2	1.0	R1	local	
(R2).k1	1.0	R2	local	

```
Setting only the following is possible:
'new_name'
'value'
```

```
[15]: set_reaction_parameters(name = '(R1).k1', value=123)
```

```
[16]: get_reaction_parameters()
```

```
[16]:
```

	value	reaction	type	mapped_to
name				
(R1).k1	123.0	R1	local	
(R1).k2	1.0	R1	local	
(R2).k1	1.0	R2	local	

## 4.5 Getting/setting reactions

```
[17]: get_reactions()
```

```
[17]:
```

	scheme	flux	particle_flux	function
name				
R1	S + E = ES	NaN	NaN	Mass action (reversible)
R2	ES -> E + P	NaN	NaN	Mass action (irreversible)

## 4.6 Setting scheme

```
[18]: set_reaction(name = 'R1', scheme = 'S + E + F = ES')
```

```
[19]: get_reactions()
```

```
[19]:
```

	scheme	flux	particle_flux	function
name				
R1	S + E + F = ES	NaN	NaN	Mass action (reversible)
R2	ES -> E + P	NaN	NaN	Mass action (irreversible)

## 4.7 Setting reaction name

```
[20]: set_reaction(name = 'R1', new_name = 'Reaction 1')
```

```
[21]: get_reactions()
```

```
[21]:
```

	scheme	flux	particle_flux	function
name				
Reaction 1	S + E + F = ES	NaN	NaN	Mass action (reversible)
R2	ES -> E + P	NaN	NaN	Mass action (irreversible)

[ ]:



## EDITING REACTION KINETICS

Previous examples showed how to create models using `basico`, and using kinetic functions from the reaction database. Here, I want to expand on that showing how to map kinetic functions to reactions involving modifiers as well. Lets start as usual by importing `basico`:

```
[1]: from basico import *
```

now lets create a new model:

```
[2]: new_model(name='Reactions');
```

we know we can create a reaction, by using the `add_reaction` command. It requires at the very least two arguments:

- `name`: the name of the reaction
- `scheme`: the reaction scheme

If nothing else is specified, this will create the reaction with the given name and reaction scheme, assigning it mass action kinetics with local parameters defaulting to a value of 0.1. All species will be created if they do not exist yet in the model. For example:

```
[3]: add_reaction('R1', 'A -> B');
```

creates the reaction R1, species A and B and a local parameter (R1).k1. With `get_reactions` we can have a look at what was created:

```
[4]: get_reactions()[['scheme', 'function', 'mapping']]
```

```
[4]:      scheme      function      mapping
name
R1    A -> B  Mass action (irreversible)  {'k1': 0.1, 'substrate': 'A'}
```

here I want to point out, the `mapping` column. It shows that the parameter `k1` is a local one, as it is mapped to a value. And that the substrate of the function is mapped to A. We can specify the mapping directly in the `add_reaction` call, or we can specify it using `set_reaction`. So for example, if we wanted to modify the reaction, to map the reactions `k1` parameter to a global quantity `global_k`, we could to that as follows:

```
[5]: add_parameter(name='global_k', initial_value=0.2)
      set_reaction('R1', mapping={'k1': 'global_k'})
```

```
[6]: get_reactions()[['scheme', 'function', 'mapping']]
```

```
[6]:      scheme      function      mapping
name
R1    A -> B  Mass action (irreversible)  {'k1': 'global_k', 'substrate': 'A'}
```

next let us assume, i wanted to use a kinetic from the function database, that includes inhibition for the reaction. Using `get_functions` we can filter the functiondatabase, for suitable functions for our reaction, and then filter for ones that contain inhibition:

```
[7]: suitable_functions = get_functions(suitable_for='R1')[['formula', 'mapping']]
suitable_inhibitions = suitable_functions[suitable_functions.index.str.contains(
↳ 'inhibition')]
suitable_inhibitions
```

	formula \		mapping
name			
Allosteric inhibition (MWC)	$V * (\text{substrate} / K_s) * (1 + (\text{substrate} / K_s))^{(n-1)} / (L * (...$		
Competitive inhibition (irr)	$V * \text{substrate} / (K_m + \text{substrate} + K_m * \text{Inhibitor} / K_i)$		
Mixed inhibition (irr)	$V * \text{substrate} / (K_m * (1 + \text{Inhibitor} / K_i) + \text{substrate} * (1 + ...$		
Noncompetitive inhibition (irr)	$V * \text{substrate} / ((K_m + \text{substrate}) * (1 + \text{Inhibitor} / K_i))$		
Substrate inhibition (irr)	$V * \text{substrate} / (K_m + \text{substrate} + K_m * (\text{substrate} / K_i)^2)$		
Uncompetitive inhibition (irr)	$V * \text{substrate} / (K_m + \text{substrate} * (1 + \text{Inhibitor} / K_i))$		
name			
Allosteric inhibition (MWC)		{'substrate': 'substrate', 'Inhibitor': 'modif...	
Competitive inhibition (irr)		{'substrate': 'substrate', 'Inhibitor': 'modif...	
Mixed inhibition (irr)		{'substrate': 'substrate', 'Inhibitor': 'modif...	
Noncompetitive inhibition (irr)		{'substrate': 'substrate', 'Inhibitor': 'modif...	
Substrate inhibition (irr)		{'substrate': 'substrate', 'Km': 'parameter', ...	
Uncompetitive inhibition (irr)		{'substrate': 'substrate', 'Inhibitor': 'modif...	

let us use Allosteric inhibition (MWC) here, lets have a look at the formula and the mapping table:

```
[8]: as_dict(suitable_inhibitions)[0]
[8]: {'name': 'Allosteric inhibition (MWC)',
      'formula': 'V*(substrate/Ks)*(1+(substrate/Ks))^(n-1)/(L*(1+Inhibitor/Ki)^
↳ n+(1+(substrate/Ks))^n)',
      'mapping': {'substrate': 'substrate',
                  'Inhibitor': 'modifier',
                  'V': 'parameter',
                  'Ks': 'parameter',
                  'n': 'parameter',
                  'L': 'parameter',
                  'Ki': 'parameter'}}
```

since this function requires a modifier, we also change the reaction scheme to include a modifier. This is done by adding a semicolon at the end of the reaction scheme, and listing the modifiers space separated there. Then we an assign that function directly.

```
[9]: set_reaction('R1', scheme='A -> B; C', function='Allosteric inhibition (MWC)')
get_reactions()[['scheme', 'function', 'mapping']]
[9]:
```

	scheme	function \		mapping
name				
R1	A -> B; C	Allosteric inhibition (MWC)		
name				
R1			{'substrate': 'A', 'Inhibitor': 'C', 'V': 0.1,...	

*Note:* that here, the mapping is not necessary, as the function has only one modifier, had we multiple modifiers defined, then we'd want to specify the mapping dictionary and map the `Inhibitor` to the respective modifier in our reaction scheme:

```
[10]: set_reaction('R1', scheme='A -> B; C D', function='Allosteric inhibition (MWC)', mapping=
      ↪{'Inhibitor': 'D'})
      get_reactions()[['scheme', 'function', 'mapping']]
```

```
[10]:          scheme          function \
name
R1      A -> B; D  Allosteric inhibition (MWC)

          mapping
name
R1      {'substrate': 'A', 'Inhibitor': 'D', 'V': 0.1,...
```

*Note:* Assigning a function that uses modifiers, *requires* that modifiers are present in the reaction scheme, or that *all* modifiers are specified in the mapping parameter. So assigning the function above would fail with error, if no modifier is declared:

```
[11]: add_reaction('error', scheme='A -> B', function='Allosteric inhibition (MWC)');
      get_reactions()[['scheme', 'function', 'mapping']]
```

```
ERROR:root:the mapping for reaction "error" with function "Allosteric inhibition (MWC)"
↪is not valid and cannot be applied.
```

```
[11]:          scheme          function \
name
R1      A -> B; D  Allosteric inhibition (MWC)
error    A -> B   Mass action (irreversible)

          mapping
name
R1      {'substrate': 'A', 'Inhibitor': 'D', 'V': 0.1, ...
error    {'k1': 0.1, 'substrate': 'A'}
```

However, it will succeed, if the modifier is specified. in the mapping parameter:

```
[12]: set_reaction('error', new_name='now_it_works', function='Allosteric inhibition (MWC)',
      ↪mapping={'Inhibitor': 'D'})
      get_reactions()[['scheme', 'function', 'mapping']]
```

```
[12]:          scheme          function \
name
R1      A -> B; D  Allosteric inhibition (MWC)
now_it_works A -> B; D  Allosteric inhibition (MWC)

          mapping
name
R1      {'substrate': 'A', 'Inhibitor': 'D', 'V': 0.1, ...
now_it_works {'substrate': 'A', 'Inhibitor': 'D', 'V': 0.1, ...
```





## PARAMETER ESTIMATION

This document describes how to use BasiCO for parameter estimation tasks. This document assumes, that the parameter estimation task was already set up using COPASI. Look for another example, to set up a parameter estimation task directly from basiCO.

We start as normal:

```
[1]: import sys
      if '../..' not in sys.path:
          sys.path.append('../..')

      from basico import *
      %matplotlib inline
```

Now we load an example model, that already has parameter estimation set up:

```
[2]: dm = load_example('LM')
```

with `get_fit_parameters` we can look at the parameters, that will be estimated, along with their bounds, and a list of experiments they apply to. If that list is empty, the parameter applies to all experiment, otherwise only the one mentioned.

```
[3]: get_fit_parameters()
```

```
[3]:
```

	lower	upper	start	affected	\
name					
(R1).k2	1e-6	1e6	1.0		[]
(R2).k1	1e-6	1e6	1.0		[]
Values[offset]	-0.2	0.4	0.1	[Experiment_1]	
Values[offset]	-0.2	0.4	0.1	[Experiment_3]	
Values[offset]	-0.2	0.4	0.1	[Experiment]	
Values[offset]	-0.2	0.4	0.1	[Experiment_4]	
Values[offset]	-0.2	0.4	0.1	[Experiment_2]	
					cn
name					
(R1).k2				CN=Root,Model=Kinetics of a Michaelian enzyme...	
(R2).k1				CN=Root,Model=Kinetics of a Michaelian enzyme...	
Values[offset]				CN=Root,Model=Kinetics of a Michaelian enzyme...	
Values[offset]				CN=Root,Model=Kinetics of a Michaelian enzyme...	
Values[offset]				CN=Root,Model=Kinetics of a Michaelian enzyme...	
Values[offset]				CN=Root,Model=Kinetics of a Michaelian enzyme...	
Values[offset]				CN=Root,Model=Kinetics of a Michaelian enzyme...	

Now lets see how well the current fit is:

```
[4]: run_parameter_estimation(method='Statistics')
```

```
[4]:
```

	lower	upper	sol	affected
name				
(R1).k2	1e-6	1e6	0.0000002	[]
(R2).k1	1e-6	1e6	44.661715	[]
Values[offset]	-0.2	0.4	0.043018	[Experiment_1]
Values[offset]	-0.2	0.4	0.054167	[Experiment_3]
Values[offset]	-0.2	0.4	-0.050941	[Experiment]
Values[offset]	-0.2	0.4	0.045922	[Experiment_4]
Values[offset]	-0.2	0.4	0.048025	[Experiment_2]

if ever you wanted to get the solution of the last run, you can execute `get_parameters_solution`

```
[5]: get_parameters_solution()
```

```
[5]:
```

	lower	upper	sol	affected
name				
(R1).k2	1e-6	1e6	0.0000002	[]
(R2).k1	1e-6	1e6	44.661715	[]
Values[offset]	-0.2	0.4	0.043018	[Experiment_1]
Values[offset]	-0.2	0.4	0.054167	[Experiment_3]
Values[offset]	-0.2	0.4	-0.050941	[Experiment]
Values[offset]	-0.2	0.4	0.045922	[Experiment_4]
Values[offset]	-0.2	0.4	0.048025	[Experiment_2]

the experimental data can be obtained like so:

```
[6]: data = get_experiment_data_from_model()
```

```
[7]: data[1]
```

```
[7]:
```

	[S]_0	Time	Values[signal]
0	0.31623	0.3	0.08568
1	0.31623	0.6	0.12849
2	0.31623	0.9	0.16599
3	0.31623	1.2	0.19788
4	0.31623	1.5	0.22046
..	...	...	...
95	0.31623	28.8	0.27933
96	0.31623	29.1	0.28938
97	0.31623	29.4	0.28208
98	0.31623	29.7	0.25544
99	0.31623	30.0	0.29552

[100 rows x 3 columns]

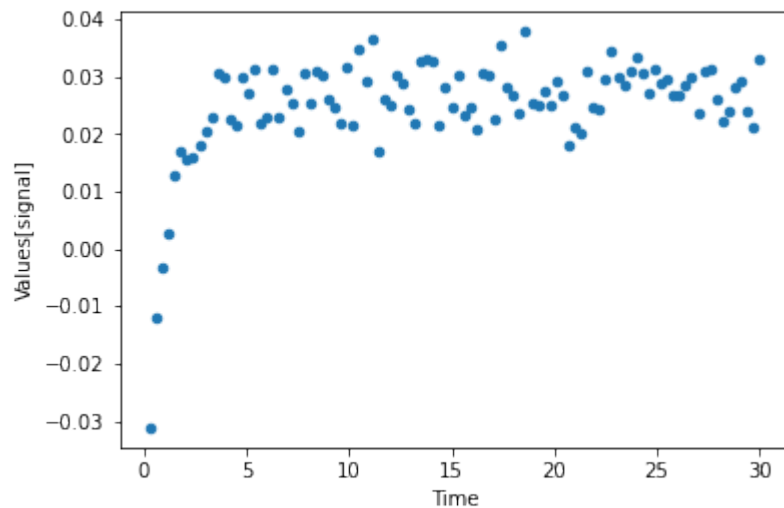
```
[8]: len(data)
```

```
[8]: 5
```

or to get a specific data set you can get it directly from the name:

```
[9]: df = get_data_from_experiment('Experiment')
```

```
[10]: df.plot(kind='scatter', x='Time', y="Values[signal]");
```



you can also look directly at the mapping, of the data. in this `independent` will mean that it is an initial value that is different in this experiment. `dependent` is the mapping to a model variable such as the transient concentration or a transient value of a model parameter.

```
[11]: get_experiment_mapping('Experiment')
```

```
[11]:
```

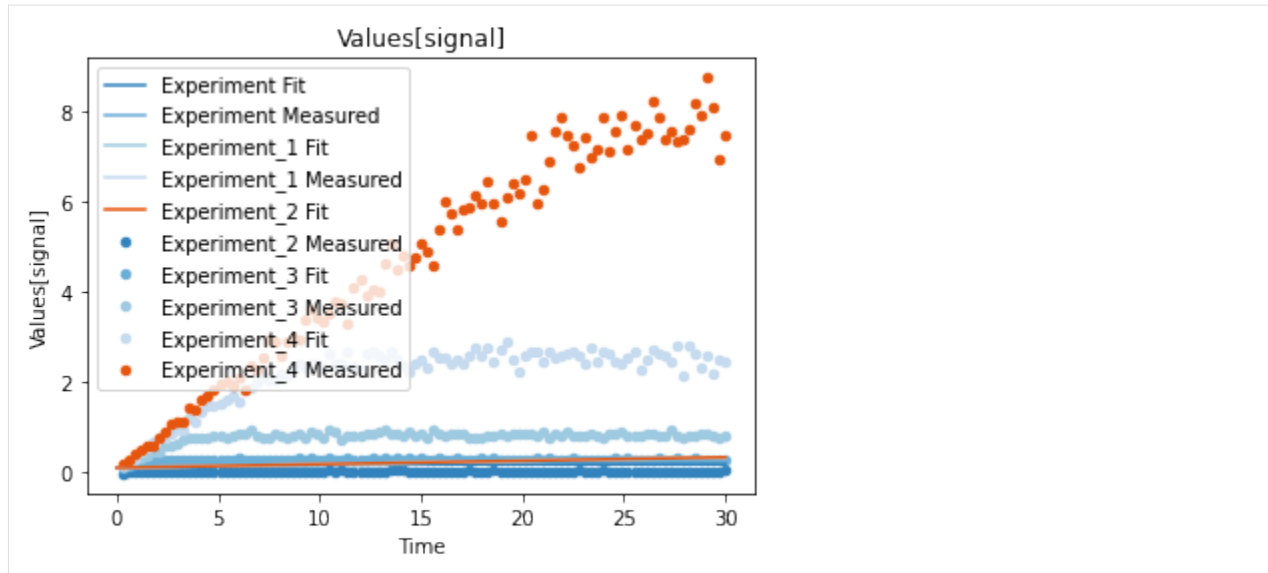
	type	mapping \
column		
0	independent	[S]_0
1	time	
2	dependent	Values[signal]

		cn
column		
0	CN=Root,Model=Kinetics of a	Michaelian enzyme...
1		
2	CN=Root,Model=Kinetics of a	Michaelian enzyme...

you can also plot the current fit, by running:

```
[12]: plot_per_dependent_variable();
```



now lets combine it, by actually running the parameter estimation (here using the Levenberg Marquardt algorithm)

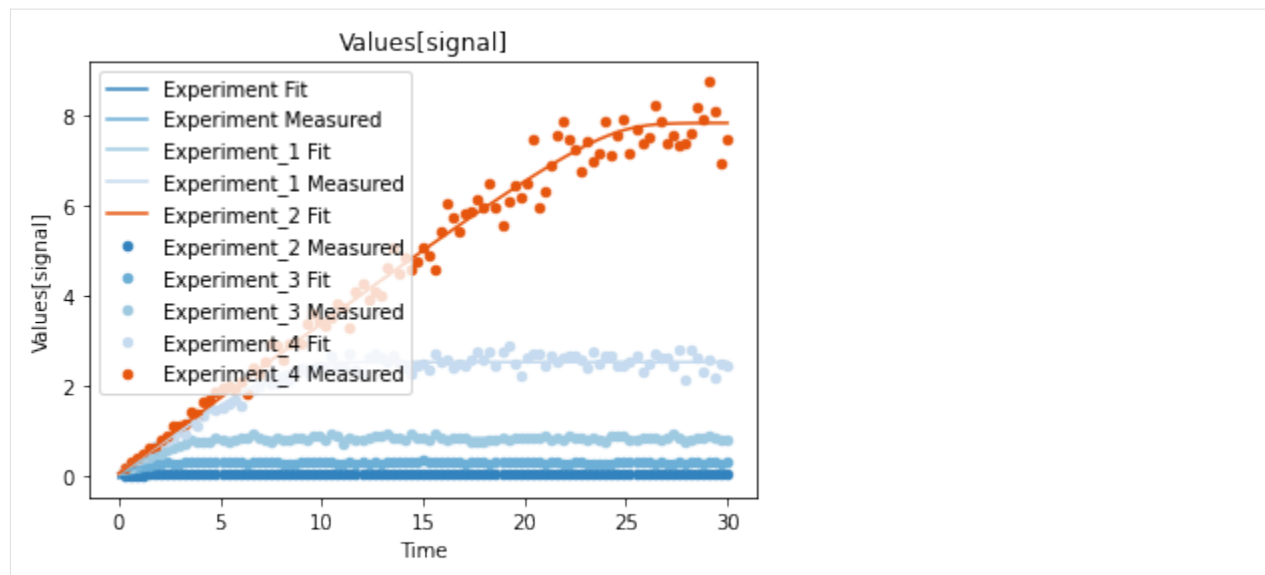
```
[13]: run_parameter_estimation(method='Levenberg - Marquardt', update_model=True)
```

```
[13]:
```

	lower	upper	sol	affected
name				
(R1).k2	1e-6	1e6	0.0000002	[]
(R2).k1	1e-6	1e6	44.729444	[]
Values[offset]	-0.2	0.4	0.043013	[Experiment_1]
Values[offset]	-0.2	0.4	0.054212	[Experiment_3]
Values[offset]	-0.2	0.4	-0.050942	[Experiment]
Values[offset]	-0.2	0.4	0.040968	[Experiment_4]
Values[offset]	-0.2	0.4	0.047976	[Experiment_2]

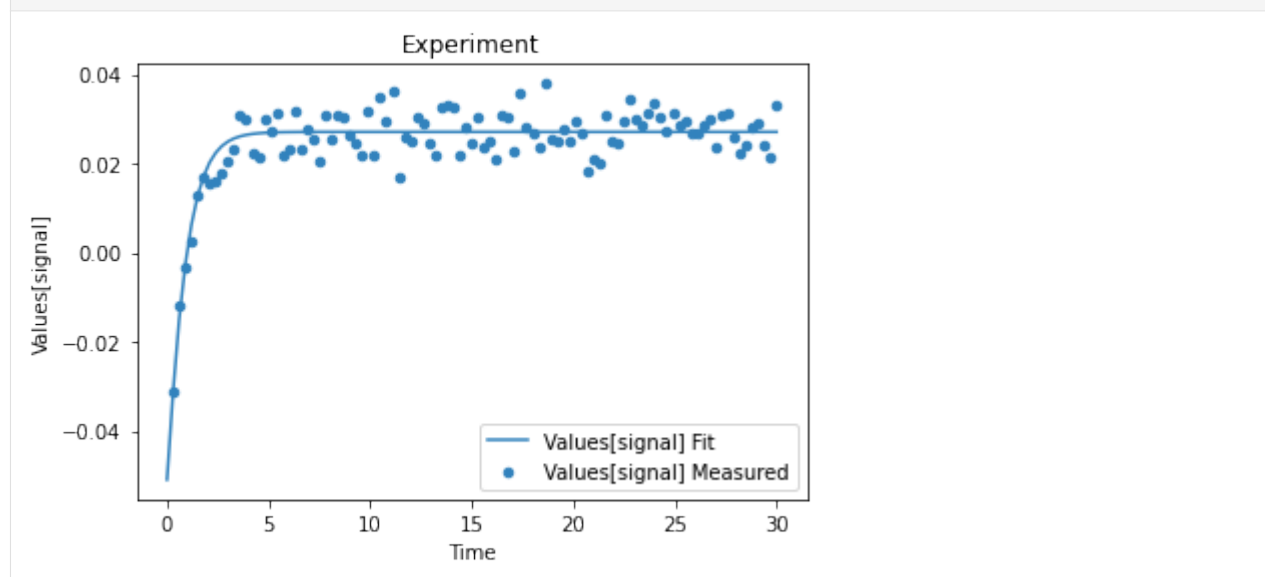
A first predefined plot function creates one plot for each defined dependent variable, containing all experiments in which it appears.

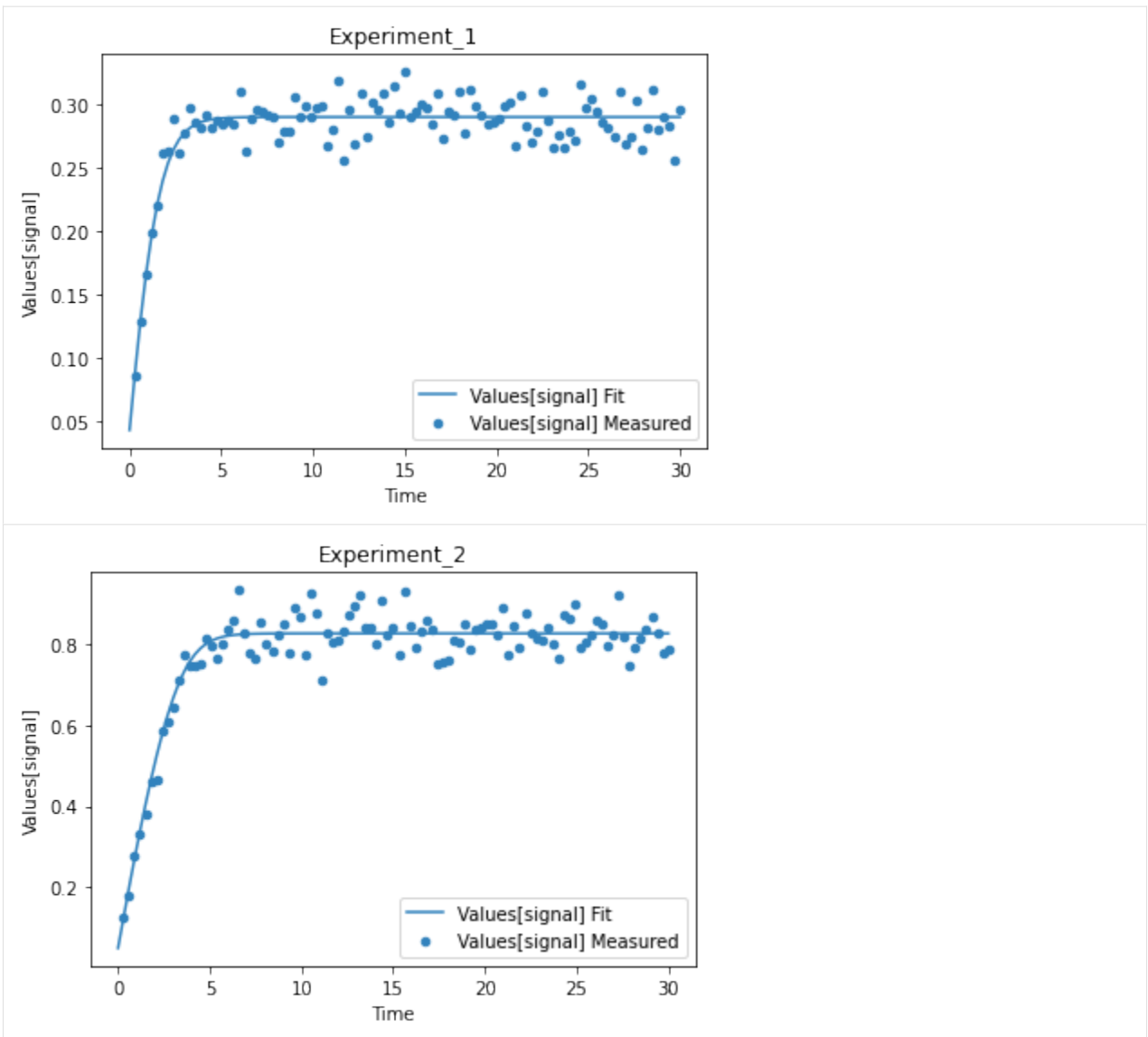
```
[14]: plot_per_dependent_variable();
```

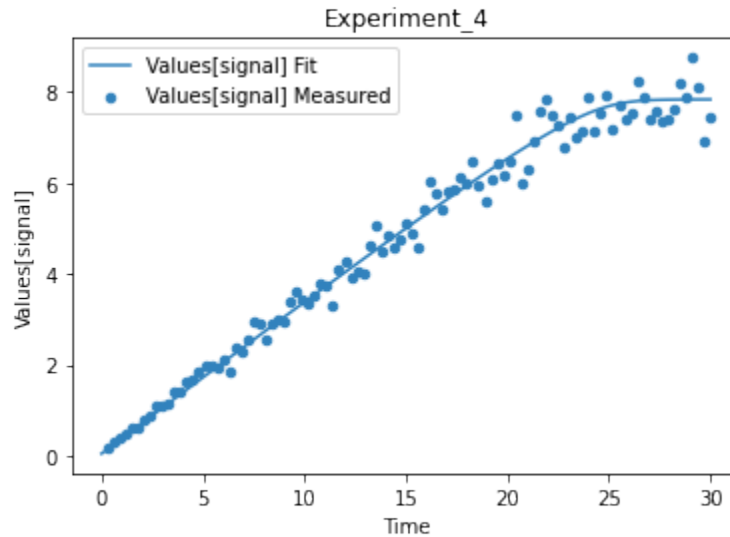
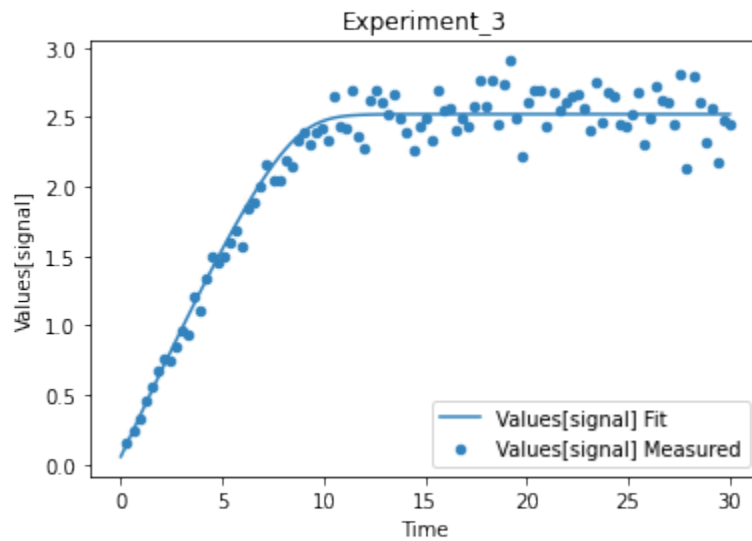


a second predefined plot function created one plot for each experiment, including all contained dependent values:

```
[15]: plot_per_experiment();
```

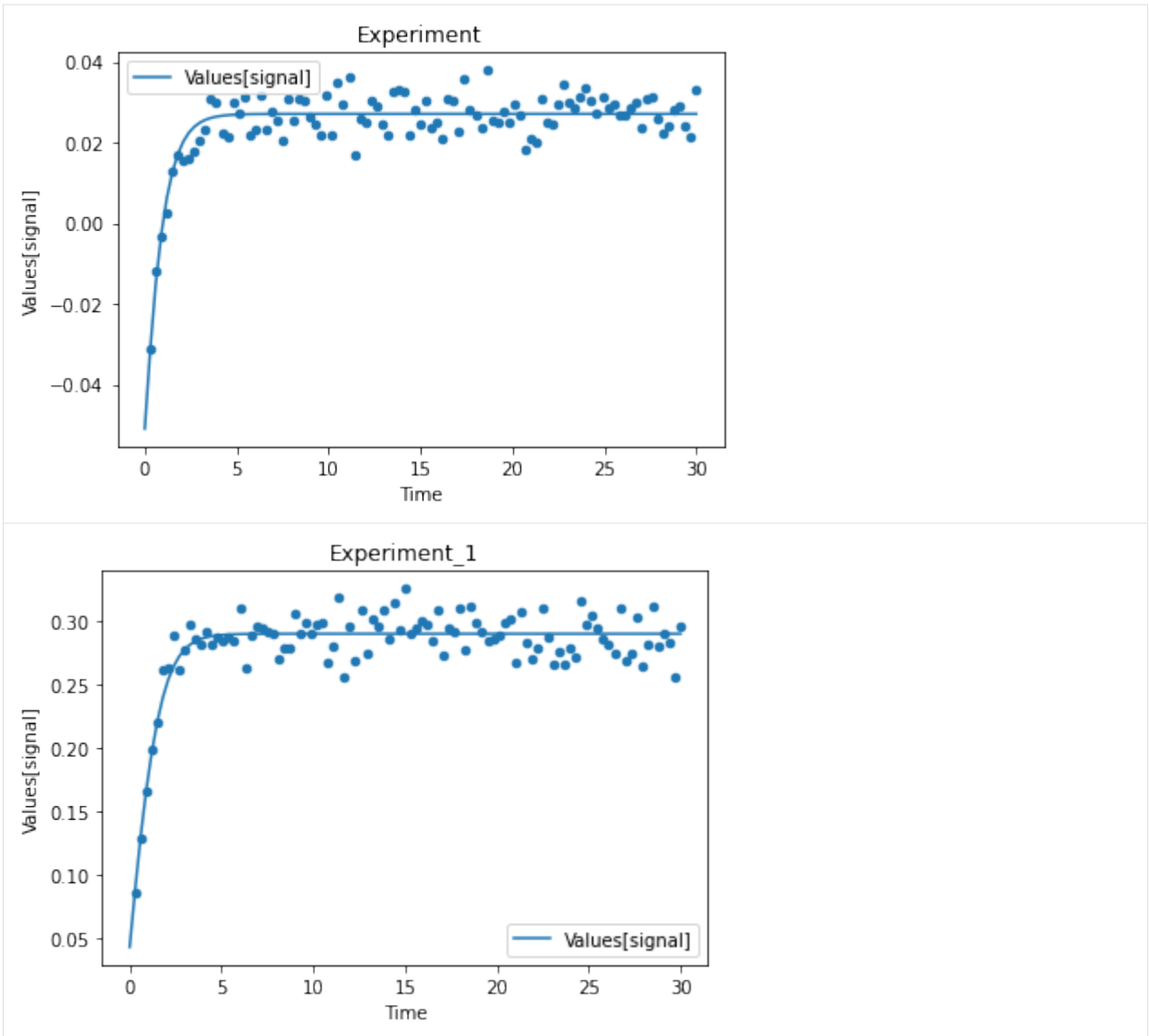




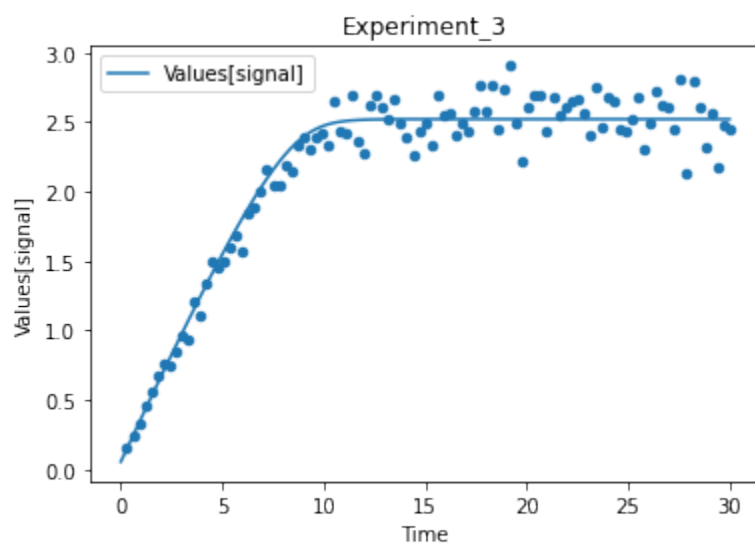
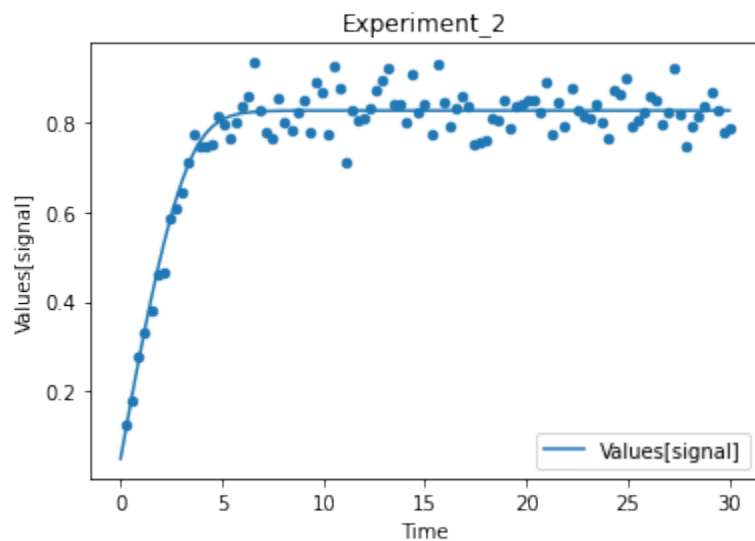


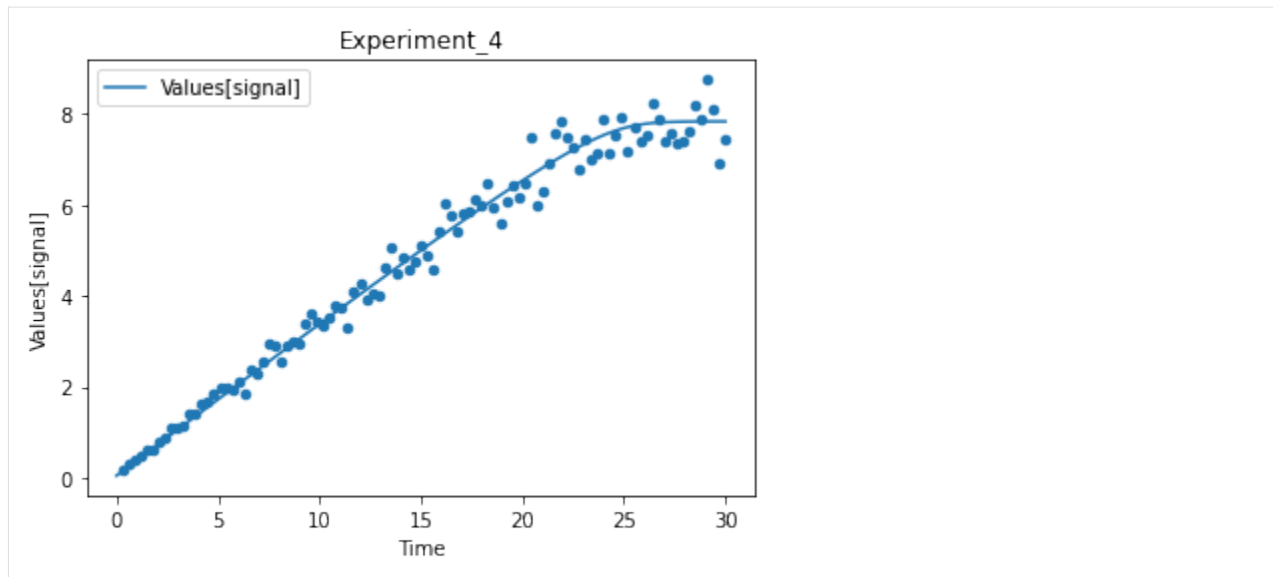
you also get all the data sets returned, so you can plot them yourself however you like

```
[16]: exp, sim = get_simulation_results()
      experiment_names = get_experiment_names()
      for i in range(len(exp)):
          ax = exp[i].plot.scatter(x='Time', y='Values[signal]')
          sim[i].reset_index().plot(x='Time', y='Values[signal]', ax=ax)
          ax.set_title(experiment_names[i])
```









## PARAMETER ESTIMATION SETUP

This example goes through the steps of setting up the parameter estimation task directly from basiCO. Experimental data is provided through pandas dataframes, and mapping of the columns will be done by convention using specially crafted column names.

We start as usual:

```
[1]: import sys
      if '../..' not in sys.path:
          sys.path.append('../..')
      from basico import *
      import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
      %matplotlib inline
```

### 7.1 preparing the data

In this example we generate the data set by modifying the brusselator model, to produce some noisy data, and we will take that data set as a starting point for the parameter estimation later on. So lets start loading the brusselator model, and adding two observable variables that follow the model species with some noise added:

```
[2]: load_example('brusselator')
      add_parameter('obs_x', type='assignment', expression='[X] + UNIFORM(0,1) - 0.5')
      add_parameter('obs_y', type='assignment', expression='[Y] + UNIFORM(0,1) - 0.5');
```

now we simulate the model, ensuring to return all result columns (not just the concentration ones)

```
[3]: result = run_time_course(start_time=0, use_number=True)
```

```
[4]: result.head()
```

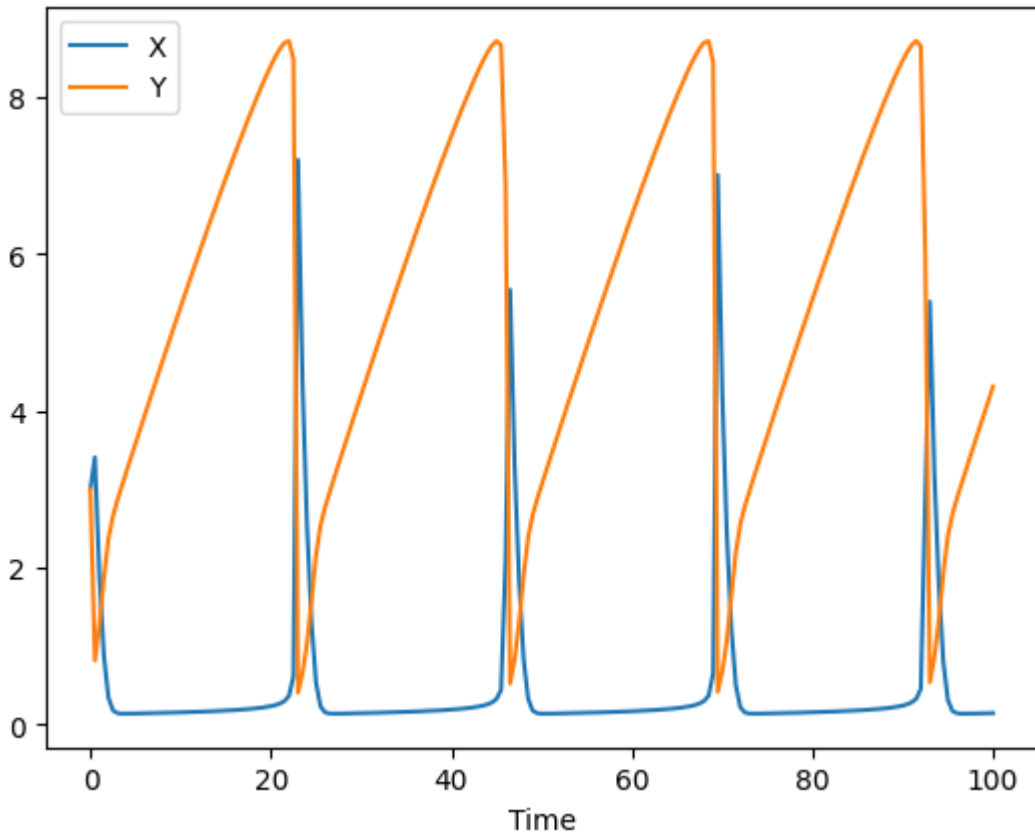
```
[4]:
```

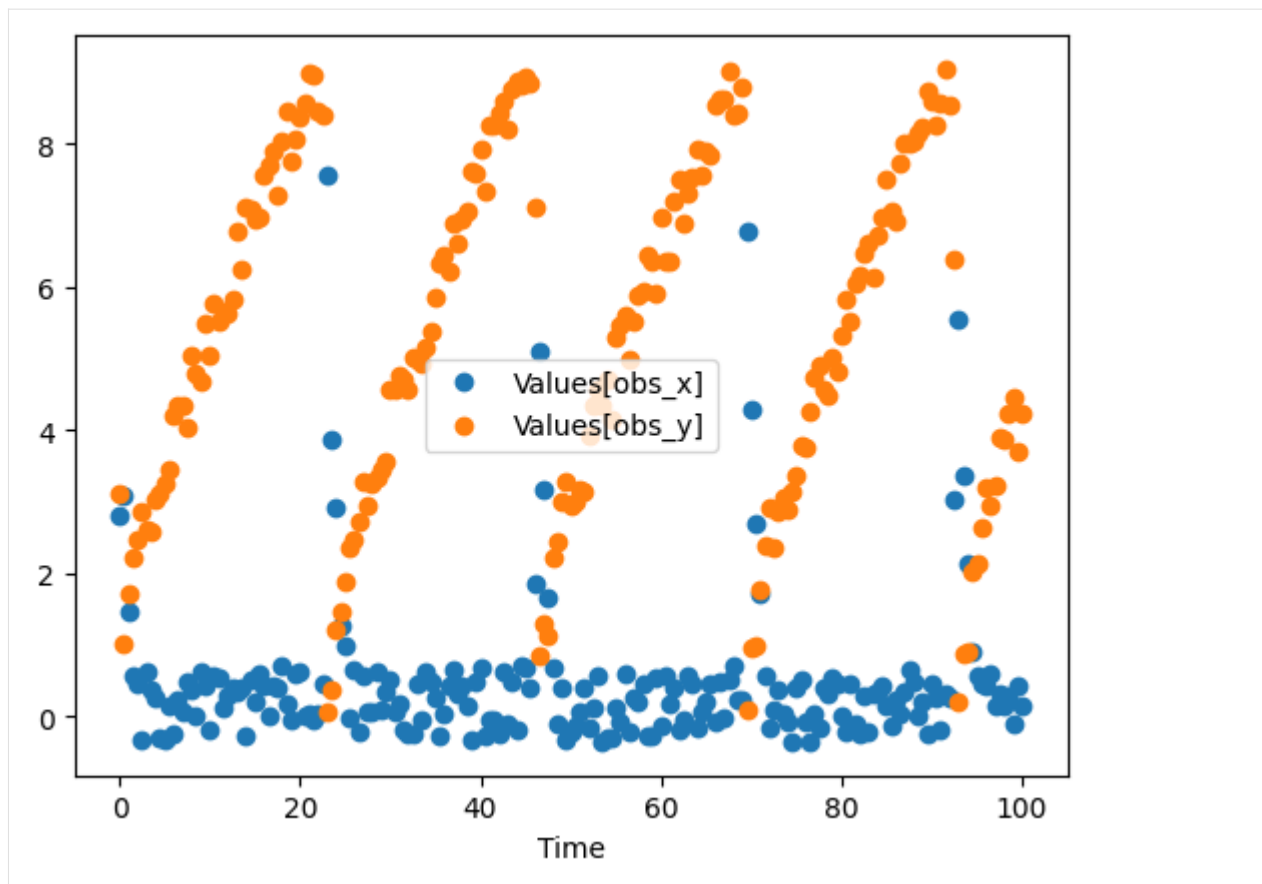
	X	Y	Values[obs_x]	Values[obs_y]
Time				
0.0	2.999996	2.999996	2.795629	3.114868
0.5	3.408155	0.817484	3.086833	1.012499
1.0	1.896454	1.276790	1.456064	1.714168
1.5	0.876253	1.872929	0.579525	2.232427
2.0	0.345934	2.368188	0.448776	2.465324

lets plot the simulation data (first plot) and the noisy data (second plot)

```
[5]: ax = result.plot(y='X')
result.plot(y='Y', ax=ax)

ax = result.plot(y='Values[obs_x]', style='o')
result.plot(y='Values[obs_y]', style='o' , ax=ax);
```





so our experimental data is the data frame with just the 2 last columns. I also rename the columns, so it makes it clear that i want to map it later to the transient concentrations. I'll also reset the index, so Time is a separate column:

```
[6]: data = result.drop(columns=['X', 'Y'])
data.rename(columns = {'Values[obs_x]':'[X]', 'Values[obs_y]':'[Y]'}, inplace=True)
data = data.reset_index()
```

```
[7]: data.head()
```

```
[7]:   Time      [X]      [Y]
0   0.0  2.795629  3.114868
1   0.5  3.086833  1.012499
2   1.0  1.456064  1.714168
3   1.5  0.579525  2.232427
4   2.0  0.448776  2.465324
```

