CRNT4SBML Documentation

Release 0.0.19

Brandon C Reyes

Aug 22, 2019

Contents:

1 CRNT4SBML **1**
- 1.1 Features . . . 1
- 1.2 Credits . . . 1

2 Installation **3**
- 2.1 Creating a Virtual Environment . . . 3
- 2.2 Stable release . . . 4
- 2.3 From sources . . . 4

3 Quick Start **5**
- 3.1 Mass Conservation Approach Example . . . 5
- 3.2 Semi-diffusive Approach Example . . . 6

4 CellDesigner Walkthrough **9**
- 4.1 Mass Conservation Approach . . . 10
- 4.2 Semi-diffusive Approach . . . 13

5 Low Deficiency Approach **15**

6 Mass Conservation Approach Walkthrough **17**

7 Semi-diffusive Approach Walkthrough **23**

8 Numerical Optimization Routine **29**
- 8.1 Feasible Point Method . . . 29
- 8.2 Hybrid Global-Local Searches . . . 30
- 8.3 Pseudocode for Optimization Method . . . 30

9 Numerical Continuation Routine **33**

10 Creating the Equilibrium Manifold **35**

11 Generating Presentable C-graphs **37**

12 Further Examples **39**
- 12.1 Low Deficiency Approach . . . 40
- 12.2 Mass Conservation Approach . . . 43
- 12.3 Semi-diffusive Approach . . . 63

13 Reference **73**
13.1 crnt4sbml_test.CRNT . . . 73
13.2 crnt4sbml_test.Cgraph . . . 77
13.3 crnt4sbml_test.LowDeficiencyApproach . . . 85
13.4 crnt4sbml_test.MassConservationApproach . . . 87
13.5 crnt4sbml_test.SemiDiffusiveApproach . . . 98

14 Contributing **105**
14.1 Types of Contributions . . . 105
14.2 Get Started! . . . 106
14.3 Pull Request Guidelines . . . 107
14.4 Tips . . . 107
14.5 Deploying . . . 107

15 Credits **109**
15.1 Development Lead . . . 109
15.2 Contributors . . . 109

16 History **111**
16.1 0.0.1 (2019-06-18) . . . 111
16.2 0.0.2 (2019-06-26) . . . 111
16.3 0.0.3 (2019-06-29) . . . 111
16.4 0.0.4 (2019-07-12) . . . 111
16.5 0.0.5 (2019-07-12) . . . 111
16.6 0.0.6 (2019-07-16) . . . 112
16.7 0.0.7 (2019-07-19) . . . 112
16.8 0.0.8 (2019-07-23) . . . 112
16.9 0.0.9 (2019-07-23) . . . 112
16.10 0.0.10 (2019-07-23) . . . 112
16.11 0.0.11 (2019-07-24) . . . 112
16.12 0.0.12 (2019-07-27) . . . 112
16.13 0.0.13 (2019-07-29) . . . 112
16.14 0.0.14 (2019-08-5) . . . 112
16.15 0.0.15 (2019-08-6) . . . 113
16.16 0.0.16 (2019-08-6) . . . 113
16.17 0.0.17 (2019-08-7) . . . 113
16.18 0.0.18 (2019-08-7) . . . 113
16.19 0.0.19 (2019-08-22) . . . 113

17 Bibliography **115**

Bibliography **117**

Index **119**

CHAPTER 1

CRNT4SBML

CRNT4SBML is a cross-platform and easily installable Python based package. CRNT4SBML is concentrated on providing a simple workflow for the testing of core CRNT methods directed at detecting bistability in cell signaling pathways endowed with mass action kinetics.

- Free software: Apache Software License 2.0
- Documentation: https://crnt4sbml-test.readthedocs.io.

1.1 Features

- Routine for testing of the Deficiency Zero and One Theorems.
- Routine for running the mass conservation approach.
- Routine for running the semi-diffusive approach.

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Creating a Virtual Environment

The preferred way to use CRNT4SBML is through a virtual environment. A virtual environment for Python is a self-contained directory tree. This environment can have a particular version of Python and Python packages. This is very helpful as it allows one to use different versions of Python and Python packages without their install conflicting with already installed versions. Here we will give a brief description of creating a virtual environment for CRNT4SBML using virtualenv. To begin we first obtain virtualenv through a pip install:

```
$ pip install virtualenv
```

Once virtualenv is installed, download the latest version of Python 3.6 (be sure to take note of the download location). Next we will create a directory to hold all of the virtual environments that we may create called "python_environments":

```
$ mkdir python_environments
```

Now that we have virtualenv and Python version 3.6, we can create the virtual environment crnt4sbml_env in the directory python_environments as follows:

```
$ cd python_environments
$ virtualenv -p /path/to/python/3.6/interpreter crnt4sbml_env
```

The flag "-p" tells virtualenv to create an environment using a specific Python interpreter, note that if a standard download of Python was followed, then the path to the interpreter is as follows "/usr/local/bin/python3.6". One can now see a directory called "crnt4sbml_env" is created in the directory python_environments. We can now activate this environment as follows:

```
$ source /path/to/python_environments/crnt4sbml_env/bin/activate
```

On the command line one should now see "(crnt4sbml_env)" on the left side of the command line, which indicates that one is now working in the virtual environment. Once the environment is activated, one can now install CRNT4SBML as follows:

```
$ pip install crnt4sbml
```

note that this will install crnt4sbml in the virtual environment crnt4sbml_env. One can only use crnt4sbml within this environment. If one wants to stop using the virtual environment, the following command can be used:

```
$ deactivate
```

"(base)" should show up on the left of the command line. One can then use the environment by using the "source" command above.

2.2 Stable release

crnt4sbml can be obtained through a standard pip install as follows:

```
$ pip install crnt4sbml
```

This is the preferred method to install crnt4sbml, as it will always install the most recent stable release. Note that crnt4sbml requires Python version 3.6.

2.3 From sources

The sources for crnt4sbml can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/breye12/crnt4sbml
```

Or download the tarball:

```
$ curl  -OL https://github.com/breye12/crnt4sbml_test/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Quick Start

To begin using CRNT4SBML, start by following the process outlined in *Installation*. Once you have correctly installed CRNT4SBML follow the steps below to obtain a general idea of how one can perform the mass conservation and semi-diffusive approach of [OMYS17]. If you are interested in running the Deficiency Zero and One theorems please consult *Low Deficiency Approach*.

3.1 Mass Conservation Approach Example

In order to run the mass conservation approach one needs to first create an SBML file of the reaction network. The SBML file representing the reaction network for this example is given by Fig1Ci.xml. It is highly encouraged that the user consult *CellDesigner Walkthrough* when considering their own individual network as the format of the SBML file must follow a certain construction to be easily used by crnt4sbml.

To run the mass conservation approach create the following python script:

```
import crnt4sbml_test

network = crnt4sbml_test.CRNT("/path/to/Fig1Ci.xml")

opt = network.get_mass_conservation_approach()

bounds = [(1e-2, 1e2)]*12
concentration_bounds = [(1e-2, 1e2)]*4

params_for_global_min, obj_fun_val_for_params = opt.run_optimization(bounds=bounds,
                                                                     concentration_
↪bounds=concentration_bounds,
                                                                     iterations=15)

multistable_param_ind = opt.run_greedy_continuity_analysis(species="s15",␣
↪parameters=params_for_global_min,
                                                           auto_parameters={
↪'PrincipalContinuationParameter': 'C3'})
```

(continues on next page)

(continued from previous page)

```
opt.generate_report()
```

This will provide the following output along with creating the directory "num_cont_graphs" in your current directory that contains multistability plots. Please note that runtimes and the number of multistability plots produced may vary among different operating systems. Please see *Mass Conservation Approach Walkthrough* for a more detailed explanation of running the mass conservation approach and the provided output.

```
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.294473

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.5590229999999998

Running feasible point method for 15 iterations ...
Elapsed time for feasible point method: 0.10936699999999977

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 17.663542

Running continuity analysis ...
Elapsed time for continuity analysis: 18.639788999999997

The number of feasible points that satisfy the constraints: 15
Total feasible points that give F(x) = 0: 6
Total number of points that passed final_check: 6
Number of multistability plots found: 6
Elements in params_for_global_min that produce multistability:
[0, 1, 2, 3, 4, 5]
```

3.2 Semi-diffusive Approach Example

To run the semi-diffusive approach one needs to create the SBML file specific for semi-diffusive networks. The SBML file representing the reaction network for this example is given by Fig1Cii.xml. It is highly encouraged that the user consult *CellDesigner Walkthrough* when considering their own individual network as the format of the SBML file must follow a certain construction to be easily used by crnt4sbml.

To run the semi-diffusive approach create the following python script:

```
import crnt4sbml_test

network = crnt4sbml_test.CRNT("path/to/Fig1Cii.xml")

opt = network.get_semi_diffusive_approach()

bounds = [(1e-3, 1e2)]*12

params_for_global_min, obj_fun_val_for_params = opt.run_optimization(bounds=bounds)

multistable_param_ind = opt.run_greedy_continuity_analysis(species="s7",␣
↪parameters=params_for_global_min,
```

(continues on next page)

(continued from previous page)

```
                                                                    auto_parameters={
↪'PrincipalContinuationParameter': 're17'})

opt.generate_report()
```