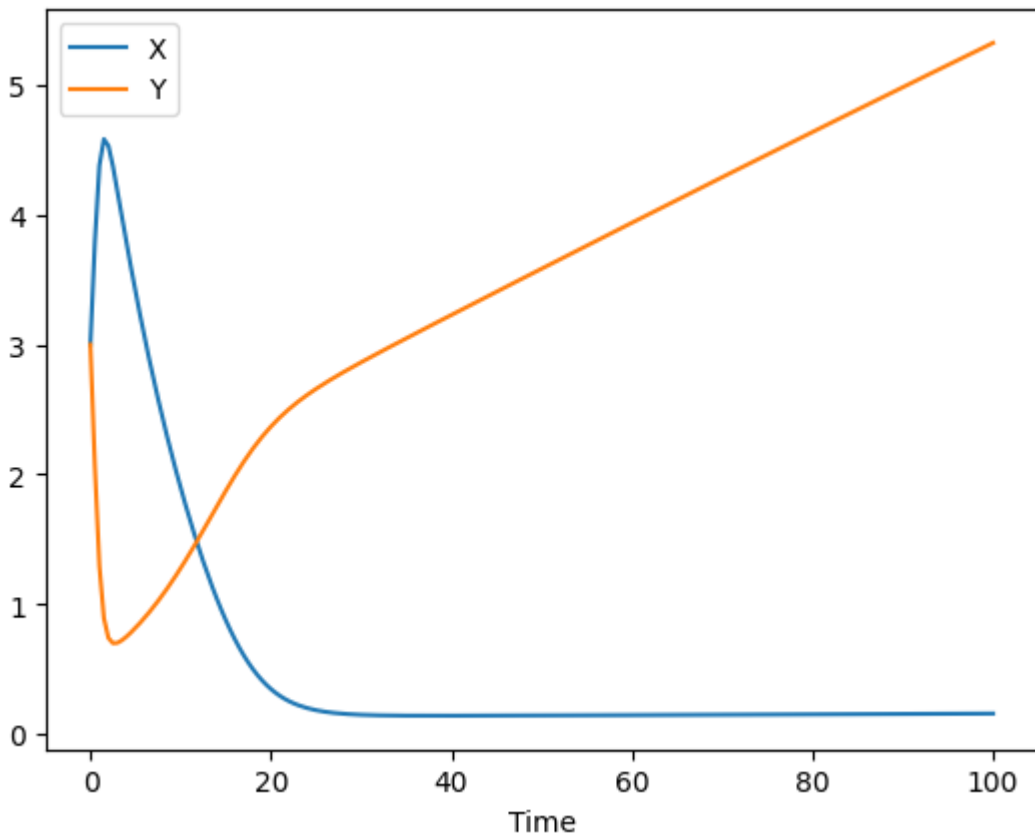
the parameters that gave rise to the solution were:

```
[8]: get_reaction_parameters()
```

```
[8]:   value reaction  type mapped_to
name
(R1).k1    1.0      R1  local
(R2).k1    1.0      R2  local
(R3).k1    1.0      R3  local
(R4).k1    1.0      R4  local
```

lets set them to something, else, so that the parameter estimation task will have something to do to find them again (we also remove the observable variables as we won't need them anymore):

```
[9]: set_reaction_parameters(['(R1).k1', '(R2).k1', '(R3).k1', '(R4).k1'], value=0.1)
      remove_parameter('obs_x')
      remove_parameter('obs_y')
      run_time_course(start_time=0).plot();
```



```
[10]: get_reaction_parameters()
```

```
[10]:
```

name	value	reaction	type	mapped_to
(R1).k1	0.1	R1	local	
(R2).k1	0.1	R2	local	
(R3).k1	0.1	R3	local	
(R4).k1	0.1	R4	local	

## 7.2 setting up the parameter estimation

the next step is to setup the parameter estimation task, first we add the experiment, and it will tell us where the experiment file has been created in case you want to delete it later.

```
[11]: add_experiment('exp1', data)
```

```
[11]: '/Users/frank/Development/basico/docs/notebooks/exp1.txt'
```

now lets verify that the experiment is there:

```
[12]: get_experiment('exp1')
```

```
[12]: <CExperiment "exp1">
```

and we have the mapping we were expecting:

```
[13]: get_experiment_mapping('exp1')
```

```
[13]:
```

	type	mapping	cn	\
column				
0	time			
1	dependent	[X]	CN=Root,Model=The Brusselator,Vector=Compartme...	
2	dependent	[Y]	CN=Root,Model=The Brusselator,Vector=Compartme...	

```

column_name
column
0          Time
1          [X]
2          [Y]
```

now we are ready to add the parameters we want to fit, in our case this will be the reaction parameters:

```
[14]: fit_items = [
        {'name': '(R1).k1', 'lower': 0.001, 'upper': 2},
        {'name': '(R2).k1', 'lower': 0.001, 'upper': 2},
        {'name': '(R3).k1', 'lower': 0.001, 'upper': 2},
        {'name': '(R4).k1', 'lower': 0.001, 'upper': 2},
    ]
```

```
[15]: set_fit_parameters(fit_items)
```

```
[16]: get_fit_parameters()
```

```
[16]:
```

	lower	upper	start	affected	\
name					
(R1).k1	0.001	2	0.1		[]
(R2).k1	0.001	2	0.1		[]
(R3).k1	0.001	2	0.1		[]
(R4).k1	0.001	2	0.1		[]

```

cn
name
(R1).k1 CN=Root,Model=The Brusselator,Vector=Reactions...
(R2).k1 CN=Root,Model=The Brusselator,Vector=Reactions...
```

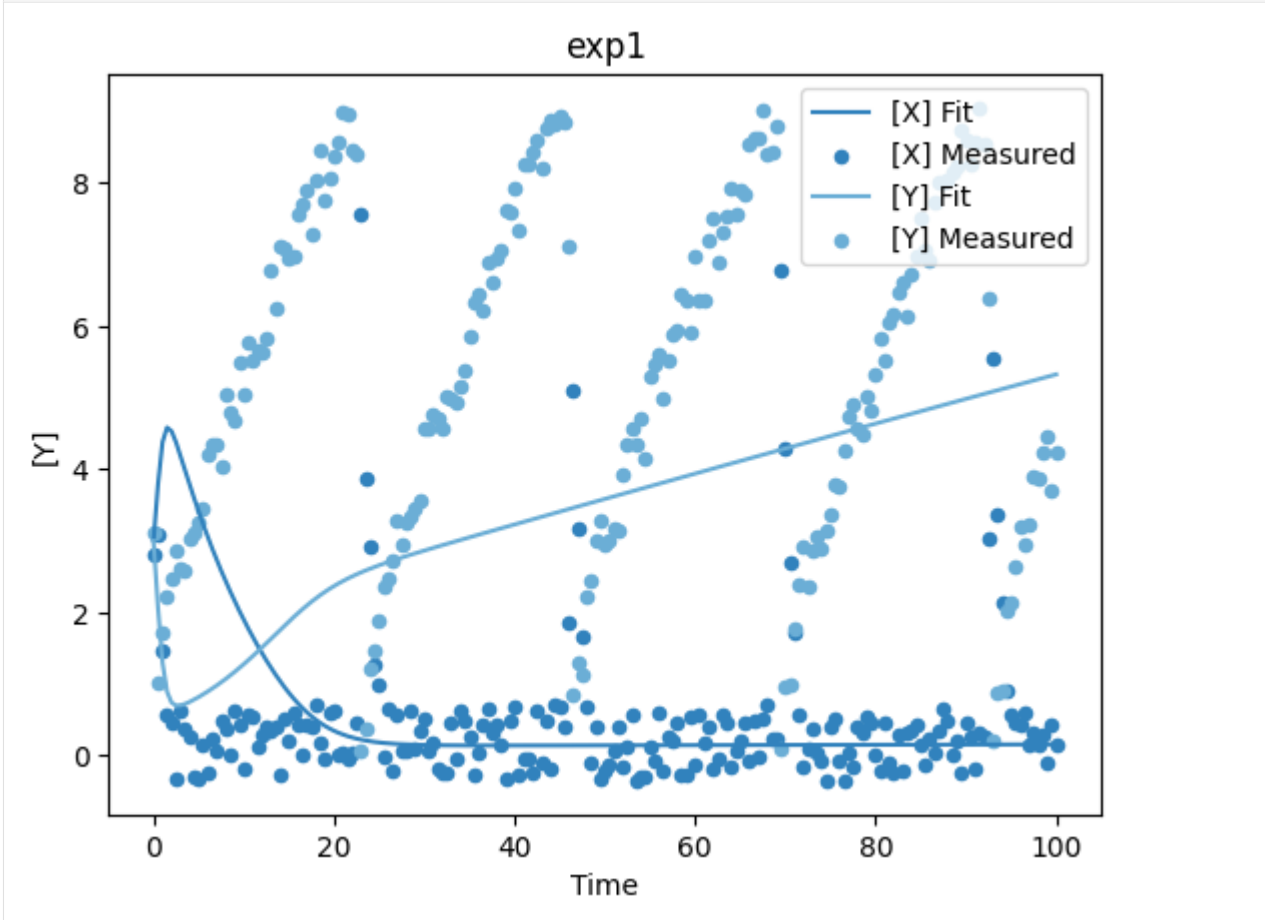
(continues on next page)

(continued from previous page)

```
(R3).k1 CN=Root,Model=The Brusselator,Vector=Reactions...
(R4).k1 CN=Root,Model=The Brusselator,Vector=Reactions...
```

and with that we are ready to run the parameter estimation. Lets see how the fit looks now, it should be bad, since we set the parameters way off:

```
[17]: plot_per_experiment();
```



Now lets run the parameter estimation, all the task names from the COPASI GUI are valid, algorithms here:

Current Solution:

- Current Solution Statistics,

Global Methods:

- Random Search,
- Simulated Annealing,
- Differential Evolution,
- Scatter Search,
- Genetic Algorithm,
- Evolutionary Programming,
- Genetic Algorithm SR,



- Evolution Strategy (SRES),
- Particle Swarm,

Local Methods:

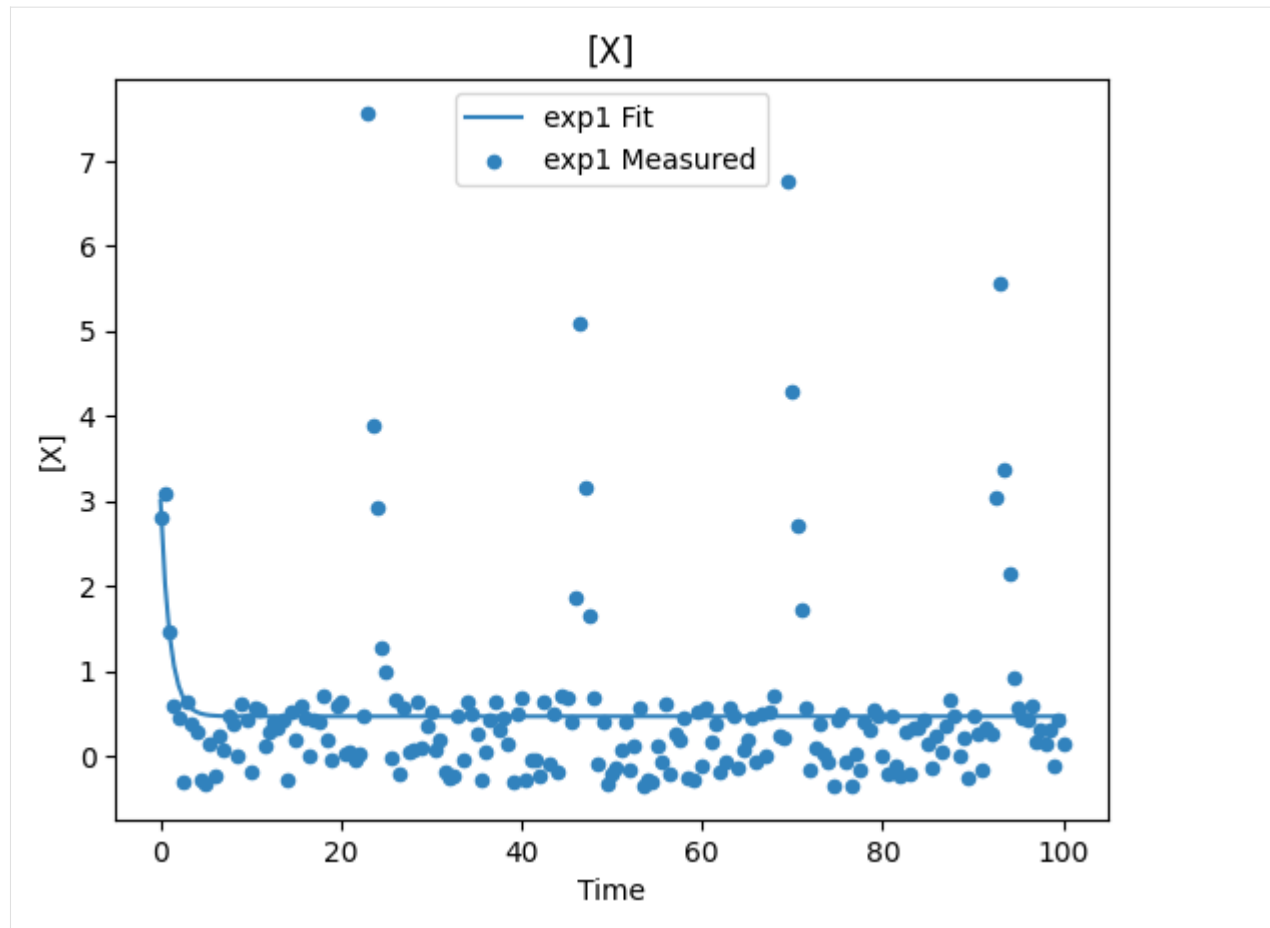
- Levenberg - Marquardt,
- Hooke & Jeeves,
- Nelder - Mead,
- Steepest Descent,
- NL2SOL,
- Praxis,
- Truncated Newton,

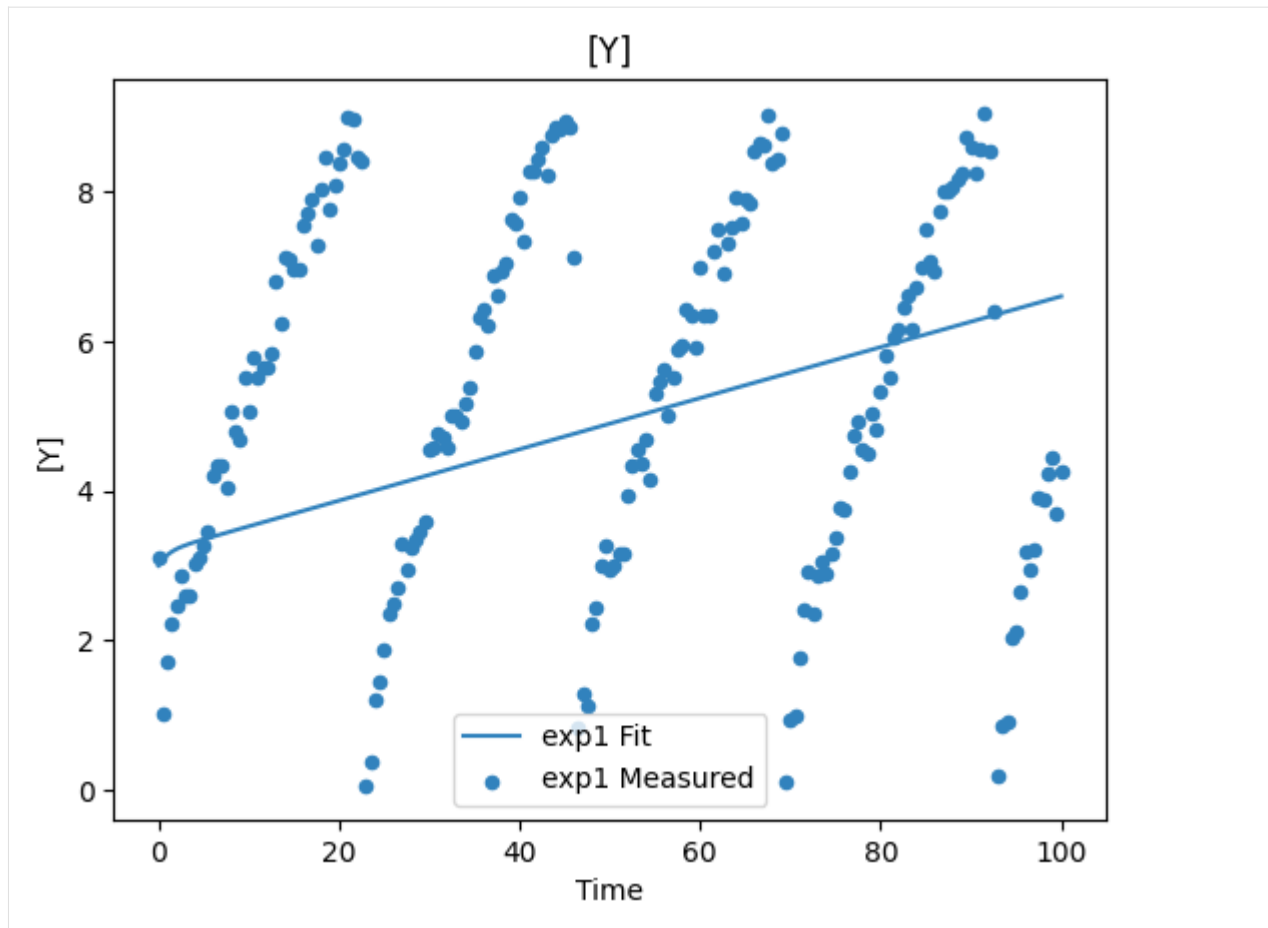
```
[18]: run_parameter_estimation(method='Evolution Strategy (SRES)', update_model=True)
```

```
[18]:
```

	lower	upper	sol	affected
name				
(R1).k1	0.001	2	0.908366	[]
(R2).k1	0.001	2	0.001000	[]
(R3).k1	0.001	2	0.025244	[]
(R4).k1	0.001	2	0.902824	[]

```
[19]: plot_per_dependent_variable();
```





## 7.3 Constraints

Additionally, you can add constraints to the parameter estimation problem. These constraint will be evaluated after each simulation, and allow to ensure, for example that solutions with concentrations outside a certain range are rejected.

Constraints are defined, just as the fit parameters specified above, when using `set_fit_constraints`. To see the constraints defined, you would use `get_fit_constraints`.

To demonstrate, lets create a constraint for this example, that rejects solutions, where the concentration of `[Y]` is outside the range of 2...10.

```
[20]: set_fit_constraints([
      {'name': 'Y', 'lower': 2, 'upper': 10}
    ])
```

`get_fit_constraints` returns a pandas dataframe of the constraints, and as usual we can transform them to a dictionary if needed using `as_dict`.

```
[21]: get_fit_constraints()
[21]:
```

	lower	upper	start	affected	\
name					
Y	2	10	6.597017		[]

(continues on next page)

(continued from previous page)

```

                                cn
name
Y      CN=Root,Model=The Brusselator,Vector=Compartmentme...

```

```
[22]: as_dict(get_fit_constraints())
```

```

[22]: {'name': 'Y',
      'lower': '2',
      'upper': '10',
      'start': 6.597017117106328,
      'affected': [],
      'cn': CN=Root,Model=The Brusselator,Vector=Compartments[compartment],
      ↪ Vector=Metabolites[Y],Reference=Concentration}

```

now lets try to run the parameter estimation again, using this constraint:

```
[23]: run_parameter_estimation(method=PE.LEVENBERG_MARQUARDT)
      get_fit_statistic()
```

```

[23]: {'obj': 295.53087769380215,
      'rms': 0.8574097253059701,
      'sd': 0.8617075453441385,
      'f_evals': 147,
      'failed_evals_exception': 0,
      'failed_evals_nan': 0,
      'constraint_evals': 2,
      'failed_constraint_evals': 0,
      'cpu_time': 0.028593999999999998,
      'data_points': 402,
      'valid_data_points': 402,
      'evals_per_sec': 0.00019451700680272107}

```

If we look at the fit statistic, we see that the evaluations failed. This happened, because the allowed interval in the constraint was too small. Setting the lower bound of the constraint, fixes that issue:

```

[24]: set_fit_constraints([
      {'name': 'Y', 'lower': 0, 'upper': 10}
    ])
      run_parameter_estimation(method=PE.LEVENBERG_MARQUARDT)
      get_fit_statistic()

```

```

[24]: {'obj': 295.53087769380215,
      'rms': 0.8574097253059701,
      'sd': 0.8617075453441385,
      'f_evals': 147,
      'failed_evals_exception': 0,
      'failed_evals_nan': 0,
      'constraint_evals': 2,
      'failed_constraint_evals': 0,
      'cpu_time': 0.026792,
      'data_points': 402,
      'valid_data_points': 402,
      'evals_per_sec': 0.00018225850340136056}

```

We've seen how to add constraints to the parameter estimation problem, and how it would look if the constraint causes the parameter estimation procedure to fail.



## PARAMETER ESTIMATION (IMPORT / EXPORT)

This example describes how to export the parameter estimation setup to a YaML file (or string), so that it can be easily modified later, and then loaded back (or applied to a different model).

### 8.1 The Format

When saving the file, it will be exported as a sequence, of experiments of the following form:

```
name: Experiment
  filename: data.txt
  type: Time-Course
  separator: "\t"
  first_row: 1
  last_row: 102
  header_row: 1
  weight_method: Mean Square
  normalize_per_experiment: true
  mapping:
    - column: '# Time'
      type: time
    - column: Values[F16BP_obs]
      type: dependent
      cn: CN=Root, ...
      object: '[Fru1,6-P2]'
```

the individual fields are:

- **name:** the name of the experiment
- **type:** Time-Course for time course data (requires a mapping of type `time` to be specified), or Steady-State for steady state data.
- **separator:** the separator being used
- **first\_row:** the beginning of the experiment in the file (1 based)
- **last\_row:** the last row of the experiment
- **header\_row:** (optional) row with header information that can be later used in the `column` field of the mappings.
- **weight\_method:** one of: Mean, Mean Square, Standard Deviation or Value Scaling.
- **normalize\_per\_experiment:** boolean indicating whether experiments should be scaled individual (True) or over all defined experimentes (False).

- **mapping**: sequence of column mappings described as follows.

The mapping descriptions contain the fields:

- **column**: either an integer index (zero based), describing which column the mapping applies to. If the experiment has header information, the column may be a string with the (case sensitive) header the mapping applies to.
- **type**: the type of the mapping for this column. One of `ignored`, `time`, `dependent` or `independent`.
- **object**: display name of the element to map to
- **cn**: (optional) the CN to the reference to map to.

For columns of type `dependent` or `independent` at least one of `object` or `cn` needs to be defined. (The `cn` value takes preference).

- **weight**: may be used for columns of type `dependent` to customize the scale to be applied to the column. If not specified it will be automatically calculated based on the selected `weight_method`.

## 8.2 Example

We start by importing basico as usual (should that fail for you just `!pip install copasi-basico`:

```
[1]: from basico import *
```

here we load an existing parameter estimation example, included with the distribution as example:

```
[2]: dm = load_example('PK')
```

now we can directly export the setup of the experimental data files as yaml string (or if you supply a filename to the function, it will be saved as file):

```
[3]: yaml_str = save_experiments_to_yaml()
print("\n".join(yaml_str.split('\n')[:24])) # just restricting the amount of yaml to be
      ↪ printed here to the first experiment
```

```
- name: Experiment
  filename: e:/development/basico/basico/data\data_2.txt
  type: Time-Course
  separator: "\t"
  first_row: 1
  last_row: 102
  weight_method: Mean Square
  normalize_per_experiment: true
  header_row: 1
  mapping:
    - column: '# Time'
      type: time
    - column: Values[F16BP_obs]
      type: dependent
      cn: CN=Root,Model=Pritchard2002_glycolysis,Vector=Compartments[cytosol],
      ↪ Vector=Metabolites[Fru1\,6-P2],Reference=Concentration
      object: '[Fru1,6-P2]'
    - column: Values[Glu_obs]
      type: dependent
      cn: CN=Root,Model=Pritchard2002_glycolysis,Vector=Compartments[cytosol],
```

(continues on next page)



(continued from previous page)

```
↪ Vector=Metabolites[Glc(int)],Reference=Concentration
  object: '[Glc(int)]'
- column: Values[Pyr_obs]
  type: dependent
  cn: CN=Root,Model=Pritchard2002_glycolysis,Vector=Compartments[cytosol],
↪ Vector=Metabolites[pyruvate],Reference=Concentration
  object: '[pyruvate]'
```

at this point you'd make modifications to you'd want to it. Remember, that the `cn` is optional. The key points to keep in mind:

- ensure that the row number for the experiment are consistent with the changes you make
- if it is a time course experiment, ensure you have a column of type `time`.
- the column specifier is either the index of the column, or the name if the experiment has headers.

once the changes are made, you can load the setup back into the model, using the `load_experiments_from_yaml` function. This will remove all existing experiments from the file first.

```
[4]: load_experiments_from_yaml(yaml_str)
```



## OPTIMIZATION

This notebook walks through the steps of setting up / running optimizations using `basico`. We start as usual:

```
[1]: from basico import *
```

### 9.1 Model

The first step is to load a model (this can be done as usual using `load_model`, `load_biomodel` or by `load_example`) or create a new one. Here I'll create one with a typical optimization problem, the `himmelblau` function:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

In `basico` this is easily done using global parameters for `x` and `y`, and then an assignment for the function

```
[2]: new_model(name="Himmelblau",
              notes="""A model implementing the himmelau function

              Maxima is known to be at (-0.270845, -0.923039) with
              max value 181.617

              4 Minima: (3,2), (-2.805118, 3.131313),
                        (-3.779310, -3.2383186),
                        (3.584428, -1.848126) with value 0

              """);
```

```
[3]: add_parameter('x', initial_value=0)
      add_parameter('y', initial_value=0)
      add_parameter('f', type='assignment',
                    expression='({Values[x].InitialValue}^2+{Values[y].InitialValue}-11)^2+(
      ↪{Values[x].InitialValue}+{Values[y].InitialValue}^2-7)^2');
```

## 9.2 The Setup

Now we setup the parameters to be varied during the optimization. For each item we need to specify, what to vary, as well as the lower and upper bounds. The utility function `get_opt_item_template` allows to retrieve all the global / local parameters and sets default bounds:

```
[4]: get_opt_item_template(include_global=True, default_lb=-1)
[4]: [{'name': 'Values[x].InitialValue', 'lower': -1, 'upper': 1000, 'start': 0.0},
      {'name': 'Values[y].InitialValue', 'lower': -1, 'upper': 1000, 'start': 0.0}]
```

lets use them:

```
[5]: set_opt_parameters(get_opt_item_template(include_global=True, default_lb=-1))
```

the next thing is to set up the objective function. Any expression with the names of model elements will work, here we want to minimize the value of the global parameter `f`:

```
[6]: set_objective_function(expression='Values[f].InitialValue', minimize=True)
```

additional settings can be modified using `set_opt_settings`, such as specifying the method to be used and their parameters:

```
[7]: set_opt_settings(settings={
      'subtask': T.TIME_COURSE,
      'method': {
        'name': PE.LEVENBERG_MARQUARDT
      })
```

to verify the setup you can use `get_opt_parameters` to retrieve all the parameters and bounds and `get_opt_settings` to retrieve all settings:

```
[8]: get_opt_settings()
[8]: {'scheduled': False,
      'update_model': False,
      'problem': {'Maximize': False,
                  'Randomize Start Values': False,
                  'Calculate Statistics': True},
      'method': {'Iteration Limit': 2000,
                  'Tolerance': 1e-06,
                  'name': 'Levenberg - Marquardt'},
      'report': {'filename': '',
                  'report_definition': 'Optimization',
                  'append': True,
                  'confirm_overwrite': True},
      'expression': 'Values[f].InitialValue',
      'subtask': 'Time-Course'}
```

## 9.3 Running the optimization

Now that everything is set up, we can simply run the optimization:

```
[9]: run_optimization()
[9]:
```

	lower	upper	sol
name			
Values[x]	-1	1000	2.999999
Values[y]	-1	1000	2.000000

we got close to one of the minima, to see more information about the run, you can use:

```
[10]: get_opt_statistic()
[10]: {'obj': 8.352969862744543e-11,
      'f_evals': 321,
      'failed_evals_exception': 0,
      'failed_evals_nan': 0,
      'constraint_evals': 0,
      'failed_constraint_evals': 0,
      'cpu_time': 0.001112,
      'evals_per_sec': 3.46417445482866e-06}
```

## 9.4 Customn Output

normally when you run an optimization `run_optimization` will return a data frame of the best parameters found, just as when you run `get_opt_solution`. So, since we get to the results in any case, there is an optional parameter that you can pass to `run_optimization`, to collect any element you would like during the run.

This is an advanced feature, as for many things we only have Common Names, that are a bit wieldy to use, still lets do that here.

**NOTE:** this will only work for real valued CN's right now

In the next run, i collect the number of function evaluation and the objective function value:

```
[11]: run_optimization(output=[
      'Values[x].InitialValue',
      'Values[y].InitialValue',
      'CN=Root,Vector=TaskList[Optimization],Problem=Optimization,Reference=Best Value'
    ])
[11]:
```

	Values[x].InitialValue	Values[y].InitialValue	\
0	0.000000	0.000000	
1	0.875001	1.375001	
2	2.108645	2.656495	
3	2.185752	2.641869	
4	2.303988	2.611787	
5	2.461318	2.552965	
6	2.637356	2.451114	
7	2.797570	2.308062	
8	2.911844	2.158941	
9	2.972712	2.053942	

(continues on next page)

(continued from previous page)

```

10          2.994778          2.010172
11          2.999459          2.000957
12          2.999970          2.000046
13          2.999998          2.000001
14          2.999999          2.000000
15          2.999999          2.000000
16          2.999999          2.000000
17          2.999999          2.000000

TaskList[Optimization].(Problem)Optimization.Best Value
0          1.700000e+02
1          9.641837e+01
2          1.987741e+01
3          1.750903e+01
4          1.400292e+01
5          9.623309e+00
6          5.245471e+00
7          2.014153e+00
8          4.593865e-01
9          4.851944e-02
10         1.711901e-03
11         1.604411e-05
12         4.073483e-08
13         1.515858e-10
14         8.410451e-11
15         8.353344e-11
16         8.352971e-11
17         8.352970e-11

```

## 9.5 Constraints

You can further constrain the optimization problem, by defining constraints that will be evaluated when the model is simulated. This is useful for example, when you want to ensure concentrations are in a specific range.

You can modify constraints using the `set_opt_constraints` function and retrieve them using `get_opt_constraints`. As example here, we constrain the solution to be greater than 170 when maximizing:

```

[29]: set_opt_constraints([
      {'name': 'Values[f]', 'lower': 170, 'upper': 200}
])

```

```

[30]: get_opt_constraints()

```

```

[30]:      lower upper      start \
name
Values[f]   170   200  181.616522

                                           cn
name
Values[f]  CN=Root,Model=Himmelblau,Vector=Values[f],Refe...

```

```
[31]: settings = get_opt_settings()
      settings['problem']['Maximize'] = True
      run_optimization(settings=settings)
      get_opt_statistic()

[31]: {'obj': 181.6165215225808,
      'f_evals': 282,
      'failed_evals_exception': 0,
      'failed_evals_nan': 0,
      'constraint_evals': 33,
      'failed_constraint_evals': 0,
      'cpu_time': 0.001709,
      'evals_per_sec': 6.060283687943263e-06}
```

*Note:* Keep in mind, that setting constraints will make finding the solution harder for the algorithms. So when in doubt, it might be a good idea to take them out.

```
[ ]:
```





## SIMPLE SIMULATIONS AND PLOTTING

In this file, we load a model from the BioModels database, and simulate it for varying durations. We start as usual:

```
[2]: from basico import *
```

### 10.1 Load a model

to load the model, we use the `load_biomodel` function, it takes in either an integer, which will be transformed into a valid biomodels id, or you can pass in a valid biomodels id to begin with:

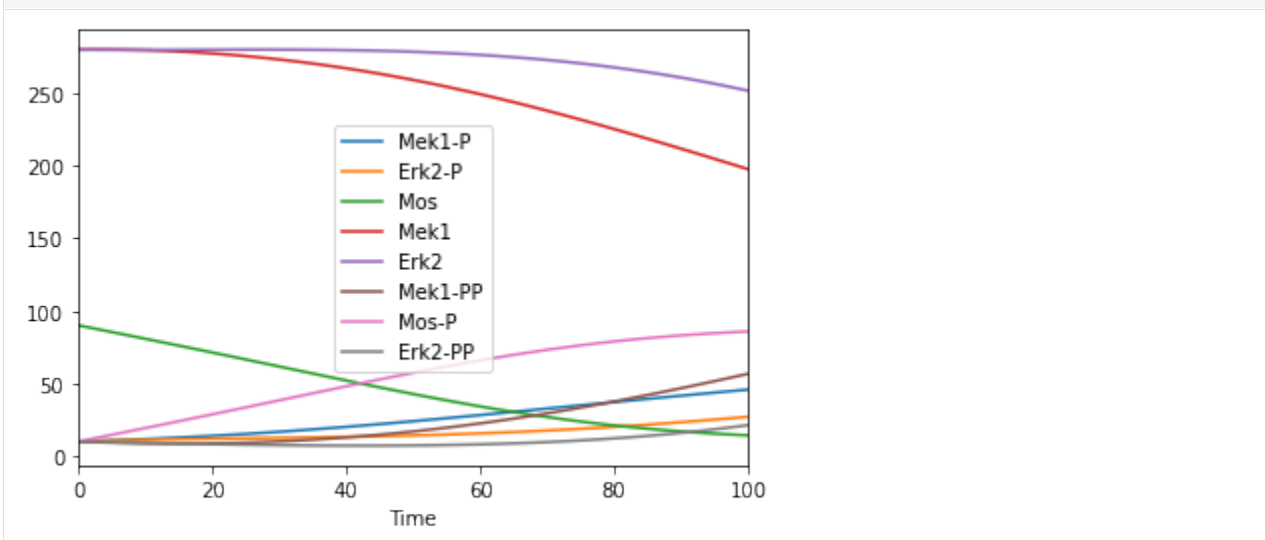
```
[3]: biomod = load_biomodel(10)
```

### 10.2 Run time course

After the model is loaded, it is ready to be simulated, here we try it for varying durations:

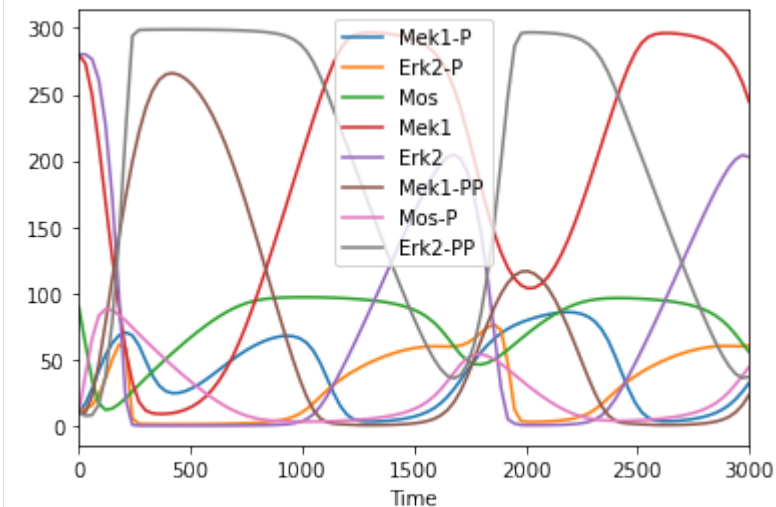
#### 10.2.1 Time course duration 100

```
[4]: tc = run_time_course(duration = 100)
tc.plot();
```



## 10.2.2 Time course duration 3000

```
[5]: tc = run_time_course(duration = 3000)
tc.plot();
```



## 10.3 Get compartments

To get an overview of what elements the model entails, we can query the individual elements, yielding each time a pandas dataframe with the information:

```
[6]: get_compartments()
```

```
[6]:
```

	type	unit	initial_size	initial_expression	dimensionality	expression \
name						
uVol	fixed	1	1.0		3	
	size	rate		key		
name						
uVol	1.0	0.0	Compartment_0			

## 10.4 Get parameters (“global quantities”)

```
[7]: get_parameters() # no global quantities
```

## 10.5 Run steady state

we can also run the model to steady state, in order to see the steady state concentrations:

```
[9]: run_steadystate()
# now call get_species, to get the steady state concentration and particle numbers
get_species()[['concentration', 'particle_number']]
```

```
[9]:
```

	compartment	type	unit	initial_concentration	\
name					
Mek1-P	uVol	reactions	nmol/l	10.0	
Erk2-P	uVol	reactions	nmol/l	10.0	
Mos	uVol	reactions	nmol/l	90.0	
Mek1	uVol	reactions	nmol/l	280.0	
Erk2	uVol	reactions	nmol/l	280.0	
Mek1-PP	uVol	reactions	nmol/l	10.0	
Mos-P	uVol	reactions	nmol/l	10.0	
Erk2-PP	uVol	reactions	nmol/l	10.0	

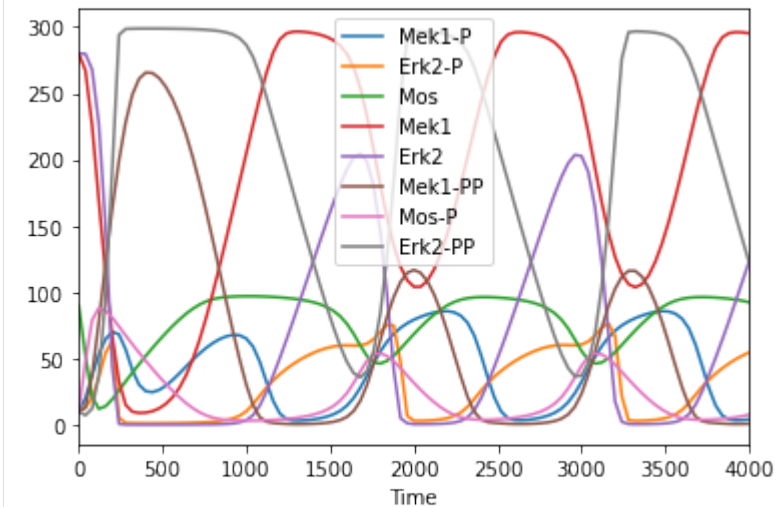
	initial_particle_number	initial_expression	expression	concentration	\
name					
Mek1-P	6.022141e+15			41.142140	
Erk2-P	6.022141e+15			99.971352	
Mos	5.419927e+16			76.635083	
Mek1	1.686199e+17			238.913726	
Erk2	1.686199e+17			102.158541	
Mek1-PP	6.022141e+15			19.944135	
Mos-P	6.022141e+15			23.364917	
Erk2-PP	6.022141e+15			97.870107	

	particle_number	rate	particle_number_rate	key
name				
Mek1-P	2.477638e+16	3.113511e-16	0.187500	Metabolite_3
Erk2-P	6.020416e+16	0.000000e+00	0.000000	Metabolite_6
Mos	4.615073e+16	-7.783777e-17	-0.046875	Metabolite_0
Mek1	1.438772e+17	0.000000e+00	0.000000	Metabolite_2
Erk2	6.152131e+16	-6.745940e-16	-0.406250	Metabolite_5
Mek1-PP	1.201064e+16	-3.113511e-16	-0.187500	Metabolite_4
Mos-P	1.407068e+16	7.783777e-17	0.046875	Metabolite_1
Erk2-PP	5.893876e+16	6.745940e-16	0.406250	Metabolite_7

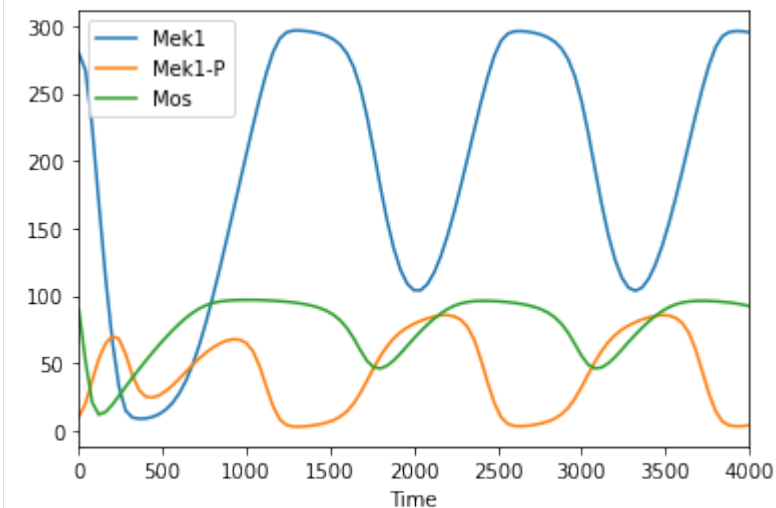
## 10.6 Use pandas syntax for indexing and plotting

```
[10]: tc = run_time_course(model = biomod, duration = 4000)
      tc.plot();
```



```
[11]: tc.loc[:, ['Mek1', 'Mek1-P', 'Mos']].plot()
```

```
[11]: <AxesSubplot:xlabel='Time'>
```



## SIMULATING A MODEL WITH BASICO

First some jupyter magic for plotting and convenience

```
[1]: %pylab
      %matplotlib inline
      import sys
      if not '../..' in sys.path:
          sys.path.append('../..')

Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

Now import basico

```
[2]: from basico import *
```

now we are ready to load a model, just adjust the file\_name variable, to match yours. The file can be a COPASI or SBML file. For this example, we use the brusselator model, that is distributed with the package.

```
[3]: file_name = get_examples('brusselator')[0]
```

```
[4]: model = load_model(file_name)
```

now we are ready to simulate. Calling run\_time\_course will run the simulation as specified in the COPASI file and return a pandas dataframe for it.

```
[5]: run_time_course().head()
```

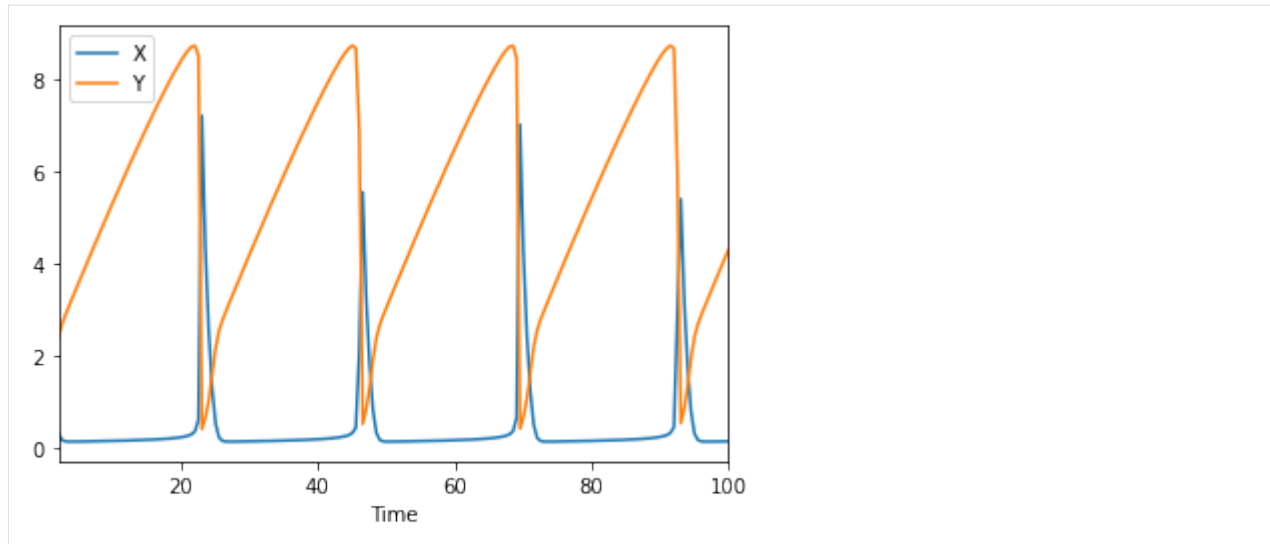
```
[5]:
```

	X	Y
Time		
2.0	0.345934	2.368188
2.5	0.182762	2.658596
3.0	0.147433	2.863681
3.5	0.141129	3.048319
4.0	0.140734	3.228345

for plotting you would then just plot that as one does

```
[6]: df = run_time_course()
      df.plot()
```

```
[6]: <AxesSubplot:xlabel='Time'>
```



## 11.1 The run\_time\_course command

you can change different options for the time course by adding named parameters into the `run_time_course` command. Supported are:

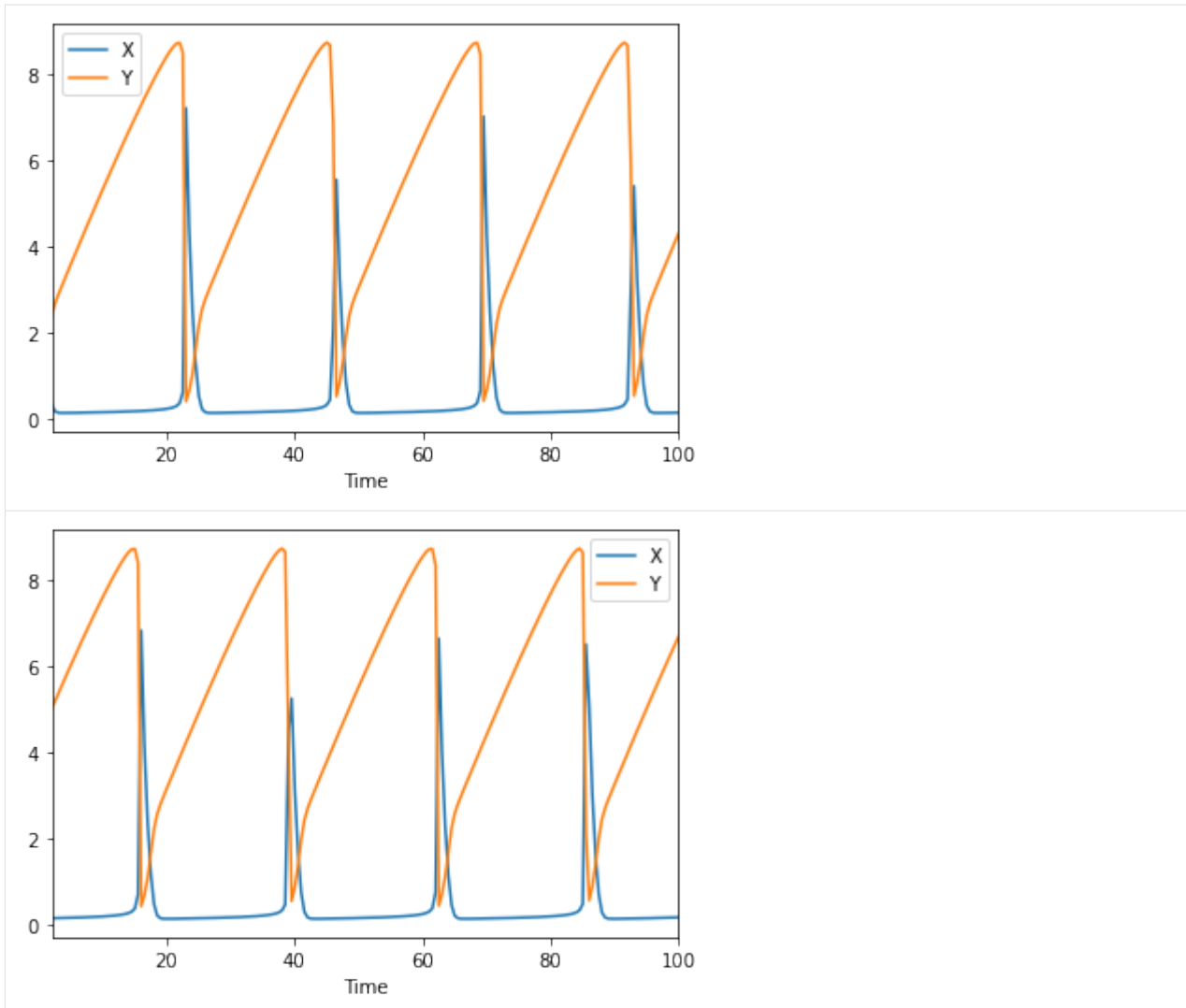
- `model`: in case you want to use another model than the one last loaded
- `scheduled`: to mark the model as scheduled
- `update_model`: to update the initial state after the simulation is run
- `duration`: to specify how long the simulation is run
- `automatic`: in case you would like automatic step size being used
- `output_event`: in case you would like to have the event values before and after the event hit listed
- `start_time`: to change the start time
- `step_number` or `intervals`: to overwrite the number of steps being used
- `method`: a method name to use for the simulation.

so lets run two simulations that will be different slightly, as we will use the `update_model` flag:

```
[7]: df1 = run_time_course(update_model=True)
     df2 = run_time_course(update_model=True)
```

```
[8]: df1.plot(), df2.plot()
```

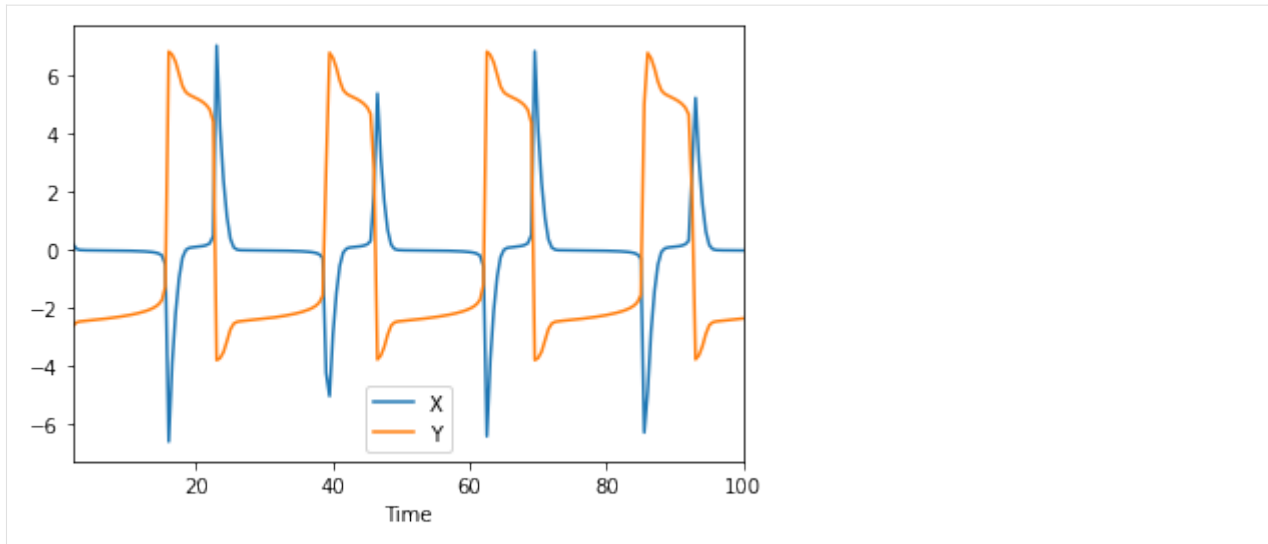
```
[8]: (<AxesSubplot:xlabel='Time'>, <AxesSubplot:xlabel='Time'>)
```



And now you could plot the difference between them too:

```
[9]: (df1-df2).plot()
```

```
[9]: <AxesSubplot:xlabel='Time'>
```



```
[10]: (df1-df2).describe()
```

```
[10]:
```

	X	Y
count	197.000000	197.000000
mean	-0.014562	-0.123916
std	1.592729	3.615933
min	-6.630084	-3.816620
25%	-0.056936	-2.428037
50%	-0.024780	-2.245888
75%	0.077394	4.837513
max	7.051794	6.844179

```
[ ]:
```



## SIMULATING WITH CUSTOM RESULTS

In this example, we'll load a model, and simulate it, fine tuning the simulation results returned. We start as usual:

```
[1]: from basico import *
```

lets load a model, i'll choose a model from the BioModels Database:

```
[2]: load_biomodel(206);
```

```
[3]: get_species()
```

```
[3]:
```

	compartment	type	unit	\
name				
ATP	compartment	reactions	mmol/l	
Triose_Gly3Phos_DHAP	compartment	reactions	mmol/l	
Acetaldehyde	compartment	reactions	mmol/l	
NAD	compartment	reactions	mmol/l	
Pyruvate	compartment	reactions	mmol/l	
Glucose	compartment	reactions	mmol/l	
extracellular acetaldehyde	compartment	reactions	mmol/l	
3PG	compartment	reactions	mmol/l	
F16P	compartment	reactions	mmol/l	

	initial_concentration	initial_particle_number	\
name			
ATP	2.00	1.204428e+21	
Triose_Gly3Phos_DHAP	0.60	3.613284e+20	
Acetaldehyde	0.08	4.817713e+19	
NAD	0.60	3.613284e+20	
Pyruvate	8.00	4.817713e+21	
Glucose	1.00	6.022141e+20	
extracellular acetaldehyde	0.02	1.204428e+19	
3PG	0.70	4.215499e+20	
F16P	5.00	3.011070e+21	

	initial_expression	expression	concentration	\
name				
ATP			NaN	
Triose_Gly3Phos_DHAP			NaN	
Acetaldehyde			NaN	
NAD			NaN	
Pyruvate			NaN	

(continues on next page)

(continued from previous page)

Glucose				NaN
extracellular acetaldehyde				NaN
3PG				NaN
F16P				NaN

	particle_number	rate	particle_number_rate	\
name				
ATP	NaN	0.0		0.0
Triose_Gly3Phos_DHAP	NaN	0.0		0.0
Acetaldehyde	NaN	0.0		0.0
NAD	NaN	0.0		0.0
Pyruvate	NaN	0.0		0.0
Glucose	NaN	0.0		0.0
extracellular acetaldehyde	NaN	0.0		0.0
3PG	NaN	0.0		0.0
F16P	NaN	0.0		0.0

	key	sbml_id
name		
ATP	Metabolite_1	at
Triose_Gly3Phos_DHAP	Metabolite_3	s3
Acetaldehyde	Metabolite_7	s6
NAD	Metabolite_4	na
Pyruvate	Metabolite_6	s5
Glucose	Metabolite_0	s1
extracellular acetaldehyde	Metabolite_8	s6o
3PG	Metabolite_5	s4
F16P	Metabolite_2	s2

[4]: get\_reactions()

		scheme	flux	particle_flux	\
name					
v1		Glucose + 2 * ATP = F16P	0.0		0.0
v2		F16P = 2 * Triose_Gly3Phos_DHAP	0.0		0.0
v3		Triose_Gly3Phos_DHAP + NAD = 3PG + ATP	0.0		0.0
v4		3PG = Pyruvate + ATP	0.0		0.0
v5		Pyruvate = Acetaldehyde	0.0		0.0
v7		ATP =	0.0		0.0
v8		Triose_Gly3Phos_DHAP = NAD	0.0		0.0
v9		"extracellular acetaldehyde" =	0.0		0.0
v10	Acetaldehyde = 0.1 * "extracellular acetaldehyde"		0.0		0.0
v6		Acetaldehyde = NAD	0.0		0.0
v0		= Glucose	0.0		0.0

	function	key	sbml_id
name			
v1	Function for v1	Reaction_0	v1
v2	Function for v2	Reaction_1	v2
v3	Function for v3	Reaction_2	v3
v4	Function for v4	Reaction_3	v4
v5	Function for v5	Reaction_4	v5

(continues on next page)

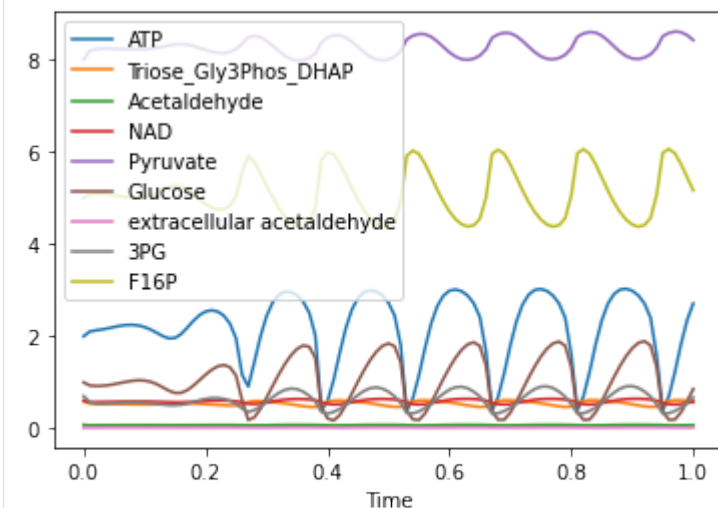
(continued from previous page)

v7	Function for v7	Reaction_5	v7
v8	Function for v8	Reaction_6	v8
v9	Function for v9	Reaction_7	v9
v10	Function for v10	Reaction_8	v10
v6	Function for v6	Reaction_9	v6
v0	Constant flux (reversible)	Reaction_10	v0

## 12.1 Simulations:

```
[5]: run_time_course(duration=1).plot()
```

```
[5]: <AxesSubplot:xlabel='Time'>
```



## 12.2 Custom Selection list

Sometimes you just want to select certain elements, for which you want to select the output. This can be done using `run_time_course_with_output`, where the first element is an array of display names, for which you'd like to collect the output

```
[6]: run_time_course_with_output(['Time', '[ATP]', 'ATP.Rate', '(v1).Flux'], duration=1)
```

```
[6]:
```

	Time	[ATP]	ATP.Rate	(v1).Flux
0	0.00	2.000000	23.471501	64.705882
1	0.01	2.103916	23.471501	52.477019
2	0.02	2.127439	23.471501	50.238184
3	0.03	2.147108	23.471501	49.133261
4	0.04	2.172374	23.471501	48.222173
..	...	...	...	...
96	0.96	0.700117	23.471501	58.593102
97	0.97	1.226780	23.471501	40.493568
98	0.98	1.833802	23.471501	29.080133
99	0.99	2.349971	23.471501	24.298749

(continues on next page)

(continued from previous page)

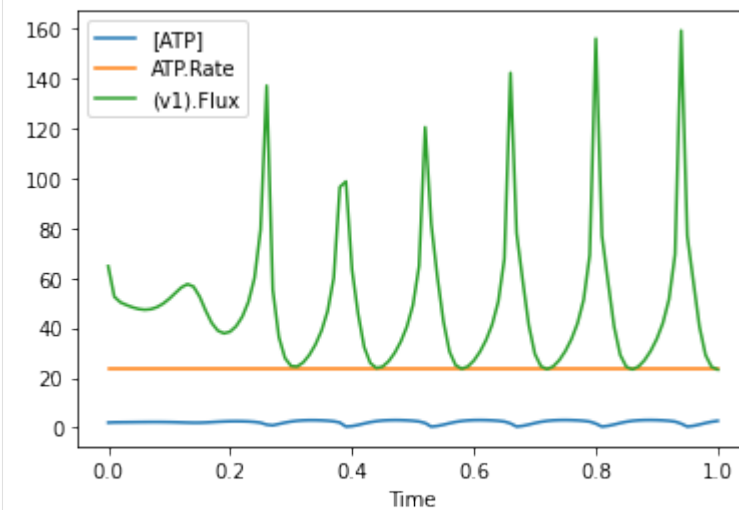
```
100  1.00  2.708829  23.471501  23.269926
```

```
[101 rows x 4 columns]
```

as you see just, as with `run_time_course` a pandas dataframe is returned, the only difference is, that the index column is not automatically set to the time column (as you might not want to collect time!).

```
[7]: run_time_course_with_output(['Time', '[ATP]', 'ATP.Rate', '(v1).Flux'], duration=1).set_index('Time').plot()
```

```
[7]: <AxesSubplot:xlabel='Time'>
```



The other difference, when using this method is, that the selection is not automatically saved to COPASI, so when you prepare a model and run it later in the COPASI UI, the output selection is not persisted. You would have to create plot elements or reports for that.

## 12.3 Custom Time Points

Some times you might only want to return values at certain times, this is possible as well (with either `run_time_course` and `run_time_course_with_output`, adding the ‘values’ argument:

```
[8]: run_time_course(values=[0, 1, 2, 4])
```

```
[8]:
```

	ATP	Triose_Gly3Phos_DHAP	Acetaldehyde	NAD	Pyruvate	\
Time						
0.0	2.000000	0.600000	0.080000	0.600000	8.000000	
1.0	2.708848	0.593576	0.074087	0.579888	8.415963	
2.0	3.001586	0.574146	0.078183	0.620547	8.217482	
4.0	2.923411	0.508574	0.080717	0.645962	8.019722	

	Glucose	extracellular	acetaldehyde	3PG	F16P
Time					
0.0	1.000000		0.020000	0.700000	5.000000
1.0	0.856600		0.023069	0.672236	5.160410

(continues on next page)

(continued from previous page)

2.0	1.290106	0.024351	0.848794	4.759722
4.0	1.851914	0.025662	0.903727	4.376452

```
[9]: run_time_course_with_output(['Time', '[ATP]', 'ATP.Rate'], values=[0, 1, 2, 4])
```

```
[9]:
```

	Time	[ATP]	ATP.Rate
0	0.0	2.000000	23.471501
1	1.0	2.708848	23.471501
2	2.0	3.001586	23.471501
3	4.0	2.923411	23.471501

## 12.4 For expert users

For some elements, we do not have an easy mapping between object and display names, to help around that issue `run_time_course_with_output` allows the use of CN's as well. Though of course they are quite error prone and not as nice to read!

```
[10]: run_time_course_with_output([
    'CN=Root,Model=Wolf2000_Glycolytic_Oscillations,Reference=Time',
    'CN=Root,Model=Wolf2000_Glycolytic_Oscillations,Vector=Compartments[compartment],
    ↪Vector=Metabolites[Glucose],Reference=ParticleNumber',
    'CN=Root,Model=Wolf2000_Glycolytic_Oscillations,Vector=Compartments[compartment],
    ↪Reference=Volume'],
    values=[0, 1, 2, 4])
```

```
[10]:
```

	Time	Glucose.ParticleNumber	Compartments[compartment].Volume
0	0.0	6.022141e+20	1.0
1	1.0	5.158565e+20	1.0
2	2.0	7.769203e+20	1.0
3	4.0	1.115249e+21	1.0

```
[ ]:
```



## PARAMETER SCANS

This notebook demonstrates how to edit / change parameter scan tasks. While most scans can be done using `basico`, by using simple loops, here we create the scans for the COPASI `Parameter Scan` task. That means, that these scans can be carried out later, using the COPASI graphical user interface, or the command line interface.

```
[1]: from basico import *
```

lets start by using the `brusselator` example, where a scan task is already set up. Using `get_scan_settings`, we can see all the individual settings:

```
[2]: load_example('brusselator')
get_scan_settings()
```

```
[2]: {'update_model': False,
      'scheduled': False,
      'subtask': 'Steady-State',
      'output_during_subtask': False,
      'continue_from_current_state': False,
      'continue_on_error': False,
      'scan_items': [{'type': 'scan',
                       'num_steps': 10,
                       'log': False,
                       'min': 0.5,
                       'max': 2.0,
                       'values': '',
                       'use_values': False,
                       'item': '(R1).k1',
                       'cn': 'CN=Root,Model=The Brusselator,Vector=Reactions[R1],ParameterGroup=Parameters,
↪Parameter=k1,Reference=Value'}]}
```

the dictionary that is returned, contains all the information stored in the COPASI file. Alternatively you could also just retrieve the scan items using `get_scan_items`.

```
[3]: get_scan_items()
```

```
[3]: [{'type': 'scan',
      'num_steps': 10,
      'log': False,
      'min': 0.5,
      'max': 2.0,
      'values': '',
      'use_values': False,
      'item': '(R1).k1',
```

(continues on next page)

(continued from previous page)

```
'cn': 'CN=Root,Model=The Brusselator,Vector=Reactions[R1],ParameterGroup=Parameters,
↪Parameter=k1,Reference=Value']}]
```

## 13.1 Modifying Scan Settings

Analogue to retrieving the settings, they can be set as well, using the same keys, as displayed above. Again, you can change all settings using `set_scan_settings`. Or just change the scan\_items using `set_scan_items`. If the scan settings dictionary contain a `scan_items` element, or if `set_scan_items` is called, then all scan items are replaced with the ones given. Alternatively `add_scan_item` can be used to add one or more scan items directly.

Lets change the scan item from above, so that the `(R1).k1` parameter is changed to the specific values of 0.5, 1.0 and 2.0:

```
[4]: set_scan_items([{'item': '(R1).k1', 'values': [0.5, 1.0, 2]}])
```

```
[5]: get_scan_items()
```

```
[5]: [{'type': 'scan',
      'num_steps': 0,
      'log': False,
      'min': 0.0,
      'max': 1.0,
      'values': [0.5, 1.0, 2.0],
      'use_values': True,
      'item': '(R1).k1',
      'cn': 'CN=Root,Model=The Brusselator,Vector=Reactions[R1],ParameterGroup=Parameters,
      ↪Parameter=k1'}]
```

scan items can be specified, either through their display name by specifying the `item` key, or by specifying the `cn` directly.

The scan item can be of one of three types:

- **scan**: this is the default (so will be used if not specific), here a model element is varied either through values, or between a specified `min` and `max` value
- **repeat**: here the subtask is repeated `num_steps` times
- **random**: here the value for the specified model element is sampled from the specified distribution (which can be `uniform`, `normal`, `poisson` or `gamma`)

For example to specify a repeat you'd use:

```
[6]: add_scan_item(type='repeat', num_steps=10)
```

```
[7]: get_scan_items()
```

```
[7]: [{'type': 'scan',
      'num_steps': 0,
      'log': False,
      'min': 0.0,
      'max': 1.0,
      'values': [0.5, 1.0, 2.0],
```

(continues on next page)



(continued from previous page)

```
'use_values': True,
'item': '(R1).k1',
'cn': 'CN=Root,Model=The Brusselator,Vector=Reactions[R1],ParameterGroup=Parameters,
↳Parameter=k1'},
{'type': 'repeat', 'num_steps': 10, 'log': None, 'min': None, 'max': None}]
```

as you can see, by using `add_scan_items`, the repeat item is added at the end of the scan list. In COPASI, when multiple scan items are defined, the semantics of those is that for each value of scan item 1, all values of scan item 2 are processed.

For scan items of type distribution, the min/max element take up different meaning:

- uniform: here the value is between the min and max value
- normal: min=mean and max=standard deviation
- poisson: min=mean and max has no meaning
- gamma: min=shape and max=scale

As a last example, lets change it to sample the initial concentration of species A from a normal distribution between around 2, and we want to do that 5 times:

```
[8]: set_scan_items([
    {
        'type': 'repeat',
        'num_steps': 5
    },
    {
        'item': '[A]_0',
        'type': 'random',
        'distribution': 'normal',
        'min': 2,
        'max': 0.1
    }
])
```

```
[9]: get_scan_items()
```

```
[9]: [{'type': 'repeat', 'num_steps': 5, 'log': None, 'min': None, 'max': None},
{'type': 'random',
'num_steps': 0,
'log': False,
'min': 2.0,
'max': 0.1,
'distribution': 'normal',
'item': '[A]_0',
'cn': 'CN=Root,Model=The Brusselator,Vector=Compartments[compartment],
↳Vector=Metabolites[A],Reference=InitialConcentration'}]
```

## 13.2 Runnin a scan:

You can run these scans in basico using the `run_scan` method. Where you can optionally pass along a `settings` parameter to reconfigure the scan, or an `output` selection, to grab some data directly.

So lets run the configured scan task, changing it to run a the steady state task, collecting the final concentrations of X and Y:

```
[10]: run_scan(settings={'subtask': T.STEADY_STATE}, output=['[A]_0', '[X]', '[Y]'])
```

```
[10]:
```

	[A]_0	[X]	[Y]
0	2.041315	2.041315	1.469639
1	1.925778	1.925778	1.557810
2	2.116724	2.116724	1.417283
3	2.088708	2.088708	1.436292
4	1.950328	1.950328	1.538201

## SENSITIVITY ANALYSIS

This notebook demonstrates how to use sensitivity analysis using `basico`.

We start as always by importing `basico`, and loading a model. Here I load the `brusselator` model from the examples.

```
[1]: from basico import *

[2]: load_example('brusselator');
     get_reactions()[['scheme']]

[2]:      scheme
name
R1      A -> X
R2    2 * X + Y -> 3 * X
R3      X + B -> Y + D
R4      X -> E
```

### 14.1 Settings

The settings for the sensitivities task are controlled using the functions `get_sensitivity_settings()` and `set_sensitivity_settings()`. The core attributes to change here would be:

- the subtask to use (can be one of 'Evaluation', 'Steady State', 'Time Series', 'Parameter Estimation', 'Optimization', 'Cross Section'). They are available in the `SENS_ST` constant class.
- `effect`: this specifies the element (or elements that we want to observe)
- `cause` and `secondary_cause`: these are the elements to be varied, that we expect to have an effect on the observed element.

If we look at the current settings, we see that the steady state subtask is run to observe what happens to non-constant concentration of species when parameter values are varied.

```
[3]: get_sensitivity_settings()

[3]: {'scheduled': False,
     'update_model': False,
     'method': {'Delta factor': 0.001,
                'Delta minimum': 1e-12,
                'name': 'Sensitivities Method'},
     'report': {'filename': '',
                'report_definition': 'Sensitivities',
                'append': True,
```

(continues on next page)

(continued from previous page)

```
'confirm_overwrite': False},
'sub_task': 'Steady State',
'effect': 'Non-Constant Concentrations of Species',
'cause': 'All Parameter Values',
'secondary_cause': 'Not Set'}
```

Valid values for `cause` and `effect` are either a specific element, specified by using either the display name or the CN of an element. Or an element from the SENS constant class.

## 14.2 Running the Analysis

`run_sensitivities()` will run the sensitivity task, after which the result is available by calling:

- `get_scaled_sensitivities()`
- `get_unscaled_sensitivities()`
- `get_summarized_sensitivities()`

```
[4]: run_sensitivities()
```

```
[5]: get_scaled_sensitivities()
```

```
[5]:      (R1).k1  (R2).k1      (R3).k1  (R4).k1
[X]  1.000000  0.000000  4.440893e-13 -0.999001
[Y] -0.999001 -0.999001  1.000000e+00  1.000000
```

```
[6]: get_unscaled_sensitivities()
```

```
[6]:      (R1).k1  (R2).k1      (R3).k1  (R4).k1
[X]  0.500000  0.000000  2.220446e-13 -0.499500
[Y] -5.993999 -5.993999  5.999993e+00  5.999993
```

```
[7]: get_summarized_sensitivities()
```

```
[7]:      0
(R1).k1  1.413507
(R2).k1  0.999001
(R3).k1  1.000000
(R4).k1  1.413507
```

Convenience overloads exist, so that the settings can directly be passed on to the run method. If `run_first=True` is given to the functions retrieving the result, the sensitivities will be computed before obtaining the result.

So for example, if we just wanted to see how the initial concentration of A would affect the transient concentration of X, we could directly call:

```
[8]: get_scaled_sensitivities(run_first=True, settings={'effect': '[X]', 'cause': '[A]_0'})
```

```
[8]: 1.000000000000004075
```

To see what effect it would have on all metabolite concentrations:

```
[9]: get_scaled_sensitivities(run_first=True,  
                             settings={'effect': SENS.NON_CONST_METAB_CONCENTRATIONS,  
                                       'cause': '[A]_0'})
```

```
[9]:      0  
[X]  1.000000  
[Y] -0.999001
```

The changed settings of the last run will persist, so that it can be reproduced in the COPASI user interface as well:

```
[10]: get_sensitivity_settings()
```

```
[10]: {'scheduled': False,  
      'update_model': False,  
      'method': {'Delta factor': 0.001,  
                 'Delta minimum': 1e-12,  
                 'name': 'Sensitivities Method'},  
      'report': {'filename': '',  
                 'report_definition': 'Sensitivities',  
                 'append': True,  
                 'confirm_overwrite': False},  
      'sub_task': 'Steady State',  
      'effect': 'Non-Constant Concentrations of Species',  
      'cause': '[A]_0',  
      'secondary_cause': 'Not Set'}
```



## WORKING WITH ARRAYS OF COMPARTMENTS

COPASI allows working with arrays of compartments, that is taking a model that is set up, and multiplying it either in a linear chain, or in an rectangular array. This example shows how to do that using basico. We start with the usual imports:

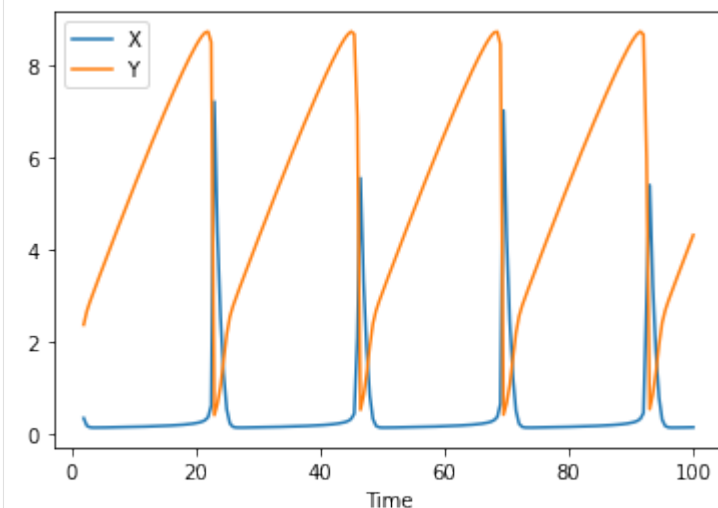
```
[1]: from basico import *
```

now we import the brusselator example model

```
[2]: dm = load_example('brusselator')
```

and obviously, we can run time courses just as usual:

```
[3]: data = run_time_course()  
data.plot();
```



For initialization it is useful, to add events to the model. The parameters are:

- name
- trigger expression
- array of arrays for assignments, with the inner array containing a target, and an expression what to assign.

for example the following would add an event e2 that triggers at time 10, and assigns 10 to X

```
[4]: add_event('e0', 'Time > 30', [['X', '10']]);
```

to display what events we have use `get_events()`

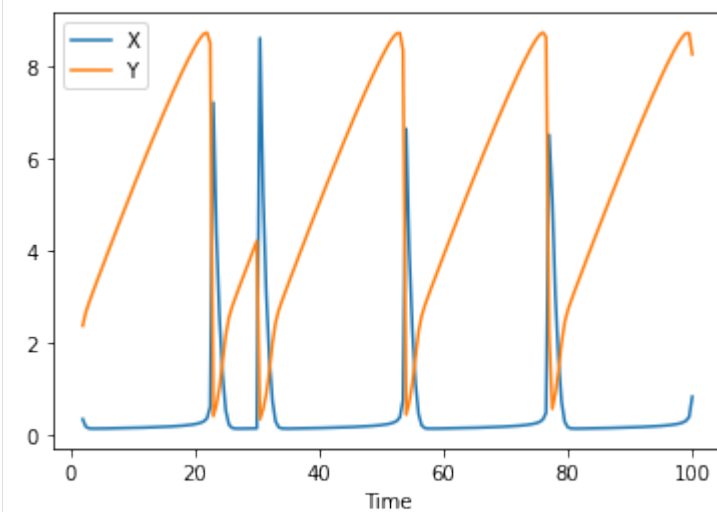
```
[5]: get_events()
```

```
[5]:      trigger delay      assignments      key \
name
e0    Time > 30      [{'target': '[X]', 'expression': '10'}] Event_0

      sbml_id
name
e0
```

so running the time course again, we do see that at time 30, the event is triggered:

```
[6]: run_time_course().plot();
```



## 15.1 Linear arrays of compartments

now lets run this model in a linear chain of 15 cells, allowing the species X and Y to diffuse linearly between them, with diffusion coefficients 0.16 and 0.08 respectively. We also remove the existing compartment afterwards, as it will not be part of the linear chain:

```
[7]: create_linear_array(15, ['X', 'Y'], [0.16, 0.8], delete_template=True)
```

now we have many more species, compartments, and reactions, all parameterized exactly as the template model was

```
[8]: print(overview())
get_species().head()
```

```
Name:          The Brusselator
# Compartments: 15
# Species:      90# Parameters:  2# Reactions:  88# Events:    1
```

```
[8]:      compartment      type      unit  initial_concentration \
name
X      compartment[0]  reactions  mmol/ml                2.999996
```

(continues on next page)



(continued from previous page)

E	compartment[0]	fixed	mmol/ml	0.000000
B	compartment[0]	fixed	mmol/ml	2.999996
D	compartment[0]	fixed	mmol/ml	0.000000
Y	compartment[0]	reactions	mmol/ml	2.999996

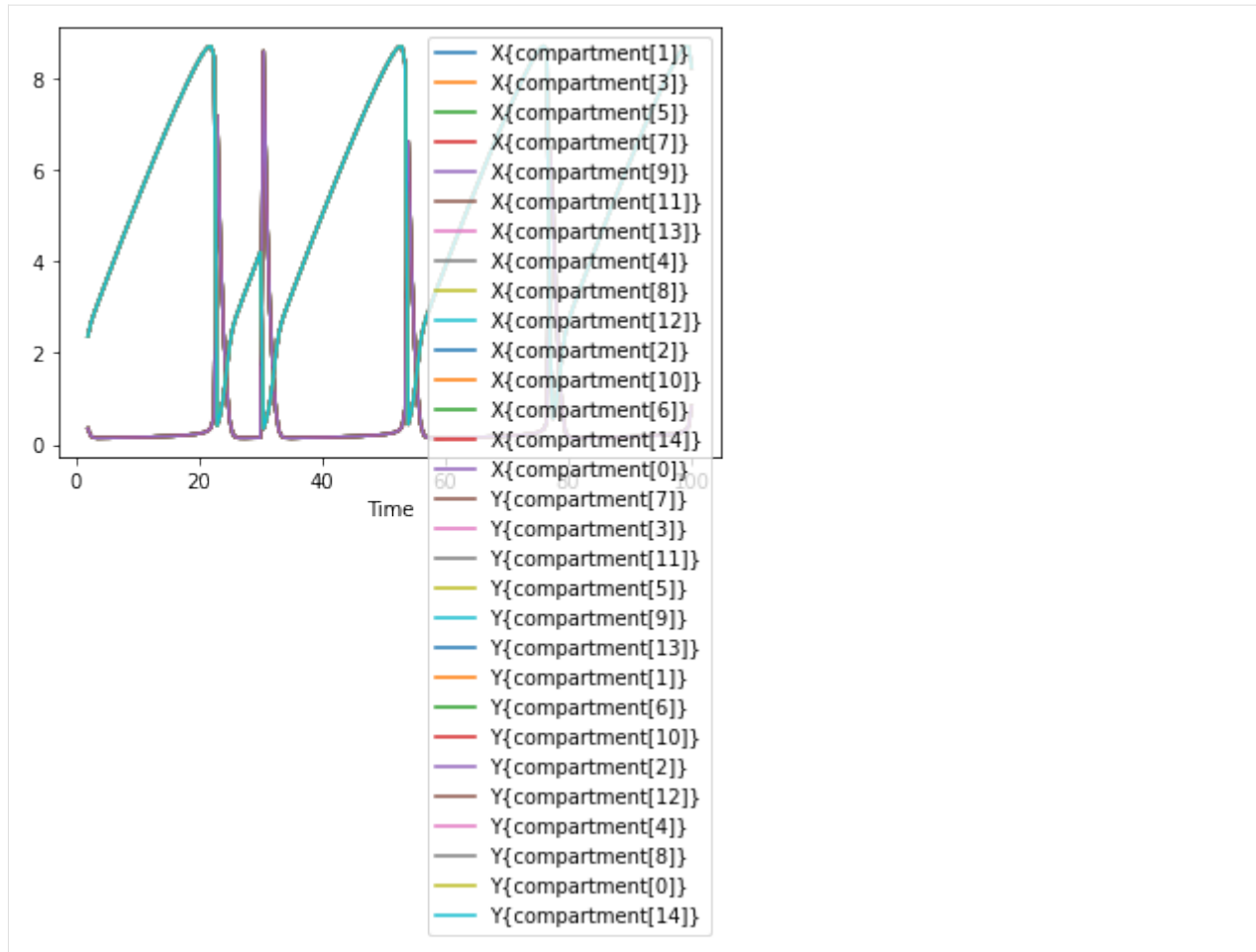
	initial_particle_number	initial_expression	expression	concentration	\
name					
X	1.806640e+21			NaN	
E	0.000000e+00			NaN	
B	1.806640e+21			NaN	
D	0.000000e+00			NaN	
Y	1.806640e+21			NaN	

	particle_number	rate	particle_number_rate	key	sbml_id
name					
X	NaN	0.0	0.0	Metabolite_6	
E	NaN	0.0	0.0	Metabolite_7	
B	NaN	0.0	0.0	Metabolite_8	
D	NaN	0.0	0.0	Metabolite_9	
Y	NaN	0.0	0.0	Metabolite_10	

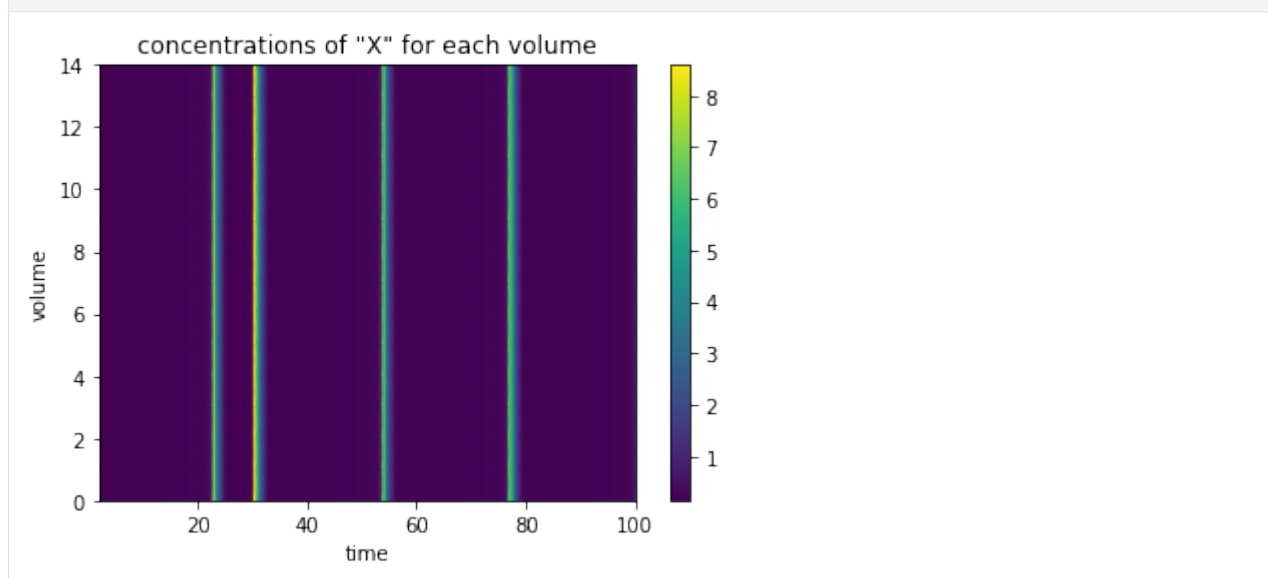
lets simulate again:

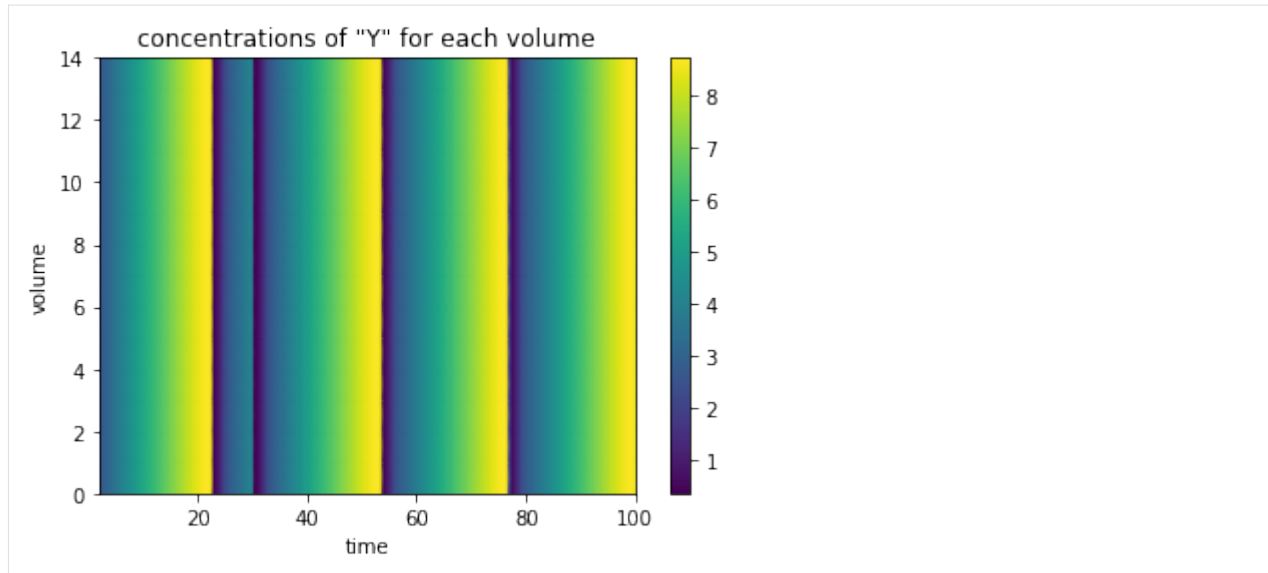
```
[9]: data = run_time_course()
data.plot();
```



Seeing all these lines plotted over each other might not be too helpful, so we can also plot this as a color mesh, with the time on the x axis, and the individual created compartments along the y axis.

```
[10]: plot_linear_time_course(data, 'compartment');
```





## 15.2 Creating 2D array of compartments

the same can be done by creating a rectangular array of compartments. Note that this results in quickly getting large amounts of model variables, so it might prove prohibitive to be used in numbers. So i'll start over, loading the model again (as i don't want the linear chain anymore)

```
[ ]: dm = load_example('brusselator')
      create_rectangular_array(15, 15, ['X', 'Y'], [0.16, 0.8], delete_template=True)
      print(overview())
```

Lots of variables indeed. So simulation will take a bit longer. But before we come to that, i would like to change the concentration of X at time 10 of the simulation, just so that we don't just use uniform values (that would be boring to look at. So i add the events as before:

```
[ ]: add_event('change_x', 'Time > 10', [['X{compartment[2,2]]', '10'],
                                           ['X{compartment[2,3]]', '10'],
                                           ['X{compartment[3,2]]', '10'],
                                           ['X{compartment[3,3]]', '10']
                                           ])
```

```
[ ]: data = run_time_course(duration=100, start_time=0)
```

plotting all times of the time course might not be too interesting, so instead i select plots only at specific times. I also override the min / max scaling of the plots, to the range between 0 and 10, if those are not given the data will be scaled to each min / max value.

```
[ ]: plot_rectangular_time_course(data, times=[10.5, 30, 60, 90], min_range=0, max_range=10);
```

To display it as animation we'll need an extra import, and you'll need to have ffmpeg installed

```
[ ]: from IPython.display import HTML
      HTML(animate_rectangular_time_course(data, metab='Y', min_range=0, max_range=10).to_
      ↪html5_video())
```



## USING CALLBACKS

When using `basico`, usually there is no feedback given when running tasks. That is fine when tasks don't take a long time, but for optimizations or parameter estimation runs, it would be nice to know how long things could go. Setting a callback also allows for interrupting a running task through interrupting by pressing `Ctrl+C`. Let's see how this works. We start as always by importing `basico`. Additionally we also import the `callbacks` module (since I consider it an experimental module at the time of writing this, it is not done automatically):

```
[1]: from basico import *  
from basico.callbacks import create_default_handler
```

For now I have wrapped the `tqdm` library, others could easily be added later on. Let me know if you have a preference! The `create_default_handler` function sets up a `tqdm` handler, that will be used for all longer running operations. It takes the following arguments:

- `delay`: delay in seconds before showing the first message (defaults to 1)
- `leave`: boolean flag, that indicates whether messages should remain on screen after being completed (defaults to `False`)
- `unit`: a string to display as unit for the iterations (by default `tqdm` will use `it/s`)

additional key value pairs will be passed on to `tqdm`. So let us just create a handler, and run some tasks:

```
[2]: create_default_handler()
```

we start with a simple example model that has parameter estimation set up on it already

```
[3]: load_example('LM');
```

it uses a local method and runs for 500 iterations, or until the tolerance of  $1e-16$  is reached:

```
[4]: get_task_settings(T.PARAMETER_ESTIMATION)['method']
```

```
[4]: {'Iteration Limit': 500,  
      'Tolerance': 1e-16,  
      'Stop after # Stalled Iterations': 0,  
      '#LogVerbosity': 0,  
      'name': 'Levenberg - Marquardt'}
```

```
[5]: run_parameter_estimation()
```

```
[5]:
```

	lower	upper	sol	affected
name				
(R1).k2	1e-6	1e6	0.0000002	[]
(R2).k1	1e-6	1e6	44.661715	[]

(continues on next page)

(continued from previous page)

```
Values[offset] -0.2 0.4 0.043018 [Experiment_1]
Values[offset] -0.2 0.4 0.054167 [Experiment_3]
Values[offset] -0.2 0.4 -0.050941 [Experiment]
Values[offset] -0.2 0.4 0.045922 [Experiment_4]
Values[offset] -0.2 0.4 0.048025 [Experiment_2]
```

lets choose just random search, which will take much longer, this will make the callbacks show up, and we will see the #function evaluations, as well as the best value reached. At any point we can interrupt the execution of the cell (by presing the stop button on the notebook), this will cause the parameter estimation to stop after the next iteration, with partial results being returned:

```
[6]: run_parameter_estimation(method=PE.RANDOM_SEARCH)
```

```
Function Evaluations: 1311 [00:01, 1309.69/s]
```

```
Best Value: 22.32126236098639 [00:01, 2516.34/s]
```

```
[6]:
```

	lower	upper	sol	affected
name				
(R1).k2	1e-6	1e6	0.000123	[]
(R2).k1	1e-6	1e6	40.864396	[]
Values[offset]	-0.2	0.4	0.108320	[Experiment_1]
Values[offset]	-0.2	0.4	0.216139	[Experiment_3]
Values[offset]	-0.2	0.4	0.010154	[Experiment]
Values[offset]	-0.2	0.4	0.368608	[Experiment_4]
Values[offset]	-0.2	0.4	-0.057231	[Experiment_2]

to remove the default handler again, the `reset_default_handler` function can be used. As it turns out, the output of the handler could not be seen on the html version of the notebook. So here a screenshot of the operation in action.

## 16.1 Timeouts

callbacks can also be used, to ensure that tasks are automatically interrupted after a certain number of seconds. So if we repeat the example from above, and stop the parameter estimation automatically after 10seconds, we could use:

```
[7]: create_default_handler(max_time=10)
```

```
[8]: run_parameter_estimation(method=PE.RANDOM_SEARCH)
```

```
Function Evaluations: 1330 [00:01, 1328.85/s]
```

```
Best Value: 18.52026239524136 [00:01, 2366.75/s]
```

```
[8]:
```

	lower	upper	sol	affected
name				
(R1).k2	1e-6	1e6	0.000002	[]
(R2).k1	1e-6	1e6	43.882694	[]
Values[offset]	-0.2	0.4	-0.061506	[Experiment_1]
Values[offset]	-0.2	0.4	0.062814	[Experiment_3]
Values[offset]	-0.2	0.4	-0.016544	[Experiment]
Values[offset]	-0.2	0.4	0.001578	[Experiment_4]
Values[offset]	-0.2	0.4	0.096818	[Experiment_2]

## WORKING WITH PLOTS

This example describes how to read / edit plot specifications, so that they can appear in the COPASI User Interface.