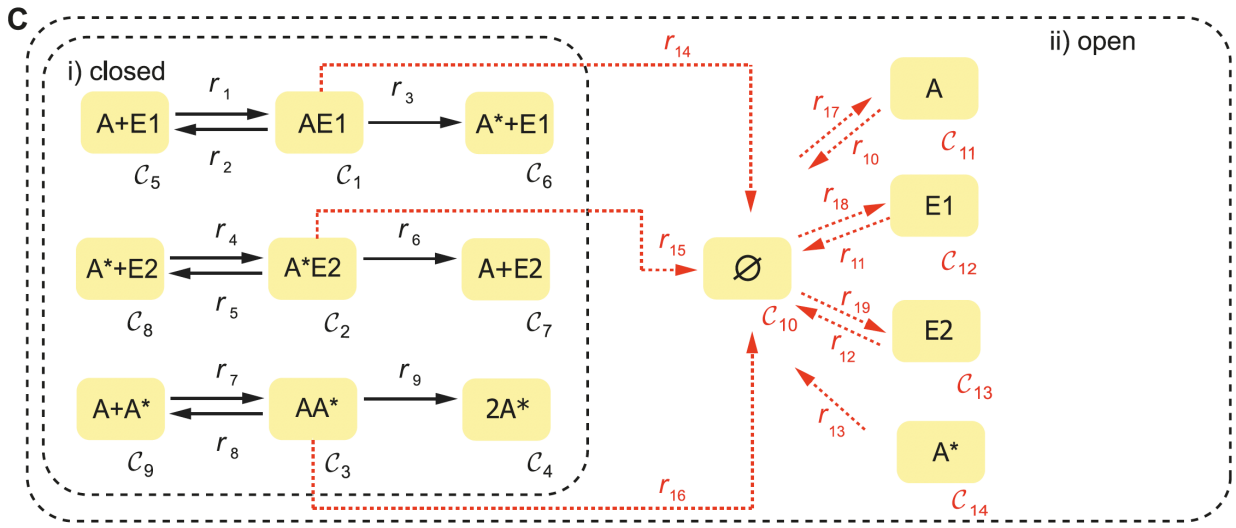
This will provide the following output along with creating the directory "num_cont_graphs" in your current directory that contains multistability plots. Please note that runtimes and the number of multistability plots produced may vary among different operating systems. Please see *Semi-diffusive Approach Walkthrough* for a more detailed explanation of running the semi-diffusive approach and the provided output.

```
Running feasible point method for 10 iterations ...
Elapsed time for feasible point method: 0.6484119999999995

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 43.25512

Running continuity analysis ...
Elapsed time for continuity analysis: 34.786898

The number of feasible points that satisfy the constraints: 10
Total feasible points that give F(x) = 0: 9
Total number of points that passed final_check: 9
Number of multistability plots found: 9
Elements in params_for_global_min that produce multistability:
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

CHAPTER 4

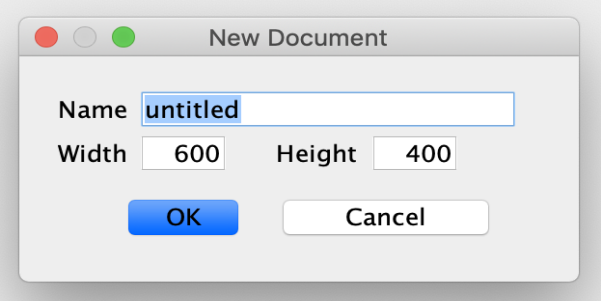
CellDesigner Walkthrough

The following is a walkthrough of how to produce Systems Biology Markup Language (SBML) files that are compatible with CRNT4SBML. The SBML file is a machine-readable format for representing biological models. Our preferred approach to constructing this file is by using CellDesigner. CellDesigner is a structured diagram editor for drawing gene-regulatory and biochemical networks. CellDesigner is a freely available software and can be downloaded by visiting celldesigner.org . Although creating this SBML file may be achievable by other means, use of CellDesigner is the only approach that has been verified to work well with the provided code. Extreme caution should be used if the user wishes to use an already established SBML file or another software that produces an SBML file. We will continue by demonstrating how to represent the C-graph of Figure 1C from [OMYS17] (provided below) for both mass conserving and semi-diffusive networks in CellDesigner. For this demonstration we will be using version 4.4.2 of CellDesigner on a Mac.



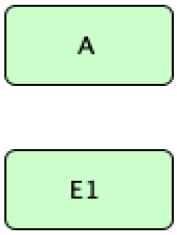
4.1 Mass Conservation Approach

For the mass conservation approach representing the C-graph in CellDesigner is fairly straightforward. To begin, launch CellDesigner and create a new document. The following new document box will then appear. The name provided in this box can be set to anything the user desires and a specific name is not required. In addition to a name, this box also asks for the dimension of white space available in the workspace. The default width of 600 and height of 400 will be appropriate for most small networks.

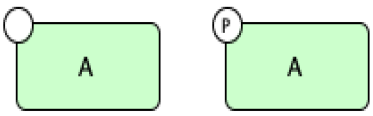


Once the workspace has been created, the species of the network can be represented in CellDesigner by creating a generic protein, which can be found in the top toolbar (as pictured below) by hovering over the symbols.

After the generic protein symbol is selected click on the workspace to create a species. A box will then appear and ask for the species name. Although no specific name is required, for visual purposes it is suggested to use a name that is similar to the name used in the C-graph. Below we have created the species A and $E1$ of the provided C-graph using generic proteins.

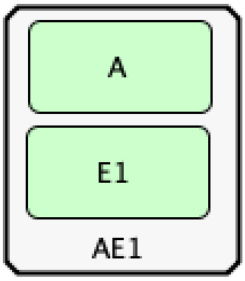


Although regular species are sufficient enough to represent a C-graph, it may also be useful to specify if a particular species is phosphorylated. This can be done by selecting the "Add/Edit Residue Modification" symbol in the top toolbar. A box will then appear and the up and down arrows can be used to select "phosphorylated". After pressing ok, a species can be phosphorylated by hovering over the generic protein and selecting one of the dots on the outline of the protein. One can tell if the species is phosphorylated by noticing if there is a circle with a "P" in the middle. Below we have a species A where the generic protein on the left is not phosphorylated and the generic protein on the right is phosphorylated.

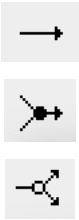


In addition to creating species we can also create chemical complexes as in the C-graph. To do this, in the top toolbox select the symbol "Complex" and click in the workspace, again a specific name is not required, but it is encouraged. One can then place species within this complex using the generic protein approach outlined above. Below is the

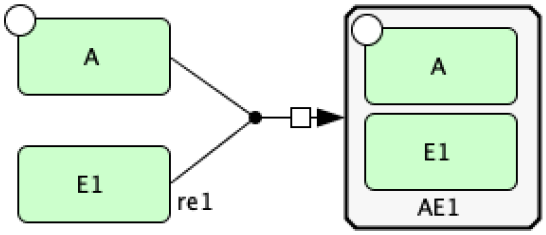
CellDesigner representation of complex $AE1$.



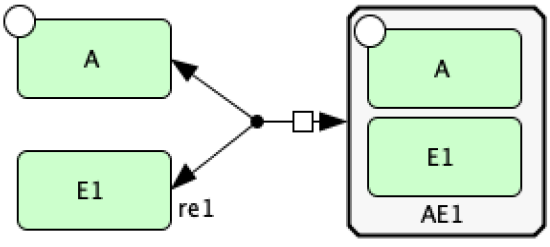
Using species and complexes the C-graph of the mass conservation approach can be completed once the reactions of the network are constructed. In CellDesigner there are three types of reactions that are important when recreating a C-graph: State Transition, Heterodimer Association, and Dissociation (depicted from top to bottom below as in CellDesigner, respectively).



We will first demonstrate Heterodimer Association and Dissociation reactions by creating reactions r_1, r_2, and r_3 of the Cgraph. We will then address State Transition reactions by creating r_9. To create reactions r_1 and r_2, first create species A and $E1$, in addition to complex $AE1$. Then using the top toolbox select "Heterodimer Association" and first select the two species A and $E1$ (order of selection does not matter) then select the complex $AE1$. This concludes the creation of r_1 and should be similar to the picture provided below.

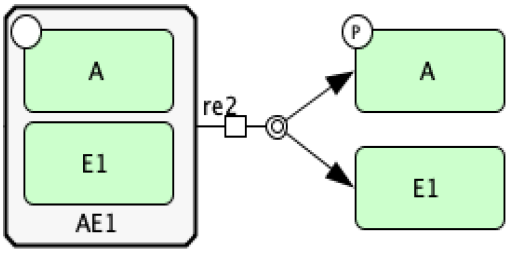


To create r_2 we need to make the heterodimer reaction reversible. To make a reaction reversible right click the reaction and select "Change Identity…", then select True under the reversible category. This provides the CellDesigner representation of r_1 and r_2 as provided below.

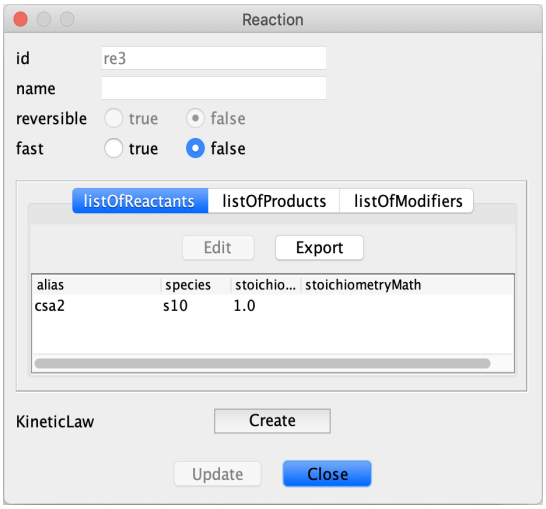


Now we create reaction r_3 using a dissociation reaction. To do this, select "Dissociation" and first select the complex $AE1$ and then select the species $E1$ and phosphorylated species A (the order of selection of the species does not

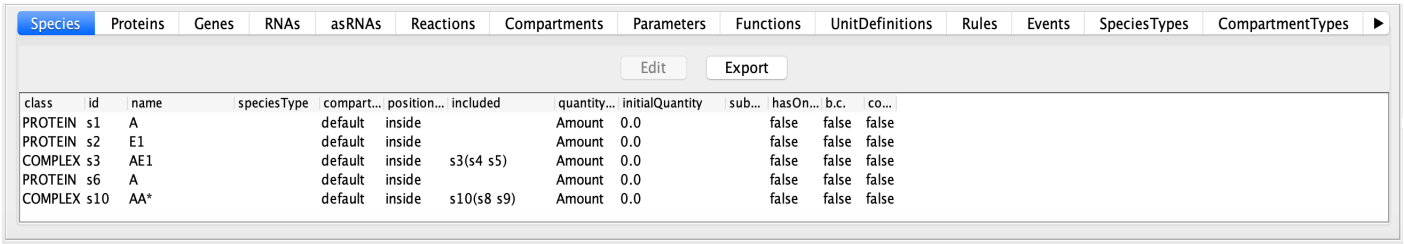
matter). This provides the CellDesigner representation of r_3 below.



The last type of reaction we will consider is a State Transition, to do this we will produce reaction r_9. After creating complex A^*A, we create reaction r_9 by selecting "State Transition" and first click the complex A^*A and then the phosphorylated species A. Although we have created a reaction we have not created r_9 exactly yet. We have not accounted for the fact that two molecules of the phosphorylated species A are produced. To specify this in CellDesigner right click the reaction and select "Edit Reaction…", this opens the following box.

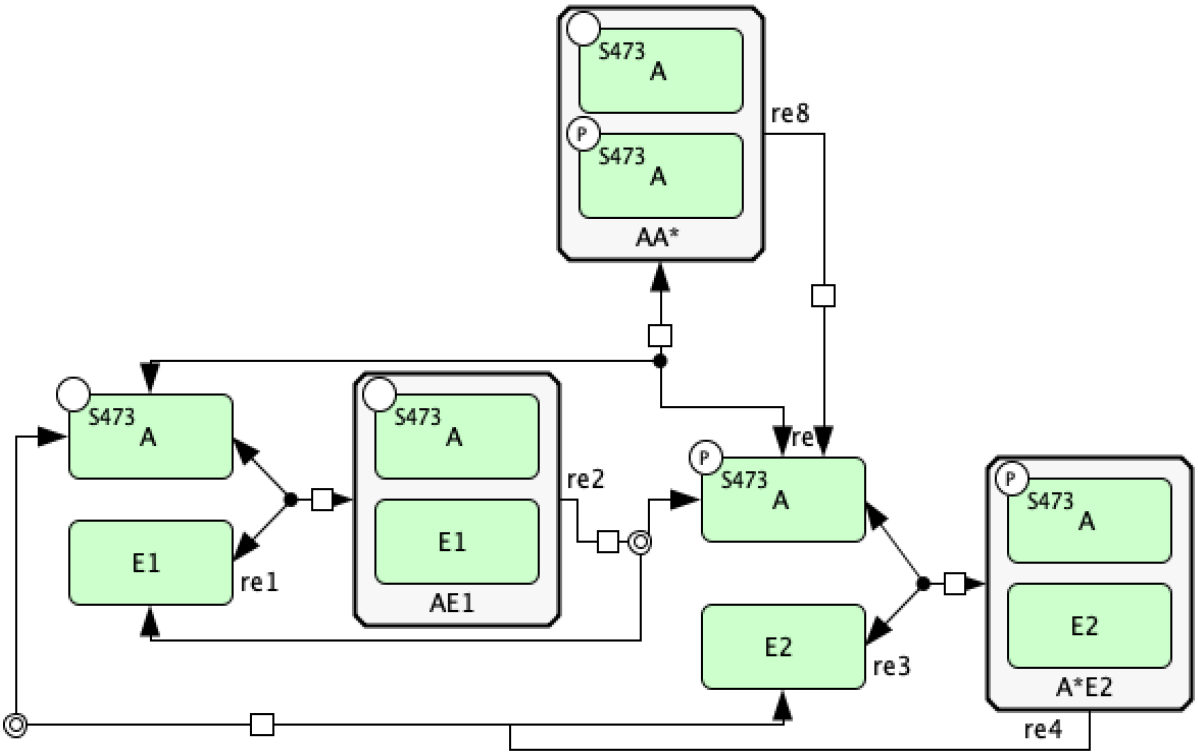


In this box one can then specify the stoichiometry of the reactants and products of the reaction. Note that the species are defined in terms of the species id, rather than the name that the user provided. To obtain the species id one can hover over a species or complex in the workspace, or one can see a list of the species by viewing the bottom box in CellDesigner and selecting the "Species" tab, an example of this box can be seen below.



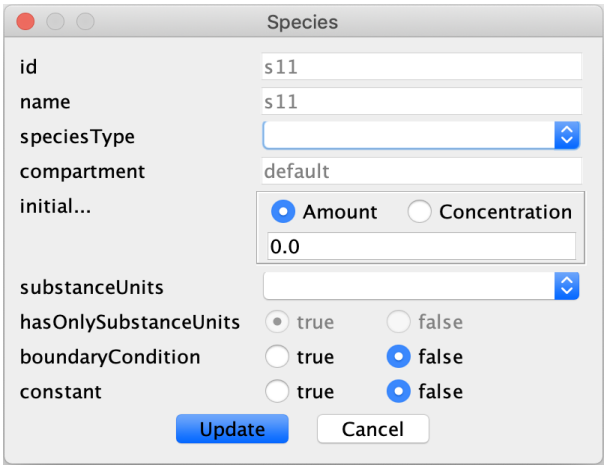
In the reaction box produced by selecting "Edit Reaction…", we can specify that two molecules of phosphorylated species A are produced by selecting the "listOfProducts" tab then clicking the species corresponding to the phosphorylated species A and then selecting Edit and changing stoichiometry to 2.0. We can confirm this change by choosing Update. A similar process can be completed if you want to change the number of molecules of any species in the reactants, but in this case one would instead choose the "listOfReactants" tab. Using the tools we have outlined so far we can represent the provided network in the C-graph using CellDesigner. One particular layout of this CellDesigner representation can be seen below. In this diagram we have manipulated the shape of the reactions by right clicking them and choosing "Add Anchor Point". Note that when saving the CellDesigner diagram, it will be saved as an xml file, this is an xml file with the layout of an SBML file. At this point no conversion to SBML is necessary and the xml

file produced can be imported into the code.

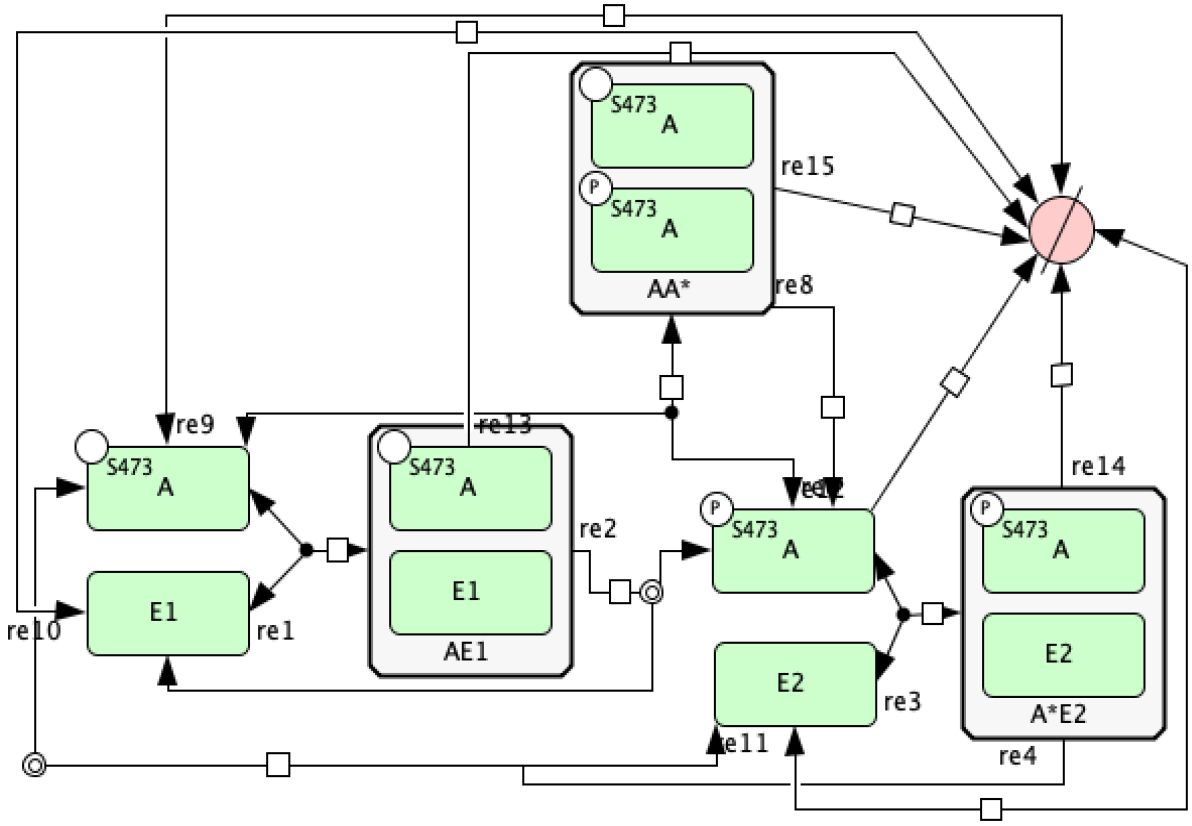


4.2 Semi-diffusive Approach

Now that we have completed the mass conserving network of the provided C-graph we will continue by implementing the semi-diffusive network. Since we are now considering the degradation and formation of a species we have to consider how to implement a source and a sink in the SBML file. Here a source is a node providing an inflow of a species and a sink is an outflow of a species. To do this, we will pick one species to be a boundary species in CellDesigner, for graphical purposes we will use the degradation symbol in CellDesigner (i.e. ∅). This symbol will serve as a sink, source, or both a sink and a source. This usage will prevent unnecessary clutter and make it simpler to create SBML files for open networks. One very important thing to note here is that **the user must specify that this species is a boundary species!** If the user does not do this then the sink/source will be considered as a normal species, this will create incorrect results and will not allow the semi-diffusive approach to be constructed. To create a boundary species right click the "Degraded" symbol in the top toolbox and then click in the workspace. At this point the item produced is just a species, although its appearance differs from a species or a complex. To make this species a source/sink right click the created item and choose "Edit species", the box provided below should appear.



In this box set boundaryCondition to true and choose "Update" to confirm the change. One last word of caution: according to the semi-diffusive approach if there is formation of a species there must also be degradation of that species. However, one can allow for just degradation of a species. Using this convention and the ideas established in the previous subsection we can recreate the open version of the provided C-graph using CellDesigner. One possible layout of this C-graph in CellDesigner is provided below.



CHAPTER 5

Low Deficiency Approach

Now that we have constructed the SBML file using the guidelines of *CellDesigner Walkthrough*, we will proceed by testing the Deficiency Zero and One Theorems of [Fei79]. We will complete this test for Fig1Ci.xml. The first step we must take is importing crnt4sbml. To do this open up a python script and add the following line:

```
import crnt4sbml
```

Next, we will take the SBML file created using CellDesigner and import it into the code. This is done by instantiating CRNT with a string representation of the path to the SBML file. An example of this instantiation is as follows:

```
c = crnt4sbml.CRNT("/path/to/Fig1Ci.xml")
```

Once this line is ran the class CRNT takes the SBML file and parses it into a Python NetworkX object which is then used to identify the basic Chemical Reaction Network Theory properties of the network. To obtain a full list of what is provided by this instantiation, please see the getter methods of *crnt4sbml_test.CRNT()*. To obtain a print out of the number of species, complexes, reactions and deficiency of the network complete the following command:

```
c.basic_report()
```

For the closed portion of the C-graph the output should be as follows:

```
Number of species: 7
Number of complexes: 9
Number of reactions: 9
Network deficiency: 2
```

It is important for the user to verify that the number of species, complexes, reactions, and if possible deficiency values are correct at this stage. To provide another check to make sure the parsing and CellDesigner model were constructed correctly, one is encouraged to print the network constructed. To do this, add the following command to the script:

```
c.print_c_graph()
```

After running this command for the constructed SBML file, the following output is obtained.

```
Reaction graph of the form
reaction -- reaction label:
s1+s2 -> s3  --  re1
s3 -> s1+s2  --  re1r
s3 -> s6+s2  --  re2
s6+s7 -> s16  --  re3
s16 -> s6+s7  --  re3r
s16 -> s7+s1  --  re4
s1+s6 -> s15  --  re6
s15 -> s1+s6  --  re6r
s15 -> 2*s6  --  re8
```

Notice that this output describes the reactions in terms of the species' id and not the species' name. Along with the reactions, the reaction labels constructed during parsing are also returned. For this example the first reaction s1+s2 -> s3 has a reaction label of 're1' and the reaction s15 -> s1+ s6 has a reaction label of 're6r'. Please note that the species id and reaction labels may be different if the user has constructed the SBML file themselves. Further information of the network can be found by analyzing the getter methods of *crnt4sbml_test.Cgraph()*.

Once one has verified that the network and CellDesigner model were created correctly, we can begin to check the properties of the network. If one is only interested in whether or not the network precludes bistability, it is best to first check the Deficiency Zero and One Theorems of Chemical Reaction Network Theory. To do this add the following lines to the script:

```
ldt = c.get_low_deficiency_approach()
ldt.report_deficiency_zero_theorem()
ldt.report_deficiency_one_theorem()
```

This provides the following output for the closed portion of the C-graph:

```
The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.
```

For information on the possible output for this run, please see *crnt4sbml_test.LowDeficiencyApproach.report_deficiency_one_theorem()* and *crnt4sbml_test.LowDeficiencyApproach.report_deficiency_zero_theorem()*.

CHAPTER 6

Mass Conservation Approach Walkthrough

Using the SBML file constructed as in *CellDesigner Walkthrough*, we will proceed by completing a more in-depth explanation of running the mass conservation approach of [OMYS17]. Note that the mass conservation approach can be ran on any network that has conservation laws, even if that network does have a sink/source. One can test whether or not there are conservation laws by seeing if the output of *crnt4sbml_test.Cgraph.get_dim_equilibrium_manifold()* is greater than zero. This tutorial will use Fig1Ci.xml. The following code will import crnt4sbml and the SBML file. For a little more detail on this process consider *Low Deficiency Approach*.

```
import crnt4sbml_test
c = crnt4sbml_test.CRNT("/path/to/Fig1Ci.xml")
```

If we then want to conduct the mass conservation approach of [OMYS17], we must first initialize the mass_conservation_approach, which is done as follows:

```
opt = c.get_mass_conservation_approach()
```

This command creates all the necessary information to construct the optimization problem to be solved. Along with this, the initialization will also attempt to obtain a linear form of the Equilibrium Manifold. Note that this process may take several minutes for larger networks. For more detail on this process consider *Creating the Equilibrium Manifold*. The following is the output provided by the initialization:

```
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.8010799999999998

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 1.1221319999999997
```

One very important value that must be provided to the optimization problem are the bounds for the decision vector of the optimization problem. For this reason, it is useful to see what decision vector was constructed. To do this one can add the following command to the script:

```
print(opt.get_decision_vector())
```

This provides the following output:

```
[re1, re1r, re2, re3, re3r, re4, re6, re6r, re8, s2, s6, s16]
```

To obtain more available functions that this initialization provides, see *crnt4sbml_test.MassConservationApproach()*. Using the decision vector provided, one can then construct the bounds which are necessary for the optimization problem by creating a list of tuples where the first element corresponds to the lower bound value of the parameter and the second element is the upper bound value of the parameter. One such set of bounds for Fig1Ci.xml could be as follows:

```
bnds = [(1e-2, 1e2)]*12
```

In addition to the bounds for the decision vector, we must also supply the bounds for those species' concentrations that are not defined in the decision vector. To see the order and those species' concentration bounds that you need to provided bounds for, we can use the following command:

```
print(opt.get_concentration_bounds_species())
```

This provides the following output:

```
[s1, s3, s7, s15]
```

This tells us that we need to provide a list of four tuples that correspond to the lower and upper bounds for the species s1, s3, s7, and s15, in that order.

For our example, a set of these bounds could be as follows:

```
conc_bnds = [(1e-2, 1e2)]*4
```

The next most important parameter for optimization is the number of initial points in the feasible point method (please see *Numerical Optimization Routine* for a detailed description of the optimization routine). It is usually good practice to run the optimization with 100 initial points and observe the minimum objective function value achieved. If an objective function value smaller than machine epsilon is not achieved, it is best to rerun the optimization with more initial points. If 10000 or more points are used and an objective function value smaller than machine epsilon is not achieved, then it is possible that the network does not produce bistability (although this test does not exclude the possibility for bistability to exist, as stated in the theory). We state the number of feasible points below.

```
num_itr = 100
```

The last values that can be defined before the optimization portion are the sys_min_val which states what value of the objective function should be considered as zero (below we set this to machine epsilon), the seed for the random number generation in the optimization method (below we set this to 0 so we can reproduce the results, None should be used if we want the method to be random), the print_flag which tells the program if the objective function value and decision vector for the feasible point and multi-start method should be printed out (here we set it to False, which means no output will be provided), and numpy_dtype which tells the program the numpy data type that should be used in the optimization method (here we set it to a float with 64 bits). Note that higher precision data types will increase the runtime of the optimization, but may produce better results. See *crnt4sbml_test.MassConservationApproach.run_optimization()* for the default values of the routine.

```
import numpy

sys_min = numpy.finfo(float).eps
sd = 0
prnt_flg = False
num_dtype = numpy.float64
```

Using these values, we run the optimization problem using the following command, which returns a list of the parameters (which correspond to the decision vectors) and corresponding objective function values that produce an objective

function value smaller than machine epsilon.

```
params_for_global_min, obj_fun_val_for_params = opt.run_optimization(bounds=bnds,␣
↪concentration_bounds=conc_bnds,
                                                                     iterations=num_
↪itr, seed=sd, print_flag=prnt_flg,
                                                                     numpy_dtype=num_
↪dtype, sys_min_val=sys_min)
```

The following is the output obtained by the constructed model:

```
Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 0.7367519999999992

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 178.14652
```

At this point it may also be helpful to generate a report on the optimization routine that provides more information. To do this execute the following command:

```
opt.generate_report()
```

This will provide the following output:

```
The number of feasible points that satisfy the constraints: 100
Total feasible points that give F(x) = 0: 67
Total number of points that passed final_check: 67
```

The first line tells one how many initial points satisfy the constraints after the feasible point method is ran. Note that there should always be a nonzero amount provided here, if a nonzero amount is not given, new bounds should be considered. The second line describes how many feasible points provide an objective function value smaller than sys_min_val. The last line outputs the number of feasible points that produce an objective function value smaller than sys_min_val that also pass all of the constraints of the optimization problem. Note that it is not uncommon for the value provided in the last line to be smaller than the value provided in the second line. Given the optimization may take a long time to complete, it may be important to save the parameters produced by the optimization. This can be done as follows:

```
numpy.save('params.npy',params_for_global_min)
```

this saves the list of numpy arrays representing the parameters into the npy file params. The user can then load these values at a later time by using the following command:

```
params_for_global_min = numpy.load('params.npy')
```

Now that we have obtained some parameters that have achieved an objective function value smaller than sys_min_val, we can conduct numerical continuation to see if the parameters produce bistability for the ODE system provided by the network. The most important parameters that must be provided by the user are the principal continuation parameter (PCP) and the species you would like to compare it against. For more information on numerical continuation and these values see *Numerical Continuation Routine*. To select the PCP one needs to know which conservation law to choose. The following command will provide the conservation laws derived by the deficiency manager:

```
print(opt.get_conservation_laws())
```

This provides the following output:

```
C1 = 1.0*s16 + 1.0*s7
C2 = 1.0*s2 + 1.0*s3
C3 = 1.0*s1 + 2.0*s15 + 1.0*s16 + 1.0*s3 + 1.0*s6
```

here the left hand side of the equation corresponds to the constant that reflects the total amount of the leading species. It is one of these constants that should be provided to the numerical continuation routine. For this example we choose a PCP of C3 (total amount of species A) and the species s15 (species AA^*) for the y-axis of the bifurcation diagram.