```
[1]: import sys
if '../..' not in sys.path:
    sys.path.append('../..')
from basico import *
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

lets start with the Brusselator example, that we have in the examples

```
[2]: load_example('brusselator');
```

now let us have a look at the plots defined in the file:

```
[3]: get_plots()

[3]:          active  log_x  log_y  \
name
Concentrations, Volumes, and Global Quantity Va...  True  False  False
Phase Plot          True  False  False

          tasks  \
name
Concentrations, Volumes, and Global Quantity Va...
Phase Plot

↪      curves
name
Concentrations, Volumes, and Global Quantity Va...  [{'name': '[X]', 'type': 'curve2d',
↪ 'channels'...
Phase Plot          [{'name': '[Y]|[X]', 'type': 'curve2d
↪ ', 'chann...
```

To filter the plot specification, by name you can specify a substring to be used:

```
[4]: get_plots('Phase Plot')

[4]:          active  log_x  log_y  tasks  \
name
```

(continues on next page)

(continued from previous page)

```
Phase Plot      True  False  False

                                                    curves
name
Phase Plot  [{'name': '[Y]|[X]', 'type': 'curve2d', 'chann...
```

the plot specification can also be retrieved directly as dictionary:

```
[5]: get_plot_dict('Phase Plot')
```

```
[5]: {'name': 'Phase Plot',
      'active': True,
      'log_x': False,
      'log_y': False,
      'tasks': '',
      'curves': [{'name': '[Y]|[X]',
                    'type': 'curve2d',
                    'channels': ['[X]', '[Y]'],
                    'color': 'auto',
                    'line_type': 'lines',
                    'line_subtype': 'solid',
                    'line_width': 2.0,
                    'symbol': 'small_cross',
                    'activity': 'during'}]}
```

Now let us add a new plot. Ensure to compare with `:func:.set_plot_dict` and `:func:.set_plot_curves` for all parameters. Note that the only parameters necessary for the curve are a name for the curve (which will be displayed in the legend), and the data channels of what information to collect. These channels again use the display names of the elements that you want to plot (so `[X]` for the concentration of species `X`).

```
[6]: add_plot('test plot', curves=[{'name': 'x vs time', 'color': '#ff8800', 'channels': ['Time
↪', '[X]']}]];
```

and verify, that it is there and has gotten all the default values expected:

```
[7]: get_plot_dict('test plot')
```

```
[7]: {'name': 'test plot',
      'active': True,
      'log_x': False,
      'log_y': False,
      'tasks': '',
      'curves': [{'name': 'x vs time',
                    'type': 'curve2d',
                    'channels': ['Time', '[X]'],
                    'color': '#ff8800',
                    'line_type': 'lines',
                    'line_subtype': 'solid',
                    'line_width': 2.0,
                    'symbol': 'small_cross',
                    'activity': 'during'}]}
```

changing is possible as well, by using `set_plot_dict` directly.

finally the plot can also be deleted using `remove_plot`.



```
[8]: remove_plot('test plot')
```



## WORKING WITH REPORTS

For the use case, where you are using `basico` to prepare files to run later (say on a cluster environment or such). It is often useful to manipulate the report definitions we store in COPASI files. This notebook demonstrate to work with them. Lets just start by importing `basico`.

```
[1]: from basico import *
```

here we simply load the brusselator example:

```
[2]: load_example('brusselator');
```

### 18.1 Retrieving Report Definitions

The first function to look into is `get_reports` which will retrieve all report definition encoded in the file and returns its content as a data frame. You can filter by name, task or whether or not the reports is an automatically generated one. So while the following line reports there are 11 reports in the model, we can ignore them if we so choose (hence not getting a result printed at the end)

```
[3]: print('there are {0} reports in the model'.format(len(get_reports())))  
get_reports(ignore_automatic=True)
```

```
there are 11 reports in the model
```

lets look at the reports that apply to steady state tasks:

```
[4]: get_reports(task=T.STEADY_STATE)
```

```
[4]:      separator  precision      task  \  
name  
Steady-State      \t          6  Steady-State  
  
                                comment  is_table  \  
name  
Steady-State  <body xmlns="http://www.w3.org/1999/xhtml">\n ...      False  
  
            header  body                                footer  
name  
Steady-State      []  []  [CN=Root,Vector=TaskList[Steady-State]]
```

analoge as to when working with plots, the reports can be retrieved as dictionaries, by specifying the name of the report to retrieve. The dictionary will either include a key `body` with lists of elements to collect (in that case `is_table` will

be True), or as seen below, a header, body and footer entries. Each entry in those lists can be either a display name, or common name (as not all elements can be retrieved via names).

The key thing to keep in mind, is that header entries will be collected before running the task, body and table entries during running of the task. And footer entries, once the task is complete.

```
[5]: get_report_dict('Steady-State')
[5]: {'name': 'Steady-State',
      'separator': '\t',
      'precision': 6,
      'task': 'Steady-State',
      'comment': '<body xmlns="http://www.w3.org/1999/xhtml">\n          Automatically_\n          generated report.\n          </body>',
      'is_table': False,
      'header': [],
      'body': [],
      'footer': ['CN=Root,Vector=TaskList[Steady-State]']}
```

## 18.2 Adding Reports

lets add a new report for a custom time course in which we collect the models time, the concentration of the species X and their rate of change:

```
[6]: add_report('X Time-Course', task=T.TIME_COURSE, table=['Time', '[X]', 'X.Rate']);
```

```
[7]: get_report_dict('X Time-Course')
[7]: {'name': 'X Time-Course',
      'separator': '\t',
      'precision': 6,
      'task': 'Time-Course',
      'comment': '',
      'is_table': True,
      'print_headers': True,
      'table': ['Time', '[X]', 'X.Rate']}
```

the value in `print_headers` indicates, whether the name of the elements will be printed in the header column.

using the `table` element directly, does only work for tasks that generate output *during* the execution of the task, as in the time course example above. Other tasks, such as steady state computation only provide results *after* the task has completed. In the following we add a report for the steady state concentration (and rate of X). We use the function `wrap_copasi_string`, to indicate that we want to have the literal string `[X]` in the header, rather than the initial concentration.

One other important thing to note, is that when specifying headers manually, the separators also need to be included manually. That can be done using the separator char manually, or the special `String Separator=\t` to indicate that it is the separator of the report. Alternatively, you can specify the boolean flag `add_sepratator=True`, so that between header, footer and body entries the separator is automatically added.

```
[9]: add_report('X Steady-State', task=T.STEADY_STATE,
               header=[wrap_copasi_string('[X]'), wrap_copasi_string('X.Rate')],
               footer=['[X]', 'X.Rate'], add_sepratator=True);
```

```
[10]: get_report_dict('X Steady-State')
[10]: {'name': 'X Steady-State',
      'separator': '\t',
      'precision': 6,
      'task': 'Steady-State',
      'comment': '',
      'is_table': False,
      'header': ['String=\\[X\\]', 'Separator=\t', 'String=X.Rate'],
      'body': [],
      'footer': ['[X]', 'Separator=\t', 'X.Rate']}
```

## 18.3 Changing values

of course all the values can be changed using the `set_report_dict` function, say we wanted to collect CSV information for the plot above, we would specify to use `,` as separator:

```
[11]: set_report_dict('X Time-Course', separator=',')
```

```
[12]: get_report_dict('X Time-Course')
[12]: {'name': 'X Time-Course',
      'separator': ',',
      'precision': 6,
      'task': 'Time-Course',
      'comment': '',
      'is_table': True,
      'print_headers': True,
      'table': ['Time', '[X]', 'X.Rate']}
```

if you at any point set the header, body or footer, the table entries will be removed.

## 18.4 Assigning to task

In order to use a report definition, it has to be assigned to a task. COPASI will only create the report if a filename is assigned to it.

```
[13]: assign_report('X Time-Course', task=T.TIME_COURSE, filename='out.txt', append=True)
```

Alternatively, it is also possible to assign a report directly using the `set_task_settings` method, so for example:

```
[14]: set_task_settings(T.TIME_COURSE,
      settings = {'report':
                  {'report_definition': 'X Time-Course'}})
```

that with `get_task_settings` you can query which report is assigned to a given task:

```
[15]: get_task_settings(T.TIME_COURSE)['report']['report_definition']
[15]: 'X Time-Course'
```



## WORKING WITH SBML IDS

Usually `basico` uses the COPASI display names, to work with model elements. That way a consistent naming scheme between the COPASI graphical user interface, and the scripts can be easily maintained. However, for someone inspecting an SBML model, it might be convenient to also look at the SBML ids and identify elements that way. For this reason the data frames returned for compartments, events, parameters, species and reactions now also contain a column `sbml_id`.

Lets start as usual with the common imports:

```
[1]: import sys
      if '../..' not in sys.path:
          sys.path.append('../..')
      from basico import *
      import numpy as np
      import matplotlib.pyplot as plt
      %matplotlib inline
```

Next lets load a model from the BioModels Database, and look at the elements:

```
[2]: load_biomodel(64);
```

Now we have not just the element name available, but also their respective `sbml_id`:

```
[3]: get_species()[['sbml_id', 'initial_concentration']]
```

```
[3]:
```

	sbml_id	initial_concentration
name		
High energy phosphates	P	6.310000
Glucose 6 Phosphate	G6P	2.450000
Triose-phosphate	TRIO	0.960000
NAD	NAD	1.200000
Acetaldehyde	ACE	0.170000
2-phosphoglycerate	P2G	0.120000
1,3-bisphosphoglycerate	BPG	0.000000
Glucose in Cytosol	GLCi	0.087000
Fructose 6 Phosphate	F6P	0.620000
Phosphoenolpyruvate	PEP	0.070000
Pyruvate	PYR	1.850000
Fructose-1,6 bisphosphate	F16P	5.510000
3-phosphoglycerate	P3G	0.900000
NADH	NADH	0.390000
ATP concentration	ATP	2.509190
ADP concentration	ADP	1.291619

(continues on next page)

(continued from previous page)

AMP concentration	AMP	0.299190
Extracellular Glucose	GLCo	50.000000
Glycogen	Glyc	0.000000
Trehalose	Trh	0.000000
CO2	CO2	1.000000
Succinate	SUCC	0.000000
Ethanol	ETOH	50.000000
Glycerol	GLY	0.150000
sum of AXP conc	SUM_P	4.100000
F2,6P	F26BP	0.020000

similarly we can get the elements by SBML id as well:

```
[4]: get_species(sbml_id='ATP')
```

```
[4]:      compartment      type      unit      initial_concentration \
name
ATP concentration      cytosol      assignment      mmol/l      2.50919

      initial_particle_number      initial_expression \
name
ATP concentration      1.511070e+21

      expression \
name
ATP concentration      ( [High energy phosphates] - [ADP concentratio...

      concentration      particle_number      rate      particle_number_rate \
name
ATP concentration      NaN      NaN      NaN      NaN

      key      sbml_id
name
ATP concentration      Metabolite_21      ATP
```

Whereas in COPASI each element has a concentration and a particle number, in SBML usually elements deal only with concentrations and amounts. To make it easy to access them, it is convenient to add the expressions for the amount to the model, so that they can be accessed at any point in time. For that a utility function exists. If `use_sbml_ids` is specified, the sbml id of the species will be used in the name (i.e: `amount(sbml_id)`), otherwise it will be named `amount(display name)`. In case `ignore_fixed` is specified, no expressions for fixed species will be created, and similarly assignment expressions can be ignored:

```
[5]: add_amount_expressions(use_sbml_ids=True, ignore_fixed=True)
```

lets look at the expressions created, we see it is just the concentration multiplied with the compartment size the species is in:

```
[6]: get_parameters(name='amount(')[['initial_value', 'expression']]
```

```
[6]:      initial_value      expression
name
amount(GLCi)      0.087000 [Glucose in Cytosol] * Compartments[cytosol].V...
amount(G6P)      2.450000 [Glucose 6 Phosphate] * Compartments[cytosol]...
amount(F6P)      0.620000 [Fructose 6 Phosphate] * Compartments[cytosol]...
```

(continues on next page)



(continued from previous page)

```

amount(F16P)      5.510000 [Fructose-1,6 biphosphate] * Compartments[cyt...
amount(TRIO)      0.960000 [Triose-phosphate] * Compartments[cytosol].Volume
amount(BPG)       0.000000 [1,3-bisphosphoglycerate] * Compartments[cytos...
amount(P3G)       0.900000 [3-phosphoglycerate] * Compartments[cytosol].V...
amount(P2G)       0.120000 [2-phosphoglycerate] * Compartments[cytosol].V...
amount(PEP)       0.070000 [Phosphoenolpyruvate] * Compartments[cytosol]...
amount(PYR)       1.850000 [Pyruvate] * Compartments[cytosol].Volume
amount(ACE)       0.170000 [Acetaldehyde] * Compartments[cytosol].Volume
amount(P)         6.310000 [High energy phosphates] * Compartments[cytoso...
amount(NAD)       1.200000 [NAD] * Compartments[cytosol].Volume
amount(NADH)      0.390000 [NADH] * Compartments[cytosol].Volume
amount(ATP)       2.509190 [ATP concentration] * Compartments[cytosol].Vo...
amount(ADP)       1.291619 [ADP concentration] * Compartments[cytosol].Vo...
amount(AMP)       0.299190 [AMP concentration] * Compartments[cytosol].Vo...

```

the `run_time_course` function now also takes a parameter to use sbml id's if they are present (it will still use the display names in case an element has no sbml id).

```
[7]: run_time_course(use_sbml_id=True)
```

```

[7]:
Time      P      G6P      TRIO      NAD      ACE      P2G      BPG  \
0.00  6.310000  2.450000  0.960000  1.200000  0.170000  0.120000  0.000000
0.01  6.530115  2.491309  2.325367  1.055557  0.011908  0.051758  0.000412
0.02  6.481760  2.428287  2.355642  1.010053  0.012240  0.039849  0.000327
0.03  6.419771  2.357805  2.326751  1.033128  0.013500  0.040591  0.000326
0.04  6.465266  2.295451  2.288924  1.076765  0.015151  0.043923  0.000361
...      ...      ...      ...      ...      ...      ...      ...
0.96  6.470716  1.124734  0.799562  1.550632  0.195576  0.049470  0.000391
0.97  6.463801  1.120309  0.798489  1.550554  0.195101  0.049284  0.000388
0.98  6.457177  1.116112  0.797470  1.550473  0.194615  0.049106  0.000385
0.99  6.450834  1.112132  0.796503  1.550390  0.194118  0.048936  0.000383
1.00  6.444763  1.108357  0.795584  1.550304  0.193613  0.048772  0.000380

Time      GLCi      F6P      PEP  ...  Values[amount(P2G)]  \
0.00  0.087000  0.620000  0.070000  ...  0.120000
0.01  0.097408  0.351027  0.075957  ...  0.051758
0.02  0.097234  0.341946  0.062268  ...  0.039849
0.03  0.099157  0.328112  0.062661  ...  0.040591
0.04  0.098315  0.323283  0.068018  ...  0.043923
...      ...      ...      ...  ...  ...
0.96  0.094281  0.127878  0.084961  ...  0.049470
0.97  0.094464  0.127152  0.084495  ...  0.049284
0.98  0.094640  0.126464  0.084049  ...  0.049106
0.99  0.094809  0.125811  0.083622  ...  0.048936
1.00  0.094971  0.125191  0.083214  ...  0.048772

Time      Values[amount(PEP)]  Values[amount(PYR)]  Values[amount(ACE)]  \
0.00      0.070000      1.850000      0.170000
0.01      0.075957      3.176736      0.011908
0.02      0.062268      3.864468      0.012240

```

(continues on next page)

(continued from previous page)

```

0.03      0.062661      4.355273      0.013500
0.04      0.068018      4.792030      0.015151
...
0.96      0.084961      9.734328      0.195576
0.97      0.084495      9.698833      0.195101
0.98      0.084049      9.663914      0.194615
0.99      0.083622      9.629595      0.194118
1.00      0.083214      9.595898      0.193613

```

```

      Values[amount(P)]  Values[amount(NAD)]  Values[amount(NADH)]  \
Time
0.00      6.310000      1.200000      0.390000
0.01      6.530115      1.055557      0.534443
0.02      6.481760      1.010053      0.579947
0.03      6.419771      1.033128      0.556872
0.04      6.465266      1.076765      0.513235
...
0.96      6.470716      1.550632      0.039368
0.97      6.463801      1.550554      0.039446
0.98      6.457177      1.550473      0.039527
0.99      6.450834      1.550390      0.039610
1.00      6.444763      1.550304      0.039696

```

```

      Values[amount(ATP)]  Values[amount(ADP)]  Values[amount(AMP)]
Time
0.00      2.509190      1.291619      0.299190
0.01      2.669381      1.191352      0.239267
0.02      2.633713      1.214334      0.251953
0.03      2.588384      1.243002      0.268613
0.04      2.621609      1.222048      0.256343
...
0.96      2.625605      1.219506      0.254889
0.97      2.620535      1.222730      0.256734
0.98      2.615684      1.225809      0.258507
0.99      2.611044      1.228747      0.260209
1.00      2.606606      1.231551      0.261843

```

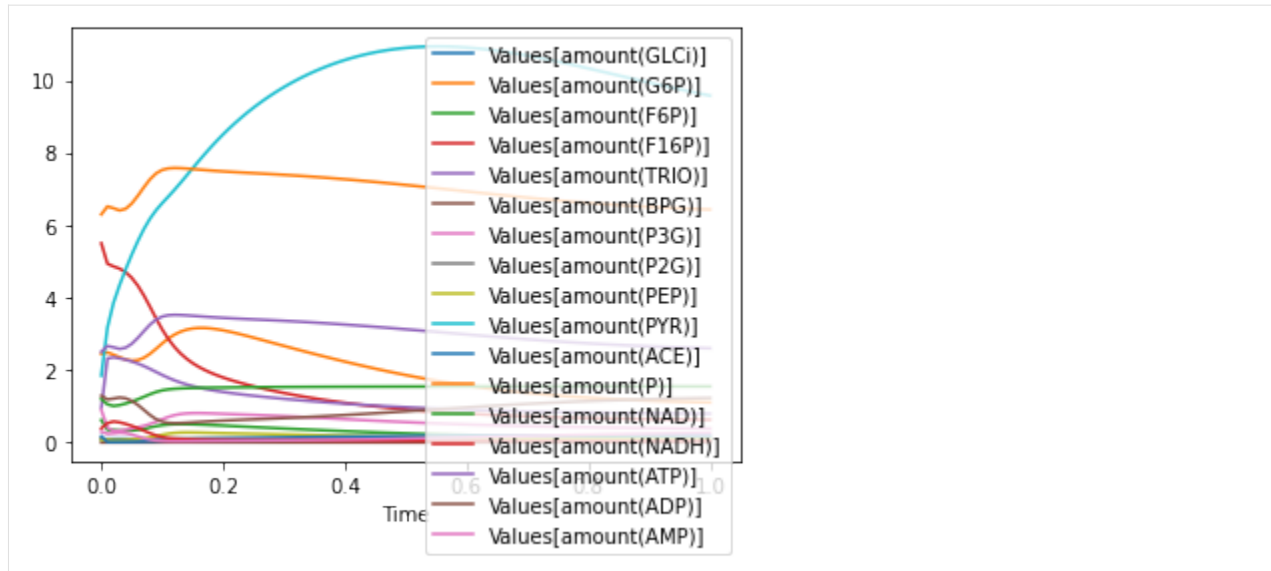
```
[101 rows x 34 columns]
```

```
[8]: df = run_time_course()
```

so lets plot just the amounts we got:

```
[9]: amount_columns = list(df.columns)
amount_columns = [name for name in amount_columns if 'amount(' in name]
```

```
[10]: df[amount_columns].plot();
```

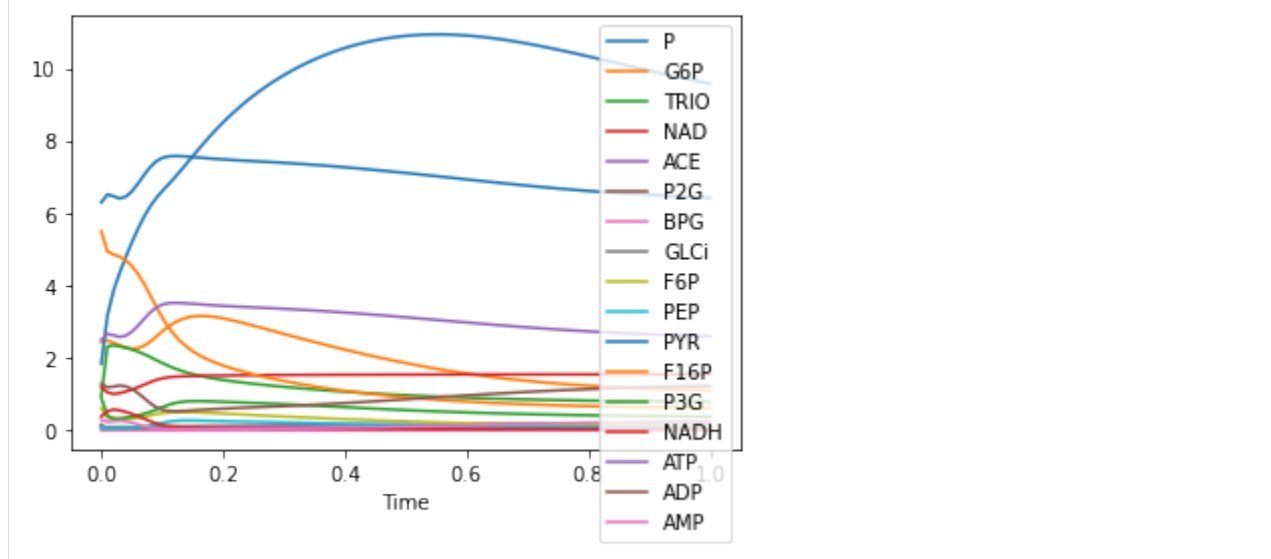


of course the added global parameters can be easily removed:

```
[11]: remove_amount_expressions()
```

and now we can plot the concentrations:

```
[14]: run_time_course(use_sbml_id=True).plot();
```



```
[ ]:
```



## WORKING WITH ANNOTATIONS

COPASI and SBML files can be annotated with [MIRIAM annotations](#). These annotations describe what the model and its constituents actually represent. In `basico` you can ask for these annotations using the `get_miriam_annotation` function, similarly, you can set them using the `set_miriam_annotation` commands.

So let's start as usual, loading `basico`:

```
[1]: import sys
    if '../..' not in sys.path:
        sys.path.append('../..')

    from basico import *
    from ipywidgets import HTML
    %matplotlib inline
```

### 20.1 Resource Lists / Updating

For the resolving of MIRIAM entries to work, occasionally COPASI needs to update the list of MIRIAM resources. This will be necessary on systems, where the graphical user interface to COAPSI has not been installed. To see whether it is necessary, you can call:

```
[2]: have_miriam_resources()
```

```
[2]: True
```

Should that call return false, simply update the resources using:

```
[3]: update_miriam_resources()
```

To see the list of terms currently known by COPASI you can use the function:

```
[4]: get_miriam_resources().head()
```

```
[4]:
```

	is_citation	uri
resource		
BIND	False	<a href="http://identifiers.org/bind">http://identifiers.org/bind</a>
ChEBI	False	<a href="http://identifiers.org/CHEBI">http://identifiers.org/CHEBI</a>
Ensembl	False	<a href="http://identifiers.org/ensembl">http://identifiers.org/ensembl</a>
Enzyme Nomenclature	False	<a href="http://identifiers.org/ec-code">http://identifiers.org/ec-code</a>
UniProt Knowledgebase	False	<a href="http://identifiers.org/uniprot">http://identifiers.org/uniprot</a>

and of course you can filter elements as usual, for example here to see the resources suitable for citations:

```
[5]: resources = get_miriam_resources()
resources[resources.is_citation == True]

[5]:
```

	is_citation	uri
resource		
PubMed	True	http://identifiers.org/pubmed
DOI	True	http://identifiers.org/doi
arXiv	True	http://identifiers.org/arxiv
ISBN	True	http://identifiers.org/isbn

## 20.2 Displaying Annotations:

Now we load a fully annotated model, to display its annotations:

```
[6]: load_biomodel(64);
```

Then we can display the Notes of the model:

```
[7]: display(HTML(get_notes()))
```

```
HTML(value='\n  <body xmlns="http://www.w3.org/1999/xhtml">\n    <p>\n      <b>Can yeast_
↪glycolysis be underst...
```

Similarly, we could get to all annotations. As with the `get_notes` call, if no element is specified, the annotations of the model element will be displayed:

```
[8]: get_miriam_annotation()

[8]: {'creators': [{'first_name': 'Lukas',
  'last_name': 'Endler',
  'email': 'lukas@ebi.ac.uk',
  'organization': 'EMBL-EBI'},
 {'first_name': 'Harish',
  'last_name': 'Dharuri',
  'email': 'hdharuri@cds.caltech.edu',
  'organization': 'California Institute of Technology'},
 {'first_name': 'Jacky L',
  'last_name': 'Snoep',
  'email': 'jls@sun.ac.za',
  'organization': 'Stellenbosh University'}],
 'references': [{'id': '10951190',
  'uri': 'http://identifiers.org/pubmed/10951190',
  'resource': 'PubMed',
  'description': ''}],
 'descriptions': [{'id': 'MODEL6623915522',
  'qualifier': 'is',
  'uri': 'http://identifiers.org/biomodels.db/MODEL6623915522',
  'resource': 'BioModels Database'},
 {'id': 'GO:0006096',
  'qualifier': 'is',
  'uri': 'http://identifiers.org/GO:0006096',
```

(continues on next page)

(continued from previous page)

```

    'resource': 'Gene Ontology'},
{'id': 'sce00010',
 'qualifier': 'is',
 'uri': 'http://identifiers.org/kegg.pathway/sce00010',
 'resource': 'KEGG Pathway'},
{'id': 'BIOMD00000000064',
 'qualifier': 'is',
 'uri': 'http://identifiers.org/biomodels.db/BIOMD00000000064',
 'resource': 'BioModels Database'},
{'id': 'REACT_723',
 'qualifier': 'is homolog to',
 'uri': 'http://identifiers.org/reactome/REACT_723',
 'resource': 'Reactome'},
{'id': '4932',
 'qualifier': 'has taxon',
 'uri': 'http://identifiers.org/taxonomy/4932',
 'resource': 'Taxonomy'}]],
'modifications': [datetime.datetime(2012, 7, 19, 18, 26, 7, tzinfo=tzutc())],
'created': datetime.datetime(2008, 9, 16, 14, 0, 6, tzinfo=tzutc())}

```

To display the annotations of a specific species, or reaction just enter its name as the element to get the information for. Here we do this for NADH to return the dictionary of descriptions as given:

```

[9]: get_miriam_annotation(name='NADH')
[9]: {'descriptions': [{ 'id': 'C00004',
    'qualifier': 'is',
    'uri': 'http://identifiers.org/kegg.compound/C00004',
    'resource': 'KEGG Compound'},
  { 'id': 'CHEBI:16908',
    'qualifier': 'is',
    'uri': 'http://identifiers.org/CHEBI:16908',
    'resource': 'ChEBI'}]}

```

## 20.3 Setting annotations

Lets start over with a new model, and annotate it as we go along:

```
[10]: new_model(name='simple model', notes='A simple decay model');
```

The `set_miriam_annotation` accepts the following arguments:

- **creators:** a list of creator dictionary entries with: `first_name`, `last_name`, `email` and `organization`
- **references:** a list of reference dictionary entries with: `id`, `uri`, `resource` and `description`
- **descriptions:** a list of description dictionary entries with: `qualifier`, `resource` and `id`
- **modifications:** a list of `DateTime` objects representing the modification dates
- **created:** a `DateTime` object representing the creation date
- **replace:** a boolean indicating, whether the current annotation entries should be replaced ('True' default), or if new entries should be added to the existing ones ('False')

```
[11]: set_miriam_annotation(creators=[
    {
        'first_name': 'Frank',
        'last_name': 'Bergmann',
        'email': 'fbergman@caltech.edu',
        'organization': 'Heidelberg University'
    }], replace=True)
```

```
[12]: get_miriam_annotation()
```

```
[12]: {'creators': [{'first_name': 'Frank',
    'last_name': 'Bergmann',
    'email': 'fbergman@caltech.edu',
    'organization': 'Heidelberg University'}],
    'created': datetime.datetime(2023, 7, 6, 9, 20, 43, tzinfo=tzutc())}
```

```
[13]: set_miriam_annotation(descriptions=[
    {
        'qualifier': 'is',
        'resource': 'BioModels Database',
        'id': 'MODEL6623915522',
    }
])
```

```
[14]: get_miriam_annotation()
```

```
[14]: {'creators': [{'first_name': 'Frank',
    'last_name': 'Bergmann',
    'email': 'fbergman@caltech.edu',
    'organization': 'Heidelberg University'}],
    'descriptions': [{'id': 'MODEL6623915522',
    'qualifier': 'is',
    'uri': 'http://identifiers.org/biomodels.db/MODEL6623915522',
    'resource': 'BioModels Database'}],
    'created': datetime.datetime(2023, 7, 6, 9, 20, 43, tzinfo=tzutc())}
```

```
[15]: set_miriam_annotation(references=[
    {
        'id': '10951190',
        'resource': 'PubMed',
    }
])
```

```
[16]: get_miriam_annotation()
```

```
[16]: {'creators': [{'first_name': 'Frank',
    'last_name': 'Bergmann',
    'email': 'fbergman@caltech.edu',
    'organization': 'Heidelberg University'}],
    'references': [{'id': '10951190',
    'uri': 'http://identifiers.org/pubmed/10951190',
    'resource': 'PubMed',
    'description': ''}],
    'created': datetime.datetime(2023, 7, 6, 9, 20, 43, tzinfo=tzutc())}
```

(continues on next page)



(continued from previous page)

```
'descriptions': [{ 'id': 'MODEL6623915522',
  'qualifier': 'is',
  'uri': 'http://identifiers.org/biomodels.db/MODEL6623915522',
  'resource': 'BioModels Database'}],
'created': datetime.datetime(2023, 7, 6, 9, 20, 43, tzinfo=tzutc())}
```

```
[17]: set_miriam_annotation(modifications=[datetime.datetime.now(datetime.timezone.utc)],
↪ replace=True)
```

```
[18]: get_miriam_annotation()
```

```
[18]: {'creators': [{ 'first_name': 'Frank',
  'last_name': 'Bergmann',
  'email': 'fbergman@caltech.edu',
  'organization': 'Heidelberg University'}],
'references': [{ 'id': '10951190',
  'uri': 'http://identifiers.org/pubmed/10951190',
  'resource': 'PubMed',
  'description': ''}],
'descriptions': [{ 'id': 'MODEL6623915522',
  'qualifier': 'is',
  'uri': 'http://identifiers.org/biomodels.db/MODEL6623915522',
  'resource': 'BioModels Database'}],
'modifications': [datetime.datetime(2023, 7, 6, 9, 20, 43, 723553, tzinfo=tzutc())],
'created': datetime.datetime(2023, 7, 6, 9, 20, 43, tzinfo=tzutc())}
```



## WORKING WITH IPYPARALLEL

In this example we want to use `basico` from `ipython`, in a `ipyparallel`, as the normal approach with using `multiprocessing` does not seem to work across operating systems within the jupyter notebooks (read: i could not make it work on Windows).

This means, that to run this example, you will need the `ipyparallel` package installed, which can be achieved by uncommenting the following line:

```
[1]: #!/pip install -q ipyparallel
```

once that is installed, you will to switch to the home screen of the jupyter notebook, and start the cluster manually:

otherwise this notebook will fail with an error like:

```
-----
FileNotFoundError                                Traceback (most recent call last)
Cell In[3], line 1
----> 1 cluster = ipp.Cluster.from_file()
      2 rc = cluster.connect_client_sync()
      3 # wait to get the engines and print their id

File ~/env/lib/python3.11/site-packages/ipyparallel/cluster/cluster.py:556, in Cluster.
from_file(cls, cluster_file, profile, profile_dir, cluster_id, **kwargs)
    554 # ensure from_file preserves cluster_file, even if it moved
    555 kwargs.setdefault("cluster_file", cluster_file)
--> 556 with open(cluster_file) as f:
    557     return cls.from_dict(json.load(f), **kwargs)

FileNotFoundError: [Errno 2] No such file or directory:
'~/ipynotebook/profile_default/security/cluster-.json'
```

Once the `ipyparallel` is installed and the cluster is running continue as follows:

```
[2]: from basico import *
import ipyparallel as ipp
```

we could create a cluster manually, in my case i just created one in the cluster menu in the jupyter setup, so to access it we just need to run:

```
[3]: cluster = ipp.Cluster.from_file()
rc = cluster.connect_client_sync()
```

(continues on next page)

(continued from previous page)

```
# wait to get the engines and print their id
rc.wait_for_engines(6); rc.ids
```

```
[3]: [0, 1, 2, 3, 4, 5, 6, 7]
```

Now we create a direct view with all of the workers:

```
[4]: dview = rc[:]
```

The model we use is the BioModel 68, since we will need the model many times, i download it once and save it to a local file:

```
[5]: m = load_biomodel(68)
save_model('bm68.cps', model=m)
remove_datamodel(m)
```

And define the worker method, the worker just loads a biomodel (in case it was not loaded into the worker before, otherwise it accesses the current model). It then chooses a random initial concentration for 2 species, and computes the steady state of the model. Finally the initial concentrations used and the fluxes computed are return. In case a seed is specified, it will be used to initialize the rng:

```
[6]: def worker_method(seed=None):
    import basico
    import random
    if seed is not None:
        random.seed(seed)
    if basico.get_num_loaded_models() == 0:
        m = basico.load_model('bm68.cps')
    else:
        m = basico.get_current_model()

    # we sample the model as described in Mendes (2009)
    cysteine = 0.3 * 10 ** random.uniform(0, 3)
    adomed = random.uniform(0, 100)

    # set the sampled initial concentration.
    basico.set_species('Cysteine', initial_concentration=cysteine, model=m)
    basico.set_species('S-adenosylmethionine', initial_concentration=adomed, model=m)

    # compute the steady state
    _ = basico.run_steadystate(model=m)

    # retrieve the current flux values
    fluxes = basico.get_reactions(model=m).flux

    # and return as tuple
    return (cysteine, adomed, fluxes[1], fluxes[2])
```

We can invoke this method synchronusly in the notebook to obtain the result:

```
[7]: worker_method()
```

```
[7]: (1.5236813977490542,
      29.805870872058716,
```

(continues on next page)

(continued from previous page)

```
0.04158430196391732,
0.9584156980360827)
```

now we want to do that many times over (passing along the index to set the seed every time to get about the same result on my machine):

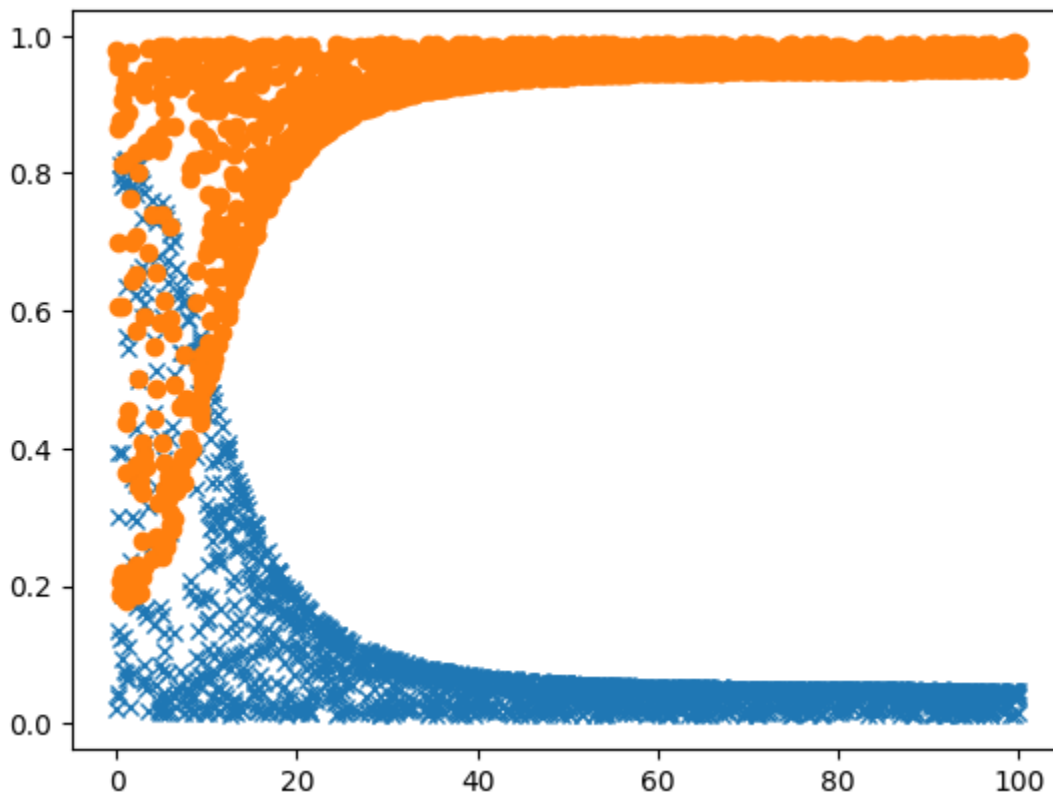
```
[8]: %%time
sync_results = []
for i in range(2000):
    sync_results.append(worker_method(i))

CPU times: user 2.06 s, sys: 55 ms, total: 2.12 s
Wall time: 2.08 s
```

here a utility method plotting the result:

```
[9]: def plot_result(results):
    cys = [x[0] for x in results]
    ado = [x[1] for x in results]
    y1 = [x[2] for x in results]
    y2 = [x[3] for x in results]
    plt.plot(ado, y1, 'x')
    plt.plot(ado, y2, 'o')
    plt.show()
```

```
[10]: plot_result(sync_results)
```



So that for me took some 10 seconds (even though the model was loaded already), so the next thing to do is to run it in parallel. Here i start 10000 runs, passing along the seed for each one, to make it reproducible:

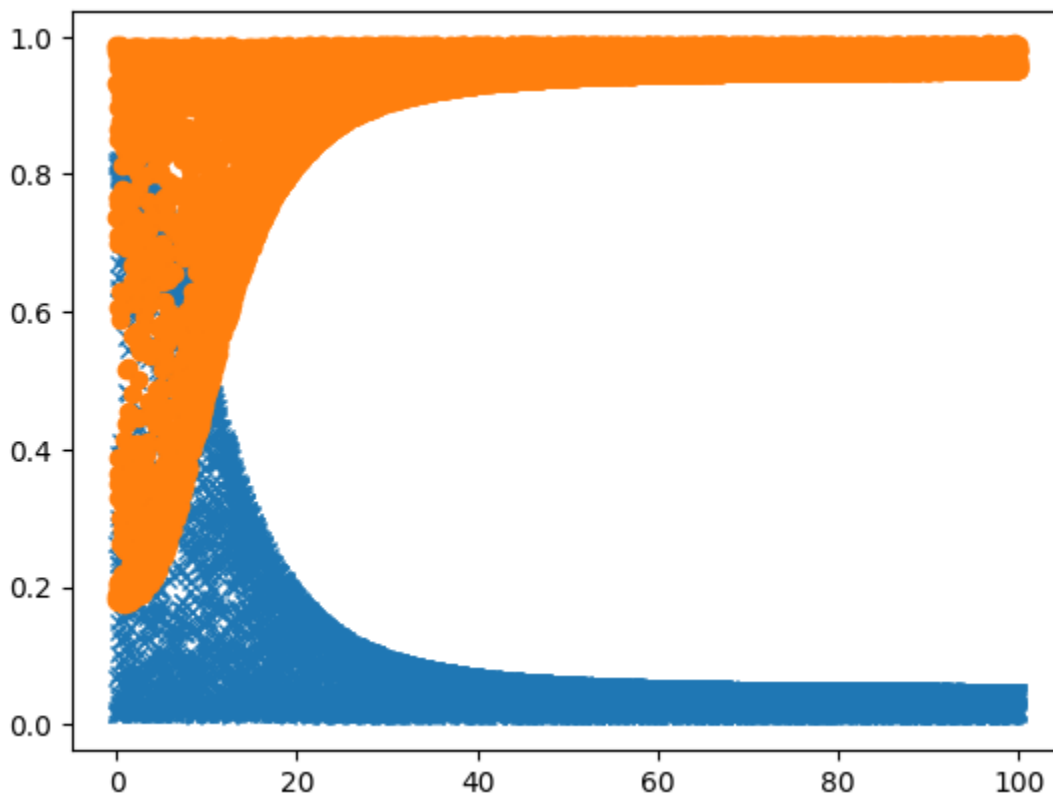
```
[11]: %%time
ar_map = dview.map_async(worker_method, [i for i in range(10000)])
ar_map.wait_for_output()
```

```
CPU times: user 412 ms, sys: 259 ms, total: 671 ms
Wall time: 3.72 s
```

```
[11]: True
```

And now we can plot that as well:

```
[12]: plot_result(ar_map.get())
```



## WORKING WITH PETAB

**PETab** is a standard file format, describing parameter estimation problems. While COPASI does not support the full feature set of PETab, with **basico** we can import PETab problems, and analyze them. This is an optional dependency, that is only installed, when **basico** is installed using:

```
pip install copasi-basico[petab]
```

assuming this is done, and that a petab problem is in the local directory, you can load it and analyze it like so:

```
[1]: from basico import *  
import basico.petab
```

```
using COPASI: 4.39.272 (Source)
```

within the `basico.petab` module we define a `PetabSimulator`, that automatically imports a PETab file, and transforms the experimental data files as needed by COPASI. It implements the `petab.simulate.Simulator` interface that can be used for other libraries using `libpetab`.

So lets start by loading a PETab problem

```
[2]: from petab import Problem  
pp = Problem.from_yaml('./Elowitz_Nature2000/Elowitz_Nature2000.yaml')
```

now lets instantiate the simulator with:

- `pp`: the petab problem
- `working_dir`: the directory where temp files can be stored

we are not specifying any additional optimization settings, so the parameter estimation task will be set to ‘Current Solution Statistics’, meaning that no actual optimization will be performed:

```
[3]: sim = basico.petab.PetabSimulator(pp, working_dir='./temp_dir/')
```

now lets simulate it:

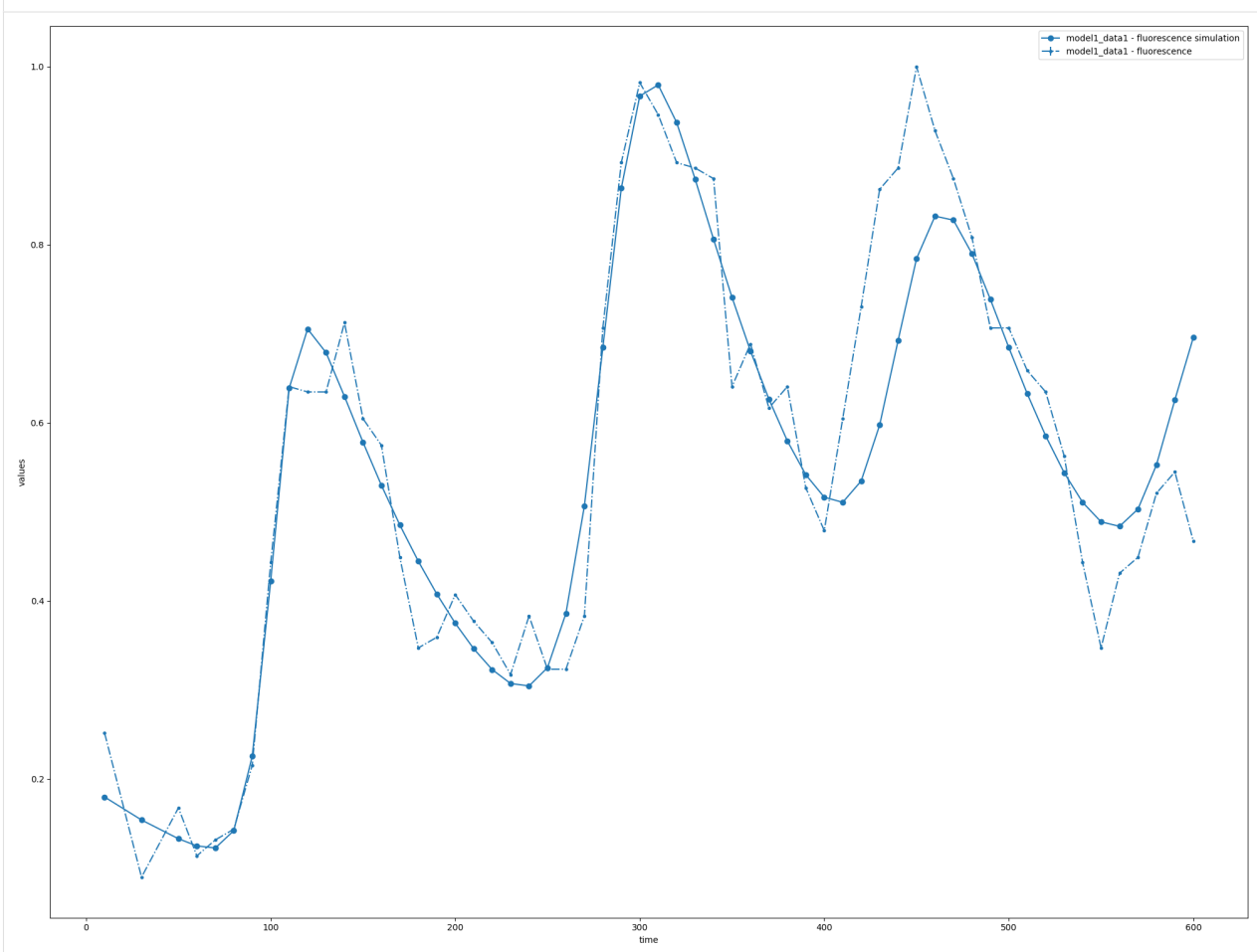
```
[4]: df = sim.simulate()
```

At this point the file can be opened directly in COPASI using `open_copasi()`, or here I use the visualization library from `libpetab`, to plot the problem and simulation obtained:

```
[5]: import petab.visualize
```

```
[6]: petab.visualize.plot_problem(pp, simulations_df=df)
```

```
[6]: {'plot1': <Axes: xlabel='time', ylabel='values'>}
```





## MODEL SELECTION

Using the `petab_select` library, model selection is also wrapped within `basico`. It will instantiate the petab problem, and explore all models according to the specified method:

```
[7]: import petab_select
```

```
[8]: problem = petab_select.Problem.from_yaml('./model_selection/petab_select_problem.yaml')
```

evaluating the problem will perform the calibrations. While COPASI uses different error scaling than other PETab scales, we would expect the same models being chosen:

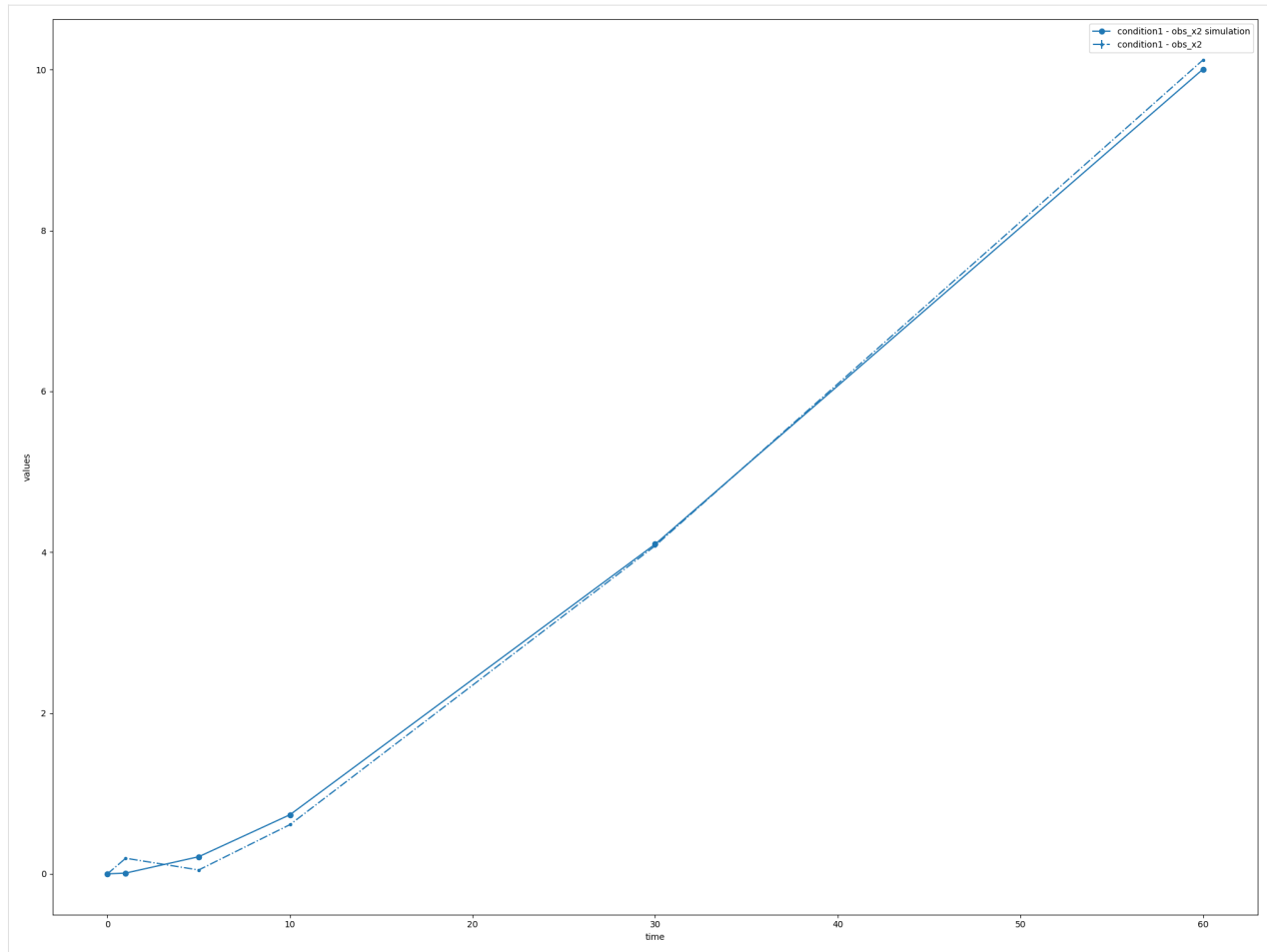
```
[9]: best = basico.petab.evaluate_problem(problem, temp_dir='./temp_dir/', delete_temp_
      ↪files=False)
      print(best.model_subspace_id)
```

```
M1_3
```

at this point the best model returned could be turned into its own PETab problem, and simulated as above:

```
[10]: pp = best.to_petab()['petab_problem']
      sim = basico.petab.PetabSimulator(pp, working_dir='./temp_dir/')
      petab.visualize.plot_problem(pp, simulations_df=sim.simulate())
```

```
[10]: {'plot1': <Axes: xlabel='time', ylabel='values'>}
```



finally lets clean up after ourselves and delete all the temp files generated:

```
[11]: import shutil
      shutil.rmtree('./temp_dir')
```

## WORKING WITH PARAMETER SETS

COPASI supports storing multiple parameter sets for a model. This tutorial describes how to work with them. We start by loading the brusselator example model:

```
[1]: from basico import *
```

```
[2]: brusselator = load_example('brus')
```

Currently we dont have any parameter sets in this file:

```
[3]: get_parameter_sets()
```

```
[3]: []
```

so lets create a new parameter set from the current state:

```
[4]: add_parameter_set('InitialState')
```

### 24.1 Looking at parameter sets

now let us look at it using the `get_parameter_sets` method. If no name is specified, `get_parameter_sets` returns all parameter sets in the file as a list of dictionaries, in each one you will find the following keys:

- **name**: the name of the parameter set
- **description**: notes if any
- **Initial Time**: a dictionary with the initial time of the model
- **Initial Compartment Sizes**: a dictionary with the compartment sizes of all compartments
- **Initial Species Values**: a dictionary with the initial concentration of all species
- **Initial Global Quantities**: a dictionary with the initial values of all global parameters
- **Kinetic Parameters**: a dictionary of all local reaction parameters

`get_parameter_sets` takes 3 arguments:

- **name**: the name of the parameter set to return (can be a substring, to return multiple)
- **exact**: whether the name has to be precisely matched
- **values\_only**: if true, only the initial values will be returned, otherwise also the type of model parameter

so you will see either:

```
[5]: get_parameter_sets(values_only=True)
[5]: [{'name': 'InitialState',
      'description': '',
      'Initial Time': {'The Brusselator': 0.0},
      'Initial Compartment Sizes': {'compartment': 1.0},
      'Initial Species Values': {'X': 2.9999959316797846,
      'Y': 2.9999959316797846,
      'A': 0.49999987545958524,
      'B': 2.9999959316797846,
      'D': 0.0,
      'E': 0.0},
      'Initial Global Quantities': {},
      'Kinetic Parameters': {'R1': {'k1': 1.0},
      'R2': {'k1': 1.0},
      'R3': {'k1': 1.0},
      'R4': {'k1': 1.0}}}]
```

or the much more verbose variant (I'll restrict it here to just one of the species to avoid clutter):

```
[6]: get_parameter_sets()[0]['Initial Species Values']['X']
[6]: {'concentration': 2.9999959316797846,
      'particle_number': 1.80664e+21,
      'parameter_type': 'species',
      'simulation_type': 'reactions'}
```

## 24.2 Creating / Modifying Parameter sets

As we saw above, we can add a new parameter set using `add_parameter_set` with only a name parameter, you can also specify the parameter set, by providing the dictionary as it was returned above. It does not have to be complete, you can also create a partial one, containing only those values you want to specify.

**NOTE:** when specifying concentrations, you will also have to specify the initial compartment volume for those species.

```
[7]: add_parameter_set('Partial', {
      'Initial Compartment Sizes': {'compartment': 1},
      'Initial Species Values': {'X': 3, 'Y': 1}})
```

```
[8]: get_parameter_sets('Partial', values_only=True)[0]
```

```
[8]: {'name': 'Partial',
      'description': '',
      'Initial Time': {},
      'Initial Compartment Sizes': {'compartment': 1.0},
      'Initial Species Values': {'X': 3.0, 'Y': 1.0},
      'Initial Global Quantities': {},
      'Kinetic Parameters': {}}
```

to change certain values in a parameter set, you can use the `set_parameter_set`, this will set all the values specified, and by default leave all entries not specified at their old values. You'd use the argument `remove_others=True`, to remove all entries not specified. For example, let me set the initial concentration of Y in the Partial parameter set to 3:

```
[9]: set_parameter_set('Partial', param_set_dict={'Initial Species Values': {'Y': 3}}, remove_
      ↪others=False)
get_parameter_sets('Partial', values_only=True)[0]['Initial Species Values']

[9]: {'X': 3.0, 'Y': 3.0}
```

To change the current model state, to use the values specified in a certain parameter set, you'd use the function `apply_parameter_set`:

```
[10]: apply_parameter_set('Partial')
get_species()[['initial_concentration']]

[10]:      initial_concentration
name
X      3.000000
Y      3.000000
A      0.500000
B      2.999996
D      0.000000
E      0.000000
```

If you wanted to update a parameter set, with all the values from the current model state, you can use `update_parameter_set`. Lets use that here, to update the `InitialState` parameter set from above, to contain the updated concentrations:

```
[11]: update_parameter_set('InitialState')
get_parameter_sets('InitialState', values_only=True)[0]['Initial Species Values']

[11]: {'X': 3.0,
      'Y': 3.0,
      'A': 0.49999987545958524,
      'B': 2.9999959316797846,
      'D': 0.0,
      'E': 0.0}
```

## 24.3 Removing parameter sets

The `remove_parameter_sets` function can be used to remove parameter sets if no argument is specified, all parameter sets will be removed, otherwise the once matching the given name. Here i just want to remove the partial one defined above:

```
[12]: remove_parameter_sets('Partial')

[13]: for p_set in get_parameter_sets():
      print(p_set['name'])

InitialState

[14]: remove_datamodel(brusselator)
```

## 24.4 Parameter sets and Parameter Estimation

Parameter sets are mainly in use in parameter estimations, with multiple experiments. Since each experiment can have different initial conditions or even dependent values can differ between experiments, the parameter sets can capture those differences, when a parameter estimation is run with `create_parametersets=True`, lets try this here for a small test model that has parameter estimation set up:

```
[15]: lm = load_example('LM')
```

we don't have any parameter sets yet, but when we run the the parameter estimation we can have some created:

```
[16]: get_parameter_sets()
```

```
[16]: []
```

```
[17]: run_parameter_estimation(create_parametersets=True)
```

```
[17]:
```

	lower	upper	sol	affected
name				
(R1).k2	1e-6	1e6	0.0000002	[]
(R2).k1	1e-6	1e6	44.661715	[]
Values[offset]	-0.2	0.4	0.043018	[Experiment_1]
Values[offset]	-0.2	0.4	0.054167	[Experiment_3]
Values[offset]	-0.2	0.4	-0.050941	[Experiment]
Values[offset]	-0.2	0.4	0.045922	[Experiment_4]
Values[offset]	-0.2	0.4	0.048025	[Experiment_2]

we can look at the fit statistic, to see which objective value was reached:

```
[18]: get_fit_statistic()['obj']
```

```
[18]: inf
```

and we also see all the generated parameter sets, one for the original model state, and one for each individual experiment:

```
[19]: for p_set in get_parameter_sets():
      print(p_set['name'])
```

```
PE: 2023-12-06T09:14:46Z Exp: Original
PE: 2023-12-06T09:14:46Z Exp: Experiment
PE: 2023-12-06T09:14:46Z Exp: Experiment_1
PE: 2023-12-06T09:14:46Z Exp: Experiment_2
PE: 2023-12-06T09:14:46Z Exp: Experiment_3
PE: 2023-12-06T09:14:46Z Exp: Experiment_4
```

we can run the parameter estimation again, maybe using a different algorithm, and this new run would give as further parameter sets:

```
[20]: run_parameter_estimation(method=PE.HOOKE_JEEVES, create_parametersets=True)
      print(f"Objective value reached: {get_fit_statistic()['obj']}")
```

```
Objective value reached: inf
```

```
[21]: for p_set in get_parameter_sets():
      print(p_set['name'])
```

```

PE: 2023-12-06T09:14:46Z Exp: Original
PE: 2023-12-06T09:14:46Z Exp: Experiment
PE: 2023-12-06T09:14:46Z Exp: Experiment_1
PE: 2023-12-06T09:14:46Z Exp: Experiment_2
PE: 2023-12-06T09:14:46Z Exp: Experiment_3
PE: 2023-12-06T09:14:46Z Exp: Experiment_4
PE: 2023-12-06T09:14:52Z Exp: Original
PE: 2023-12-06T09:14:52Z Exp: Experiment
PE: 2023-12-06T09:14:52Z Exp: Experiment_1
PE: 2023-12-06T09:14:52Z Exp: Experiment_2
PE: 2023-12-06T09:14:52Z Exp: Experiment_3
PE: 2023-12-06T09:14:52Z Exp: Experiment_4

```

since the objective value is better for the second run, let us assume we want to remove the parameter sets from the first run. We can do this like so:

```

[22]: psets = get_parameter_sets()
first_time_stamp = psets[0]['name'][:psets[0]['name'].rfind('Exp:')]
remove_parameter_sets(first_time_stamp, exact=False)

```

```

[23]: for p_set in get_parameter_sets():
        print(p_set['name'])

PE: 2023-12-06T09:14:52Z Exp: Original
PE: 2023-12-06T09:14:52Z Exp: Experiment
PE: 2023-12-06T09:14:52Z Exp: Experiment_1
PE: 2023-12-06T09:14:52Z Exp: Experiment_2
PE: 2023-12-06T09:14:52Z Exp: Experiment_3
PE: 2023-12-06T09:14:52Z Exp: Experiment_4

```

```

[24]: remove_datamodel(lm)

```

Once a model has several parameter sets, we can also use them in a parameter scan. To give an example, here we load the brusselator example again, create a parameter set for the initial state, and then one for the steady state. Then running a scan over the two parameter sets we can see them both being applied:

```

[25]: brusselator = load_example('brus')
add_parameter_set('InitialState')
run_steadystate(update_model=True)
add_parameter_set('steady_state')

```

lets look at the parametersets we have, and the concentrations of X and Y in them

```

[26]: for param_set in get_parameter_sets():
        print(param_set['name'])
        concentrations = param_set['Initial Species Values']
        for species in ['X', 'Y']:
            print(f"{species}: {concentrations[species]['concentration']}")

InitialState
X: 2.9999959316797846
Y: 2.9999959316797846
steady_state

```

(continues on next page)

(continued from previous page)

```
X: 0.49999987545958524
Y: 5.999993357842892
```

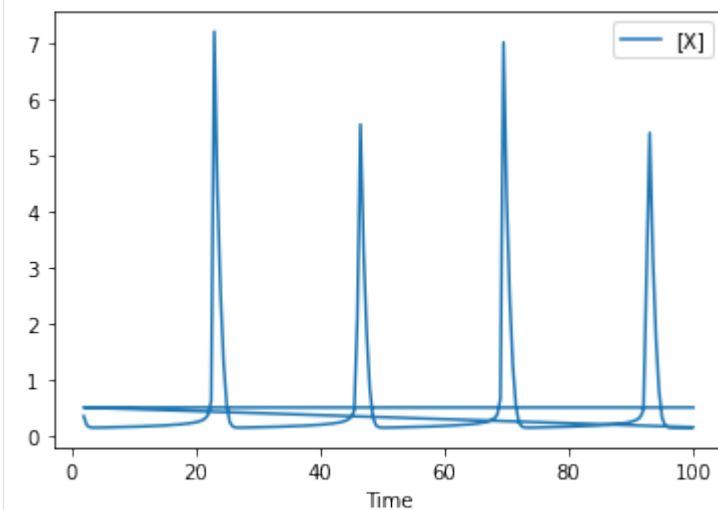
Now we can setup a scan over both parameter sets as example, for that we switch the subtask to time course, capture output while the task is running and set the scan item to be the parameter sets:

```
[27]: set_scan_settings( subtask=T.TIME_COURSE, output_during_subtask=True, scan_items=[
      {'parameter_sets':['InitialState', 'steady_state'],}
    ])
```

when we now run this task, and capture the output for time and the concentration for X we want to see the time course to be performed twice, once for the initial state, and once for steadystate:

```
[28]: run_scan(output=['Time', '[X]']).plot(x='Time', y='[X]')
```

```
[28]: <Axes: xlabel='Time'>
```





## PROFILE LIKELIHOOD

To perform a basic profile likelihood analysis of a fit obtained, `basico` implements the [Schaber method](#) that in turn fixes each optimization parameter, while reoptimizing the remaining parameters. This scan is performed in both directions. Ideally, what we would want to see for identified parameters, is that the result gets worse, as we go away from the found solution. If we find a flat line, that means the parameter is not identifiable, if we find a curve that is flat on one side, we can only identify the corresponding bound.

Since this could potentially take quite some time `basico` breaks this task into three steps:

- generating different `copasi` models for each of the scans. (these files could then be moved to a cluster environment for parallel computation)
- running the the scans (here `basico` uses the `python multiprocessing` library), which places reports into a specified directory
- plotting the result, by specifying the directory where the reports have been stashed.

Let's try this on an example model.

```
[1]: from basico import *  
from basico.task_profile_likelihood import prepare_files, process_dir, plot_data
```

### 25.1 Preparing files

The first step is to prepare the files, here we load an example model, run the parameter estimation

```
[2]: example_model = load_example('LM')  
run_parameter_estimation(method=PE.HOOKE_JEEVES, update_model=True)
```

```
[2]:
```

	lower	upper	sol	affected
name				
(R1).k2	1e-6	1e6	26.353601	[]
(R2).k1	1e-6	1e6	46.699200	[]
Values[offset]	-0.2	0.4	0.046461	[Experiment_1]
Values[offset]	-0.2	0.4	0.077161	[Experiment_3]
Values[offset]	-0.2	0.4	-0.049828	[Experiment]
Values[offset]	-0.2	0.4	0.030610	[Experiment_4]
Values[offset]	-0.2	0.4	0.058074	[Experiment_2]

at this point we also want to make sure, that we have reached a good fit, so we look at the fit statistic:

```
[3]: get_fit_statistic()
```

```
[3]: {'obj': 12.501488024131111,
      'rms': 0.15812329381929224,
      'sd': 0.15924191621103437,
      'f_evals': 3008,
      'failed_evals_exception': 0,
      'failed_evals_nan': 0,
      'cpu_time': 5.0625,
      'data_points': 500,
      'valid_data_points': 500,
      'evals_per_sec': 0.0016830119680851063}
```

now we create a temporary directory, and save the file and experimental data into it. After that i unload the model.

```
[4]: from tempfile import mkdtemp
data_dir = mkdtemp()
original_model = os.path.join(data_dir, 'original.cps')
save_model(original_model)
remove_datamodel(example_model)
```

the code has just been ported from an earlier version, so lets debug it for now

```
[5]: #import logging
#logging.basicConfig()
#logging.getLogger().setLevel(logging.DEBUG)
```

the data will be stored in those files:

```
[6]: #print(data_dir)
#print(original_model)
```

now we go ahead and prepare all the files, this is done by the `prepare_files` function that accepts the following arguments:

- `filename`: the template file with the fit we want to analyze
- `data_dir`: the directory in which to store the files in

the remaining parameters are optional:

- `iterations=50`: if not specified hooks & Jeeves / Levenberg Marquardt will perform 50 iterations
- `scan_interval=40`: if not specified 40 scan intervals will be taken (40 in each direction!)
- `lower_adjustment='-50%'`: the multiplier for the lower bound of the scan. By default the interval will be -50% of the found parameter value.
- `upper_adjustment='+50%'`: the multiplier for the upper bound of the scan, defaults to '+50%' found
- `modulation=0.01`: modulation parameter for Levenberg Marquardt
- `tolerance=1e-06`: tolerance for the optimization method
- `disable_plots=True`: if true, other plots in the file will be removed
- `disable_tasks=True`: if true, other tasks in the file will be deleted
- `use_hooke=False`: if true, use hooks & jeeves otherwise Levenberg Marquadt
- `prefix="out_"`: the prefix for each file generated.

For this example we just do a scan of 8 values in each direction for 20 iterations:

```
[7]: prepare_files(original_model, scan_interval=8, iterations=20, data_dir=data_dir)
```

```
[7]: {'num_params': 7,
      'num_data': 500,
      'obj': 12.501488024131111,
      'param_sds': {'(R1).k2': 5.433042366390387,
                    '(R2).k1': 0.5087112193172452,
                    'Values[offset].InitialValue': 0.01143802470807778}}
```

## 25.2 Processing the files:

Here we use the python multiprocessing pool, to launch CopasiSE to process all the files we have generated in the data directory we process 4 files at the same time, but you can supply a `pool_size` parameter to change that. This step can be skipped, if you copy the files to a cluster environment and run them there.

To specify a specific CopasiSE version to use you can pass it along the module's `COPASI_SE` field. For example like so:

```
basico.task_profile_likelihood.COPASI_SE = '/opt/COPASI/COPASI-4.38.268-Linux-64bit/bin/
CopasiSE'
```

alternatively, `process_dir` also accepts the following named arguments:

- `copasi_se`: if you dont want to change the global default for CopasiSE, but just pass it along here
- `max_time`: normally, we'd let the model run to completion, but if you specify a max time in seconds, then the computation will be stopped after that time.

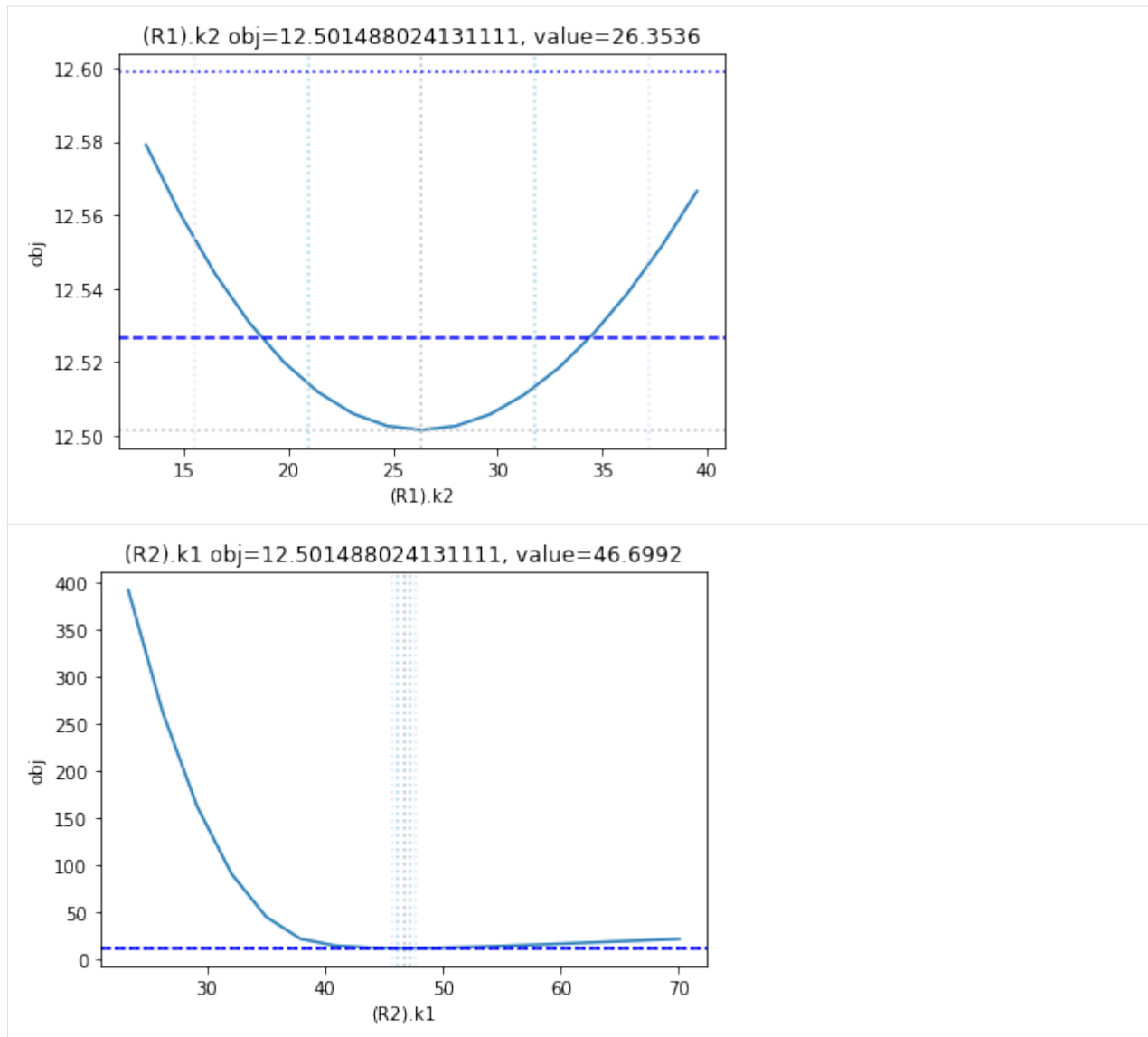
so lets runt the files we have generated above, and restrict each run to 10 seconds:

```
[8]: process_dir(data_dir, max_time=10)
```

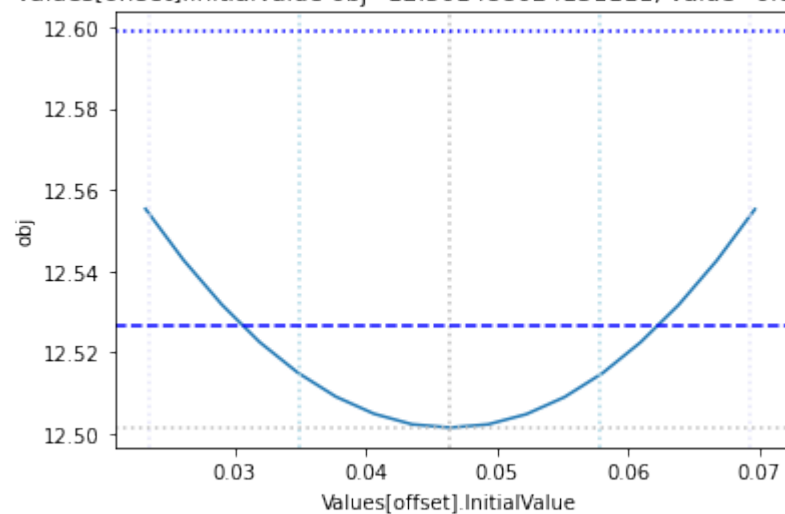
## 25.3 Looking at the results

The `plot_data` function reads all the report files from the `data_dir`, and combines the scans from both directions to one plot.

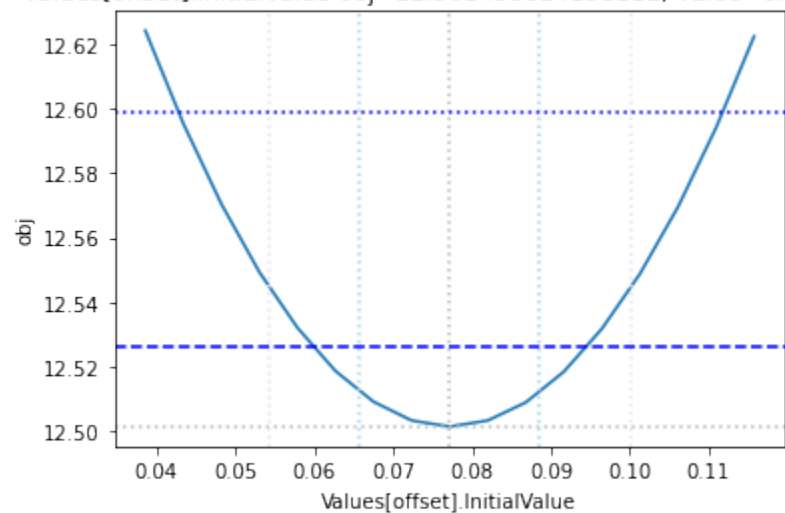
```
[9]: plots = plot_data(data_dir)
```

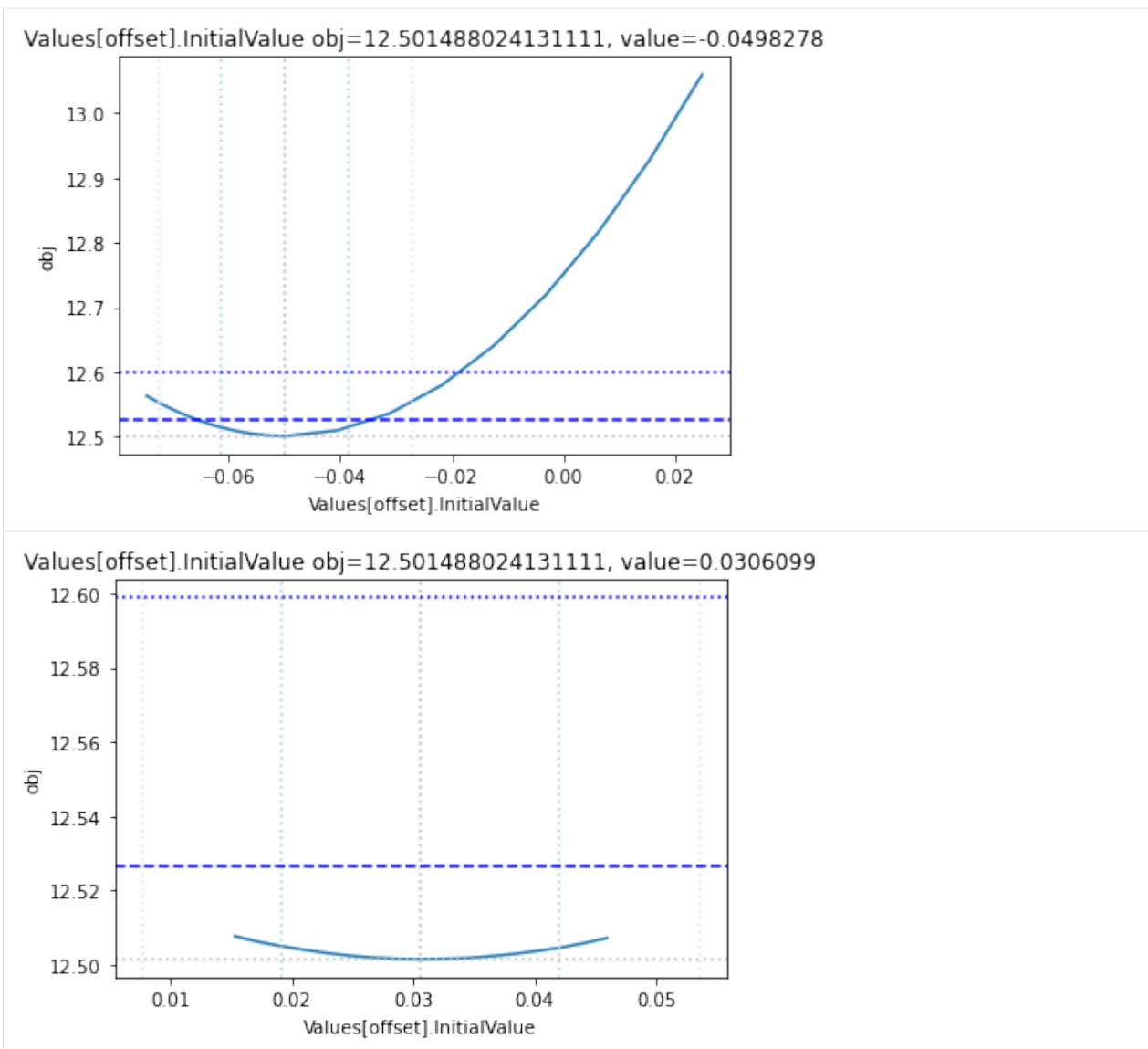


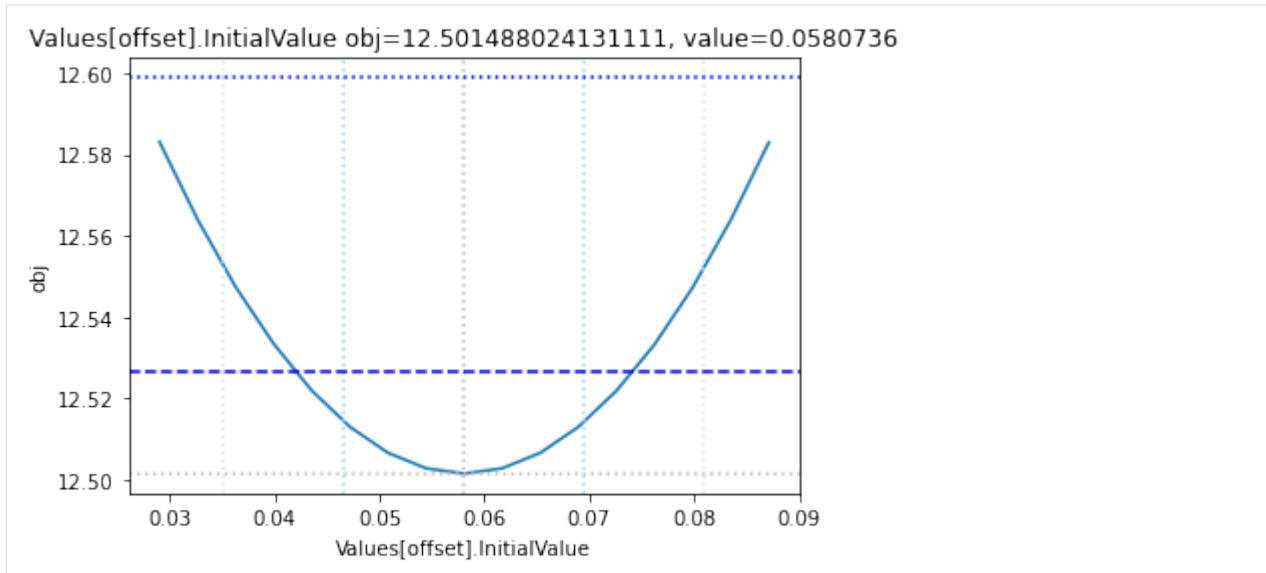
Values[offset].InitialValue obj=12.501488024131111, value=0.0464614



Values[offset].InitialValue obj=12.501488024131111, value=0.077161







and finally, let us remove the temporary files:

```
[10]: #import shutil
      #shutil.rmtree(data_dir, ignore_errors=True)
```

## 25.4 Everything together

Lets use a second example, this time we recreate the model from Schabers publication from scratch, and then run the parameter estimation using a convenience method:

```
[11]: from basico import *
      from basico.task_profile_likelihood import get_profiles_for_model
```

```
[12]: new_model(name='Schaber Example', notes="""
The example from the supplement, originally described in

Schaber, J. and Klipp, E. (2011) Model-based inference of biochemical parameters and
dynamic properties of microbial signal transduction networks, Curr Opin Biotechnol, 22,
109-
116 (https://doi.org/10.1016/j.copbio.2010.09.014)

""");
```

```
[13]: add_function('mod. MA',
                  'k * S * T',
                  'irreversible',
                  mapping={'S': 'modifier', 'T': 'substrate'})
);
```

```
[14]: add_reaction('v1', 'T -> Tp; S', function='mod. MA')
      add_reaction('v2', 'Tp -> T')
```

(continues on next page)

(continued from previous page)

```
add_reaction('decay', 'S ->', mapping={'k1': 0.5})
get_reactions()[['scheme', 'function', 'mapping']]
```

```
[14]:
```

	scheme	function	mapping
name			
v1	T -> Tp; S	mod. MA	{'k': 0.1, 'S': 'S', 'T': 'T'}
v2	Tp -> T	Mass action (irreversible)	{'k1': 0.1, 'substrate': 'Tp'}
decay	S ->	Mass action (irreversible)	{'k1': 0.5, 'substrate': 'S'}

```
[15]: set_species('Tp', initial_concentration=0)
# observable
add_species('TpFit', status='assignment', expression='{[Tp]}/0.5')
get_species()[['initial_concentration']]
```

```
[15]:
```

	initial_concentration
name	
S	1.0
T	1.0
Tp	0.0
TpFit	0.0

```
[16]: add_experiment('Data set 1', pd.DataFrame(data = {
    'Time': [1, 2, 4, 6],
    '[TpFit]': [1, 0.88, 0.39, 0.22],
    '#sd': [0.09, 0.09, 0.09, 0.09]
}));
```

```
[17]: set_fit_parameters([
    {'name': '(v1).k', 'lower': 0, 'upper': 100},
    {'name': '(v2).k1', 'lower': 0, 'upper': 100},
])

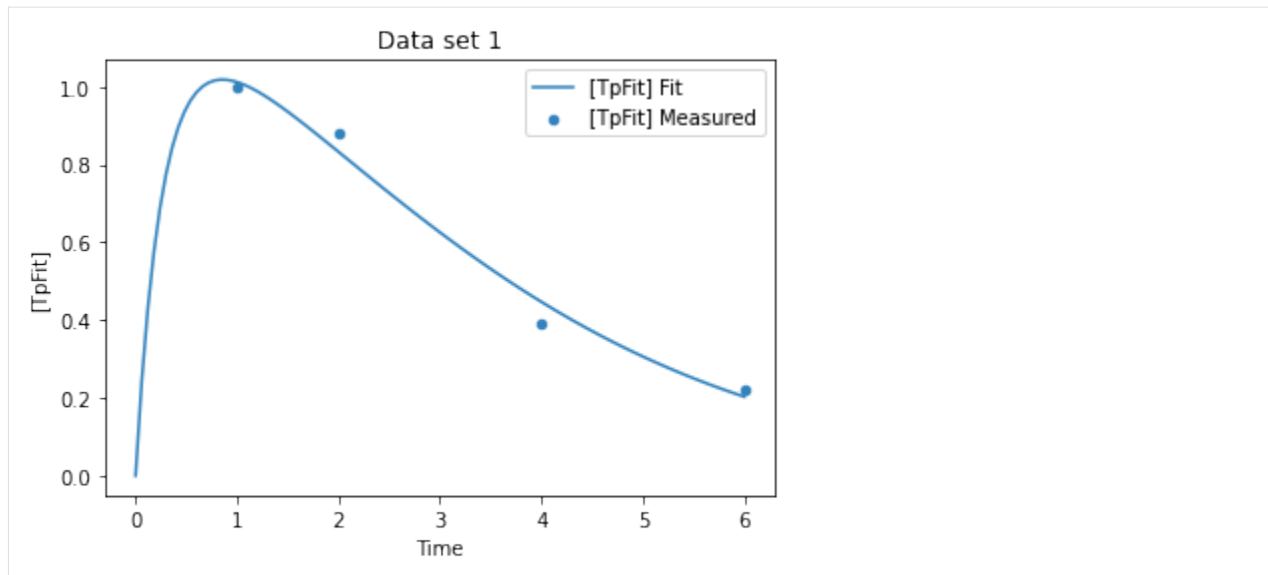
sol = run_parameter_estimation(method=PE.HOOKE_JEEVES, update_model=True)
sol[['sol']]
```

```
[17]:
```

	sol
name	
(v1).k	2.267945
(v2).k1	1.425208

```
[18]: plot_per_experiment(solution=sol);
```



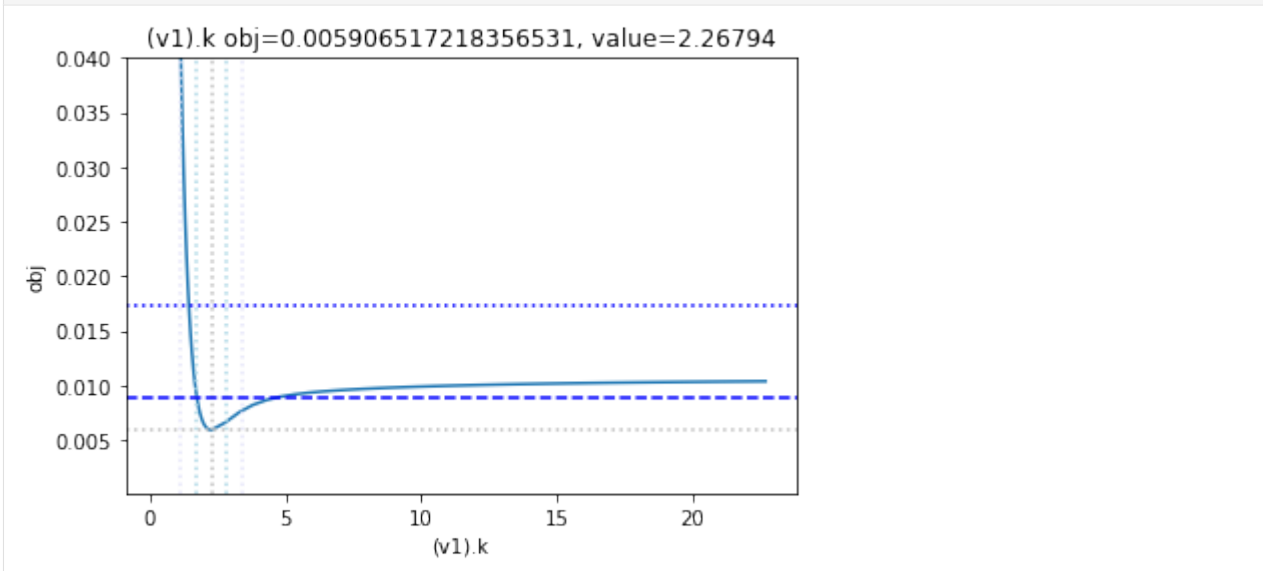


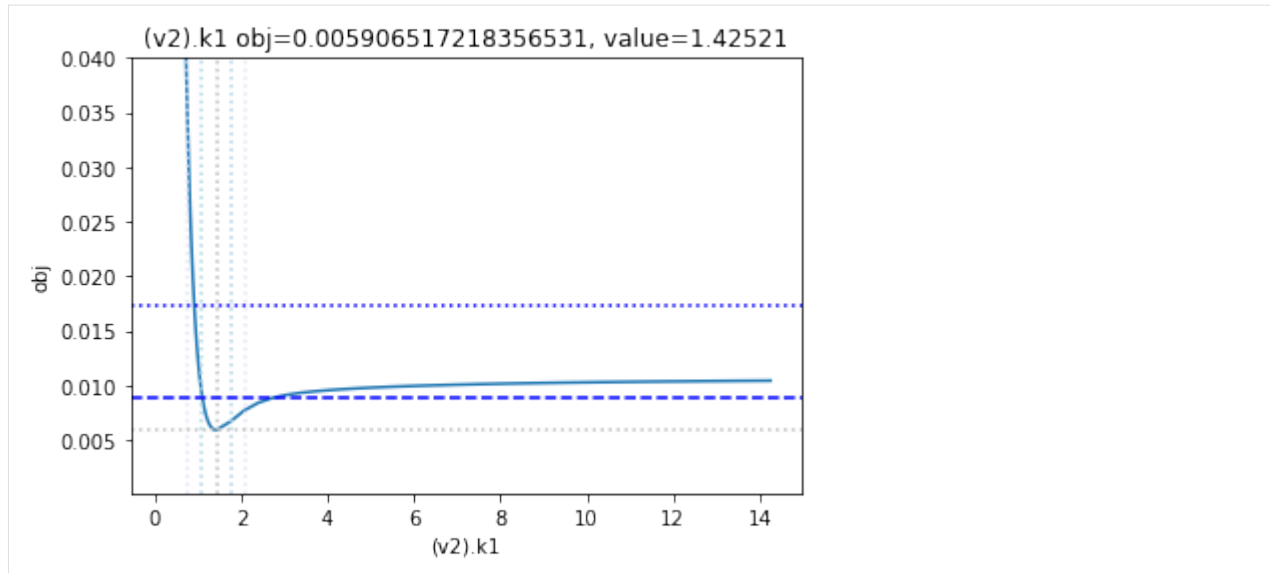
while the fit looks ok, lets have a look at the likelihood profile

the utility method `get_profiles_for_model` combines all of the above and runs the profile likelihood for the currently loaded model and fit. You can supply all the arguments of `prepare_files` and `process_dir`. By default `get_profiles_for_model` will use a temporary directory, and remove the files afterwards. If you would like the files to stay on your disc, you can supply a `data_dir`, in which all files will be generated, and reports placed.