```
spcs = "s15"
PCP_x = "C3"
```

Now we can call the numerical continuation routine. First we set the species and pass in the parameters we obtained from the optimization routine. The next input we provide is a dictionary representation of the AUTO 2000 parameters, to obtain a description of these parameters and more options refer to `AUTO parameters`. Please note that one should **not** set 'SBML' or 'ScanDirection' in these parameters as these are automatically assigned. It is absolutely necessary to set PrincipalContinuationParameter in this dictionary.

To show some functionality, here we set the maximum stepsize for numerical continuation, DSMAX. The default value for DSMAX is 0.1, however, for certain runs of the numerical continuation this may produce jagged plots. Smaller values should be used if one wants to obtain a smoother plot, although it should be noted that this will increase the runtime of the numerical continuation. We also state the principal continuation parameter range by defining 'RL0' and 'RL1', the lower and upper bound for the parameter, respectively.

Once we have set the AUTO parameters, we tell the numerical continuation routine whether or not to print out the labels obtained by the numerical continuation routine. Please refer to *Numerical Continuation Routine* for a description of this print out. The last value we provide is the string representation of the directory where we would like to store the multistability plots, if any are found (here we choose to create the stability_graphs directory in the current directory).

Using this input we can now run the numerical continuation routine on the parameters that pass the constraints of the optimization problem and produce an objective function value smaller than sys_min_val. This is done below.

```
multistable_param_ind = opt.run_continuity_analysis(species=spcs, parameters=params_
↪for_global_min,
                                                    auto_parameters={
↪'PrincipalContinuationParameter': PCP_x,
                                                                     'RL0': 0.1, 'RL1
↪': 30, 'DSMAX': 0.1},
                                                    print_lbls_flag=False, dir_path=".
↪/stability_graphs")
```

In addition to putting the multistability plots found into the path dir_path, this routine will also return the indices of params_for_global_min that correspond to these plots. Also note that if multistability plots are produced, the plot names will have the following form: PCP_species id_index of params_for_global_min _multistable_region.png. Where multistable_region is an integer that corresponds to the different regions of multistability. Note that this value is often just zero. The output provided by the numerical continuation run is as follows (note you may receive errors from failed point sets, you may ignore these):

```
Running continuity analysis ...
Elapsed time for continuity analysis = 158.717126
```

Again, we can generate a report that will contain the numerical optimization routine output and the now added information provided by the numerical continuation run.

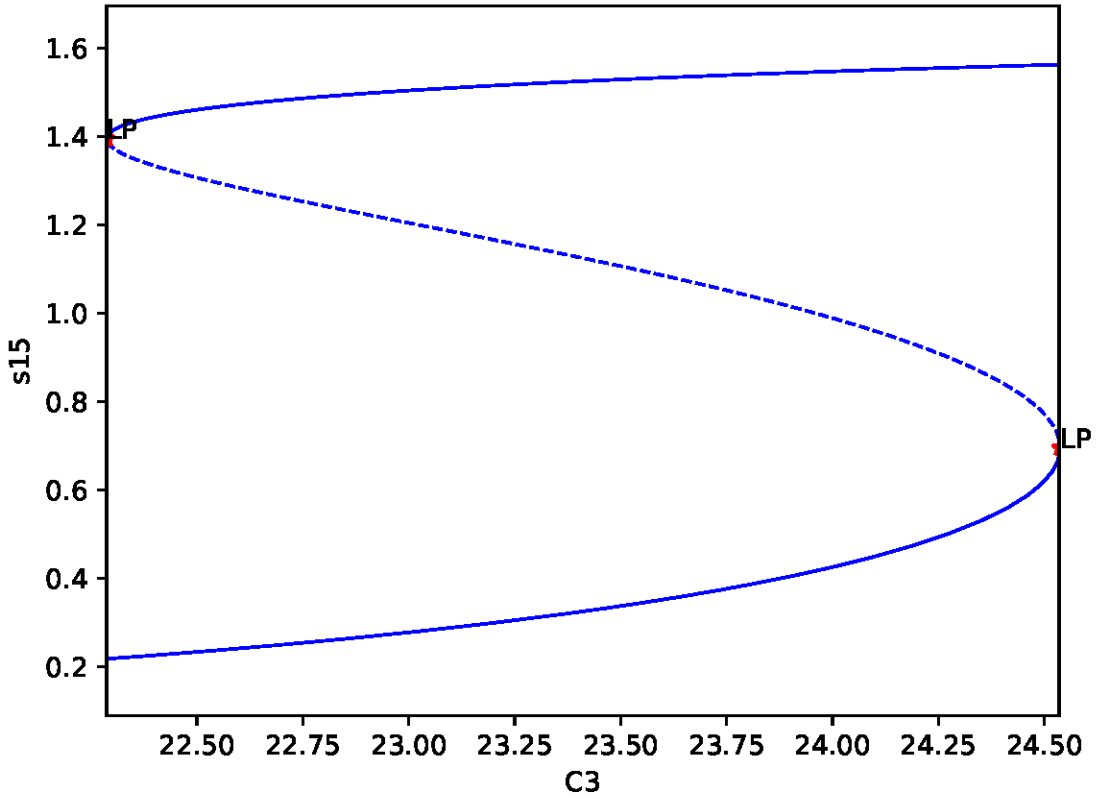
```
opt.generate_report()
```

This provides the following output that describes that of the 67 parameter sets that passed the constraints of the optimization problem, 14 of them produce multistability for the given input. In addition to this, it also tells one the

indices in params_for_global_min that produce multistability. In practice, larger ranges for the principal continuation parameter may be needed, but this will increase the runtime of the numerical continuation routine.

```
The number of feasible points that satisfy the constraints: 100
Total feasible points that give F(x) = 0: 67
Total number of points that passed final_check: 67
Number of multistability plots found: 14
Elements in params_for_global_min that produce multistability:
[4, 5, 13, 16, 21, 23, 26, 27, 31, 35, 52, 53, 63, 64]
```

The following is a bistability plot produced by element 35 of params_for_global_min. Here the solid blue line indicates stability, the dashed blue line is instability, and the red stars are the special points produced by the numerical continuation.



In addition to providing this more hands on approach to the numerical continuation routine, we also provide a greedy version of the numerical continuation routine. With this approach the user just needs to provide the species, parameters, and PCP. This routine does not guarantee that all multistability plots will be found, but it does provide a good place to start finding multistability plots. Once the greedy routine is ran, it is usually best to return to the more hands on approach described above. Note that as stated by the name, this approach is computationally greedy and will take a longer time than the more hands on approach. Below is the code used to run the greedy numerical continuation:

```
multistable_param_ind = opt.run_greedy_continuity_analysis(species=spcs,␣
↪parameters=params_for_global_min,
                                                           auto_parameters={
↪'PrincipalContinuationParameter': PCP_x})
```

(continues on next page)

(continued from previous page)

```
opt.generate_report()
```

This provides the following output:

```
Running continuity analysis ...
Elapsed time for continuity analysis: 301.959491

Number of multistability plots found: 66
Elements in params_for_global_min that produce multistability:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
↪ 24, 25, 26, 27, 28, 29, 30,
 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 49, 50, 51, 52,␣
↪53, 54, 55, 56, 57, 58, 59,
 60, 61, 62, 63, 64, 65, 66]
```

Note that some of these plots will be jagged or have missing sections in the plot. To produce better plots the hands on approach should be used.

For more examples of running the mass conservation approach please see *Further Examples*.

CHAPTER 7

Semi-diffusive Approach Walkthrough

Using the SBML file constructed as in *CellDesigner Walkthrough*, we will proceed by completing a more in-depth explanation of running the semi-diffusive approach of [OMYS17]. This tutorial will use Fig1Cii.xml. The following code will import crnt4sbml and the SBML file. For a little more detail on this process consider *Low Deficiency Approach*.

```
import crnt4sbml_test
c = crnt4sbml_test.CRNT("/path/to/Fig1Ci.xml")
```

If we then want to conduct the semi-diffusive approach of [OMYS17], we must first initialize the optimization_based_injectivity_manager, which is done as follows:

```
opt = c.get_semi_diffusive_approach()
```

This command creates all the necessary information to construct the optimization problem to be solved. Unlike the mass conservation, there should be no output provided by this initialization. Note that if a boundary species is not provided or there are conservation laws present, then the semi-diffusive approach will not be able to be ran. If conservation laws are found, then the mass conservation approach should be ran.

As in the mass conservation approach, it is very important to view the decision vector constructed for the optimization routine. In the semi-diffusive approach, the decision vector produced is in terms of the fluxes of the reactions. To make the decision vector more clear, the following command will print out the decision vector and also the corresponding reaction labels.

```
opt.print_decision_vector()
```

This provides the following output. As in the mass conservation approach, if your are using an SBML file you created yourself, the output may differ.

```
Decision vector for optimization:
[v_2, v_3, v_4, v_5, v_6, v_7, v_9, v_11, v_13, v_15, v_17, v_18]

Reaction labels for decision vector:
['re1r', 're3', 're3r', 're6', 're6r', 're2', 're8', 're17r', 're18r',
 're19r', 're21', 're22']
```

As in the mass conservation approach one can just obtain the decision vector by completing the following command:

```
print(opt.get_decision_vector())
```

Using the decision vector as a reference, we can now provide the bounds for the optimization routine. A possible definition of these bounds for Fig1Cii is as follows:

```
bounds = [(1e-3, 1e2)]*12
```

Another important check that should be completed for the semi-diffusive approach is to verify that that the key species, non key species, and boundary species are correct. This can be done after initializing the semi-diffusive approach as follows:

```
print(opt.get_key_species())
print(opt.get_non_key_species())
print(opt.get_boundary_species())
```

This provides the following results for our example:

```
['s1', 's2', 's7']

['s3', 's6', 's8', 's11']

['s21']
```

Using this information, we can now run the optimization in a similar manner to the mass conservation approach. First we will initialize some variables for demonstration purposes. In practice, the user should only need to define the bounds and number of iterations to run the optimization routine. For more information on the defaults of the optimization routine, see *crnt4sbml_test.SemiDiffusiveApproach.run_optimization()*.

```
import numpy
num_itr = 100
sys_min = numpy.finfo(float).eps
sd = 0
prnt_flg = False
num_dtype = numpy.float64
```

We now run the optimization routine for the semi-diffusive approach:

```
params_for_global_min, obj_fun_val_for_params = opt.run_optimization(bounds=bounds,␣
↪iterations=num_itr, seed=sd,
                                                                     print_flag=prnt_
↪flg, numpy_dtype=num_dtype,
                                                                     sys_min_val=sys_
↪min)
```

The following is the output obtained by the constructed model:

```
Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 3.785042

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 383.349612
```

At this point it may also be helpful to generate a report on the optimization routine that provides more information. To do this execute the following command:

```
opt.generate_report()
```

This provides the following output:

```
The number of feasible points that satisfy the constraints: 100
Total feasible points that give F(x) = 0: 82
Total number of points that passed final_check: 82
```

Similar to the mass conservation approach, we can run numerical continuation for the semi-diffusive approach. Note that the principal continuation parameter (PCP) now corresponds to a reaction rather than a constant as in the mass conservation approach. However, the actual continuation will be performed with respect to the flux of the reaction. The y-axis of the continuation can then be set by defining the species, here we choose the species s7. For the semi-diffusive network we conduct the numerical continuation for the semi-diffusive approach as follows:

```
multistable_param_ind = opt.run_continuity_analysis(species='s7', parameters=params_
↪for_global_min,
                                                    auto_parameters={
↪'PrincipalContinuationParameter': 're17',
                                                                     'RL0': 0.1, 'RL1
↪': 100, 'A0': 0.0,
                                                                     'A1': 10000})
```

For more information on the AUTO parameters provided, please see `AUTO parameters`. This provides the following output:

```
Running continuity analysis ...
Elapsed time for continuity analysis: 309.387856
```

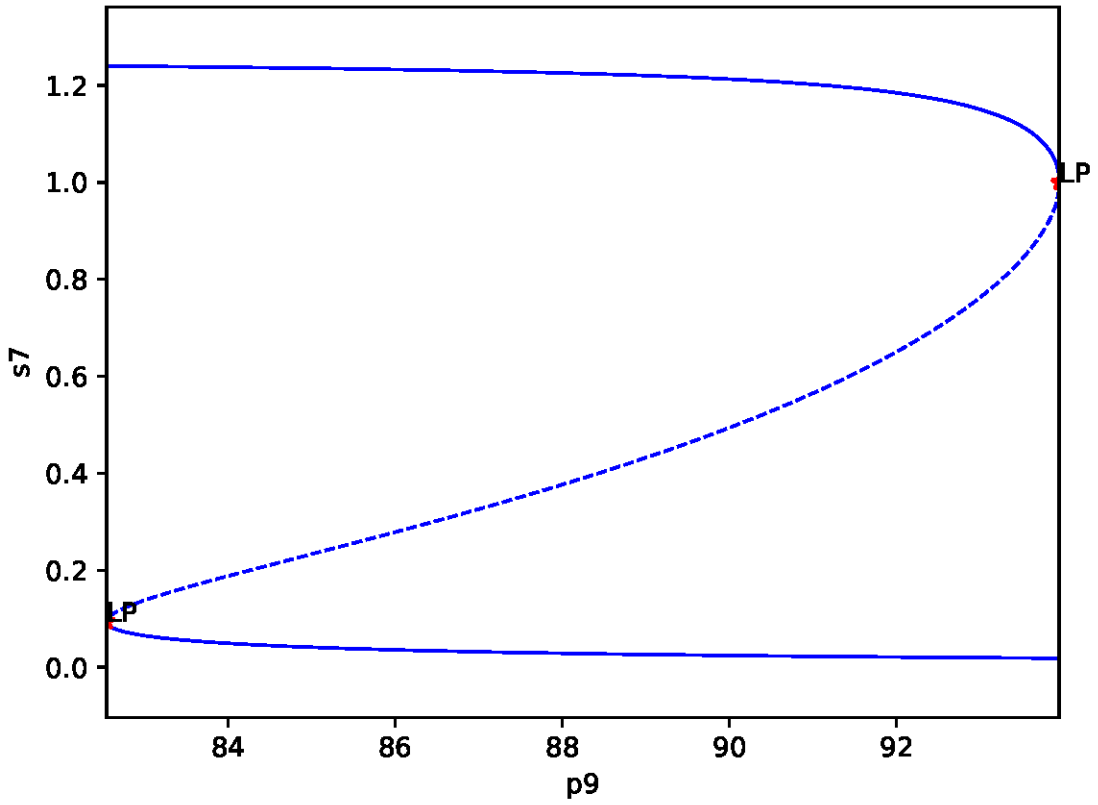
Note that you may receive Error reports to the screen, but these may be ignored for this particular exampels. Again we can generate a report that will contain the numerical optimization routine output and the now added information provided by the numerical continuation run:

```
opt.generate_report()
```

This provides the following output:

```
The number of feasible points that satisfy the constraints: 100
Total feasible points that give F(x) = 0: 82
Total number of points that passed final_check: 82
Number of multistability plots found: 22
Elements in params_for_global_min that produce multistability:
[0, 3, 4, 6, 14, 17, 19, 21, 25, 26, 28, 35, 39, 41, 43, 47, 48, 52, 53, 68, 79, 80]
```

Similar to the mass conservation approach, we obtain multistability plots in the directory provided by the dir_path option in run_continuity_analysis (here it is the default value), where the plots follow the following format PCP (in terms of p as in the theory) _species id_index of params_for_global_min_multistable_region.png. Where multistable_region is an integer that corresponds to the different regions of multistability. Note that this value is often just zero. The following is one multistability plot produced by index 4 of params_for_global_min.



In addition to providing this more hands on approach to the numerical continuation routine, we also provide a greedy version of the numerical continuation routine. With this approach the user just needs to provide the species, parameters, and PCP. This routine does not guarantee that all multistability plots will be found, but it does provide a good place to start finding multistability plots. Once the greedy routine is ran, it is usually best to return to the more hands on approach described above. Note that as stated by the name, this approach is computationally greedy and will take a longer time than the more hands on approach. Below is the code used to run the greedy numerical continuation:

```
multistable_param_ind = opt.run_greedy_continuity_analysis(species="s7",␣
↪parameters=params_for_global_min,
                                                           auto_parameters={
↪'PrincipalContinuationParameter': 're17'})

opt.generate_report()
```

This provides the following output:

```
Running continuity analysis ...
Elapsed time for continuity analysis: 522.959491

Number of multistability plots found: 75
Elements in params_for_global_min that produce multistability:
[0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25,␣
↪26, 27, 28, 29, 31, 32, 33, 34,
 35, 36, 37, 38, 39, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,␣
↪57, 58, 59, 60, 61, 62, 63,
 64, 65, 66, 67, 68, 69, 70, 72, 73, 74, 75, 76, 77, 78, 79, 80]
```

Note that some of these plots will be jagged or have missing sections in the plot. To produce better plots the hands on approach should be used.

For more examples of running the semi-diffusive approach please see *Further Examples*.

CHAPTER 8

Numerical Optimization Routine

In [OMYS17] it is suggested to use the ESS algorithm in the MEIGO toolbox to solve the constrained global optimization problem. Although evolutionary algorithms such as ESS can perform very well, they often need to be coupled with multi-start procedures to produce sensible results for complex reaction networks. In addition to this, to use the MEIGO toolbox within Python, a Python interface to R is required. This is not desirable, and for this reason we have constructed our own multi-start routine that compares favorably with the ESS routine for a general class of reaction networks.

The optimization routine utilizes two steps to achieve a minimization of the objective function:

1. Multi-level feasible point method
2. Hybrid global-local searches beginning at the feasibility points

8.1 Feasible Point Method

Both the mass conservation and semi-diffusive approach have constraints on the decision vector provided. These extra constraints coupled with the global optimization problem are difficult to solve and can often require many multi-starts to find a solution. This is due to the fact that multi-start routines often start at randomly generated values pulled from a uniform distribution, which do not satisfy the constraints. One way to begin the multi-start procedure in favorable positions is to generate starting points that already satisfy the constraints of the problem. We do this by conducting a feasible point method.

The feasible point method attempts to minimize the following objective function

$$f(\mathbf{x}) = \sum_{i=1}^{I}[v(g_i(\mathbf{x}))]^2 + \sum_{j=1}^{J}[v(\mathbf{x}_j)]^2.$$

where $v(\cdot)$ are violation functions for the constraint equations, $g_i(\cdot)$, and variable bounds, $\mathbf{x}_j$. The violation functions are defined as follows

Constraint Type	Violation Function
$g_i(\mathbf{x}) \leq b$	$max(0, g_i(\mathbf{x}) - b)$
$g_i(\mathbf{x}) \geq b$	$max(0, b - g_i(\mathbf{x}))$
$g_i(\mathbf{x}) = b$	$\lvert g_i(\mathbf{x}) - b \rvert$

Variable Bounds	Violation Function
$\mathbf{x}_j \leq b$	$max(0, \mathbf{x}_j - b)$
$\mathbf{x}_j \geq b$	$max(0, b - \mathbf{x}_j)$
$\mathbf{x}_j = b$	$\lvert\mathbf{x}_j - b\rvert$

this is called a penalty method and is outlined in Chapter 18 of [Chi14]. For the mass conservation approach the constraint equations are defined as $c_i \geq 0$, where c_i are the species' concentration expressions derived from the equilibrium manifold. The variable bounds for this approach are then defined by the bounds established for the decision vector. For the semi-diffusive approach the constraint equations are defined as $\bar{p}_i(\mu) \geq 0$ if c_i is a key species. Note that the constraint of $\bar{p}_i(\mu) = 0$ if c_i is not a key species is not considered in the optimization directly as they are satisfied by direct substitution. The variable bounds are again the bounds established for the decision vector. Notice that in both approaches we do not consider the rank constraints. In practice these are very difficult to satisfy via direct optimization. However, if the objective function is minimized, then the rank constraints have a very high likelihood of being satisfied.

Once the penalty function $f(\cdot)$ is constructed we can then continue by minimizing it. We do this by conducting a multi-level multi-start method. First we generate a user defined amount of decision vectors using a random uniform distribution and then put them in the user defined bounds. Next, we minimize $f(\cdot)$ using SciPy's SLSQP function with a tolerance of 1e-16. Although it is often sufficient to just run SLSQP, in some cases if a minimum of zero is not achieved by this run, it is beneficial to also perform a minimization using Nelder-Mead starting from the minimum point found by SLSQP. To reduce runtimes, we do not run the Nelder-Mead routine if SLSQP returns an objective function value that is sufficiently small.

8.2 Hybrid Global-Local Searches

Using those decision vectors produced by the feasible point method, we now address the global optimization problem. For the mass conservation approach we let the objective function be:

$F(\mathbf{x}) = det(G)^2 + f(\mathbf{x})$,

and for the semi-diffusive approach we let the objective function be:

$F(\mathbf{x}) = det(S_{to} diag(\mu) Y_r^T)^2 + f(\mathbf{x})$,

where $f(\mathbf{x})$ is the objective function formed by the feasible point method. Using the decision vectors produced by the feasible point method as starting points, we then run SciPy's global optimization algorithm Basin-hopping. In addition to running this global optimization, we employ Nelder-Mead as a local minimizer. If the local minimizer returns an objective function value smaller than a user defined value of sys_min_val, then the result solution array from the minimizer is saved and returned to the user.

8.3 Pseudocode for Optimization Method

Establish bounds for decision vector.

Randomly generate $niter$ parameter sets of decision vectors within the given bounds, say $samples$.

For $i = 1$ to $niter$

Let $samples_i$ be a starting point for the feasible point method where $f(\mathbf{x})$ is the objective function

If $samples_i$ **provides** $f(\mathbf{x}) \leq$ **machine epsilon** Run hybrid global-local search for $F(\mathbf{x})$ objective function with $\mathbf{x}$ as starting point, providing $\mathbf{x}_{best}$.

Store $\mathbf{x}_{best}$ and function values that are smaller than sys_min_val

Else Throw away $samples_i$

CHAPTER 9

Numerical Continuation Routine

To conduct the numerical continuation of the points produced by the mass conservation and semi-diffusive approaches, we use the very well developed software AUTO. In particular, we use the updated version AUTO 2000 made accessible through Libroadrunner and its extension rrplugins [SBG+15]. In the examples we have provided throughout this documentation we choose a set configuration of the parameters to run on all of the points found by the optimization routine. Although this is sufficient for detecting if bistability occurs in a particular network, if one wants to identify possible true physiological values, then it is best to consider each point individually while varying AUTO parameters. This is because if the points exist with varying ranges in the point sets then a set AUTO configuration may miss the detection of bistability for certain parameter settings.

Given most users may be unfamiliar with numerical continuation, in this section we provide some tips to consider when conducting the numerical continuation routine. To begin, it is first suggested to consider the available parameters in `AUTO parameters`. Note that as said in earlier sections, one should **not** set 'SBML' or 'ScanDirection' in these parameters as these are automatically assigned. Further descriptions of these parameters can be found in the older AUTO documentation. Of the available parameters, the most important are NMX, RL0, RL1, A1, DSMIN, and DSMAX, although more advanced users may find other parameters useful. The following is a short description of these parameters:

1. NMX is the maximum number of steps the numerical continuation is able to take. If one is using smaller values for DSMIN and or DSMAX it is suggested that NMX be increased. Note that an increase in NMX may result in longer run times.

2. DSMIN is the minimum continuation step size. A smaller DSMIN value may be beneficial if the values for the species' concentrations or principal continuation parameter is smaller than the default value provided. Larger values may be helpful in some contexts, but for most examples the parameter should be left at its default value.

3. DSMAX is the maximum continuation step size. A large DSMAX is necessary when considering the physiological values provided by *`crnt4sbml_test.CRNT.get_physiological_range()`* as this produces larger values for the species' concentrations and principal continuation parameters. A smaller DSMAX is also beneficial for both producing smoother plots and identifying special points. Although a smaller DSMAX will increase the runtime of the continuation.

4. RL0 is the lower bound for the principal continuation parameter (PCP). This value should be set at least a magnitude smaller than the starting value of the PCP, with 0.0 being the absolute minimum value that should be provided.

5. RL1 is the upper bound for the principal continuation parameter (PCP). This value should be set at least a magnitude

larger than the starting value of the PCP. An arbitrarily large value should not be used as this range can drastically affect the discovery of limit points and require fine tuning of DSMAX and DSMIN.

6. A1 is the upper bound on the principal solution measure. The principal solution measure used for differential equations is the L_2-norm defined as follows where $NDIM$ is the number of species and $U_k(x)$ is the solution to the ODE system for species k

$\sqrt{\int_0^1 \sum_{k=1}^{NDIM} U_k(x)^2 dx}$.

Although this parameter is somewhat difficult to monitor in the current setup of the continuity analysis, it is usually best to set it as a magnitude or two larger than the largest upper bound established on the species' concentrations.

To configure these parameters, it may be useful to see what special points are produced by the numerical continuation run. This can be done in both approaches by adding 'print_lbls_flag=True' to the run_continuity_analysis functions. For a description of the possible points that may be produced consider the section 'Special Points' in the XPP AUTO documentation. For the purposes of detecting bistability, the most important special points are limit points (LP). These points often mark a change in the stability of the ODE system and are more likely to produce overlapping stable and unstable branches that lead to bistability. It is the search for these special points that should guide the configuration of the AUTO parameters.

In addition to limit points, finding a set of two endpoints can be useful in determining if the ranges for the principal continuation parameter are appropriate. If no endpoints are found, then it is likely that the bounds chosen for the principal continuation parameter need to be changed. Note that when 'print_lbls_flag=True' is added to the run_continuity_analysis functions, the numerical continuation is first ran in the Positive direction and if no multistability is found, then the routine is ran in the Negative direction. This may result in two printouts per point provided. This switching of directions can often produce better results for numerical continuation runs.

CHAPTER 10

Creating the Equilibrium Manifold

For the mass conservation approach of [OMYS17] there are multiple ways that one can form the equilibrium manifold, $H(\alpha, c, k)$. In the approach we have constructed, we have chosen the equilibrium manifold that will result in two characteristics. The first of which is that the decision vector ultimately chosen will consist of only kinetic constants and species' concentrations. The reason for this is that we would like to remove the need for the user to provide bounds on the so called deficiency parameters, α. These bounds in practice can be somewhat difficult to find as they are not tied to any physical aspect of the network. The second characteristic we impose is that the manifold will be as close as possible to being linear with respect to the deficiency parameters and those species not in the decision vector. If the manifold is close to being linear in these variables, then solving for them is much simpler resulting in a shorter solve time for SymPy's solve function and the avoidance of unsolvable instances of the problem.

We now describe the process taken to find the decision vector and resulting equilibrium manifold. As stated in [OMYS17], the choice of the decision vector is as follows:

$x = (k_1, ..., k_R, \alpha_1, ..., \alpha_\lambda)$ for proper and over-dimensioned networks

and

$x = (k_1, ..., k_R, \alpha_1, ..., \alpha_\delta, c_1, ..., c_{\lambda-\delta})$ for under-dimensioned networks.

Although these decision vectors can be used, it is apparent that if they are chosen then the user will need to provide bounds for the deficiency parameters, α. However, as can be inferred by the statement on the bottom of page 7 in the S1 Appendix of [OMYS17], as long as the parameters α and k are fixed and we can form equation (2.7), then the results of Proposition 1 of page 8 follow. This allows one to choose the decision vector to be as follows for proper and over/under - dimensioned networks:

$x = (k_1, ..., k_R, \theta_1, ..., \theta_\lambda)$,

where the θ values are nonidentical choices of the species' concentrations, c.

Now that we have reformed the decision vector to be in terms of just kinetic constants and species' concentrations, the next step is to choose the θ values such that the equilibrium manifold is as close to being linear as possible. To do this, we first generate $\frac{N!}{s!(N-s)!}$ choices of θ values using $\binom{N}{s}$, where N is the number of species and s is the rank of the stoichiometric matrix. Using each of these sets of θ values, we then test how many rows of (9) in [OMYS17] are linear in those species' concentrations that are not in θ by testing if the second order derivatives of the expression in the row is zero. This is essentially testing if the expression is jointly linear with respect to a given set of species' concentrations not in θ.

In practice going through all $\frac{N!}{s!(N-s)!}$ choices can be expensive for large networks, to reduce this runtime we exit this routine if all rows of (9) are linear in those species' concentrations not in θ and choose this set of θ variables for our decision vector. After choosing the set of θ variables, we then choose $H(\alpha, c, k)$ by selecting $M - \ell$ independent rows of (9). This process of selecting $H(\alpha, c, k)$ is reflected in the run of crnt4sbml by the following output produced by *crnt4sbml_test.CRNT.get_mass_conservation_approach()*