We additionally use the `scale_mode` parameter, that allows to limit the y\_axis to desired limits.

```
[19]: get_profiles_for_model(lower_adjustment=0.1, upper_adjustment=10, scale_mode={'top': 0.04, 'bottom': 0.0001});
```





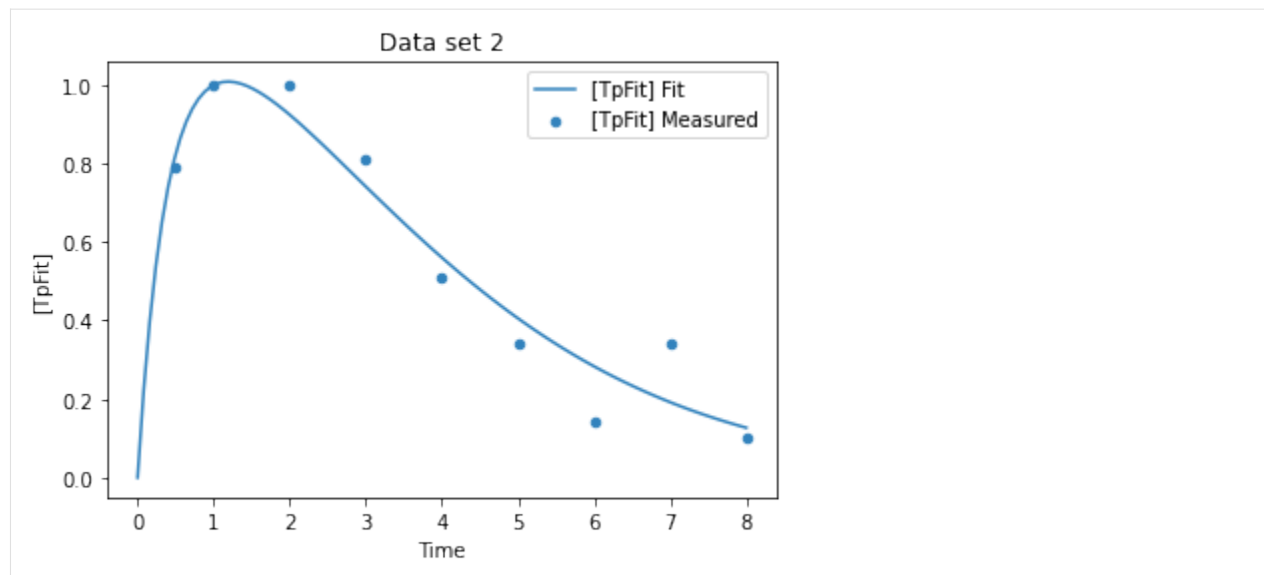
as the result from the publication suggests, the parameter is not identifiable. No lets the run the second dataset as well:

```
[20]: remove_experiments()
      add_experiment('Data set 2',pd.DataFrame(data = {
          'Time': [0.5, 1, 2, 3, 4, 5, 6, 7, 8],
          '[TpFit]': [0.79, 1, 1, 0.81, 0.51, 0.34, 0.14, 0.34, 0.1],
          '#sd': [0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]
      }))

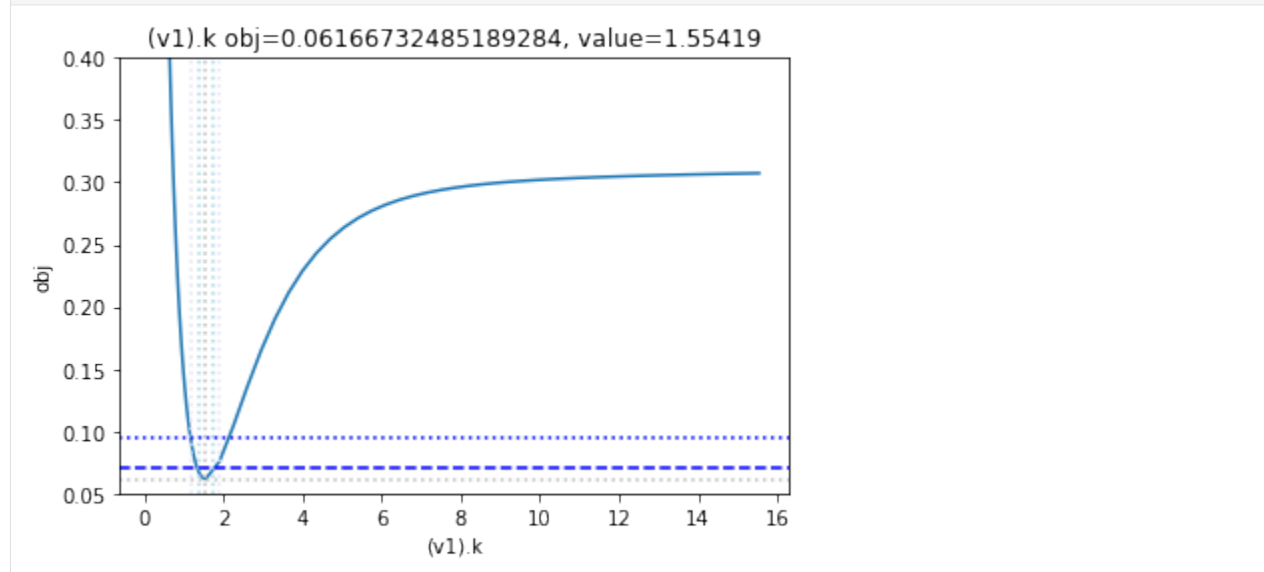
      sol = run_parameter_estimation(method=PE.HOOKE_JEEVES, update_model=True)
      sol[['sol']]
```

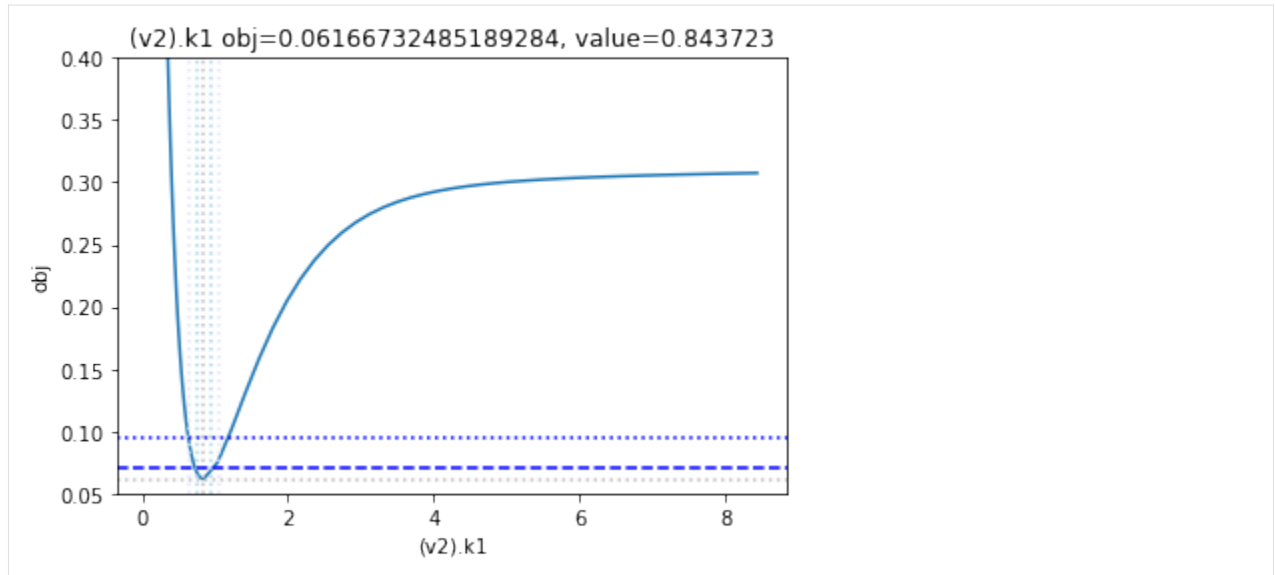
```
[20]:          sol
      name
      (v1).k  1.554192
      (v2).k1  0.843723
```

```
[21]: plot_per_experiment(solution=sol);
```



```
[22]: get_profiles_for_model(lower_adjustment=0.1, upper_adjustment=10, scale_mode={'top': 0.4,
↪ 'bottom': 0.05});
```

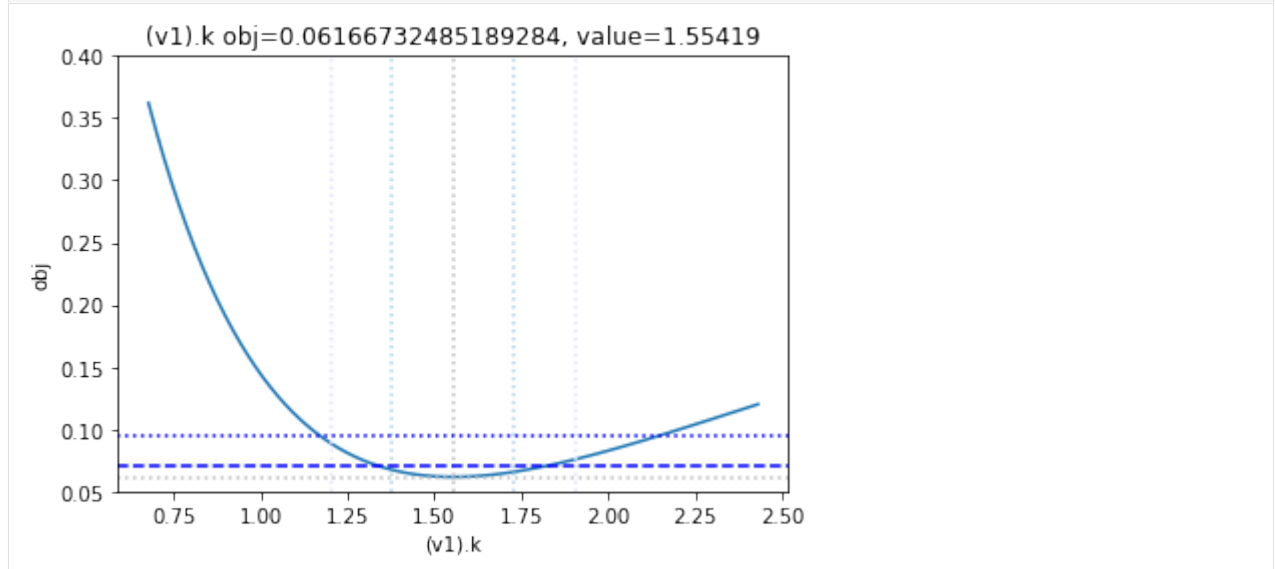


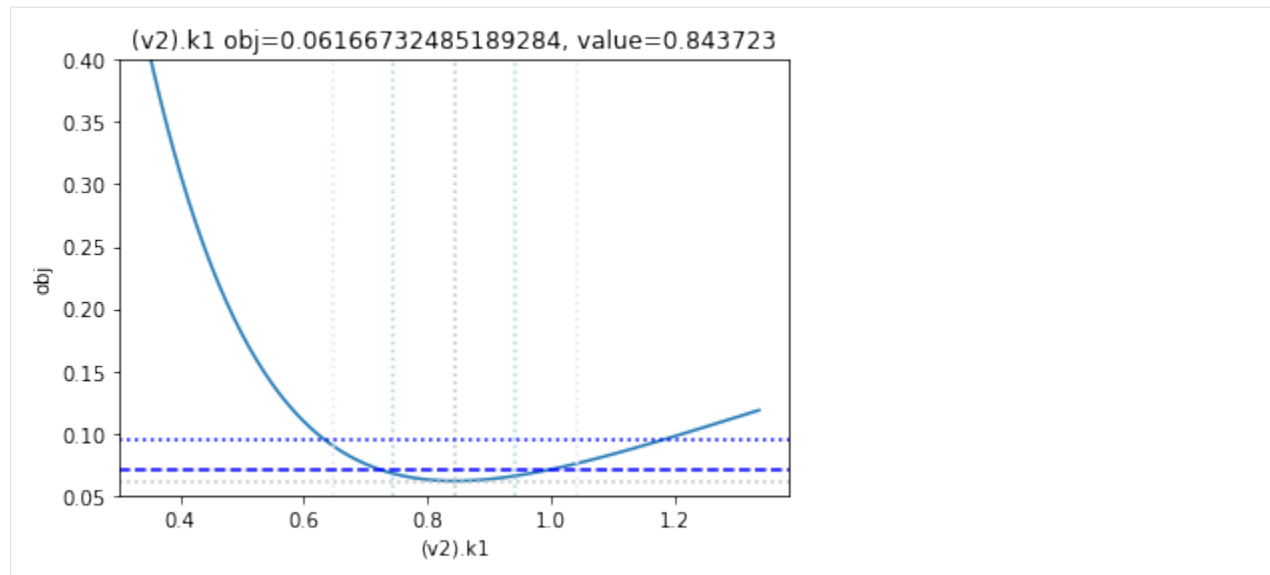


looking at the profile we find what we want to see.

A new option added, is to generate the profiles for a certain number of standard deviations as calculated by the COPASI fit. Lets try this here, by going to the ranges of -5 / +5 standard deviations from the value found:

```
[23]: get_profiles_for_model(lower_adjustment="-5SD", upper_adjustment="+5SD", scale_mode={'top'
↳ ': 0.4, 'bottom': 0.05});
```







## METABOLIC CONTROL ANALYSIS

This notebook demonstrates how to use Metabolic Control Analysis using `basico`. We start as always, by importing `basico`:

```
[1]: from basico import *
```

next we load the Brusselator example model

```
[2]: brusselator = load_example('brusselator')
```

running the MCA can be done explicitly by calling `run_mca`, or implicitly by passing along `run_first=True` to the functions that retrieve the elasticities or control coefficients. The task needs to be run only once so that all matrices are available. These functions additionally include a `scaled` flag, that indicates, whether the scaled or unscaled values should be returned. By default `run_first=False` and `scaled=True`.

So here we pass `run_first=True` for the first call, and then just retrieve the remaining functions. First the elasticities.

Running the MCA can be done explicitly by calling `run_mca`, or implicitly by passing along `run_first=True` to the functions that retrieve the elasticities or control coefficients. The task needs to be run only once so that all matrices are available. These functions additionally include a `scaled` flag, that indicates whether the scaled or unscaled values should be returned. By default, `run_first=False` and `scaled=True`.

So here we pass `run_first=True` for the first call, and then just retrieve the remaining functions.

Elasticities define how much impact do changes of the metabolite concentration  $S_i$  have on the reaction rate  $v_k$ .

```
[3]: get_elasticities(run_first=True)
```

```
[3]:      X    Y
(R1)  0.0  0.0
(R2)  2.0  1.0
(R3)  1.0  0.0
(R4)  1.0  0.0
```

Concentration control coefficients tell us about how much impact changes of a single reaction rate have on the steady state concentrations of the metabolites. The sum of entries in each row should be 0, the `Summation Error` column shows the deviation from that, that can be the result of numeric operations.

Concentration coefficients are only available if a steady state was found when running the task.

```
[4]: get_concentration_control_coefficients(scaled=True)
```

```
[4]:      (R1)  (R2)  (R3)  (R4)  'Summation Error'
X      1.0   0.0   0.0  -1.0                0.0
Y     -1.0  -1.0   1.0   1.0                0.0
```

Finally, the flux control coefficients tell us about how much changes of a single reaction rate impact the steady state flux of (another) reaction. The sum of each row should be 1, deviations will again be shown in the `Summation Error` column when the scaled results are retrieved.

Flux control coefficients are only available if a steady state was found when running the task.

```
[5]: get_flux_control_coefficients()
```

```
[5]:
```

	(R1)	(R2)	(R3)	(R4)	'Summation Error'
(R1)	1.0	0.000000e+00	0.0	0.0	0.000000e+00
(R2)	1.0	-2.220446e-16	1.0	-1.0	0.000000e+00
(R3)	1.0	0.000000e+00	1.0	-1.0	2.220446e-16
(R4)	1.0	0.000000e+00	0.0	0.0	0.000000e+00



## WORKING WITH WIDGETS

Here we create an interactive example, using ipywidgets and matplotlib for plotting, lets first import those. If the script does not run for you try:

```
!pip install -q --upgrade ipympl ipywidgets matplotlib
!jupyter labextension install jupyter-matplotlib
```

and reload the kernel.

```
[1]: %matplotlib widget
import ipywidgets as widgets
import matplotlib.pyplot as plt
import numpy as np

from basico import *
```

We start by creating the `lorenz` model, right from the ODEs:

```
[2]: new_model(name='Lorenz Model')

add_species('X', initial_concentration=0)
add_species('Y', initial_concentration=1)
add_species('Z', initial_concentration=1.05)

add_parameter('beta', initial_value=8./3.)
add_parameter('rho', initial_value=28.)
add_parameter('sigma', initial_value=10.0)

add_equation('d[X]/dt=sigma*([Y]-[X])')
add_equation('d[Y]/dt=rho*[X]-[Y]-[X]*[Z]')
add_equation('d[Z]/dt=[X]*[Y]-beta*[Z]')
```

Now we use the ipywidgets interact decorator, to allow to modify the values for sigma, rho and beta.

```
[3]: fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")

@widgets.interact(sigma=(1, 50, 0.01), rho=(0.1, 50, .01), beta=(-0.1, 10, 0.01),
```

(continues on next page)

(continued from previous page)

```

duration=(10, 1000), start_time=(0, 10))
def update(sigma = 10, rho=28, beta=8.0/3.0, duration=100, start_time=0.01):
    # remove existing lines
    [l.remove() for l in ax.lines]

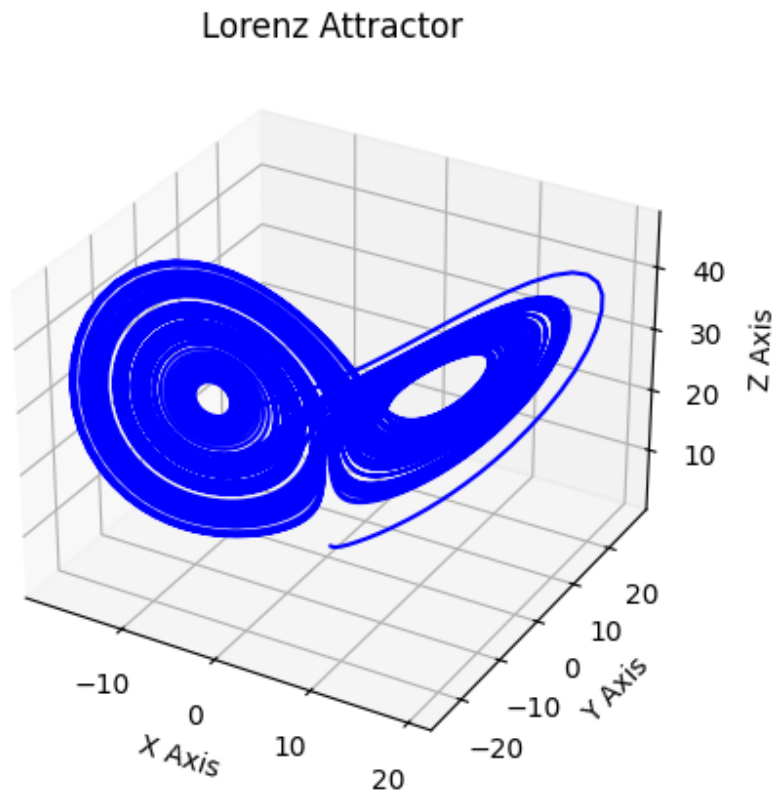
    # set new parameter values
    set_parameters('sigma', initial_value=sigma)
    set_parameters('rho', initial_value=rho)
    set_parameters('beta', initial_value=beta)

    # run time course
    df = run_time_course_with_output(['[X]', '[Y]', '[Z]'], duration=duration,
    intervals=10000, start_time=start_time)

    # create 3d plot for x,y and z (remember to set color otherwise it'll be overwritten)
    ax.plot(df['[X]'].values, df['[Y]'].values, df['[Z]'].values, color='blue');

interactive(children=(FloatSlider(value=10.0, description='sigma', max=50.0, min=1.0,
step=0.01), FloatSlider(...

```



I've noticed, that the result is not visible when rendering the result as html, so here a gif showing the result:

## 28.1 basico package

BasiCO is a simplified interface to COPASI.

This module provides convenience functions to quickly get a model loaded and simulated:

### Example

```
>>> from basico import *
>>> load_biomodel(10)
>>> run_time_course().plot()
```

### 28.1.1 Submodules

#### 28.1.2 basico.biomodels module

A submodule for accessing the BioModels API.

This submodule accesses the BioModels REST api as described on:

[<https://www.ebi.ac.uk/biomodels/docs/>](https://www.ebi.ac.uk/biomodels/docs/)

### Examples

```
>>> # get info for a specific model
>>> info = get_model_info(12)
>>> print(info['name'], info['files']['main'][0]['name'])
```

```
>>> # get all files for one model
>>> files = get_files_for_model(12)
>>> print(files['main'][0]['name'])
```

```
>>> # get content of specific model
>>> sbml = get_content_for_model(12)
>>> print(sbml)
```

```
>>> # search for model
>>> models = search_for_model('repressilator')
>>> for model in models:
>>>     print(model['id'], model['name'], model['format'])
```

`basico.biomodels.download_from(url)`

Convenience method reading content from a URL.

This convenience method uses `urlopen` on either python 2.7 or 3.x

**Parameters**

**url** (*str*) – the url to read from

**Returns**

the contents of the URL as *str*

**Return type**

*str*

`basico.biomodels.download_json(url)`

Convenience method reading the content of the url as JSON.

**Parameters**

**url** (*str*) – the url to read from

**Returns**

a python object representing the json content loaded

**Return type**

*dict*

`basico.biomodels.get_content_for_model(model_id, file_name=None)`

Downloads the specified file from biomodels

**Parameters**

- **model\_id** – the model id as int, or string
- **file\_name** – the filename to download (or None, to download the main file)

**Returns**

the content of the specified file

`basico.biomodels.get_files_for_model(model_id)`

Retrieves the json structure for all files for the given biomodel.

The structure is of form:

```
>>> get_files_for_model(10)
{
  'additional': [
    {'description': 'Auto-generated Scilab file',
     'fileSize': '3873',
     'name': 'BIOMD0000000010.sci'},
    ...
  ],
  'main': [
    {'fileSize': '31568',
     'name': 'BIOMD0000000010_url.xml'}
```

(continues on next page)

(continued from previous page)

```

    }
  ]
}

```

**Parameters****model\_id** – the model id (as int or string)**Returns**

json structure

```
basico.biomodels.get_model_info(model_id)
```

Return the model info for the provided *model\_id*.**Parameters****model\_id** – either an integer, or a valid model id**Returns**

a python object describing the model

```
basico.biomodels.search_for_model(query, offset=0, num_results=10, sort='id-asc')
```

Queries the biomodel database

Queries the database, for information about the query system see: <[https://www.ebi.ac.uk/biomodels-static/jummp-biomodels-help/model\\_search.html](https://www.ebi.ac.uk/biomodels-static/jummp-biomodels-help/model_search.html)>**Example**

```

>>> search_for_model('glycolysis')
[...
  {
    'format': 'SBML',
    'id': 'BIOMD0000000206',
    'lastModified': '2012-07-04T23:00:00Z',
    'name': 'Wolf2000_Glycolytic_Oscillations',
    'submissionDate': '2008-11-27T00:00:00Z',
    'submitter': 'Harish Dharuri',
    'url': 'https://www.ebi.ac.uk/biomodels/BIOMD0000000206'
  }
]

```

Note by default, it will include only manually curated models, to obtain Non-curated models you would use:

```

>>> search_for_model('Hodgkin AND curationstatus:"Non-curated"')
[...
  {
    'format': 'SBML',
    'id': 'MODEL1006230012',
    'lastModified': '2012-02-02T00:00:00Z',
    'name': 'Stewart2009_ActionPotential_PurkinjeFibreCells',
    'submissionDate': '2010-06-22T23:00:00Z',
    'submitter': 'Camille Laibe',
    'url': 'https://www.ebi.ac.uk/biomodels/MODEL1006230012'
  }
]

```

(continues on next page)

(continued from previous page)

```
}
]
```

### Parameters

- **query** – the query to use (it will be encoded with quote\_plus before send out, so it is safe to use spaces)
- **offset** – offset (defaults to 0)
- **num\_results** – number of results to obtain (defaults to 10)
- **sort** – sort criteria to be used (defaults to id-asc)

### Returns

the search result as [{}]

## 28.1.3 basico.compartment\_array\_tools module

This module provides convenience functions for array of compartments.

COPASI can duplicate the current model in either a rectangular or linear manner, with diffusion reactions added in between the created model. This allows for a simplified spatial simulation.

This submodule adds functions, to create such an array, to delete the template model. Additionally some basic plotting functionality is available as well.

### Example

```
>>> dm = load_example('brusselator')
>>> create_rectangular_array(10, 10, ['X', 'Y'], [0.16, 0.8], delete_template=True)
>>> set_species(['X{compartment[1,1]}',
...           'X{compartment[1,2]}',
...           'X{compartment[2,1]}',
...           'X{compartment[2,2]}'], initial_concentration=10)
```

```
>>> add_event('E0', 'Time > 10', [['X{compartment[1,1]}', '10'],
...                               ['X{compartment[1,2]}', '10'],
...                               ['X{compartment[2,1]}', '10'],
...                               ['X{compartment[2,2]}', '10']])
```

```
>>> data = run_time_course(start_time=0, duration=500)
>>> animate_rectangular_time_course_as_image(data, metabs=["X", "Y"], min_range=0, max_
↳ range=10)
```

```
basico.compartment_array_tools.animate_rectangular_time_course(data, metab=None, prefix=None,
                                                                shading='gouraud',
                                                                min_range=nan, max_range=nan,
                                                                filename=None, **kwargs)
```

Plots the simulation data on the loaded model assuming that this is in the format of an array of compartments as generated by COPASI. This will create a figure for each species at each of the specified time points.

**Parameters**

- **data** – timecourse simulation result
- **metab** – optional parameter specifying the species to animate. If not given, one the first metab will be chosen
- **prefix** – string of compartment prefix to indicate which compartment should be visualized. This is expected to not include the indices, so it would be ‘compartment’ rather than ‘compartment[0]’
- **shading** – optional shading for the color mesh, defaults to ‘gouraud’ can also be ‘flat’
- **min\_range** – optional min range, defaults to NaN, meaning that it is to be the minimum value of the data
- **max\_range** – optional max range, defaults to NaN, meaning that it is to be the maximum value of the data
- **filename** – optional filename to a file to which to save the animation to
- **kwargs** –
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

the FuncAnimation constructed

```
basico.compartment_array_tools.animate_rectangular_time_course_as_image(data, metabs=None,
                                                                           prefix=None,
                                                                           min_range=nan,
                                                                           max_range=nan,
                                                                           filename=None,
                                                                           **kwargs)
```

Plots the data as image.

**Parameters**

- **data** – data frame with results to plot
- **metabs**
- **prefix** – string of compartment prefix to indicate which compartment should be visualized. This is expected to not include the indices, so it would be ‘compartment’ rather than ‘compartment[0]’
- **min\_range** – optional min range, defaults to NaN, meaning that it is to be the minimum value of the data
- **max\_range** – optional max range, defaults to NaN, meaning that it is to be the maximum value of the data
- **filename** – optional filename to save the image as
- **kwargs** –
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

FuncAnimation result

```
basico.compartment_array_tools.create_linear_array(num_steps, species=None,
                                                    diffusion_coefficients=None,
                                                    compartment_names=None,
                                                    delete_template=False, **kwargs)
```

Utility function to create a linear duplicating the specified species, their reactions in the given compartments and created diffusion reactions between the newly created array

#### Parameters

- **num\_steps** – the number of steps to create
- **species** – array of species names, that should be diffusing between compartments
- **compartment\_names** – optional compartment names (will default to the compartment the species is in)
- **diffusion\_coefficients** – optional array of diffusion coefficients in the same order as the species (otherwise they will be set to 1)
- **delete\_template** – if True, the original template model in the specified compartment will be deleted.
- **kwargs** –
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

```
basico.compartment_array_tools.create_rectangular_array(num_steps_x, num_steps_y, species=None,
                                                         diffusion_coefficients=None,
                                                         compartment_names=None,
                                                         delete_template=False, **kwargs)
```

Utility function to create a rectangular array duplicating the specified species, their reactions in the given compartments and created diffusion reactions between the newly created array

#### Parameters

- **num\_steps\_x** – the number of compartments to create along the x direction
- **num\_steps\_y** – the number of compartments to create along the y direction
- **species** – array of species names, that should be diffusing between compartments
- **compartment\_names** – optional compartment names (will default to the compartment the species is in)
- **diffusion\_coefficients** – optional array of diffusion coefficients in the same order as the species (otherwise they will be set to 1)
- **delete\_template** – if True, the original template model in the specified compartment will be deleted.
- **kwargs** –
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None



`basico.compartment_array_tools.delete_compartments(selection, **kwargs)`

utility function for deleting a selection of compartments from the datamodel. This will also delete the species and reactions included.

#### Parameters

- **selection** – an array of tuples of indices at which the compartments should be deleted
- **kwargs** –
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.compartment_array_tools.plot_linear_time_course(data, prefix=None, metab_names=None, shading='gouraud', min_range=nan, max_range=nan, **kwargs)`

Plots the simulation data on the loaded model assuming that this is in the format of an array of compartments as generated by COPASI. This will create a figure for each species

#### Parameters

- **data** – timecourse simulation result
- **prefix** – string of compartment prefix to indicate which compartment should be visualized. This is expected to not include the indices, so it would be 'compartment' rather than 'compartment[0]'
- **metab\_names** – optional array of metabolite names that should be plotted (defaults to all)
- **shading** – optional shading for the color mesh, defaults to 'gouraud' can also be 'flat'
- **min\_range** – optional min range, defaults to NaN, meaning that it is to be the minimum value of the data
- **max\_range** – optional max range, defaults to NaN, meaning that it is to be the maximum value of the data
- **kwargs** –
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

array with tuple of figures and their axis

`basico.compartment_array_tools.plot_rectangular_time_course(data, times=None, prefix=None, shading='gouraud', min_range=nan, max_range=nan, **kwargs)`

Plots the simulation data on the loaded model assuming that this is in the format of an array of compartments as generated by COPASI. This will create a figure for each species at each of the specified time points.

#### Parameters

- **data** – timecourse simulation result
- **times** – optional parameter specifying the times for which to plot the results. If not given, one figure will be created for each output time point in the data

- **prefix** – string of compartment prefix to indicate which compartment should be visualized. This is expected to not include the indices, so it would be ‘compartment’ rather than ‘compartment[0]’
- **shading** – optional shading for the color mesh, defaults to ‘gouraud’ can also be ‘flat’
- **min\_range** – optional min range, defaults to NaN, meaning that it is to be the minimum value of the data
- **max\_range** – optional max range, defaults to NaN, meaning that it is to be the maximum value of the data
- **kwargs** –
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

array with tuple of figures and their axis

### 28.1.4 basico.jws\_online module

Convenience module to access JWS models

This module provides convenience functions for accessing models from JWS Online <<https://jjj.mib.ac.uk>> using the netherlands endpoint.

#### Example

```
>>> # get all models with ATP
>>> atp_models = get_models_for_species('atp')
```

```
>>> # get all models with PFK
>>> pfk_models = get_models_for_reaction('pfk')
```

```
>>> # get all models (this will take a while)
>>> all = get_all_models()
```

```
>>> # get manuscript information
>>> manuscript = get_manuscript('teusink')
>>> print(manuscript['title'], manuscript['abstract'], manuscript['url'])
```

```
>>> # get info for a specific model
>>> info = get_model_info('teusink')
>>> print(info['name'], info['status'])
```

```
>>> # get content of specific model
>>> sbml = get_sbml_model('teusink')
>>> print(sbml)
```

`basico.jws_online.download_from(url)`

Convenience method reading content from a URL.

This convenience method uses `urlopen` on either python 2.7 or 3.x

**Parameters****url** (*str*) – the url to read from**Returns**

the contents of the URL as str

**Return type**

str

`basico.jws_online.download_json(url)`

Convenience method reading the content of the url as JSON.

**Parameters****url** (*str*) – the url to read from**Returns**

a python object representing the json content loaded

**Return type**

dict

`basico.jws_online.get_all_models()`

Returns the list of all models

```
>>> get_all_models()
[...
  {
    'slug': 'zi1',
    'name': 'zi1',
    'cbm': False,
    'status': 'CURATED',
    'details': 'https://jjj.bio.vu.nl/rest/models/zi1/',
    'manuscript': 'https://jjj.bio.vu.nl/rest/models/zi1/manuscript/',
    'experiments': 'https://jjj.bio.vu.nl/rest/models/zi1/experiments/'
  }
]
```

**Returns**

list of model ids

`basico.jws_online.get_manuscript(model_id)`

Returns information about the model manuscript

```
>>> get_manuscript('wolf')
```

**Parameters****model\_id** – valid model slug**Returns**

manuscript structure (list of dictionary)

`basico.jws_online.get_mathematica_model(model_id)`

Return the mathematica model for the slug

**Parameters****model\_id** – valid model slug

### Returns

the model as mathematica notebook

`basico.jws_online.get_model_info(model_id)`

Returns information about the JWS model

```
>>> get_model_info('wolf')
{
  'slug': 'wolf',
  'id': 'wolf',
  'name': 'wolf',
  'cbm': False,
  'status': 'CURATED',
  'species_set': ['at (ATP)', ... ],
  'reaction_set': ['v_1 (glucose transporter)', ...],
  'event_set': [],
  'parameter_set': ['atot',...]
```

### Parameters

**model\_id** – a valid jws slug

### Returns

structure with information about the model [{}]

`basico.jws_online.get_models_for_reaction(reaction)`

Searches for models containing a specific reaction

```
>>> get_models_for_reaction('pfk')
```

### Parameters

**reaction** – name of the reaction to search for

### Returns

(list of dictionary) of all models containing the reaction

`basico.jws_online.get_models_for_species(species)`

Searches for models containing a specific chemical species

```
>>> get_models_for_species('atp')
```

### Parameters

**species** – the species name to search for

### Returns

(list of dictionary) of all models containing this species

`basico.jws_online.get_sbml_model(model_id)`

Returns the SBML for the model.

### Parameters

**model\_id** – valid model slug

### Returns

the model as sbml string

### 28.1.5 basico.model\_info module

The model\_info module contains basic functionality for interrogating the model.

Here all functionality for interrogating and manipulating the model is hosted. For each of the elements:

- compartments
- species
- parameters
- events
- reactions

you will find functions to add, get, set, and remove them.

**class** basico.model\_info.T

Bases: object

Constants for Task names

Convert between task names to enums

```
>>> T.from_enum(0)
Steady-State
```

```
>>> T.to_enum('Steady-State')
0
```

CROSS\_SECTION = 'Cross Section'

EFM = 'Elementary Flux Modes'

LNA = 'Linear Noise Approximation'

LYAPUNOV\_EXPONENTS = 'Lyapunov Exponents'

MCA = 'Metabolic Control Analysis'

MOIETIES = 'Moieties'

OPTIMIZATION = 'Optimization'

PARAMETER\_ESTIMATION = 'Parameter Estimation'

SCAN = 'Scan'

SENSITIVITIES = 'Sensitivities'

STEADY\_STATE = 'Steady-State'

TIME\_COURSE = 'Time-Course'

TIME\_COURSE\_SENSITIVITIES = 'Time-Course Sensitivities'

TIME\_SCALE\_SEPARATION = 'Time Scale Separation Analysis'

classmethod all\_task\_names()

classmethod from\_enum(int\_value)

**classmethod** `to_enum(value)`

`basico.model_info.add_amount_expressions(**kwargs)`

Utility function that adds model values for all metabolites to the model to compute the amount

The global parameters created will be named `amount(metab_name)`, and so can be accessed at any time. Should the amount already exist, it will not be modified.

**Parameters**

**kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

`basico.model_info.add_compartment(name, initial_size=1.0, **kwargs)`

Adds a new compartment to the model.

**Parameters**

- **name** (*str*) – the name for the new compartment
- **initial\_size** (*float*) – the initial size for the compartment
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - all other parameters from `set_compartment()`.

**Returns**

the compartment added

`basico.model_info.add_default_plot(name, **kwargs)`

Adds a default plot to the list of plots

**Parameters**

- **name** (*str*) – name of the default plot
- **kwargs** – optional arguments
  - *new\_name*: to rename the plot specification
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

none or the name of the plot created

**Return type**

str or None

`basico.model_info.add_equation(eqn, time_symbol='t', **kwargs)`

This function allows to add arbitrary equations to the current model.

This function allows adding arbitrary ODE's / assignments to the model. Nonexisting model entities will be created.

**Parameters**

- **eqn** (*str*) – the equation for example of form:  $d[X]/dt = k1 * \exp(\{Time\})$

- **time\_symbol** (*str*) – optional symbol that will be used for time (defaults to *t*)
- **kwargs**

**Returns**

`basico.model_info.add_event(name, trigger, assignments, **kwargs)`

Adds a new event to the model.

**Parameters**

- **name** (*str*) – the name for the new event
- **trigger** (*str*) – the trigger expression to be used. The expression can consist of all display names. for example *Time > 10* would make the event trigger at time 10.
- **assignments** (*[(str, str)]*) – All the assignments that should be made, when the event fires. This should be a list of tuples where the first element is the name of the element to change, and the second element the assignment expression.
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - *'new\_name'*: the new name for the event
  - *'delay'*: the delay expression
  - *'priority'*: the priority expression
  - *'persistent'*: boolean indicating if the event is persistent
  - *'delay\_calculation'*: **boolean indicating whether just the assignment is delayed, or the calculation as well**
  - *'fire\_at\_initial\_time'*: boolean indicating if the event should fire at the initial time

**Returns**

the newly created event

`basico.model_info.add_event_assignment(name, assignment, exact=False, **kwargs)`

Adds an event assignment to the named event

**Parameters**

- **name** (*str*) – the name (or substring of name) of an event
- **assignment** (*[(str, str)]* or *(str, str)*) – tuple or list of tuples of event assignments of form (target, expression)
- **exact** (*bool*) – boolean indicating whether the named expression has to be exact

**Returns**

None

`basico.model_info.add_function(name, infix, type='general', mapping=None, **kwargs)`

Adds a new function definition if none with that name already exists

**Parameters**

- **name** (*str*) – the name for the new function
- **infix** (*str*) – the formula for the new function (e.g:  $V * S / (K + S)$ )

- **type** (*str*) – optional flag specifying whether the function is ‘reversible’, ‘irreversible’ or ‘general’
- **mapping** (*dict*) – optional dictionary mapping the elements of the infix to their usage. If not specified, the usage will default to *parameter*, other values possible would be *substrate*, *product*, *modifier*, *volume* or *time*. One example for the infix for the infix above we would specify that *S* is *substrate*:  

```
{ 'S': 'substrate' }
```
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

`basico.model_info.add_parameter(name, initial_value=1.0, **kwargs)`

Adds a new global parameter to the model.

#### Parameters

- **name** (*str*) – the name for the new global parameter
- **initial\_value** (*float*) – optional the initial value of the parameter (defaults to 1)
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - all other parameters from `set_parameters()`.

#### Returns

the newly created parameter

`basico.model_info.add_parameter_set(name, param_set_dict=None, **kwargs)`

Adds a new parameter set to the model with the values from the dictionary

#### Parameters

- **name** (*str*) – name of the parameter set to add
- **param\_set\_dict** (*dict or None*) – dictionary with the parameter set values if empty, the current state of the model will be used
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

`basico.model_info.add_plot(name, **kwargs)`

Adds a new plot specification to the model.

#### Parameters

- **name** (*str*) – the name for the new plot specification
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - all other parameters from `set_plot_dict()`.



**Returns**

the plot

```
basico.model_info.add_reaction(name, scheme, **kwargs)
```

Adds a new reaction to the model

**Parameters**

- **name** (*str*) – the name for the new reaction
- **scheme** (*str*) – the reaction scheme for the new reaction, if it includes Species that do not exist yet in the model they will be created. So for example a scheme of  $A \rightarrow B$  would create species  $A$  and  $B$  if they would not exist in the model, before creating the irreversible reaction.
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - all other parameters from `set_reaction()`.

**Returns**

the newly created reaction

```
basico.model_info.add_report(name, **kwargs)
```

Adds a new report specification to the model.

**Examples**

The following would adds a report definition 'Time Course' to include Time and the concentration of S, in a report that is separated by tabs.

```
>>> add_report('Time Course', body=['Time', '[S]'])
```

The following defines a report for the Steady State concentration of S. To disambiguate, that the string '[S]' in the header should be used literally, we call the function `wrap_copasi_string`.

```
>>> add_report('Steady State', task=T.STEADY_STATE, header=[wrap_copasi_string('[S]')], footer=['[S]'])
```

**Parameters**

- **name** (*str*) – the name for the new plot specification
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - all other parameters from `set_report_dict()`.

**Returns**

the report definition

```
basico.model_info.add_species(name, compartment_name="", initial_concentration=1.0, **kwargs)
```

Adds a new species to the model.

**Parameters**

- **name** (*str*) – the name for the new species
- **compartment\_name** (*str*) – optional the name of the compartment in which the species should be created, it will default to the first compartment. If no compartment is present, a unit compartment named *compartment* will be created.
- **initial\_concentration** (*float*) – optional the initial concentration of the species
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - all other parameters from `set_species()`.

#### Returns

the newly created species

`basico.model_info.apply_parameter_set(name, exact=False, **kwargs)`

Applies the parameter set with the given name to the model

#### Parameters

- **name** – the name of the parameter set or a substring of the name
- **exact** – boolean indicating whether the name has to match exactly
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

`basico.model_info.as_dict(df, fold_list=True)`

Convenience function returning the data frame as dictionary

#### Parameters

- **df** (*pd.DataFrame*) – the data frame
- **fold\_list** (*bool*) – optional boolean indicating whether to fold lists into a single dictionary, if there is only one entry (defaults to true)

#### Returns

the contents of the dataframe as `[{ }]` if there are multiple ones, otherwise the dictionary if just one, or None

#### Return type

List[Dict] or Dict or None

`basico.model_info.assign_report(name, task, filename="", append=True, confirm_overwrite=True, **kwargs)`

Assigns the named report to the specified task

#### Parameters

- **name** (*str*) – the name of the report definition to assign
- **task** (*Union[int, str, COPASI.CCopasiTask]*) – the task to assign the report to
- **filename** (*str*) – the filename to write the result to or `'`, if it is the empty, it resets the target of the task, and COPASI will not create that report
- **append** (*bool*) – boolean indicating whether output should be appended (defaults to True)

- **confirm\_overwrite** (*bool*) – boolean indicating whether the copasi should ask before overwriting a file
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

```
basico.model_info.get_cn(name_or_reference, initial=False, **kwargs)
```

Gets the cn of the named element or none

**Parameters**

- **name\_or\_reference** (*str* or *COPASI.CDataObject*) – display name of model element
- **initial** (*bool*) – if True, an initial reference cn will be returned, rather than a transient one
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

the cn if found or None

**Return type**

str or None

```
basico.model_info.get_compartments(name=None, exact=False, **kwargs)
```

Returns all information about the compartments as pandas dataframe.

**Parameters**

- **name** (*str*) – optional filter expression for the compartment, if it is not included in the name, the compartment will not be added to the data set.
- **exact** (*bool*) – boolean indicating, that the name has to be exact
- **kwargs** – optional arguments:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

a pandas dataframe with the information about the compartment

**Return type**

pandas.DataFrame

```
basico.model_info.get_copasi_messages(num_messages_before, filters=None)
```

Returns error messages that occurred while initializing or running the simulation

**Parameters**

- **num\_messages\_before** – number of messages before calling initialization or process
- **filters** (*list of str or str or None*) – optional list of filter expressions of what messages to ignore

**Returns**

error messages in form of a string

`basico.model_info.get_default_plot_names(filter=None, **kwargs)`

Returns a list of default plot names

#### Parameters

- **filter** – optional filter of substring to be in the name
- **kwargs**

#### Returns

`basico.model_info.get_events(name=None, exact=False, **kwargs)`

Returns all information about the events as pandas dataframe.

#### Parameters

- **name** (*str*) – optional filter expression for the event, if it is not included in the event name, the event will not be added to the data set.
- **exact** (*bool*) – boolean indicating whether the name has to be exact
- **kwargs** – optional arguments:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

a pandas dataframe with the information about the event

#### Return type

`pandas.DataFrame`

`basico.model_info.get_functions(name=None, **kwargs)`

Returns all available functions as pandas dataframe.

#### Parameters

- **name** (*str*) – optional filter expression for the functions, if it is not included in the name, the function will not be added to the data set.
- **kwargs** – optional arguments:
  - *reversible*: to further filter for functions that are only reversible
  - *suitable\_for*: an optional reaction for which to filter the function list. Only functions suitable for the reaction will be returned
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

a pandas dataframe with the information about the functions

#### Return type

`pandas.DataFrame`

`basico.model_info.get_jacobian_matrix(apply_initial_values=False, **kwargs)`

Returns the jacobian matrix of the model at the current state

#### Parameters

- **apply\_initial\_values** (*bool*) – if set to the the initial values will be applied, otherwise the jacobian from the current state will be returned
- **kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

the stoichiometry matrix of the current model

**Return type**

pd.DataFrame

`basico.model_info.get_miriam_annotation(**kwargs)`

Returns the elements miriam annotations as dictionary of the form:

```
{
  'created': datetime, 'creators': [{
    'first_name': '...', 'last_name': '...', 'email': '...', 'organization': '...'
  },...],
  'references': [{
    'id': '...', 'uri': 'identifiers.org uri', 'resource': 'human readable name of resource', 'description',
    '...'
  },...],
  'descriptions': [{
    'id': '...', 'qualifier': 'human readable qualifier string', 'uri': 'identifiers.org uri', 'resource': 'name
    of the resource referenced'
  },...], 'modifications': [datetime,...]
}
```

**Parameters**

**kwargs** – optional parameters, recognized are:

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *name*: the display name of the element to set the notes on. otherwise the main model will be taken.
- *element*: any model element

**Returns**

the elements annotation as dictionary as described

**Return type**

{}

`basico.model_info.get_miriam_resources(compact=True)`

Retrieves the current MIRIAM resources from the configuration

**Parameters**

**compact** (*bool*) – whether to return a compact version of the resources (default: True)

**Returns**

dataframe with the list of current miriam resources

**Return type**

pandas.DataFrame

`basico.model_info.get_model_name(**kwargs)`

Returns the name of the current model.

**Parameters**

**kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

the name of the model

`basico.model_info.get_model_units(**kwargs)`

Returns all model units as dictionary.

**Parameters**

**kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

a dictionary containing the model units in the form: | { | 'time\_unit': '', | 'quantity\_unit': '', | 'length\_unit': '', | 'area\_unit': '', | 'volume\_unit': '', | }

`basico.model_info.get_notes(**kwargs)`

Returns all notes on the element or model.

**Parameters**

**kwargs** – optional parameters, recognized are:

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *name*: the display name of the element to set the notes on. otherwise the main model will be taken.
- *element*: any model element

**Returns**

the notes string (plain text, or xhtml)

**Return type**

str

`basico.model_info.get_parameter_sets(name=None, exact=False, values_only=False, **kwargs)`

Returns the list of parameter sets

**Parameters**

- **name** (*str*) – name of the parameter set to return (or a substring of the name)
- **exact** (*bool*) – boolean indicating whether the name has to be exact or not (default: False)
- **values\_only** (*bool*) – boolean indicating whether to return only the values of the entries of the parameter set or a dictionary describing it (default: False)
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

`basico.model_info.get_parameters(name=None, exact=False, **kwargs)`

Returns all information about the global parameters as pandas dataframe.

#### Parameters

- **name** (*str*) – optional filter expression for the parameters, if it is not included in the name, the parameter will not be added to the data set.
- **exact** (*bool*) – boolean indicating that the name has to be exact
- **kwargs** – optional arguments:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

a pandas dataframe with the information about the parameter

#### Return type

`pandas.DataFrame`

`basico.model_info.get_plot_dict(plot_spec, **kwargs)`

Returns the information for the specified plot

#### Parameters

- **plot\_spec** (*Union[str, int, COPASI.CPlotSpecification]*) – the name, index or plot specification object
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

dictionary of the form: `{ | 'name': 'Phase Plot', | 'active': True, | 'log_x': False, | 'log_y': False, | 'tasks': '', | 'curves': [ | { | 'name': '[Y]||[X]', # the name of the curve | 'type': 'curve2d', # type of the curve (one of curve2d, histoItem1d, bandedGraph | or spectrogram) | 'channels': ['[X]', '[Y]'], # display names of all the items to be plotted | 'color': 'auto', # color as hex rgb value (i.e. '#ff0000' for red) or 'auto' | 'line_type': 'lines', # the line type (one of lines, points, symbols or | lines_and_symbols) | 'line_subtype': 'solid', # line subtype (one of solid, dotted, dashed, dot_dash or | dot_dot_dash) | 'line_width': 2.0, # line width | 'symbol': 'small_cross', # the symbol to be used (one of small_cross, large_cross | or circle) | 'activity': 'during' # when the data should be collected (one of 'before', 'during', 'after') | from task | } | }`

`basico.model_info.get_plots(name=None, **kwargs)`

Returns all information about the plot definitions as pandas dataframe.