```
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: xx
```

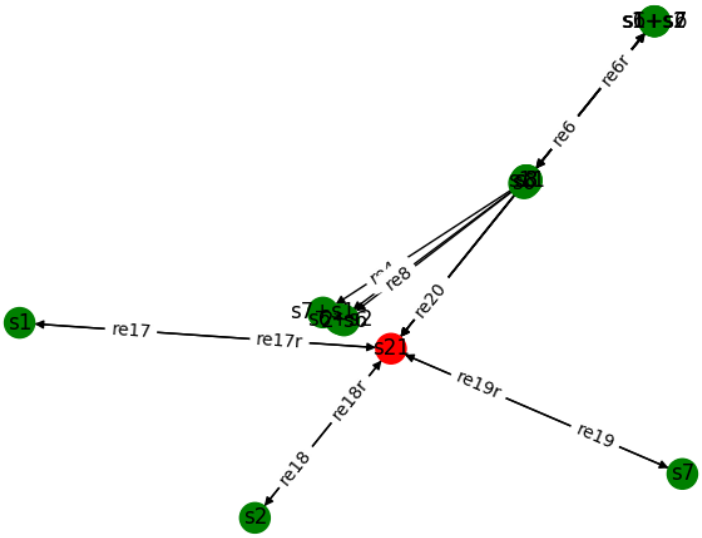
Once we have selected the equilibrium manifold, we then use the manifold to solve for all the deficiency parameters and species' concentrations not in θ using SymPy's solve function. This allows us to create expressions for each species' concentration. This process may take several minutes, so we have provided the following output to the screen to aid the user:

```
Solving for species' concentrations ...
Elapsed time for finding species' concentrations: xx
```

CHAPTER 11

Generating Presentable C-graphs

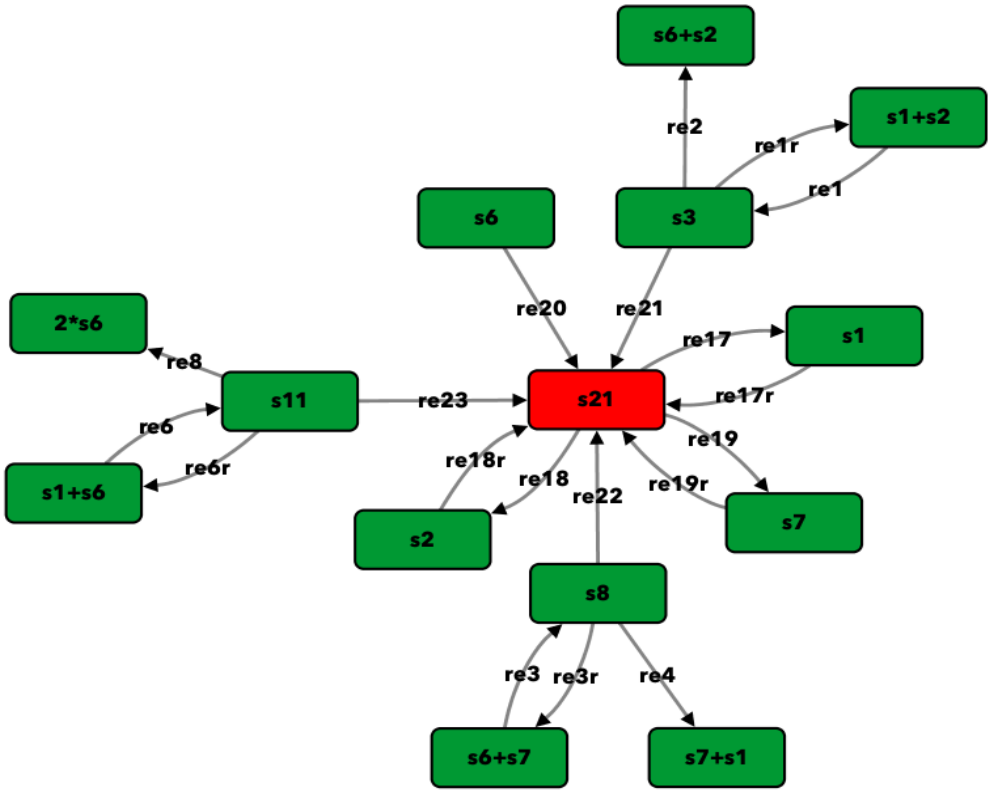
In practice complex networks can be difficult to display in terms of the CellDesigner format. For this reason, it is usually simpler to present networks in terms of C-graphs. Although CRNT4SBML provides the functions `crnt4sbml_test.CRNT.plot_c_graph()` and `crnt4sbml_test.CRNT.plot_save_c_graph()` to plot and save C-graphs using Matplotlib, respectively, for large networks these displays can be cluttered. For example, consider the following semi-diffusive network:



As mentioned in the NetorkX documentation , the graph visualization tools provided are not up to par with other graph visualization tools. For this reason, we suggest using the cross-platform and easily installable tool Cytoscape to create presentable C-graphs. Cytoscape allows one to import a network defined in the GraphML format which it can then use to create a C-graph. To create a GraphML format of the provided network, CRNT4SBML contains the function `crnt4sbml_test.CRNT.get_network_graphml()`. Note that this function only extracts the nodes, edges, and edge labels. Below we use use Fig1Cii.xml to demonstrate turning a network into a GraphML file.

```
import crnt4sbml_test

c = crnt4sbml_test.CRNT("path/to/Fig1Cii.xml")

c.get_network_graphml()
```

This will provide a GraphML file for the Fig1Cii network in the current directory under the name network.graphml. We may then use this file within Cytoscape by opening up the application navigating to the menu bar selecting file -> Import -> Network from File... then selecting network.graphml from the appropriate directory. Some simple modifications lead to the following C-graph.



CHAPTER 12

Further Examples

In this section we present multiple examples for the mass conservation and semi-diffusive approaches. In addition to this, we provide some examples satisfying the deficiency theorems. Before each example we depict the CellDesigner layout and C-graph generated using the instructions in *Generating Presentable C-graphs*. Those nodes that represent zero complexes are colored red while regular nodes are green.

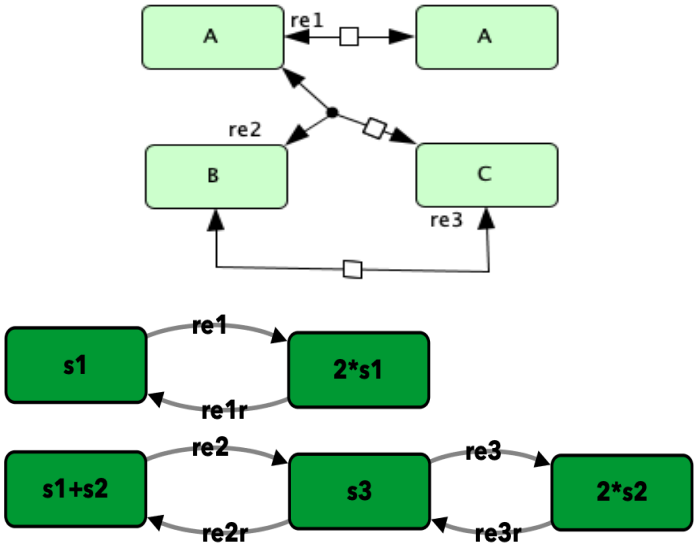
Contents

- *Further Examples*
  - *Low Deficiency Approach*
    - *Network 3.13 of [Fei79]*
    - *Figure 1Aii of [OMYS17]*
    - *Example 3.D.3 of [Fei79]*
  - *Mass Conservation Approach*
    - *Closed graph of Figure 5A of [OMYS17]*
    - *Gene regulatory network with two mutually repressing genes from [OMYS14]*
    - *Enzymatic reaction with inhibition by substrate from [OMBA09]*
    - *Enzymatic reaction with simple substrate cycle from [HC87]*
    - *Signal transduction from [CSRGR05]*
    - *G1/S transition in the cell cycle of Saccharomyces cerevisiae from [CFRS07]*
    - *Figure 6A of [OMYS17]*
    - *Double insulin binding*
    - *p85-p110-PTEN*
    - *Closed version of Figure 4B from [OMYS17]*

        * *Closed version of Figure 4C from [OMYS17]*
    - *Semi-diffusive Approach*
        * *Figure 5B of [OMYS17]*
        * *Open version of Figure 5A from [OMYS17]*
        * *Figure 4B from [OMYS17]*
        * *Figure 4C from [OMYS17]*

12.1 Low Deficiency Approach

12.1.1 Network 3.13 of [fein_lecture]



To run this example download the SBML `file` and script `run_feinberg_ex3_13`. After running this script we obtain the following output:

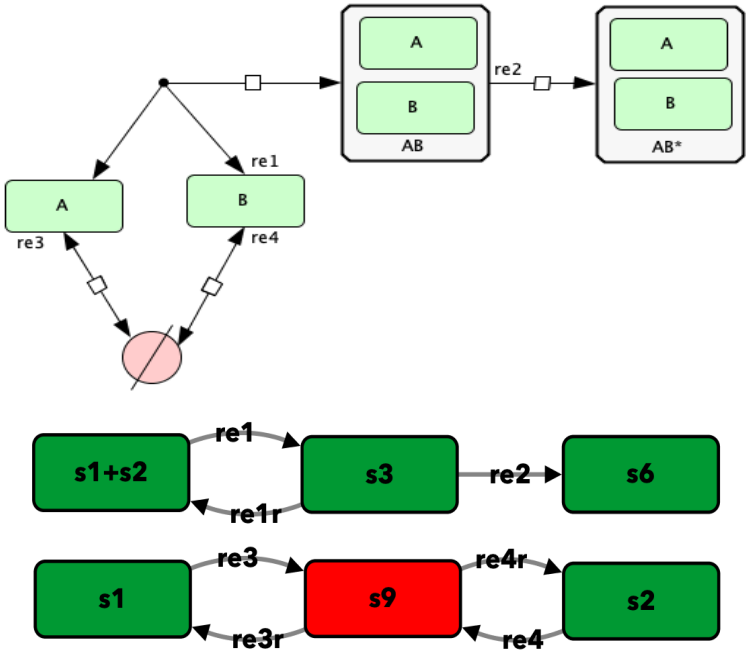
```
Number of species: 3
Number of complexes: 5
Number of reactions: 6
Network deficiency: 0

Reaction graph of the form
reaction -- reaction label:
s1 -> 2*s1  --  re1
2*s1 -> s1  --  re1r
s1+s2 -> s3  --  re2
s3 -> s1+s2  --  re2r
s3 -> 2*s2  --  re3
2*s2 -> s3  --  re3r

By the Deficiency Zero Theorem, there exists within each positive
stoichiometric compatibility class precisely one equilibrium.
Thus, multiple equilibria cannot exist for the network.
```

(continues on next page)

(continued from previous page)

```
The network does not satisfy Deficiency One Theorem.
Network satisfies one of the low deficiency theorems.
One should not run the optimization-based methods.
```

12.1.2 Figure 1Aii of [irene]



To run this example download the SBML `file` and script `run_fig1Aii`. After running this script we obtain the following output:

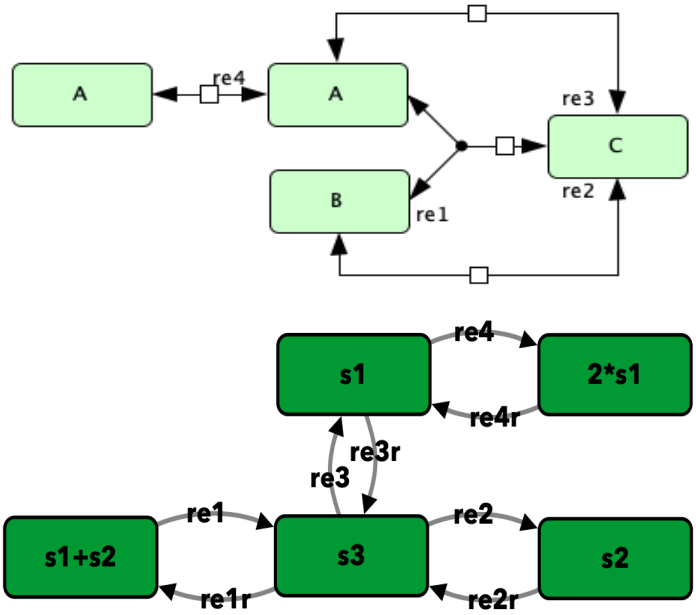
```
Number of species: 4
Number of complexes: 6
Number of reactions: 7
Network deficiency: 0

Reaction graph of the form
reaction -- reaction label:
s1+s2 -> s3  --  re1
s3 -> s1+s2  --  re1r
s3 -> s6  --  re2
s1 -> s9  --  re3
s9 -> s1  --  re3r
s2 -> s9  --  re4
s9 -> s2  --  re4r

By the Deficiency Zero Theorem, the differential equations
cannot admit a positive equilibrium or a cyclic composition
trajectory containing a positive composition. Thus, multiple
equilibria cannot exist for the network.