#### Parameters

- **name** (*str*) – optional filter expression for the plots, if it is not included in the plot name, the plot will not be added to the data set.
- **kwargs** – optional arguments:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

a pandas dataframe with the information about the plot see also `get_plot_dict()`

#### Return type

`pandas.DataFrame`

`basico.model_info.get_reaction_mapping(reaction, **kwargs)`

Returns the reaction mapping of the given reaction

#### Parameters

- **reaction** (*str* or *COPASI.CReaction*) – name of a reaction, or the reaction object
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

the dictionary with the reaction mapping

#### Return type

{}

`basico.model_info.get_reaction_parameters(name=None, **kwargs)`

Returns all local parameters as pandas dataframe.

This also includes global parameters that are mapped to local ones.

#### Parameters

- **name** (*str*) – optional filter expression, if it is not included in the name, the function will not be added to the data set.
- **kwargs** – optional arguments:
  - *reaction\_name*: to further filter for local parameters of only certain reactions (that contain the substring)
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

a pandas dataframe with the information about local parameters

#### Return type

`pandas.DataFrame`

`basico.model_info.get_reactions(name=None, exact=False, **kwargs)`

Returns all reactions as pandas dataframe.

#### Parameters

- **name** (*str*) – optional filter expression, if it is not included in the name, the reaction will not be added to the data set.
- **exact** (*bool*) – boolean indicating, that the name has to be exact
- **kwargs** – optional arguments:
  - *reaction\_name*: to further filter for local parameters of only certain reactions (that contain the substring)
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

a pandas dataframe with the information about local parameters

#### Return type

`pandas.DataFrame`



`basico.model_info.get_reduced_jacobian_matrix(apply_initial_values=False, **kwargs)`

Returns the jacobian matrix of the reduced model at the current state

#### Parameters

- **apply\_initial\_values** (*bool*) – if set to the the initial values will be applied, otherwise the jacobian from the current state will be returned
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

the stoichiometry matrix of the reduced current model

#### Return type

pd.DataFrame

`basico.model_info.get_reduced_stoichiometry_matrix(**kwargs)`

Returns the reduced stoichiometry matrix of the model

#### Returns

the stoichiometry matrix of the current model

#### Return type

pd.DataFrame

`basico.model_info.get_report_dict(report, **kwargs)`

Returns all information about the plot as dictionary

#### Parameters

- **report** (*COPASI.CReportDefinition or int or str*) – report definition, index or name
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

a dictionary with all information about the plot

#### Return type

dict

`basico.model_info.get_reports(name=None, ignore_automatic=False, task=None, **kwargs)`

Returns the reports as dataframe

#### Parameters

- **name** – optional filter by name
- **ignore\_automatic** – if true, only manually created reports are returned
- **task** – optional task name, to the get the report for
- **kwargs** – optional arguments:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

a data frame with all the report information

**Return type**

pd.DataFrame

`basico.model_info.get_scheduled_tasks(**kwargs)`

Returns the list of scheduled tasks

**Parameters**

**kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

list of tasks that are scheduled

**Return type**

[str]

`basico.model_info.get_species(name=None, exact=False, **kwargs)`

Returns all information about the species as pandas dataframe.

Example:

Assume you have the brusselator example loaded `load_example('brusselator')`

```
>>> get_species()
```

returns you a dataframe of all species with the species name as index.

```
>>> get_species('X')
```

returns you only those species, that include *X* in the name.

**Parameters**

- **name** (*str*) – optional filter expression for the species, if it is not included in the species name, the species will not be added to the data set.
- **exact** (*bool*) – if true, the name has to match precisely the name of the species
- **kwargs** – optional arguments to further filter down the species. recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - *compartment*: to filter down only species in specific compartments
  - *type*: to filter for species of specific simulation type

**Returns**

a pandas dataframe with the information about the species

**Return type**

pandas.DataFrame

`basico.model_info.get_stoichiometry_matrix(**kwargs)`

Returns the stoichiometry matrix of the model

**Returns**

the stoichiometry matrix of the current model

**Return type**

pd.DataFrame

`basico.model_info.get_task_settings(task, basic_only=True, **kwargs)`

Returns the settings of the given task

**Parameters**

- **task** (*COPASI.CCopasiTask* or *str*) – the task to read the settings of
- **basic\_only** (*bool*) – boolean flag, indicating that only the basic parameters should be returned
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

dict of task settings

**Return type**

{}

`basico.model_info.get_time_unit(**kwargs)`

Returns the time unit of the model

`basico.model_info.get_value(name_or_reference, initial=False, **kwargs)`

Gets the value of the named element or nones

**Parameters**

- **name\_or\_reference** (*str* or *COPASI.CDataObject*) – display name of model element
- **initial** (*bool* or *None*) – if True, an initial value will be returned, rather than a transient one. If set to *None*, the default reference will be returned and not coerced.
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

the value if found or None

**Return type**

float or None

`basico.model_info.have_miriam_resources()`

Utility function returning whether MIRIAM resources are available

**Returns**

boolean indicating whether there are MIRIAM resources available or not

**Return type**

bool

`basico.model_info.remove_amount_expressions(**kwargs)`

Utility function that removes model values created using `add_amount_expressions`.

The global parameters created will be named `amount(metab_name)`, and so can be accessed at any time.

**Parameters**

- kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

### Returns

None

`basico.model_info.remove_compartment(name, **kwargs)`

Deletes the named compartment (and everything included)

### Parameters

- **name** (*str*) – the name of a compartment in the model
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

### Returns

None

`basico.model_info.remove_event(name, **kwargs)`

Deletes the named event

### Parameters

- **name** (*str*) – the name of an event in the model
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

### Returns

None

`basico.model_info.remove_function(name, **kwargs)`

Removes the function with the given name

### Parameters

- **name** (*str*) – the name of the function to be removed
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

### Returns

`basico.model_info.remove_parameter(name, **kwargs)`

Deletes the named global parameter

This will also delete any model element that uses this parameter, so if it appears in any model expression, the elements using these expressions will also be deleted. To prevent that, use the recursive parameter.

### Parameters

- **name** (*str* | *List[str]*) – the name of a parameter in the model
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

### Returns

None

`basico.model_info.remove_parameter_sets(name=None, exact=False, **kwargs)`

remove the named parameter set(s)

#### Parameters

- **name** (*str*) – name of the parameter set to remove (or a substring of the name)
- **exact** (*bool*) – boolean indicating whether the name has to be exact
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

`basico.model_info.remove_plot(name, **kwargs)`

Deletes the named plot

#### Parameters

- **name** (*str*) – the name of an plot in the model
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.remove_reaction(name, **kwargs)`

Deletes the named reaction

#### Parameters

- **name** (*str*) – the name of a reaction in the model
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.remove_report(name, **kwargs)`

Deletes the named report

#### Parameters

- **name** (*str*) – the name of a report in the model
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.remove_report_from_task(task, **kwargs)`

Clears the report filename from the specified task

#### Parameters

- **task** (*Union[int, str, COPASI.CCopasiTask]*) – the task to assign the report to
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.remove_species(name, **kwargs)`

Deletes the named species

#### Parameters

- **name** (*str*) – the name of a species in the model
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.remove_user_defined_functions()`

Removes all user defined functions along with all elements that still use them

`basico.model_info.run_scheduled_tasks(include_plots=True, include_general_plots=False, plots=None, reports=None, **kwargs)`

Runs all scheduled tasks, optionally producing plots and reports

#### Parameters

- **include\_plots** – boolean indicating whether to produce the plots associated with the task
- **include\_general\_plots** – boolean indicating whether to produce the general plots
- **plots** – optional list of plot dataframes computed by the task
- **reports** – optional list of report dataframes computed by the task

#### Returns

Figure produced if *include\_plots* is true, otherwise None

`basico.model_info.run_task(task_name, include_plots=True, include_general_plots=False, plots=None, reports=None, **kwargs)`

Utility function that runs the named task and returns the result

#### Parameters

- **task\_name** (*str*) – the name of the task e.g. ‘Time-Course’ to run the timecourse task. See the `basico.T` for all the task names.
- **include\_plots** (*bool*) – boolean indicating whether to produce the plots associated with the task
- **include\_general\_plots** (*bool*) – boolean indicating whether to produce the general plots (those not specified to a particular task)
- **plots** (*list[pandas.DataFrame] or None*) – optional list of plot dataframes computed by the task
- **reports** (*list[pandas.DataFrame] or None*) – optional list of report dataframes computed by the task

**Returns**

Figures produced if *include\_plots* is true, otherwise None

`basico.model_info.set_compartment(name=None, exact=False, **kwargs)`

Sets properties of the named compartment

**Parameters**

- **name** (*str*) – the name of the compartment (or a substring of the name)
- **exact** (*bool*) – boolean indicating whether the name has to be exact
- **kwargs** – optional arguments
  - *new\_name*: the new name for the compartment
  - *initial\_value* or *initial\_size*: to set the initial size of the compartment
  - *value* or *size*: to set the transient size of the compartment
  - *initial\_expression*: the initial expression for the compartment
  - *status* or *type*: the type of the compartment one of *fixed*, *assignment* or *ode*
  - *expression*: the expression for the compartment (only valid when type is *ode* or *assignment*)
  - *dimensionality*: sets the dimensionality of the compartment (int value 1..3)
  - *notes*: sets notes for the compartment (either plain text, or valid xhtml)
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

`basico.model_info.set_element_name(element, new_name, **kwargs)`

Sets the name of the element

**Parameters**

- **element** (*COPASI.CDataObject* or *str*) – the element whose name to change
- **new\_name** (*str*) – the new name for the element
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

`basico.model_info.set_event(name, exact=False, trigger=None, assignments=None, **kwargs)`

Sets properties of the named event

**Parameters**

- **name** (*str*) – the name of the event (or a substring of the name)
- **exact** (*bool*) – boolean indicating, that the name has to be exact
- **trigger** (*str* or *None*) – the trigger expression to be used. The expression can consist of all display names. for example *Time > 10* would make the event trigger at time 10.
- **assignments** (*[(str, str)]* or *None*) – All the assignments that should be made, when the event fires. This should be a list of tuples where the first element is the name of the element to change, and the second element the assignment expression.

- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - 'new\_name': the new name for the event
  - 'delay': the delay expression
  - 'priority': the priority expression
  - 'persistent': boolean indicating if the event is persistent
  - 'delay\_calculation': boolean indicating whether just the assignment is delayed, or the calculation as well
  - 'fire\_at\_initial\_time': boolean indicating if the event should fire at the initial time

#### Returns

`basico.model_info.set_miriam_annotation(created=None, creators=None, references=None, descriptions=None, modifications=None, replace=True, **kwargs)`

Sets the MIRIAM annotations for the provided element or model

#### Parameters

- **created** (*datetime.datetime or None*) – the date/time to set as the objects creation time or None, if not to be set

- **creators** (*list or None*) –

None, if not to be modified, otherwise list of creators of the form:

```
{
    'first_name': '...',
    'last_name': '...',
    'email': '...',
    'organization': '...'
}
```

- **references** (*list or None*) –

None if not to be modified, otherwise list of references of the form:

```
{
    'resource': 'human readable name of resource',
    'id': '...',
    'uri': 'identifiers.org uri',
    'description', '...'
}
```

only the uri needs to be provided, or alternatively id + resource.

- **descriptions** (*list or None*) –

None if not to be modified, otherwise list of descriptions of the form:

```
{
    'resource': 'human readable name of resource',
```



```

        'id': '...',
        'qualifier': '...',
        'uri': 'identifiers.org uri',
    }

```

only the uri needs to be provided, or alternatively id + resource.

- **modifications** (*list or None*) –  
None if not to be modified, otherwise list of datetime objects representing modification dates
- **replace** – Boolean indicating whether existing entries should be removed.

#### Returns

None

`basico.model_info.set_model_name(new_name, **kwargs)`

Renames the model to the provided new name

#### Parameters

- **new\_name** (*str*) – the new name of the model
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.set_model_unit(**kwargs)`

Sets the model units.

#### Parameters

- kwargs** – optional parameters
- *time\_unit*: time unit expression
  - *substance\_unit* or *quantity\_unit*: substance unit expression
  - *length\_unit*: length unit expression
  - *area\_unit*: area unit expression
  - *volume\_unit*: volume unit expression
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

`basico.model_info.set_notes(notes, **kwargs)`

Sets notes on the provided element

#### Parameters

- **notes** (*str*) – the notes to be set, can be either plain text, or valid xhtml
- **kwargs** – optional parameters, recognized are:
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - *name*: the display name of the element to set the notes on.

otherwise the main model will be taken.

- *element*: any model element

#### Returns

None

`basico.model_info.set_parameter_set(name, exact=False, param_set_dict=None, remove_others=False, **kwargs)`

sets the named parameter sets to the given dictionary values

#### Parameters

- **name** (*str*) – name of the parameter set to change (or a substring of the name)
- **exact** (*bool*) – boolean indicating whether the name has to be exact
- **param\_set\_dict** (*dict or None*) – dictionary with the parameter set values
- **remove\_others** (*bool*) – boolean indicating whether to remove entries, that are not specified in the dictionary (default: False)
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

`basico.model_info.set_parameters(name=None, exact=False, **kwargs)`

Sets properties of the named parameter(s).

#### Parameters

- **name** (*str*) – the name of the parameter (or a substring of the name)
- **exact** (*bool*) – boolean indicating whether the name has to be exact or not
- **kwargs** – optional arguments
  - *new\_name*: the new name for the parameter
  - *unit*: the unit expression to be set
  - *initial\_value*: to set the initial value for the parameter
  - *value*: set the transient value for the parameter
  - *initial\_expression*: the initial expression
  - *status* or *type*: the type of the parameter one of *fixed*, *assignment* or *ode*
  - *expression*: the expression for the parameter (only valid when type is *ode* or *assignment*)
  - *notes*: sets notes for the parameter (either plain text, or valid xhtml)
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.set_plot_curves(plot_spec, curves, **kwargs)`

Sets all curves of the named plot specification (all curves will be replaced)

#### Parameters

- **plot\_spec** (*Union[str, int, COPASI.CPlotSpecification]*) – the name, index or plot specification object
- **curves** ([{}]) –  
list of dictionaries of curve items to be added. For example

```
[
    {
        'name': '[Y]([X])', # the name of the curve
        'type': 'curve2d', # type of the curve (one of curve2d, histoItem1d,
        bandedGraph
        or spectrogram)
        'channels': ['[X]', '[Y]'], # display names of all the items to be plotted
        'color': 'auto', # color as hex rgb value (i.e '#ff0000' for red) or 'auto'
        'line_type': 'lines', # the line type (one of lines, points, symbols or
        lines_and_symbols)
        'line_subtype': 'solid', # line subtype (one of solid, dotted, dashed, dot_dash
        or
        dot_dot_dash)
        'line_width': 2.0, # line width
        'symbol': 'small_cross', # the symbol to be used (one of small_cross,
        large_cross or circle)
        'activity': 'during' # when the data should be collected (one of 'before',
        'during', 'after')
    }
]
```

Additionally, histograms may have the bin size in a *increment* element. Spectrograms can have

*log\_z*, *color\_map* (one of 'Default', 'Yellow-Red', 'Grayscale' or 'Blue-White-Red'),  
'bilinear' (should the plot interpolate between values), 'contours' (at which values to draw contours)  
'max\_z' (max z value).

- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from [get\\_current\\_model\(\)](#) will be taken)

### Returns

None

`basico.model_info.set_plot_dict(plot_spec, active=True, log_x=False, log_y=False, tasks="", **kwargs)`

Sets properties of the named plot specification.

### Parameters

- **plot\_spec** (*Union[str, int, COPASI.CPlotSpecification]*) – the name, index or plot specification object
- **active** (*bool*) – boolean indicating whether plot should be active (defaults to true)
- **log\_x** (*bool*) – boolean indicating that the x axis should be logarithmic
- **log\_y** (*bool*) – boolean indicating that the y axis should be logarithmic

- **tasks** (*str*) –  
task type (or colon separated list of task types) for which the plot should be brought up
- **kwargs** – optional arguments
  - *new\_name*: to rename the plot specification
  - *model*: to specify the data model to be used (if not specified the one from [get\\_current\\_model\(\)](#) will be taken)
  - *curves*: dictionary in the format as described in [set\\_plot\\_curves\(\)](#).

#### Returns

None

`basico.model_info.set_reaction(name=None, exact=False, **kwargs)`

Sets attributes of the named reaction.

#### Parameters

- **name** (*str*) – the name of the reaction (or a substring of the name)
- **exact** (*bool*) – boolean indicating whether the name has to be exact
- **kwargs** – optional arguments
  - *new\_name*: new name of the reaction
  - *scheme*: the reaction scheme, new species will be created automatically
  - *function*: the function from the function database to set
  - *mapping*: an optional dictionary that maps model elements to the function parameters. (can be any volume, species, modelvalue or in case of local parameters a value)
  - *notes*: sets notes for the reaction (either plain text, or valid xhtml)
  - *model*: to specify the data model to be used (if not specified the one from [get\\_current\\_model\(\)](#) will be taken)

#### Returns

None

`basico.model_info.set_reaction_mapping(reaction, mapping, **kwargs)`

Sets the reaction mapping of the parameters as specified in the mapping dictionary

#### Parameters

- **reaction** (*str* or *COPASI.CReaction*) – the name of the reaction (or reaction object)
- **mapping** (*{}*) –  
**dictionary that maps model elements to the function**  
parameters. (can be any volume, species, modelvalue or in case of local parameters a value)
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from [get\\_current\\_model\(\)](#) will be taken)

**Returns**

boolean indicating whether the reaction was changed

**Return type**

bool

`basico.model_info.set_reaction_parameters(name=None, **kwargs)`

Sets local parameter values.

**Parameters**

- **name** (*str*) – the name of the parameter (or a substring of the name)
- **kwargs** – optional arguments
  - *reaction\_name*: if specified only parameters of the named reaction will be changed
  - *value*: the new value of the parameter to set. (only one of *value* / *mapped\_to* should be defined)
  - *mapped\_to*: the name of a global parameter to map the local parameter to
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

`basico.model_info.set_report_dict(spec, precision=None, separator=None, table=None, print_headers=True, header=None, body=None, footer=None, task=None, comment=None, add_separator=None, **kwargs)`

Sets properties of the named report definition.

Examples:

The following would set a report definition ‘Time Course’ to include Time and the concentration of S, in a report that is separated by tabs.

```
>>> set_report_dict('Time Course', body=['Time', '[S]'])
```

The following defines a report for the Steady State concentration of S. To disambiguate, that the string ‘[S]’ in the header should be used literally, we call the function `wrap_copasi_string`.

```
>>> set_report_dict('Steady State', task=T.STEADY_STATE, header=[wrap_copasi_string(
    ↳ '[S]')], footer=['[S]'])
```

**Parameters**

- **spec** (*Union[[str](#), [int](#), [COPASI.CReportDefinition](#)]*) – the name, index or report definition object
- **precision** (*Optional[[int](#)]*) – number of digits to print (defaults to 6)
- **separator** (*Optional[[str](#)]*) – the separator to use between elements (defaults to )
- **table** (*[[str](#)]*) – a list of CNs or display names of elements to collect in a table. If *table* is specified the header, body, footer argument will be ignored. Note that setting table elements is only useful for tasks that generate output *during* the task. If that is not the case, you will have to specify the footer and header element directly.
- **print\_headers** (*bool*) – optional arguments, indicating whether table headers will be printed (only applies when the *table* argument is given)

- **header** (*[str]*) – a list of CNs or display names of elements to collect in the header column
- **body** (*[str]*) – a list of CNs or display names of elements to collect in the body rows
- **footer** (*[str]*) – a list of CNs or display names of elements to collect in the footer column
- **task** (*Optional[str]*) –  
task name for which the report should be used
- **comment** (*Optional[str]*) – a documentation string for the report (can be either string, or xhtml string)
- **add\_separator** (*Optional[bool]*) – an optional boolean flag, to automatically add separators between header, body and footer entries since this is not necessary for table entries.
- **kwargs** – optional arguments
  - *new\_name*: to rename the report
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.set_scheduled_tasks(task_name, **kwargs)`

Sets the scheduled tasks

Only the tasks with the listed names will be set to be scheduled.

#### Parameters

- **task\_name** (*str or [str]*) – name or list of names of tasks set to be scheduled
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

None

`basico.model_info.set_species(name=None, exact=False, **kwargs)`

Sets properties of the named species

#### Parameters

- **name** (*str*) – the name of the species (or a substring of the name)
- **exact** (*bool*) – boolean indicating, that the name has to be exact
- **kwargs** – optional arguments
  - *new\_name*: the new name for the species
  - *initial\_concentration*: to set the initial concentration for the species
  - *initial\_particle\_number*: to set the initial particle number for the species
  - *initial\_expression*: the initial expression for the species
  - *concentration*: the new transient concentration for the species
  - *particle\_number*: the new transient particle number for the species

- *status* or *type*: the type of the species one of *fixed*, *assignment* or *ode*
- *expression*: the expression for the species (only valid when type is *ode* or *assignment*)
- *notes*: sets notes for the species (either plain text, or valid xhtml)
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

`basico.model_info.set_task_settings(task, settings, **kwargs)`

Applies the task settings present in the settings object

**Parameters**

- **task** (`COPASI.CCopasiTask` or `str`) – the task to set
- **settings** (`dict`) – dictionary in the same format as the ones obtained from `get_task_settings()`
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

`basico.model_info.set_time_unit(unit, **kwargs)`

Sets the time unit of the model.

**Parameters**

- **unit** (`str`) – the time unit expression
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

`basico.model_info.set_value(name_or_reference, new_value, initial=False, **kwargs)`

Gets the value of the named element or nones

**Parameters**

- **name\_or\_reference** (`str` or `COPASI.CDataObject`) – display name of model element
- **new\_value** (`float`) – the new value to set
- **initial** (`bool` or `None`) – if True, an initial value will be set, rather than a transient one. If set to `None`, the default reference will be returned and not coerced.
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

the value if found or None

**Return type**

float or None

`basico.model_info.simplify_names(**kwargs)`

Simplifies the names of the model elements by removing all special characters and replacing them with under-scores

**Parameters**

**kwargs**

- **model**: the model to simplify
- **drop**: a list of substrings to drop from the names

**Returns**

None

`basico.model_info.update_miriam_resources()`

This method downloads the latest miriam resources from the COPASI website and stores the configuration

`basico.model_info.update_parameter_set(name, exact=False, **kwargs)`

Updates the specified parameter set with values from the model

**Parameters**

- **name** – the name of the parameter set or a substring of the name
- **exact** – boolean indicating whether the name has to match exactly
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

`basico.model_info.wrap_copasi_string(text)`

Utility function wrapping the given text into a COPASI string

**Parameters**

**text** – the text to wrap

**Returns**

an escaped COPASI string, that can be used in reports

## 28.1.6 basico.model\_io module

This module hosts all method for loading/saving new models.

All methods for loading / saving a new model are within this module. As well as the functionality to get/set the currently loaded model. Whenever a function needs the currently loaded model, and no model was provided in the *kwargs*, the `get_current_model()` function retrieves the model loaded last. There are also convenience methods to load models directly from BioModels or from url.

`basico.model_io.create_datamodel()`

creates a new data model

**Returns**

new data model

**Return type**

COPASI.CDataModel

`basico.model_io.get_current_model()`

Returns the current model.

This function returns the current model. That is the model loaded / created last. If no model exists a new one will be created first.



**Returns**

the current model

**Return type**

COPASI.CDataModel

`basico.model_io.get_examples(selector="")`

Returns the filenames of examples bundled with this version.

A number of example models are included with the distribution. This method returns the filenames to those examples. Filtered by the argument.

**Parameters**

**selector** (*str*) – a filter expression to be used, only files matching *\*{selector}.[cps|xml]* will be returned

**Returns**

the list of examples matching

**Return type**

[*str*]

`basico.model_io.get_model_from_dict_or_default(d, key='model')`

Convenience function returning the data model from the dictionary

**Parameters**

- **d** – the dictionary optionally containing the data model
- **key** – the key that the model is under (defaults to 'model')

**Returns**

the data model if found, or the one from `get_current_model()`

**Return type**

COPASI.CDataModel

`basico.model_io.get_num_loaded_models()`

**Returns**

the number of loaded models

`basico.model_io.load_biomodel(model_id, remove_user_defined_functions=False)`

Loads a model from the BioModels Database.

**Parameters**

- **model\_id** (*Union[int, str]*) – either an integer of the biomodels id, or a valid biomodels id
- **remove\_user\_defined\_functions** (*bool*) – optional flag, indicating that user defined functions should be removed before loading the model. Since function definitions are global, this can be helpful to ensure that function names remain the same as in the loaded file. (default: False)

**Returns****Return type**

COPASI.CDataModel

`basico.model_io.load_example(selector)`

Loads the example matching the selector.

**Parameters**

**selector** (*str*) – the filter expression to use for the examples see `get_examples()`

**Returns**

the loaded model, or None, if none matched

**Return type**

COPASI.CDataModel or None

`basico.model_io.load_model(location, remove_user_defined_functions=False)`

Loads the model and sets it as current

**Parameters**

- **location** (*str*) – either a filename, url or raw string of a COPASI / SBML model
- **remove\_user\_defined\_functions** (*bool*) – optional flag, indicating that user defined functions should be removed before loading the model. Since function definitions are global, this can be helpful to ensure that function names remain the same as in the loaded file. (default: False)

**Returns**

the loaded model

**Return type**

COPASI.CDataModel

`basico.model_io.load_model_from_string(content)`

Loads either COPASI model / SBML model from the raw string given.

**Parameters**

**content** (*str or bytes*) – the copasi / sbml model serialized as string

**Returns**

the loaded model

**Return type**

COPASI.CDataModel

`basico.model_io.load_model_from_url(url)`

Loads either COPASI model / SBML model from the url.

**Parameters**

**url** (*str*) – url to a copasi / sbml model

**Returns**

the loaded model

**Return type**

COPASI.CDataModel

`basico.model_io.new_model(**kwargs)`

Creates a new model and sets it as current.

**Parameters**

**kwargs** – optional arguments

- **name** (*str*): the name for the new model
- **quantity\_unit** (*str*): the unit to use for species
- **time\_unit** (*str*): the time unit to use
- **volume\_unit** (*str*): the unit to use for 3D compartments
- **area\_unit** (*str*): the unit to use for 2D compartments
- **length\_unit** (*str*): the unit to use for 1D compartments
- **remove\_user\_defined\_functions** (*bool*): whether to remove user defined functions when creating the model
- **notes**: sets notes for the model (either plain text, or valid xhtml)

**Returns**

the new model

**Return type**

COPASI.CDataModel

`basico.model_io.open_copasi(filename="", **kwargs)`

Saves the model as COPASI file and opens it in COPASI.

The file will be written to a temporary file, and then it will be executed, so that the application registered to open it will start.

**Parameters**

- **filename** (*str*) – the file name to write to, if not given a temp file will be created that will be deleted at the end of the python session.
- **kwargs** – optional arguments:
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

`basico.model_io.overview(model=None)`

Returns a basic representation of the model.

**Parameters****model** (COPASI.CDataModel or None) – the model to get the overview for**Returns**

a string, consisting of name, # compartments, # species, # parameters, # reaction

**Return type**

str

`basico.model_io.print_model(model=None)`

Prints the model overview.

See also `overview()`**Parameters****model** (COPASI.CDataModel) – the model**Returns**

None

`basico.model_io.remove_datamodel(model)`

Removes the model from the internal list of loaded models.

The model is removed from the list of models, and current model, before it is freed

**Parameters****model** – the model to be removed

:type model:COPASI.CDataModel :return: None

`basico.model_io.remove_loaded_models()`

Removes all loaded models

**Returns**

None

`basico.model_io.save_model(filename, **kwargs)`

Saves the model to the given filename.

**Parameters**

- **filename** (*str*) – the file to be written

- **kwargs** – optional arguments:

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *type* (str): *copasi* to write COPASI files, *sbml* to write SBML files (defaults to *copasi*), *sedml* to write SED-ML files, *omex* to write a COMBINE archive
- *overwrite* (bool): whether the file should be overwritten if present (defaults to True)
- *sbml\_level* (int): SBML level to export
- *sbml\_version* (int): SBML version to export
- *sedml\_level* (int): SEDML level to export
- *sedml\_version* (int): SEDML version to export
- *export\_copasi\_miriam* (bool): whether to export copasi miriam annotations
- *export\_incomplete* (bool): whether to export incomplete SBML model
- *include\_copasi* (bool): whether to include the COPASI file in the COMBINE archive (defaults to True)
- *include\_data* (bool): whether to include the data file in the COMBINE archive (defaults to True)
- *include\_sbml* (bool): whether to include the SBML file in the COMBINE archive (defaults to True)
- *include\_sedml* (bool): whether to include the SED-ML file in the COMBINE archive (defaults to False)

#### Returns

None

`basico.model_io.save_model_and_data(filename, **kwargs)`

Saves the model to the give filename, along with all experimental data files.

#### Parameters

- **filename** – the filename of the COPASI file to write
- **filename** – str
- **kwargs** – optional arguments:
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *delete\_data\_on\_exit* (bool): a flag indicating whether the files should be deleted at the end of the python session (defaults to False)

#### Returns

None

`basico.model_io.save_model_to_string(**kwargs)`

Saves the current model to string

#### Parameters

**kwargs** – optional arguments:

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *type* (str): *copasi* to write COPASI files, *sbml* to write SBML files (defaults to *copasi*), *sedml* to write SED-ML files
- *sbml\_level* (int): SBML level to export
- *sbml\_version* (int): SBML version to export
- *sedml\_level* (int): SEDML level to export
- *sedml\_version* (int): SEDML version to export

#### Returns

the copasi model as string

#### Return type

str

```
basico.model_io.set_current_model(model)
```

Sets the current model.

The current model, is the one that all functions will use if not is provided explicitly.

**Parameters**

**model** (*COPASI.CDataModel*) – the model to be set as current

**Returns**

the model

**Return type**

*COPASI.CDataModel*

## 28.1.7 basico.task\_parameterestimation module

Submodule with utility methods for carrying out and plotting of parameter estimations.

The main function provided by this submodule is `run_parameter_estimation()`. Without any parameters, the previously set up parameter estimation as stored in the file will be carried out. And the parameters found will be returned.

It is also possible to set up parameter estimation problems from scratch. To make it as simple as possible, pandas data frames are used, the mapping from the columns to the model element will be done implicitly by naming the columns like the corresponding model elements.

### Example

```
>>> from basico import *
>>> m = model_io.load_example("LM-test1")
>>> print(get_fit_parameters())
>>> print(get_parameters_solution())
>>> run_parameter_estimation(method='Levenberg - Marquardt')
>>> print(get_parameters_solution())
```

```
class basico.task_parameterestimation.PE
```

Bases: object

Constants for Parameter estimation method names

Convert between method names to enums

```
>>> PE.from_enum(0)
'Current Solution Statistics'
```

```
>>> PE.to_enum('Current Solution Statistics')
17
```

```
CURRENT_SOLUTION = 'Current Solution Statistics'
```

```
DIFFERENTIAL_EVOLUTION = 'Differential Evolution'
```

```
EVOLUTIONARY_PROGRAMMING = 'Evolutionary Programming'
```

```
EVOLUTIONARY_STRATEGY_SRES = 'Evolution Strategy (SRES)'
```

```

GENETIC_ALGORITHM = 'Genetic Algorithm'
GENETIC_ALGORITHM_SR = 'Genetic Algorithm SR'
HOOKE_JEEVES = 'Hooke & Jeeves'
LEVENBERG_MARQUARDT = 'Levenberg - Marquardt'
NELDER_MEAD = 'Nelder - Mead'
NL2SOL = 'NL2SOL'
PARTICLE_SWARM = 'Particle Swarm'
PRAXIS = 'Praxis'
RANDOM_SEARCH = 'Random Search'
SCATTER_SEARCH = 'Scatter Search'
SIMULATED_ANNEALING = 'Simulated Annealing'
STEEPEST_DESCENT = 'Steepest Descent'
TRUNCATED_NEWTON = 'Truncated Newton'

classmethod all_method_names()
classmethod from_enum(int_value)
classmethod to_enum(value)

```

`basico.task_parameterestimation.add_experiment(name, data, **kwargs)`

Adds a new experiment to the model.

This method adds a new experiment file to the parameter estimation task. The provided data frame will be written into the current directory as *experiment\_name.txt* unless a filename has been provided.

The mapping between the columns and the model elements should be done by having the columns of the data frame be model element names in question. So for example *[A]* to note that the transient concentrations of a species *A* is to be mapped as dependent variable. or *[A]\_0* to note that the initial concentration of a species *A* is to be mapped as independent variable.

#### Parameters

- **name** (*str*) – the name of the experiment
- **data** (*pandas.DataFrame*) – the data frame with the experimental data
- **kwargs**
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *file\_name* (*str*): the file name to save the experimental data to (otherwise it will be name.txt)
- *data\_dir* (*str*): the directory to save the experimental data to (otherwise it will be the current directory)

#### Returns

the filename of the generated data file

#### Return type

*str*

`basico.task_parameterestimation.add_experiment_from_dict(exp_dict, **kwargs)`

Adds an experiment from dictionary

**Parameters**

**exp\_dict**

**Returns**

`basico.task_parameterestimation.get_data_from_experiment(experiment, **kwargs)`

Returns the data of the given experiment as dataframe

**Parameters**

- **experiment** – the experiment
- **kwargs**
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *rename\_headers* (bool): if true (default) the columns of the headers will be renamed with the names of the element it is mapped to. Also all ignored columns will be removed from the dataset

**Returns**

dataframe with experimental data

**Return type**

pandas.DataFrame

`basico.task_parameterestimation.get_experiment(experiment, **kwargs)`

Returns the specified experiment.

**Parameters**

- **experiment** (*int or str or COPASI.CExperiment*) – experiment name or index
- **kwargs**
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

the experiment or an error if none existent

`basico.task_parameterestimation.get_experiment_data_from_model(model=None)`

Returns all experimental data from the model

**Parameters**

**model** (*COPASI.CDataModel or None*) – the model to get the data from

**Returns**

list of dataframes with experimental data (with columns renamed and unmapped columns dropped)

**Return type**

[pandas.DataFrame]

`basico.task_parameterestimation.get_experiment_dict(experiment, **kwargs)`

Returns all information about the experiment as dictionary

**Parameters**

- **experiment** – copasi experiment, experiment name or index
- **kwargs** – optional arguments
- *model*: to specify the data model to be used (if not specified

- the one from `get_current_model()` will be taken)
- *raise\_error*: boolean indicating that an error should be raised if the experimentfile is not present (default: False)
- *return\_relative*: to indicate that relative experiment filenames should be returned (default: True)

#### Returns

all information about the experiment as dictionary

`basico.task_parameterestimation.get_experiment_filenames(model=None)`

Returns filenames of all experiments

#### Parameters

**model** (*COPASI.CDataModel* or *None*) – the model to get the data from

#### Returns

list of filenames of experimental data

#### Return type

[str]

`basico.task_parameterestimation.get_experiment_mapping(experiment, **kwargs)`

Retrieves a data frame of the experiment mapping.

The resulting data frame will have the columns: \* *column* (int): index of the column in the file \* *type* (str): 'time', 'dependent', 'independent' or 'ignored' \* 'mapping' (str): the name of the element it is mapped to \* 'cn' (str): internal identifier

#### Parameters

- **experiment** – the experiment to get the mapping from
- **kwargs**
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

data frame with the mapping as described

#### Return type

pandas.DataFrame

`basico.task_parameterestimation.get_experiment_names(**kwargs)`

Returns the list of experiment names

#### Parameters

**kwargs**

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

list of experiment names defined

#### Return type

[str]

`basico.task_parameterestimation.get_fit_constraints(model=None)`

Returns a data frame with all fit constraints

The resulting dataframe will have the following columns:

- *name*: the name of the fit parameter
- *lower*: the lower bound of the parameter



- *upper*: the upper bound of the parameter
- *start*: the start value
- *affected*: a list of all experiments (names) the fit parameter should apply to. If empty the parameter should be varied for all experiments.
- *cn*: internal identifier

#### Parameters

**model** (*COPASI.CDataModel* or *None*) – the model to get the fit parameters from

#### Returns

data frame with the fit parameters

#### Return type

pandas.DataFrame

`basico.task_parameterestimation.get_fit_item_template(include_local=False, include_global=False, default_lb=0.001, default_ub=1000, model=None)`

Returns a template list of items to be used for the parameter estimation

#### Parameters

- **include\_local** (*bool*) – boolean, indicating whether to include local parameters
- **include\_global** (*bool*) – boolean indicating whether to include global parameters
- **default\_lb** (*float*) – default lower bound to be used
- **default\_ub** (*float*) – default upper bound to be used
- **model** (*COPASI.CDataModel* or *None*) – the model or None

#### Returns

List of dictionaries, with the local / global parameters in the format needed by: [set\\_fit\\_parameters\(\)](#).

#### Return type

[{}]

`basico.task_parameterestimation.get_fit_parameters(model=None)`

Returns a data frame with all fit parameters

The resulting dataframe will have the following columns:

- *name*: the name of the fit parameter
- *lower*: the lower bound of the parameter
- *upper*: the upper bound of the parameter
- *start*: the start value
- *affected*: a list of all experiments (names) the fit parameter should apply to. If empty the parameter should be varied for all experiments.
- *cn*: internal identifier

#### Parameters

**model** (*COPASI.CDataModel* or *None*) – the model to get the fit parameters from

#### Returns

data frame with the fit parameters

#### Return type

pandas.DataFrame

```
basico.task_parameterestimation.get_fit_statistic(include_parameters=False,
                                                  include_experiments=False, include_fitted=False,
                                                  **kwargs)
```

Return information about the last fit.

#### Parameters

- **include\_parameters** (*bool*) – whether to include information about the parameters in a result entry with key *parameters*
- **include\_experiments** (*bool*) – whether to include information about the experiments in a result entry with key *experiments*
- **include\_fitted** (*bool*) – whether to include information about the fitted values in a result entry with key *fitted*
- **kwargs**
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

dictionary with the fit statistic

#### Return type

{}

```
basico.task_parameterestimation.get_parameters_solution(model=None)
```

Returns the solution found for the fit parameters as data frame

The resulting data frame will have the columns:

- *name*: the name of the parameter
- *lower*: the parameters lower bound
- *upper*: the parameters upper bound
- *sol*: the solution found in the last run (or NaN, if not run yet, or no solution found)
- *affected*: the experiments this parameter applies to (or an empty list if it applies to all)

#### Parameters

**model** (*COPASI.CDataModel* or *None*) – the model to use, or None

#### Returns

data frame as described

#### Return type

pandas.DataFrame

```
basico.task_parameterestimation.get_simulation_results(values_only=False,
                                                       update_parameters=True, **kwargs)
```

Runs the current solution statistics and returns result of simulation and experimental data

#### Parameters

- **values\_only** (*bool*) – if true, only time points at the measurements will be returned
- **update\_parameters** (*bool*) – if set true, the model will be updated with the parameters found from the solution. (defaults to True)
- **kwargs**
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *solution*: a solution data frame to use, if not specified a current solution statistic will be computed

**Returns**

tuple of lists of experiment data, and a list of simulation data

**Return type**

([pandas.DataFrame],[pandas.DataFrame])

`basico.task_parameterestimation.load_experiments_from_dict(experiments, **kwargs)`

Loads all experiments from the specified experiment description

All existing experiments will be replaced with the ones from the specified file.

**Parameters**

- **experiments** – list of experiment dictionaries
- **kwargs** – optional arguments
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

`basico.task_parameterestimation.load_experiments_from_yaml(experiment_description, **kwargs)`

Loads all experiments from the specified experiment description

All existing experiments will be replaced with the ones from the specified file.

**Parameters**

- **experiment\_description** – filename or yamlstring
- **kwargs** – optional arguments
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

`basico.task_parameterestimation.num_experiment_files(**kwargs)`

Return the number of experiment files defined.

**Parameters****kwargs**

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

number of experiment files

**Return type**

int

`basico.task_parameterestimation.num_validations_files(**kwargs)`

Returns the number of cross validation experiment files

**Parameters****kwargs**

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

number of cross validation experiment files

**Return type**

int

`basico.task_parameterestimation.plot_per_dependent_variable(**kwargs)`

This function creates a figure for each dependent variable, with traces for all experiments.

**Parameters**

**kwargs**

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

array of tuples (fig, ax) for each figure created

`basico.task_parameterestimation.plot_per_experiment(**kwargs)`

This function creates one figure per experiment defined, with plots of all dependent variables and their fit in it.

**Parameters**

**kwargs**

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

array of tuples (fig, ax) for the figures created

`basico.task_parameterestimation.prune_simulation_results(simulation_results)`

Removes all columns & time points from the simulation set, that are not available in the measurement set

**Parameters**

**simulation\_results** – the simulation result as obtained by `get_simulation_results`

**Returns**

`basico.task_parameterestimation.remove_experiments(**kwargs)`

Removes all experiments from the model

**Parameters**

**kwargs**

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

`basico.task_parameterestimation.remove_fit_parameters(**kwargs)`

Removes all fit items

**Parameters**

**kwargs**

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

`basico.task_parameterestimation.run_parameter_estimation(**kwargs)`

Runs the parameter estimation task as specified:

The following are valid methods to be used for the parameter estimation task.

Current Solution:

- *Current Solution Statistics*,

Global Methods:

- *Random Search*,
- *Simulated Annealing*,
- *Differential Evolution*,
- *Scatter Search*,
- *Genetic Algorithm*,
- *Evolutionary Programming*,
- *Genetic Algorithm SR*,
- *Evolution Strategy (SRES)*,
- *Particle Swarm*,

Local Methods:

- *Levenberg - Marquardt*,
- *Hooke & Jeeves*,
- *Nelder - Mead*,
- *Steepest Descent*,
- *NL2SOL*,
- *Praxis*,
- *Truncated Newton*,

### Parameters

#### **kwargs**

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *method* (str): one of the strings from above
- *randomize\_start\_values* (bool): if true, parameters will be randomized before starting otherwise the parameters starting value will be taken.
- *calculate\_statistics* (bool): if true, the statistics will be calculated at the end of the task
- *create\_parametersets* (bool): if true, parameter sets will be created for all experiments
- *use\_initial\_values* (bool): whether to use initial values
- *scheduled* (bool): sets whether the task is scheduled or not
- *update\_model* (bool): sets whether the model should be updated, or reset to initial conditions.
- *settings* (dict): **a dictionary with settings to use, in the same format as the ones obtained from `get_task_settings()`**
- *write\_report* (bool): overrides the writing of a report file of filename is specified. (defaults to True)

### Returns

the solution for the fit parameters see `get_parameters_solution()`.

### Return type

pandas.DataFrame

`basico.task_parameterestimation.save_experiments_to_dict(**kwargs)`

Returns a list of dictionaries with the parameter estimation experiments

### Parameters

**kwargs** – optional arguments

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

- *raise\_error*: boolean indicating that an error should be raised if the experimentfile is not present (default: False)
- *return\_relative*: to indicate that relative experiment filenames should be returned (default: True)

#### Returns

the parameter estimation experiments as list of dictionary

#### Return type

[{}]

`basico.task_parameterestimation.save_experiments_to_yaml(filename=None, **kwargs)`

Saves the experiment to yaml

#### Parameters

- **filename** – optional filename to write to
- **kwargs** – optional arguments
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

the yaml string

`basico.task_parameterestimation.set_fit_constraints(fit_constraints, model=None)`

Replaces all existing fit constraints with the ones provided

#### Parameters

- **fit\_constraints** (*pandas.DataFrame* or [{}]) – the fit parameters as pandas data frame of list of dictionaries with keys:
  - 'name' str: the display name of the model element to map the column to.
  - 'lower': the lower bound of the parameter
  - 'upper': the upper bound of the parameter
  - 'start' (float, optional): the start value
  - 'affected' (list[str], optional): a list of affected experiment names.
  - 'cn' (str, optional): internal identifier
- **model** (*COPASI.CDataModel* or *None*) – the model or None

#### Returns

None

`basico.task_parameterestimation.set_fit_parameters(fit_parameters, model=None)`

Replaces all existing fit items with the ones provided

#### Parameters

- **fit\_parameters** (*pandas.DataFrame* or [{}]) – the fit parameters as pandas data frame of list of dictionaries with keys:
  - 'name' str: the display name of the model element to map the column to.
  - 'lower': the lower bound of the parameter
  - 'upper': the upper bound of the parameter
  - 'start' (float, optional): the start value
  - 'affected' (list[str], optional): a list of affected experiment names.

- ‘cn’ (str, optional): internal identifier
- **model** (*COPASI.CDataModel* or *None*) – the model or None

**Returns**

None

### 28.1.8 basico.task\_steadystate module

This module is concerned with bringing the model to steady state.

The `run_steadystate()` method brings the currently loaded model to steady state supporting a number of parameters you could set. Once done, additional calls to functions like `get_species()` or `get_reactions()` will contain the steady state values.

**Example**

```
>>> run_steadystate()
>>> get_species()
```

`basico.task_steadystate.run_steadystate(**kwargs)`

Brings the model to steady state.

The function will use the `get_current_model()` unless one is provided.

**Parameters**

**kwargs** – optional arguments:

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *use\_initial\_values* (bool): whether to use initial values
- *scheduled* (bool): sets whether the task is scheduled or not
- *update\_model* (bool): sets whether the model should be updated, or reset to initial conditions.
- *criterion* (str): specifies the acceptance criterion to be used for a steady state can be one of:
  - \* *Distance and Rate*: both the Distance and the Rate criterion have to be fulfilled to accept
  - \* *Distance*: the distance criterion
  - \* *Rate*: the rate of change has to be sufficiently small
- *settings* (dict): a dictionary with settings to use, in the same format as the ones obtained from `get_task_settings()`

**Returns**

integer status information whether the steady state was reached:

- 0: not found
- 1: steady state found
- 2: equilibrium steady state found

- 3: steady state with negative concentrations found

**Return type**

int

### 28.1.9 basico.task\_timecourse module

This module encapsulates methods for running time course simulations.

main method provided is the `run_time_course()` method, that will simulate the given model (or the current `get_current_model()`).

#### Examples

To run a time course for the duration of 10 time units use

```
>>> run_time_course(10)
```

To run a time course for the duration of 10 time units, in 50 simulation steps use

```
>>> run_time_course(10, 50)
```

To run a time course from 0, for the duration of 10 time units, in 50 simulation steps use:

```
>>> run_time_course(0, 10, 50)
```

all parameters can also be given as key value pairs.