The network does not satisfy Deficiency One Theorem.
```

(continues on next page)

(continued from previous page)

```
Network satisfies one of the low deficiency theorems.
One should not run the optimization-based methods.
```

12.1.3 Example 3.D.3 of [fein_lecture]

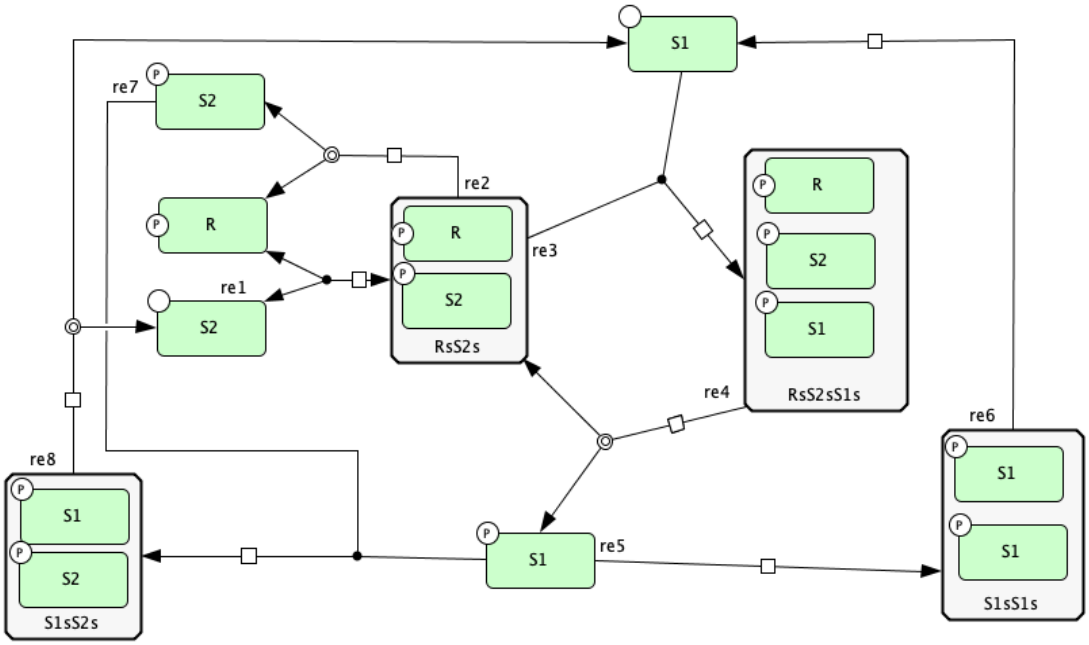


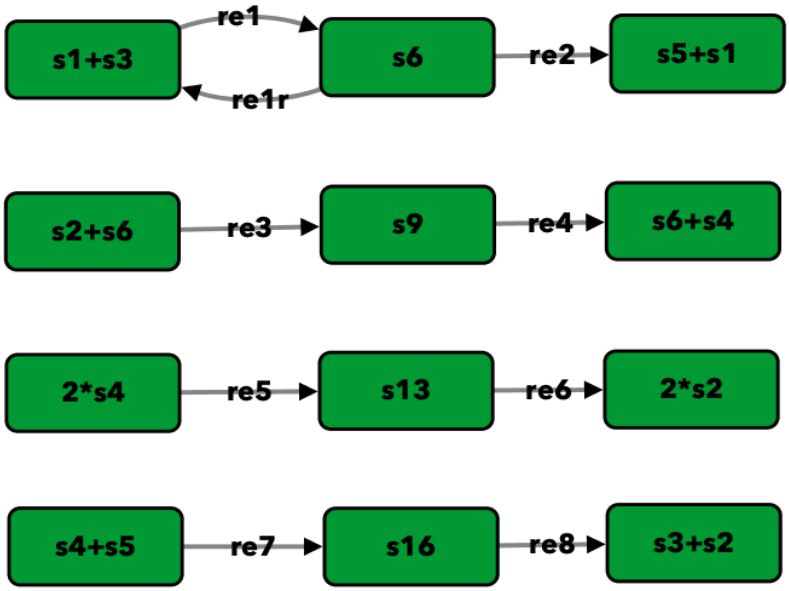
To run this example download the SBML `file` and script `run_feinberg_ex_3_D_3`. After running this script we obtain the following output:

```
Number of species: 3
Number of complexes: 5
Number of reactions: 8
Network deficiency: 1

Reaction graph of the form
reaction -- reaction label:
s1+s2 -> s3  --  re1
s3 -> s1+s2  --  re1r
s3 -> s2  --  re2
s2 -> s3  --  re2r
s3 -> s1  --  re3
s1 -> s3  --  re3r
s1 -> 2*s1  --  re4
2*s1 -> s1  --  re4r

The network does not satisfy Deficiency Zero Theorem.
By the Deficiency One Theorem, the differential equations
admit precisely one equilibrium in each positive stoichiometric
compatibility class. Thus, multiple equilibria cannot exist
for the network.

Network satisfies one of the low deficiency theorems.
One should not run the optimization-based methods.
```

12.2 Mass Conservation Approach

12.2.1 Closed graph of Figure 5A of [irene]





To run this example download the SBML `file` and script `run_closed_fig5A`. After running this script we obtain the following output:

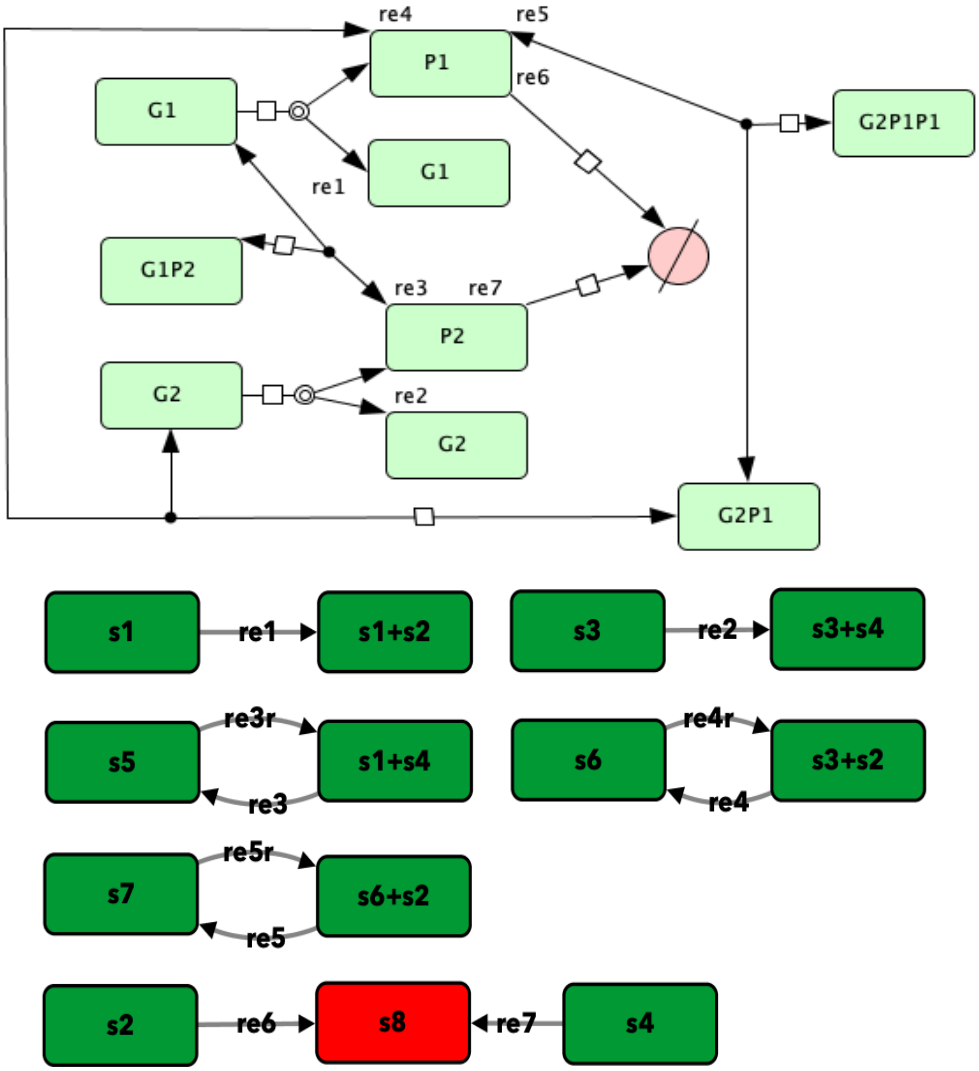
```
Number of species: 9
Number of complexes: 12
Number of reactions: 9
Network deficiency: 2

```

(continues on next page)

(continued from previous page)

```
Reaction graph of the form
reaction -- reaction label:
s1+s3 -> s6  --  re1
s6 -> s1+s3  --  re1r
s6 -> s5+s1  --  re2
s2+s6 -> s9  --  re3
s9 -> s6+s4  --  re4
2*s4 -> s13  --  re5
s13 -> 2*s2  --  re6
s4+s5 -> s16  --  re7
s16 -> s3+s2  --  re8

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 1.5645950000000006

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.20860400000000023

Decision Vector:
[re1, re1r, re2, re3, re4, re5, re6, re7, re8, s3, s4, s6]

Species for concentration bounds:
[s1, s2, s5, s9, s13, s16]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 31.838014

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 118.78041900000002

Running continuity analysis ...
Elapsed time for continuity analysis: 41.44564199999999

The number of feasible points that satisfy the constraints: 92
Total feasible points that give F(x) = 0: 30
Total number of points that passed final_check: 30
Number of multistability plots found: 9
Elements in params_for_global_min that produce multistability:
[3, 4, 5, 6, 7, 8, 11, 15, 24]
```

12.2.2 Gene regulatory network with two mutually repressing genes from [irene2014]



To run this example download the SBML `file` and script `run_irene2014`. After running this script we obtain the following output:

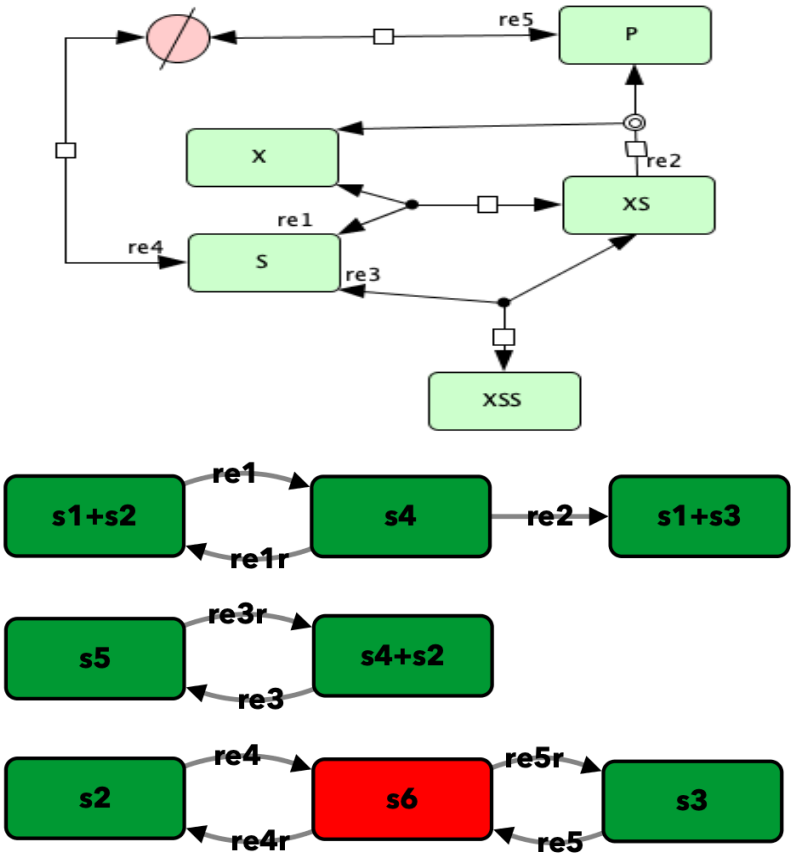
```
Number of species: 7
Number of complexes: 13
Number of reactions: 10
Network deficiency: 2

Reaction graph of the form
reaction -- reaction label:
s1 -> s1+s2  --  re1
s3 -> s3+s4  --  re2
s1+s4 -> s5  --  re3
s5 -> s1+s4  --  re3r
s3+s2 -> s6  --  re4
s6 -> s3+s2  --  re4r
```

(continues on next page)

(continued from previous page)