`basico.task_timecourse.create_data_handler`(*output\_selection*, *during=None*, *after=None*, *before=None*, *model=None*)

Creates an output handler for the given selection

**Parameters**

- **output\_selection** (*[str]*) – list of display names or cns, of elements to capture
- **during** (*[str]*) – optional list of elements from the output selection, that should be collected during the run of the task
- **after** (*[str]*) – optional list of elements from the output selection, that should be collected after the run of the task
- **before** (*[str]*) – optional list of elements from the output selection, that should be collected before the run of the task
- **model** – the model in which to resolve the display names

**Returns**

tuple of the data handler from which to retrieve output later, and their columns

**Return type**

(COPASI.CDataHandler, [])

`basico.task_timecourse.get_data_from_data_handler`(*dh*, *columns*)

`basico.task_timecourse.run_time_course`(\**args*, \*\**kwargs*)

Simulates the current or given model, returning a data frame with the results

**Parameters**

- **args** – positional arguments



- 1 argument: the duration to simulate the model
- 2 arguments: the duration and number of steps to take
- 3 arguments: start time, duration, number of steps
- **kwargs** – additional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - *use\_initial\_values* (bool): whether to use initial values
  - *scheduled* (bool): sets whether the task is scheduled or not
  - *update\_model* (bool): sets whether the model should be updated, or reset to initial conditions.
  - *method* (str): sets the simulation method to use (otherwise the previously set method will be used)  
support methods:
    - \* *deterministic / lsoda*: the LSODA implementation
    - \* *stochastic*: the Gibson & Bruck Gillespie implementation
    - \* *directMethod*: Gillespie Direct Method
    - \* others: *hybrid*, *hybridode45*, *hybridlsoda*, *adaptivesa*, *tauleap*, *radau5*, *sde*
  - *duration* (float): the duration in time units for how long to simulate
  - *automatic* (bool): whether to use automatic determined steps (True), or the specified interval / number of steps
  - *output\_event* (bool): if true, output will be collected at the time a discrete event occurs.
  - *values* ([float]): if given, output will only returned at the output points specified  
for example use *values=[0, 1, 4]* to return output only for those three times
  - *start\_time* (float): the output start time. If the model is not at that start time, a simulation  
will be performed in one step, to reach it before starting to collect output.
  - *step\_number* or *intervals* (int): the number of output steps. (will only be used if *automatic*  
or *stepsize* is not used.
  - *stepsize* (float): the output step size (will only be used if *automatic* is False).
  - *seed* (int): set the seed that will be used if *use\_seed* is true, using this stochastic trajectories can  
be repeated
  - 'use\_seed' (bool): if true, the specified seed will be used.
  - *a\_tol* (float): the absolute tolerance to be used
  - *r\_tol* (float): the relative tolerance to be used
  - *max\_steps* (int): the maximum number of internal steps the integrator is allowed to use.
  - *use\_concentrations* (bool): whether to return just the concentrations (default)

- *use\_numbers* (bool): return all elements collected

### Returns

data frame with simulation results

### Return type

pandas.DataFrame

`basico.task_timecourse.run_time_course_with_output(output_selection, *args, **kwargs)`

Simulates the current model, returning only the data specified in the `output_selection` array

### Parameters

- **output\_selection** – selection of elements to return, for example ['Time', '[ATP]', 'ATP.Rate'] to return the time column, ATP concentration and the rate of change of ATP. The strings can be either the Display names as can be found in COPASI, or the CN's of the elements.
- **args** – positional arguments
  - 1 argument: the duration to simulate the model
  - 2 arguments: the duration and number of steps to take
  - 3 arguments: start time, duration, number of steps
- **kwargs** – additional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
  - *use\_initial\_values* (bool): whether to use initial values
  - *scheduled* (bool): sets whether the task is scheduled or not
  - *update\_model* (bool): sets whether the model should be updated, or reset to initial conditions.
  - *method* (str): sets the simulation method to use (otherwise the previously set method will be used)  
support methods:
    - \* *deterministic / lsoda*: the LSODA implementation
    - \* *stochastic*: the Gibson & Bruck Gillespie implementation
    - \* *directMethod*: Gillespie Direct Method
    - \* others: *hybrid*, *hybridode45*, *hybridlsoda*, *adaptivesa*, *tauleap*, *radau5*, *sde*
  - *duration* (float): the duration in time units for how long to simulate
  - *automatic* (bool): whether to use automatic determined steps (True), or the specified interval / number of steps
  - *output\_event* (bool): if true, output will be collected at the time a discrete event occurs.
  - *values* ([float]): if given, output will only returned at the output points specified for example use *values*=[0, 1, 4] to return output only for those three times
  - *start\_time* (float): the output start time. If the model is not at that start time, a simulation will be performed in one step, to reach it before starting to collect output.

- *step\_number* or *intervals* (int): the number of output steps. (will only be used if *automatic* or *stepsize* is not used).
- *stepsize* (float): the output step size (will only be used if *automatic* is False).
- *seed* (int): set the seed that will be used if *use\_seed* is true, using this stochastic trajectories can be repeated
- 'use\_seed' (bool): if true, the specified seed will be used.
- *a\_tol* (float): the absolute tolerance to be used
- *r\_tol* (float): the relative tolerance to be used
- *max\_steps* (int): the maximum number of internal steps the integrator is allowed to use.

### 28.1.10 basico.task\_scan module

Utility functions for dealing with scan tasks

While usually it is easier to encode scan functionality directly in python code, the utility methods here make it easy to set up scan tasks as supported by COPASI directly. That way they can be carried out from the command line version of COPASI, or from the graphical user interface.

`basico.task_scan.add_scan_item(**kwargs)`

Adds the scan item to the model

#### Parameters

**kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *type* (str): the type for the scan item can be one of *scan*, *repeat*, *random* or *parameter\_set*. If not specified *scan* will be used.
- *cn* (str): the cn of the element to use in the scan item (use when no suitable display names for the item you are interested in exist) if you specify cn, don't use the *item* optional paramter
- *item* (str): the display name of the item you want to use, for example `[Signal]_0` for the initial concentration of species *Signal*, or `(r1).k` for the local parameter *k* of reaction *r1*.
- *values* (str or [float]): if you want to scan over specific values, rather than a range, specify them here e.g.: `[0.1, 0.5, 1, 2]`. Using this parameter also sets *use\_values* to *True*
- *parameter\_sets* ([str]): if you want to scan over parameter sets, add the list of parameter set names here.
- *use\_values* (bool): indicates that the values specified should be used rather than the min / max range
- *num\_steps* (int): the number of steps in the range of [min, max] or the number of repeats.

- *min (float)*: minimum value for the range or first distribution parameter
- *max (float)*: maximum value for the range, or the 2nd distributon parameter
- *log (bool)*: boolean indicating that the range should be used logarithmically.
- *distribution (str)*: one of *uniform*, *normal*, *poisson* or *gamma*

### Returns

`basico.task_scan.get_scan_items(**kwargs)`

Retrieves the scan items specified on the scan task:

```
>>> get_scan_items()
[
  {
    'type': 'scan',
    'num_steps': 10,
    'log': False,
    'min': 0.5,
    'max': 2.0,
    'item': '(R1).k1',
  }
]
```

### Parameters

**kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

### Returns

array of dictionary with the scan items specified

### Return type

[{}]

`basico.task_scan.get_scan_items_frame(**kwargs)`

Returns all the scan items as pandas DataFrame

### Parameters

**kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

### Returns

data frame of scan items

`basico.task_scan.get_scan_settings(**kwargs)`

Returns the scan settings as dictionary

```
>>> get_scan_settings()
{
  'update_model': False,
  'scheduled': False,
  'subtask': 'Steady-State',
  'output_during_subtask': False,
  'continue_from_current_state': False,
  'continue_on_error': False,
```

(continues on next page)

(continued from previous page)

```
'scan_items': [ ... ]
}
```

**Parameters****kwargs** – optional parameters

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

dictionary of scan settings

**Return type**

{}

```
basico.task_scan.run_scan(**kwargs)
```

Runs the scan task

**Parameters****kwargs** – optional parameters

- *settings*: optional dictionary with the scan settings
- *scan\_items*: a list of scan items see `set_scan_items()`
- *output* ([str]): optional list of cns or display names, of elements to collect in the scan.
- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None if output is not specified, otherwise the collected output as dataframe.

**Return type**

None or pd.DataFrame

```
basico.task_scan.set_scan_items(scan_items, **kwargs)
```

Replaces the scan items in the task, with the scan items passed in

If you just wanted to change a specific entry, you could would retrieve the current set of scan items, make the change and then set them again:

```
>>> scan_items = get_scan_items()
```

the list of scan items is returned as array of dictionary items, and can be freely modified (say change the number of steps of the first item to a new value), or add new entries, remove some

```
>>> scan_items[0]['num_steps'] = 20
```

Finally a call to scan items, will replace the ones in the model with the ones from the array.

```
>>> set_scan_items(scan_items)
```

**Parameters**

- **scan\_items** ([{}]) – list of dictionaries as returned by `get_scan_items()`.
- **kwargs** – optional parameters
  - *model*: to specify the data model to be used (if not specified)

the one from `get_current_model()` will be taken)

#### Returns

None

`basico.task_scan.set_scan_settings(**kwargs)`

Changes the scan settings

#### Parameters

**kwargs** – optional parameters

- **settings**: dictionary with the scan settings
- **subtask**: sub task name
- **output\_during\_subtask**: boolean, indicating whether output should be collected during the subtask execution
- **continue\_from\_current\_state**: boolean indicating, whether the subtask should be reset to initial values (False) or not (True)
- **continue\_on\_error**: boolean indicating, whether executions should continue, in case one subtask execution failed (True) or not.
- **scan\_items**: a list of scan items see `set_scan_items()`
- **model**: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

#### Returns

### 28.1.11 basico.task\_optimization module

Submodule with utility methods for setting up and carrying out optimization tasks.

The main function provided by this submodule is `run_optimization()`. Without any parameters, the previously set up parameter estimation as stored in the file will be carried out. And the parameters found will be returned.

#### Example

Create an optimization problem

```
>>> from bascio import *
>>> new_model(name='Square')
>>> add_parameter('x', initial_value=0)
>>> add_parameter('y', initial_value=0)
>>> add_parameter('f', type='assignment',
...               expression='(1+{Values[x].InitialValue})^2+(1 + {Values[y].
↪InitialValue})^2')
```

Setup the optimization

```
>>> set_opt_settings({
...     'expression': 'Values[f].InitialValue',
...     'subtask': T.TIME_COURSE,
...     'method': {'name': PE.LEVENBERG_MARQUARDT}
... })
```

Specify which parameters to vary:

```
>>> set_opt_parameters(get_opt_item_template(include_global=True))
```

Run the optimization

```
>>> run_optimization()
```

Get statistic, to see how good the fit was:

```
>>> get_opt_statistic()
```

`basico.task_optimization.get_opt_constraints(model=None)`

Returns a data frame with all constraints to be adhered to during optimization

The resulting dataframe will have the following columns:

- *name*: the name of the fit parameter
- *lower*: the lower bound of the parameter
- *upper*: the upper bound of the parameter
- *start*: the start value
- *cn*: internal identifier

#### Parameters

**model** (*COPASI.CDataModel* or *None*) – the model to get the optitems from

#### Returns

data frame with the constraints

#### Return type

`pandas.DataFrame`

`basico.task_optimization.get_opt_item_template(include_local=False, include_global=False, default_lb=0.001, default_ub=1000, model=None)`

Returns a template dictionary with optimization items

#### Parameters

- **include\_local** (*bool*) – boolean, indicating whether to include local parameters
- **include\_global** (*bool*) – boolean indicating whether to include global parameters
- **default\_lb** (*float*) – default lower bound to be used
- **default\_ub** (*float*) – default upper bound to be used
- **model** (*COPASI.CDataModel* or *None*) – the model or None

#### Returns

List of dictionaries, with the local / global parameters in the format needed by: `set_opt_parameters()`.

#### Return type

`[{}]`

`basico.task_optimization.get_opt_parameters(model=None)`

Returns a data frame with all parameters to be varied during optimization

The resulting dataframe will have the following columns:

- *name*: the name of the fit parameter
- *lower*: the lower bound of the parameter
- *upper*: the upper bound of the parameter
- *start*: the start value

- *cn*: internal identifier

#### Parameters

**model** (*COPASI.CDataModel* or *None*) – the model to get the optitems from

#### Returns

data frame with the fit parameters

#### Return type

pandas.DataFrame

`basico.task_optimization.get_opt_settings(model=None, basic_only=True)`

Returns a dictionary with the optimization setup

The result int dictionary includes the objective function, and the subtask.

#### Parameters

- **model** – the model or None for the current one
- **basic\_only** (*bool*) – boolean flag indicating whether only basic settings should be returned

#### Returns

dictionary with settings

`basico.task_optimization.get_opt_solution(model=None)`

Returns the solution found for the optimization parameters as data frame

The resulting data frame will have the columns:

- *name*: the name of the parameter
- *lower*: the parameters lower bound
- *upper*: the parameters upper bound
- *sol*: the solution found in the last run (or NaN, if not run yet, or no solution found)

#### Parameters

**model** (*COPASI.CDataModel* or *None*) – the model to use, or None

#### Returns

data frame as described

#### Return type

pandas.DataFrame

`basico.task_optimization.get_opt_statistic(**kwargs)`

Return information about the last optimization run.

#### Parameters

**kwargs**

- *model*: to specify the data model to be used (if not specified the one from [get\\_current\\_model\(\)](#) will be taken)

#### Returns

dictionary with the statistic

#### Return type

{}

`basico.task_optimization.run_optimization(expression=None, output=None, settings=None, **kwargs)`

Runs the optimization

#### Parameters

- **expression** – optional objective function to be used



- **settings** – optional settings dictionary
- **output** – optional list of output to collect
- **kwargs** – optional arguments

**Returns**

pandas data frame of the specified output, or the resulting solution for the parameters

`basico.task_optimization.set_objective_function(expression, maximize=None, minimize=None, model=None)`

Sets the objective function to be used

**Parameters**

- **expression** – the expression to be used as objective function, it can contain any display names of model elements. So for example to minimize an initial value of a global parameter  $x$  the expression would look like `Values[x].InitialValue`.
- **maximize** – optional boolean indicating whether the expression should be maximized
- **minimize** – optional boolean indicating whether the expression should be minimized
- **model** – the model to be used or None

**Returns**

None

`basico.task_optimization.set_opt_constraints(opt_constraints, model=None)`

Replaces all existing opt constraints with the ones provided

**Parameters**

- **opt\_constraints** (*pandas.DataFrame* or *[{}]*) – the optimization parameters as pandas data frame of list of dictionaries with keys:
  - 'name' str: the display name of the model element to map the column to.
  - 'lower': the lower bound of the parameter
  - 'upper': the upper bound of the parameter
  - 'start' (float, optional): the start value
  - 'cn' (str, optional): internal identifier
- **model** (*COPASI.CDataModel* or *None*) – the model or None

**Returns**

None

`basico.task_optimization.set_opt_parameters(opt_parameters, model=None)`

Replaces all existing opt items with the ones provided

**Parameters**

- **opt\_parameters** (*pandas.DataFrame* or *[{}]*) – the optimization parameters as pandas data frame of list of dictionaries with keys:
  - 'name' str: the display name of the model element to map the column to.
  - 'lower': the lower bound of the parameter
  - 'upper': the upper bound of the parameter
  - 'start' (float, optional): the start value
  - 'cn' (str, optional): internal identifier

- **model** (*COPASI.CDataModel* or *None*) – the model or None

#### Returns

None

`basico.task_optimization.set_opt_settings(settings, model=None)`

Changes the optimization setup

#### Parameters

- **settings** – a dictionary as returned by [get\\_opt\\_parameters\(\)](#)
- **model** – the model to be used, nor None

#### Returns

### 28.1.12 basico.task\_sensitivities module

Utility methods for working with the sensitivity task

**class** `basico.task_sensitivities.EnumHelper`

Bases: `object`

Utility class for dealing with mapping from names to enums easier

**classmethod** `from_enum(int_value)`

**classmethod** `get_all_names()`

**classmethod** `to_enum(value)`

**class** `basico.task_sensitivities.SENS`

Bases: [EnumHelper](#)

Class for Sensitivity object lists

**ALL\_INITIAL\_VALUES** = 'All initial Values'

**ALL\_LOCAL\_PARAMETER\_VALUES** = 'Local Parameter Values'

**ALL\_ODE\_VARIABLES** = 'All independent Variables of the model'

**ALL\_PARAMETER\_AND\_INITIAL\_VALUES** = 'All Parameter and Initial Values'

**ALL\_PARAMETER\_VALUES** = 'All Parameter Values'

**ALL\_VARIABLES** = 'All Variables of the model'

**ASS\_COMPARTMENT\_VOLUMES** = 'Values of Compartment Volumes with Assignment'

**ASS\_GLOBAL\_PARAMETER\_VALUES** = 'Values of Global Quantities with Assignment'

**ASS\_METAB\_CONCENTRATIONS** = 'Concentrations of Species with Assignment'

**ASS\_METAB\_NUMBERS** = 'Numbers of Species with Assignment'

**COMPARTMENTS** = 'Compartments'

**COMPARTMENT\_INITIAL\_VOLUMES** = 'Compartment Initial Volumes'

**COMPARTMENT\_RATES** = 'Compartment Volume Rates'

```
COMPARTMENT_VOLUMES = 'Compartment Volumes'
CONST_COMPARTMENT_VOLUMES = 'Constant Compartment Volumes'
CONST_GLOBAL_PARAMETER_INITIAL_VALUES = 'Constant Global Quantity Values'
CONST_METAB_CONCENTRATIONS = 'Constant Concentrations of Species'
CONST_METAB_NUMBERS = 'Constant Numbers of Species'
EMPTY_LIST = 'Not Set'
GLOBAL_PARAMETERS = 'Global Quantity'
GLOBAL_PARAMETER_INITIAL_VALUES = 'Global Quantity Initial Values'
GLOBAL_PARAMETER_RATES = 'Global Quantity Rates'
GLOBAL_PARAMETER_VALUES = 'Global Quantity Values'
METABS = 'Species'
METAB_CONCENTRATIONS = 'Concentrations of Species'
METAB_CONC_RATES = 'Concentration Rates'
METAB_INITIAL_CONCENTRATIONS = 'Initial Concentrations'
METAB_INITIAL_NUMBERS = 'Initial Numbers'
METAB_NUMBERS = 'Numbers of Species'
METAB_PART_RATES = 'Particle Rates'
METAB_TRANSITION_TIME = 'Transition Time'
NON_CONST_COMPARTMENT_VOLUMES = 'Non-Constant Compartment Volumes'
NON_CONST_GLOBAL_PARAMETER_VALUES = 'Non-Constant Global Quantity Values'
NON_CONST_METAB_CONCENTRATIONS = 'Non-Constant Concentrations of Species'
NON_CONST_METAB_NUMBERS = 'Non-Constant Numbers of Species'
ODE_COMPARTMENT_VOLUMES = 'Values of Compartment Volumes with ODE'
ODE_GLOBAL_PARAMETER_VALUES = 'Values of Global Quantities with ODE'
ODE_METAB_CONCENTRATIONS = 'Concentrations of Species with ODE'
ODE_METAB_NUMBERS = 'Numbers of Species with ODE'
REACTIONS = 'Reactions'
REACTION_CONC_FLUXES = 'Concentration Fluxes of Reactions'
REACTION_METAB_CONCENTRATIONS = 'Concentrations of Species determined by Reactions'
REACTION_METAB_NUMBERS = 'Numbers of Species determined by Reactions'
REACTION_PART_FLUXES = 'Particle Fluxes of Reactions'
```

REDUCED\_JACOBIAN\_EV\_IM = 'Imaginary part of eigenvalues of the reduced jacobian'

REDUCED\_JACOBIAN\_EV\_RE = 'Real part of eigenvalues of the reduced jacobian'

SINGLE\_OBJECT = 'Single Object'

**class** basico.task\_sensitivities.SENS\_ST

Bases: [EnumHelper](#)

Enumeration of Sensitivity Subtasks

CrossSection = 'Cross Section'

Evaluation = 'Evaluation'

Optimization = 'Optimization'

ParameterEstimation = 'Parameter Estimation'

SteadyState = 'Steady State'

TimeSeries = 'Time Series'

basico.task\_sensitivities.get\_scaled\_sensitivities(\*\*kwargs)

Returns the scaled sensitivity matrix as pandas data frame

**Parameters**

**kwargs** – optional arguments

- *model*: to specify the data model to be used (if not specified the one from [get\\_current\\_model\(\)](#) will be taken)
- *run\_first*: boolean flag indicating that the task should be run first (defaults to false)
- *settings*: a dictionary with the settings to apply first
- *use\_initial\_values*: boolean flag indicating whether initial values should be used (true by default)

**Returns**

the scaled sensitivity matrix

**Return type**

pd.DataFrame

basico.task\_sensitivities.get\_sensitivity\_settings(basic\_only=True, \*\*kwargs)

Returns the settings of the sensitivity task

**Parameters**

- **basic\_only** (*bool*) – if true, only basic parameters will be returned
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from [get\\_current\\_model\(\)](#) will be taken)

**Returns**

the settings as dictionary

**Return type**

dict

`basico.task_sensitivities.get_summarized_sensitivities(**kwargs)`

Returns the summarized sensitivity matrix as pandas data frame

The summarized result is computed as the 2 norm of the scaled result, thus collapsing the dimension

#### Parameters

**kwargs** – optional arguments

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *run\_first*: boolean flag indicating that the task should be run first (defaults to false)
- *settings*: a dictionary with the settings to apply first
- *use\_initial\_values*: boolean flag indicating whether initial values should be used (true by default)

#### Returns

the summarized sensitivity matrix

#### Return type

pd.DataFrame or None

`basico.task_sensitivities.get_unscaled_sensitivities(**kwargs)`

Returns the unscaled sensitivity matrix as pandas data frame

#### Parameters

**kwargs** – optional arguments

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *run\_first*: boolean flag indicating that the task should be run first (defaults to false)
- *settings*: a dictionary with the settings to apply first
- *use\_initial\_values*: boolean flag indicating whether initial values should be used (true by default)

#### Returns

the unscaled sensitivity matrix

#### Return type

pd.DataFrame

`basico.task_sensitivities.run_sensitivities(**kwargs)`

Runs the sensitivity task, the result is obtained

by calling:

- `get_scaled_sensitivities`
- `get_unscaled_sensitivities` or
- `get_summarized_sensitivities`

#### Parameters

**kwargs** – optional arguments

- *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)
- *settings*: a dictionary with the settings to apply first
- *use\_initial\_values*: boolean flag indicating whether initial values should be used (true by default)

**Returns**

None

`basico.task_sensitivities.set_sensitivity_settings(settings, **kwargs)`

Applies the settings to the sensitivity task

**Parameters**

- **settings** (*dict*) – the setting dictionary with keys, as obtained by `get_sensitivity_settings`
- **kwargs** – optional arguments
  - *model*: to specify the data model to be used (if not specified the one from `get_current_model()` will be taken)

**Returns**

None

## PYTHON MODULE INDEX

### b

- `basico`, [149](#)
- `basico.biomodels`, [149](#)
- `basico.compartment_array_tools`, [152](#)
- `basico.jws_online`, [156](#)
- `basico.model_info`, [159](#)
- `basico.model_io`, [186](#)
- `basico.task_optimization`, [208](#)
- `basico.task_parameterestimation`, [191](#)
- `basico.task_scan`, [205](#)
- `basico.task_sensitivities`, [212](#)
- `basico.task_steadystate`, [201](#)
- `basico.task_timecourse`, [202](#)





## A

add\_amount\_expressions() (in module *basico.model\_info*), 160  
 add\_compartment() (in module *basico.model\_info*), 160  
 add\_default\_plot() (in module *basico.model\_info*), 160  
 add\_equation() (in module *basico.model\_info*), 160  
 add\_event() (in module *basico.model\_info*), 161  
 add\_event\_assignment() (in module *basico.model\_info*), 161  
 add\_experiment() (in module *basico.task\_parameterestimation*), 192  
 add\_experiment\_from\_dict() (in module *basico.task\_parameterestimation*), 192  
 add\_function() (in module *basico.model\_info*), 161  
 add\_parameter() (in module *basico.model\_info*), 162  
 add\_parameter\_set() (in module *basico.model\_info*), 162  
 add\_plot() (in module *basico.model\_info*), 162  
 add\_reaction() (in module *basico.model\_info*), 163  
 add\_report() (in module *basico.model\_info*), 163  
 add\_scan\_item() (in module *basico.task\_scan*), 205  
 add\_species() (in module *basico.model\_info*), 163  
 ALL\_INITIAL\_VALUES (*basico.task\_sensitivities.SENS* attribute), 212  
 ALL\_LOCAL\_PARAMETER\_VALUES (*basico.task\_sensitivities.SENS* attribute), 212  
 all\_method\_names() (*basico.task\_parameterestimation.PE* class method), 192  
 ALL\_ODE\_VARIABLES (*basico.task\_sensitivities.SENS* attribute), 212  
 ALL\_PARAMETER\_AND\_INITIAL\_VALUES (*basico.task\_sensitivities.SENS* attribute), 212  
 ALL\_PARAMETER\_VALUES (*basico.task\_sensitivities.SENS* attribute), 212  
 all\_task\_names() (*basico.model\_info.T* class method), 159  
 ALL\_VARIABLES (*basico.task\_sensitivities.SENS* attribute), 212  
 animate\_rectangular\_time\_course() (in module

*basico.compartment\_array\_tools*), 152  
 animate\_rectangular\_time\_course\_as\_image() (in module *basico.compartment\_array\_tools*), 153  
 apply\_parameter\_set() (in module *basico.model\_info*), 164  
 as\_dict() (in module *basico.model\_info*), 164  
 ASS\_COMPARTMENT\_VOLUMES (*basico.task\_sensitivities.SENS* attribute), 212  
 ASS\_GLOBAL\_PARAMETER\_VALUES (*basico.task\_sensitivities.SENS* attribute), 212  
 ASS\_METAB\_CONCENTRATIONS (*basico.task\_sensitivities.SENS* attribute), 212  
 ASS\_METAB\_NUMBERS (*basico.task\_sensitivities.SENS* attribute), 212  
 assign\_report() (in module *basico.model\_info*), 164

## B

basico  
     module, 149  
 basico.biomodels  
     module, 149  
 basico.compartment\_array\_tools  
     module, 152  
 basico.jws\_online  
     module, 156  
 basico.model\_info  
     module, 159  
 basico.model\_io  
     module, 186  
 basico.task\_optimization  
     module, 208  
 basico.task\_parameterestimation  
     module, 191  
 basico.task\_scan  
     module, 205  
 basico.task\_sensitivities  
     module, 212  
 basico.task\_steadystate  
     module, 201  
 basico.task\_timecourse  
     module, 202

## C

COMPARTMENT\_INITIAL\_VOLUMES (*basico.task\_sensitivities.SENS* attribute), 212

COMPARTMENT\_RATES (*basico.task\_sensitivities.SENS* attribute), 212

COMPARTMENT\_VOLUMES (*basico.task\_sensitivities.SENS* attribute), 212

COMPARTMENTS (*basico.task\_sensitivities.SENS* attribute), 212

CONST\_COMPARTMENT\_VOLUMES (*basico.task\_sensitivities.SENS* attribute), 213

CONST\_GLOBAL\_PARAMETER\_INITIAL\_VALUES (*basico.task\_sensitivities.SENS* attribute), 213

CONST\_METAB\_CONCENTRATIONS (*basico.task\_sensitivities.SENS* attribute), 213

CONST\_METAB\_NUMBERS (*basico.task\_sensitivities.SENS* attribute), 213

create\_data\_handler() (in module *basico.task\_timecourse*), 202

create\_datamodel() (in module *basico.model\_io*), 186

create\_linear\_array() (in module *basico.compartment\_array\_tools*), 153

create\_rectangular\_array() (in module *basico.compartment\_array\_tools*), 154

CROSS\_SECTION (*basico.model\_info.T* attribute), 159

CrossSection (*basico.task\_sensitivities.SENS\_ST* attribute), 214

CURRENT\_SOLUTION (*basico.task\_parameterestimation.PE* attribute), 191

## D

delete\_compartments() (in module *basico.compartment\_array\_tools*), 154

DIFFERENTIAL\_EVOLUTION (*basico.task\_parameterestimation.PE* attribute), 191

download\_from() (in module *basico.biomodels*), 150

download\_from() (in module *basico.jws\_online*), 156

download\_json() (in module *basico.biomodels*), 150

download\_json() (in module *basico.jws\_online*), 157

## E

EFM (*basico.model\_info.T* attribute), 159

EMPTY\_LIST (*basico.task\_sensitivities.SENS* attribute), 213

EnumHelper (class in *basico.task\_sensitivities*), 212

Evaluation (*basico.task\_sensitivities.SENS\_ST* attribute), 214

EVOLUTIONARY\_PROGRAMMING (*basico.task\_parameterestimation.PE* attribute), 191

EVOLUTIONARY\_STRATEGY\_SRES (*basico.task\_parameterestimation.PE* attribute),

191

## F

from\_enum() (*basico.model\_info.T* class method), 159

from\_enum() (*basico.task\_parameterestimation.PE* class method), 192

from\_enum() (*basico.task\_sensitivities.EnumHelper* class method), 212

## G

GENETIC\_ALGORITHM (*basico.task\_parameterestimation.PE* attribute), 191

GENETIC\_ALGORITHM\_SR (*basico.task\_parameterestimation.PE* attribute), 192

get\_all\_models() (in module *basico.jws\_online*), 157

get\_all\_names() (*basico.task\_sensitivities.EnumHelper* class method), 212

get\_cn() (in module *basico.model\_info*), 165

get\_compartments() (in module *basico.model\_info*), 165

get\_content\_for\_model() (in module *basico.biomodels*), 150

get\_copasi\_messages() (in module *basico.model\_info*), 165

get\_current\_model() (in module *basico.model\_io*), 186

get\_data\_from\_data\_handler() (in module *basico.task\_timecourse*), 202

get\_data\_from\_experiment() (in module *basico.task\_parameterestimation*), 193

get\_default\_plot\_names() (in module *basico.model\_info*), 165

get\_events() (in module *basico.model\_info*), 166

get\_examples() (in module *basico.model\_io*), 187

get\_experiment() (in module *basico.task\_parameterestimation*), 193

get\_experiment\_data\_from\_model() (in module *basico.task\_parameterestimation*), 193

get\_experiment\_dict() (in module *basico.task\_parameterestimation*), 193

get\_experiment\_filenames() (in module *basico.task\_parameterestimation*), 194

get\_experiment\_mapping() (in module *basico.task\_parameterestimation*), 194

get\_experiment\_names() (in module *basico.task\_parameterestimation*), 194

get\_files\_for\_model() (in module *basico.biomodels*), 150

get\_fit\_constraints() (in module *basico.task\_parameterestimation*), 194

get\_fit\_item\_template() (in module *basico.task\_parameterestimation*), 195  
get\_fit\_parameters() (in module *basico.task\_parameterestimation*), 195  
get\_fit\_statistic() (in module *basico.task\_parameterestimation*), 195  
get\_functions() (in module *basico.model\_info*), 166  
get\_jacobian\_matrix() (in module *basico.model\_info*), 166  
get\_manuscript() (in module *basico.jws\_online*), 157  
get\_mathematica\_model() (in module *basico.jws\_online*), 157  
get\_miriam\_annotation() (in module *basico.model\_info*), 167  
get\_miriam\_resources() (in module *basico.model\_info*), 167  
get\_model\_from\_dict\_or\_default() (in module *basico.model\_io*), 187  
get\_model\_info() (in module *basico.biomodels*), 151  
get\_model\_info() (in module *basico.jws\_online*), 158  
get\_model\_name() (in module *basico.model\_info*), 167  
get\_model\_units() (in module *basico.model\_info*), 168  
get\_models\_for\_reaction() (in module *basico.jws\_online*), 158  
get\_models\_for\_species() (in module *basico.jws\_online*), 158  
get\_notes() (in module *basico.model\_info*), 168  
get\_num\_loaded\_models() (in module *basico.model\_io*), 187  
get\_opt\_constraints() (in module *basico.task\_optimization*), 209  
get\_opt\_item\_template() (in module *basico.task\_optimization*), 209  
get\_opt\_parameters() (in module *basico.task\_optimization*), 209  
get\_opt\_settings() (in module *basico.task\_optimization*), 210  
get\_opt\_solution() (in module *basico.task\_optimization*), 210  
get\_opt\_statistic() (in module *basico.task\_optimization*), 210  
get\_parameter\_sets() (in module *basico.model\_info*), 168  
get\_parameters() (in module *basico.model\_info*), 168  
get\_parameters\_solution() (in module *basico.task\_parameterestimation*), 196  
get\_plot\_dict() (in module *basico.model\_info*), 169  
get\_plots() (in module *basico.model\_info*), 169  
get\_reaction\_mapping() (in module *basico.model\_info*), 169  
get\_reaction\_parameters() (in module *basico.model\_info*), 170  
get\_reactions() (in module *basico.model\_info*), 170  
get\_reduced\_jacobian\_matrix() (in module *basico.model\_info*), 171  
get\_reduced\_stoichiometry\_matrix() (in module *basico.model\_info*), 171  
get\_report\_dict() (in module *basico.model\_info*), 171  
get\_reports() (in module *basico.model\_info*), 171  
get\_sbml\_model() (in module *basico.jws\_online*), 158  
get\_scaled\_sensitivities() (in module *basico.task\_sensitivities*), 214  
get\_scan\_items() (in module *basico.task\_scan*), 206  
get\_scan\_items\_frame() (in module *basico.task\_scan*), 206  
get\_scan\_settings() (in module *basico.task\_scan*), 206  
get\_scheduled\_tasks() (in module *basico.model\_info*), 172  
get\_sensitivity\_settings() (in module *basico.task\_sensitivities*), 214  
get\_simulation\_results() (in module *basico.task\_parameterestimation*), 196  
get\_species() (in module *basico.model\_info*), 172  
get\_stoichiometry\_matrix() (in module *basico.model\_info*), 172  
get\_summarized\_sensitivities() (in module *basico.task\_sensitivities*), 214  
get\_task\_settings() (in module *basico.model\_info*), 172  
get\_time\_unit() (in module *basico.model\_info*), 173  
get\_unscaled\_sensitivities() (in module *basico.task\_sensitivities*), 215  
get\_value() (in module *basico.model\_info*), 173  
GLOBAL\_PARAMETER\_INITIAL\_VALUES (in *basico.task\_sensitivities.SENS* attribute), 213  
GLOBAL\_PARAMETER\_RATES (in *basico.task\_sensitivities.SENS* attribute), 213  
GLOBAL\_PARAMETER\_VALUES (in *basico.task\_sensitivities.SENS* attribute), 213  
GLOBAL\_PARAMETERS (*basico.task\_sensitivities.SENS* attribute), 213  
**H**  
have\_miriam\_resources() (in module *basico.model\_info*), 173  
HOOKE\_JEEVES (*basico.task\_parameterestimation.PE* attribute), 192  
**L**  
LEVENBERG\_MARQUARDT (in *basico.task\_parameterestimation.PE* attribute), 192  
LNA (*basico.model\_info.T* attribute), 159  
load\_biomodel() (in module *basico.model\_io*), 187  
load\_example() (in module *basico.model\_io*), 187

load\_experiments\_from\_dict() (in module *basico.task\_parameterestimation*), 197  
load\_experiments\_from\_yaml() (in module *basico.task\_parameterestimation*), 197  
load\_model() (in module *basico.model\_io*), 188  
load\_model\_from\_string() (in module *basico.model\_io*), 188  
load\_model\_from\_url() (in module *basico.model\_io*), 188  
LYAPUNOV\_EXPONENTS (*basico.model\_info.T* attribute), 159

## M

MCA (*basico.model\_info.T* attribute), 159  
METAB\_CONC\_RATES (*basico.task\_sensitivities.SENS* attribute), 213  
METAB\_CONCENTRATIONS (*basico.task\_sensitivities.SENS* attribute), 213  
METAB\_INITIAL\_CONCENTRATIONS (*basico.task\_sensitivities.SENS* attribute), 213  
METAB\_INITIAL\_NUMBERS (*basico.task\_sensitivities.SENS* attribute), 213  
METAB\_NUMBERS (*basico.task\_sensitivities.SENS* attribute), 213  
METAB\_PART\_RATES (*basico.task\_sensitivities.SENS* attribute), 213  
METAB\_TRANSITION\_TIME (*basico.task\_sensitivities.SENS* attribute), 213  
METABS (*basico.task\_sensitivities.SENS* attribute), 213  
module  
    *basico*, 149  
    *basico.biomodels*, 149  
    *basico.compartment\_array\_tools*, 152  
    *basico.jws\_online*, 156  
    *basico.model\_info*, 159  
    *basico.model\_io*, 186  
    *basico.task\_optimization*, 208  
    *basico.task\_parameterestimation*, 191  
    *basico.task\_scan*, 205  
    *basico.task\_sensitivities*, 212  
    *basico.task\_steadystate*, 201  
    *basico.task\_timecourse*, 202  
MOIETIES (*basico.model\_info.T* attribute), 159

## N

NELDER\_MEAD (*basico.task\_parameterestimation.PE* attribute), 192  
new\_model() (in module *basico.model\_io*), 188  
NL2SOL (*basico.task\_parameterestimation.PE* attribute), 192  
NON\_CONST\_COMPARTMENT\_VOLUMES (*basico.task\_sensitivities.SENS* attribute), 213  
NON\_CONST\_GLOBAL\_PARAMETER\_VALUES (*basico.task\_sensitivities.SENS* attribute), 213

NON\_CONST\_METAB\_CONCENTRATIONS (*basico.task\_sensitivities.SENS* attribute), 213  
NON\_CONST\_METAB\_NUMBERS (*basico.task\_sensitivities.SENS* attribute), 213  
num\_experiment\_files() (in module *basico.task\_parameterestimation*), 197  
num\_validations\_files() (in module *basico.task\_parameterestimation*), 197

## O

ODE\_COMPARTMENT\_VOLUMES (*basico.task\_sensitivities.SENS* attribute), 213  
ODE\_GLOBAL\_PARAMETER\_VALUES (*basico.task\_sensitivities.SENS* attribute), 213  
ODE\_METAB\_CONCENTRATIONS (*basico.task\_sensitivities.SENS* attribute), 213  
ODE\_METAB\_NUMBERS (*basico.task\_sensitivities.SENS* attribute), 213  
open\_copasi() (in module *basico.model\_io*), 189  
OPTIMIZATION (*basico.model\_info.T* attribute), 159  
Optimization (*basico.task\_sensitivities.SENS\_ST* attribute), 214  
overview() (in module *basico.model\_io*), 189

## P

PARAMETER\_ESTIMATION (*basico.model\_info.T* attribute), 159  
ParameterEstimation (*basico.task\_sensitivities.SENS\_ST* attribute), 214  
PARTICLE\_SWARM (*basico.task\_parameterestimation.PE* attribute), 192  
PE (class in *basico.task\_parameterestimation*), 191  
plot\_linear\_time\_course() (in module *basico.compartment\_array\_tools*), 155  
plot\_per\_dependent\_variable() (in module *basico.task\_parameterestimation*), 197  
plot\_per\_experiment() (in module *basico.task\_parameterestimation*), 198  
plot\_rectangular\_time\_course() (in module *basico.compartment\_array\_tools*), 155  
PRAXIS (*basico.task\_parameterestimation.PE* attribute), 192  
print\_model() (in module *basico.model\_io*), 189  
prune\_simulation\_results() (in module *basico.task\_parameterestimation*), 198

## R

RANDOM\_SEARCH (*basico.task\_parameterestimation.PE* attribute), 192  
REACTION\_CONC\_FLUXES (*basico.task\_sensitivities.SENS* attribute), 213  
REACTION\_METAB\_CONCENTRATIONS (*basico.task\_sensitivities.SENS* attribute), 213



REACTION\_METAB\_NUMBERS (in module *basico.task\_sensitivities*), 213  
 REACTION\_PART\_FLUXES (in module *basico.task\_sensitivities*), 213  
 REACTIONS (in module *basico.task\_sensitivities*), 213  
 REDUCED\_JACOBIAN\_EV\_IM (in module *basico.task\_sensitivities*), 213  
 REDUCED\_JACOBIAN\_EV\_RE (in module *basico.task\_sensitivities*), 214  
 remove\_amount\_expressions() (in module *basico.model\_info*), 173  
 remove\_compartment() (in module *basico.model\_info*), 174  
 remove\_datamodel() (in module *basico.model\_io*), 189  
 remove\_event() (in module *basico.model\_info*), 174  
 remove\_experiments() (in module *basico.task\_parameterestimation*), 198  
 remove\_fit\_parameters() (in module *basico.task\_parameterestimation*), 198  
 remove\_function() (in module *basico.model\_info*), 174  
 remove\_loaded\_models() (in module *basico.model\_io*), 189  
 remove\_parameter() (in module *basico.model\_info*), 174  
 remove\_parameter\_sets() (in module *basico.model\_info*), 174  
 remove\_plot() (in module *basico.model\_info*), 175  
 remove\_reaction() (in module *basico.model\_info*), 175  
 remove\_report() (in module *basico.model\_info*), 175  
 remove\_report\_from\_task() (in module *basico.model\_info*), 175  
 remove\_species() (in module *basico.model\_info*), 176  
 remove\_user\_defined\_functions() (in module *basico.model\_info*), 176  
 run\_optimization() (in module *basico.task\_optimization*), 210  
 run\_parameter\_estimation() (in module *basico.task\_parameterestimation*), 198  
 run\_scan() (in module *basico.task\_scan*), 207  
 run\_scheduled\_tasks() (in module *basico.model\_info*), 176  
 run\_sensitivities() (in module *basico.task\_sensitivities*), 215  
 run\_steadystate() (in module *basico.task\_steadystate*), 201  
 run\_task() (in module *basico.model\_info*), 176  
 run\_time\_course() (in module *basico.task\_timecourse*), 202  
 run\_time\_course\_with\_output() (in module *basico.task\_timecourse*), 204

## S

save\_experiments\_to\_dict() (in module *basico.task\_parameterestimation*), 199  
 save\_experiments\_to\_yaml() (in module *basico.task\_parameterestimation*), 200  
 save\_model() (in module *basico.model\_io*), 189  
 save\_model\_and\_data() (in module *basico.model\_io*), 190  
 save\_model\_to\_string() (in module *basico.model\_io*), 190  
 SCAN (in module *basico.model\_info*), 159  
 SCATTER\_SEARCH (in module *basico.task\_parameterestimation*), 192  
 search\_for\_model() (in module *basico.biomodels*), 151  
 SENS (class in *basico.task\_sensitivities*), 212  
 SENS\_ST (class in *basico.task\_sensitivities*), 214  
 SENSITIVITIES (in module *basico.model\_info*), 159  
 set\_compartment() (in module *basico.model\_info*), 177  
 set\_current\_model() (in module *basico.model\_io*), 190  
 set\_element\_name() (in module *basico.model\_info*), 177  
 set\_event() (in module *basico.model\_info*), 177  
 set\_fit\_constraints() (in module *basico.task\_parameterestimation*), 200  
 set\_fit\_parameters() (in module *basico.task\_parameterestimation*), 200  
 set\_miriam\_annotation() (in module *basico.model\_info*), 178  
 set\_model\_name() (in module *basico.model\_info*), 179  
 set\_model\_unit() (in module *basico.model\_info*), 179  
 set\_notes() (in module *basico.model\_info*), 179  
 set\_objective\_function() (in module *basico.task\_optimization*), 211  
 set\_opt\_constraints() (in module *basico.task\_optimization*), 211  
 set\_opt\_parameters() (in module *basico.task\_optimization*), 211  
 set\_opt\_settings() (in module *basico.task\_optimization*), 212  
 set\_parameter\_set() (in module *basico.model\_info*), 180  
 set\_parameters() (in module *basico.model\_info*), 180  
 set\_plot\_curves() (in module *basico.model\_info*), 180  
 set\_plot\_dict() (in module *basico.model\_info*), 181  
 set\_reaction() (in module *basico.model\_info*), 182  
 set\_reaction\_mapping() (in module *basico.model\_info*), 182  
 set\_reaction\_parameters() (in module *basico.model\_info*), 183

set\_report\_dict() (in module *basico.model\_info*),  
     183  
 set\_scan\_items() (in module *basico.task\_scan*), 207  
 set\_scan\_settings() (in module *basico.task\_scan*),  
     208  
 set\_scheduled\_tasks() (in module *basico.model\_info*), 184  
 set\_sensitivity\_settings() (in module *basico.task\_sensitivities*), 216  
 set\_species() (in module *basico.model\_info*), 184  
 set\_task\_settings() (in module *basico.model\_info*),  
     185  
 set\_time\_unit() (in module *basico.model\_info*), 185  
 set\_value() (in module *basico.model\_info*), 185  
 simplify\_names() (in module *basico.model\_info*), 185  
 SIMULATED\_ANNEALING (basico.task\_parameterestimation.PE attribute),  
     192  
 SINGLE\_OBJECT (basico.task\_sensitivities.SENS attribute), 214  
 STEADY\_STATE (basico.model\_info.T attribute), 159  
 SteadyState (basico.task\_sensitivities.SENS\_ST attribute), 214  
 STEEPEST\_DESCENT (basico.task\_parameterestimation.PE attribute),  
     192

## T

T (class in *basico.model\_info*), 159  
 TIME\_COURSE (basico.model\_info.T attribute), 159  
 TIME\_COURSE\_SENSITIVITIES (basico.model\_info.T attribute), 159  
 TIME\_SCALE\_SEPARATION (basico.model\_info.T attribute), 159  
 TimeSeries (basico.task\_sensitivities.SENS\_ST attribute), 214  
 to\_enum() (basico.model\_info.T class method), 159  
 to\_enum() (basico.task\_parameterestimation.PE class method), 192  
 to\_enum() (basico.task\_sensitivities.EnumHelper class method), 212  
 TRUNCATED\_NEWTON (basico.task\_parameterestimation.PE attribute),  
     192

## U

update\_miriam\_resources() (in module *basico.model\_info*), 186  
 update\_parameter\_set() (in module *basico.model\_info*), 186

## W

wrap\_copasi\_string() (in module *basico.model\_info*),  
     186