```
s6+s2 -> s7  --  re5
s7 -> s6+s2  --  re5r
s2 -> s8  --  re6
s4 -> s8  --  re7

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.6639610000000005

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.3587790000000002

Decision Vector:
[re1, re2, re3, re3r, re4, re4r, re5, re5r, re6, re7, s2, s4]

Species for concentration bounds:
[s1, s3, s5, s6, s7]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 27.412999999999997

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 119.703018

Running continuity analysis ...
Elapsed time for continuity analysis: 112.23523899999995

The number of feasible points that satisfy the constraints: 96
Total feasible points that give F(x) = 0: 93
Total number of points that passed final_check: 93
Number of multistability plots found: 21
Elements in params_for_global_min that produce multistability:
[13, 14, 25, 27, 29, 30, 32, 39, 46, 48, 49, 53, 58, 64, 66, 73, 75, 78, 82, 88, 90]
```

12.2.3 Enzymatic reaction with inhibition by substrate from [irene2009]



To run this example download the SBML `file` and script `run_irene2009`. After running this script we obtain the following output:

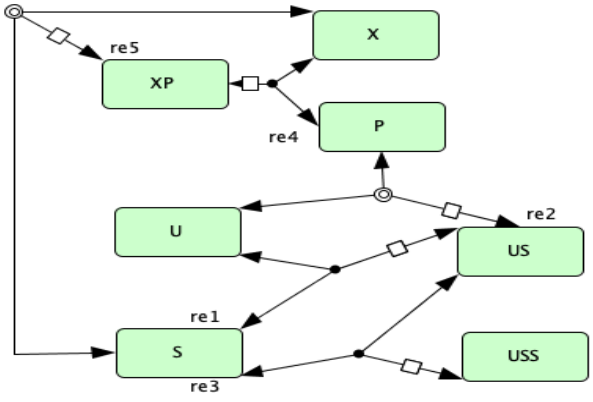
```
Number of species: 5
Number of complexes: 8
Number of reactions: 9
Network deficiency: 1

Reaction graph of the form
reaction -- reaction label:
s1+s2 -> s4  --  re1
s4 -> s1+s2  --  re1r
s4 -> s1+s3  --  re2
s4+s2 -> s5  --  re3
s5 -> s4+s2  --  re3r
s2 -> s6  --  re4
s6 -> s2  --  re4r
s3 -> s6  --  re5
s6 -> s3  --  re5r

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.12490000000000023
```

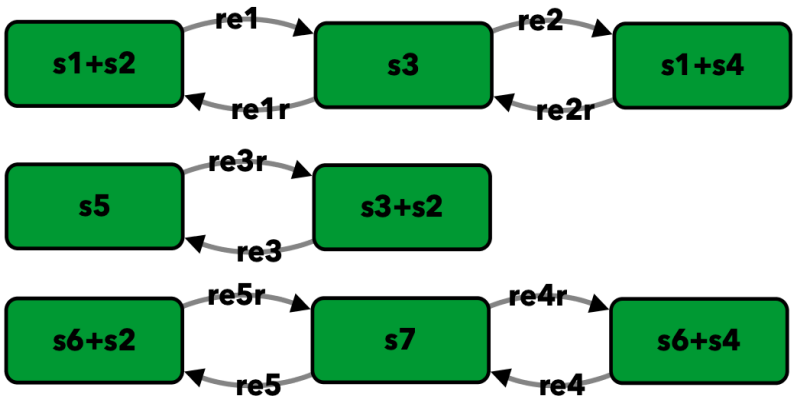
(continues on next page)

(continued from previous page)

```
Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.1801330000000001

Decision Vector:
[re1, re1r, re2, re3, re3r, re4, re4r, re5, re5r, s2]

Species for concentration bounds:
[s1, s3, s4, s5]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 17.950815

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 73.953789

Running continuity analysis ...
Elapsed time for continuity analysis: 77.51778000000002

The number of feasible points that satisfy the constraints: 100
Total feasible points that give F(x) = 0: 83
Total number of points that passed final_check: 83
Number of multistability plots found: 50
Elements in params_for_global_min that produce multistability:
[1, 6, 7, 12, 14, 16, 17, 19, 20, 21, 22, 23, 25, 26, 29, 31, 32, 35, 36, 39, 40, 41,␣
↪42, 45, 46, 47, 49, 51, 52, 54, 55, 56, 57, 58, 59, 60, 61, 65, 67, 68, 70, 71, 72,␣
↪73, 76, 77, 78, 79, 80, 81]
```

12.2.4 Enzymatic reaction with simple substrate cycle from [HERVAGAULT1987439]





To run this example download the SBML `file` and script `run_hervagault_canu`. After running this script we obtain the following output:

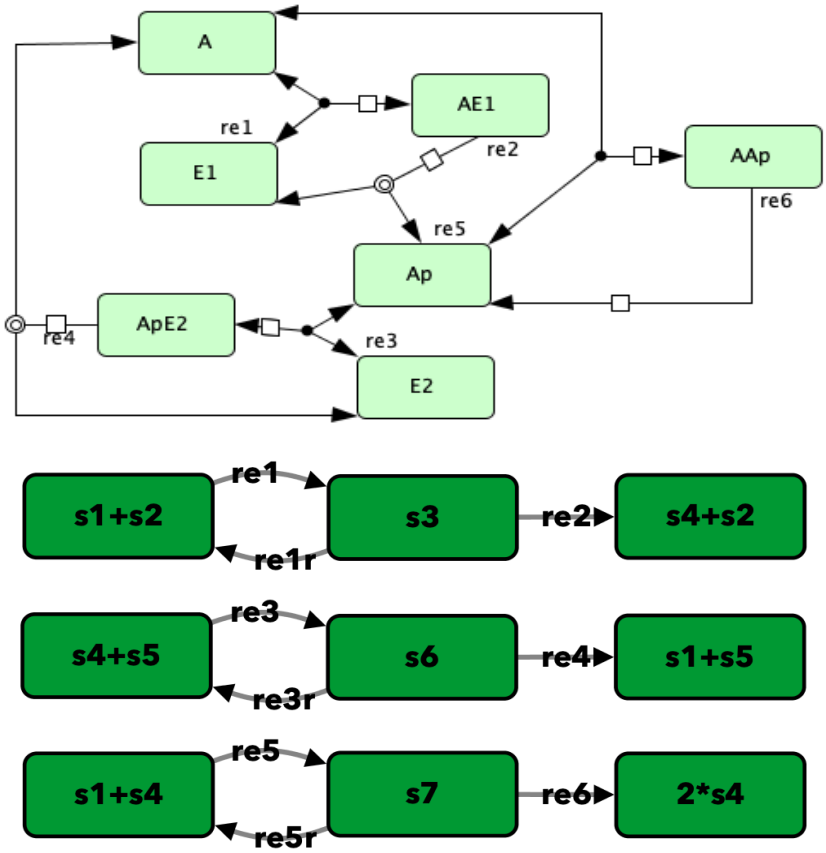
```
Number of species: 7
Number of complexes: 8
Number of reactions: 10
Network deficiency: 1

Reaction graph of the form
reaction -- reaction label:
s1+s2 -> s3  --  re1
s3 -> s1+s2  --  re1r
s3 -> s1+s4  --  re2
s1+s4 -> s3  --  re2r
s3+s2 -> s5  --  re3
s5 -> s3+s2  --  re3r
s6+s4 -> s7  --  re4
s7 -> s6+s4  --  re4r
s7 -> s6+s2  --  re5
s6+s2 -> s7  --  re5r

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.24900900000000004

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.8099070000000008

Decision Vector:
[re1, re1r, re2, re2r, re3, re3r, re4, re4r, re5, re5r, s2, s4, s7]

Species for concentration bounds:
[s1, s3, s5, s6]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 22.274808

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 172.20124700000002
```

(continues on next page)

(continued from previous page)

```
Running continuity analysis ...
Elapsed time for continuity analysis: 19.03241

The number of feasible points that satisfy the constraints: 93
Total feasible points that give F(x) = 0: 24
Total number of points that passed final_check: 24
Number of multistability plots found: 20
Elements in params_for_global_min that produce multistability:
[1, 2, 3, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
```

12.2.5 Signal transduction from [conradi2005]



To run this example download the SBML `file` and script `run_conradi2005`. After running this script we obtain the following output:

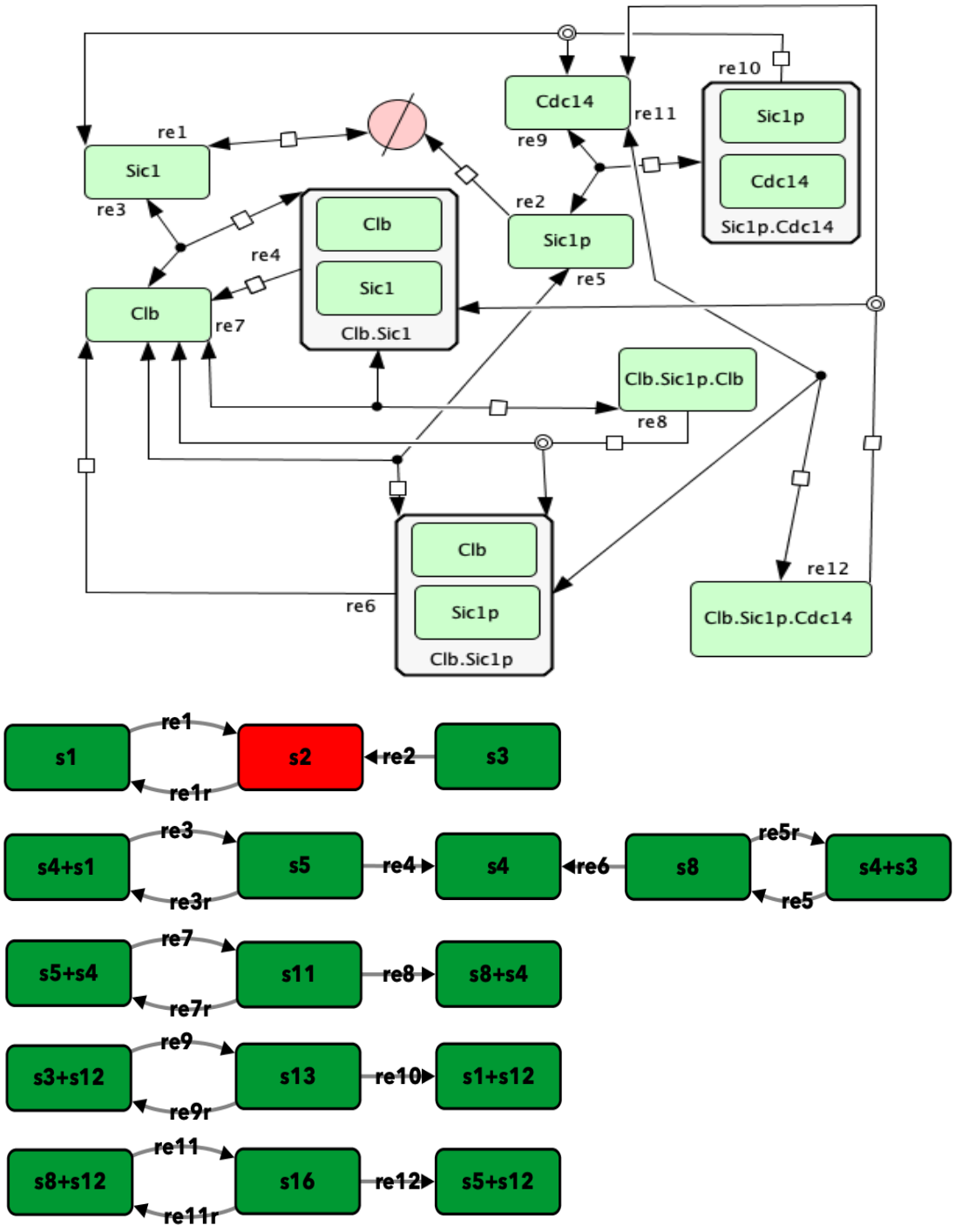
```
Number of species: 7
Number of complexes: 9
Number of reactions: 9
Network deficiency: 2

Reaction graph of the form
reaction -- reaction label:
s1+s2 -> s3  --  re1
```

(continues on next page)

(continued from previous page)

```
s3 -> s1+s2  --  re1r
s3 -> s4+s2  --  re2
s4+s5 -> s6  --  re3
s6 -> s4+s5  --  re3r
s6 -> s1+s5  --  re4
s1+s4 -> s7  --  re5
s7 -> s1+s4  --  re5r
s7 -> 2*s4  --  re6

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.4383759999999999

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.6672840000000004

Decision Vector:
[re1, re1r, re2, re3, re3r, re4, re5, re5r, re6, s2, s4, s7]

Species for concentration bounds:
[s1, s3, s5, s6]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 4.570956999999999

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 113.44187000000001

Running continuity analysis ...
Elapsed time for continuity analysis: 34.630101999999994

The number of feasible points that satisfy the constraints: 99
Total feasible points that give F(x) = 0: 38
Total number of points that passed final_check: 38
Number of multistability plots found: 30
Elements in params_for_global_min that produce multistability:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 14, 15, 17, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29,␣
↪30, 31, 33, 34, 35, 36, 37]
```

12.2.6 G1/S transition in the cell cycle of Saccharomyces cerevisiae from [Conradi2007]



To run this example download the SBML `file` and script `run_conradi2007`. After running this script we obtain the following output:

```
Number of species: 9
Number of complexes: 17
```

(continues on next page)

(continued from previous page)

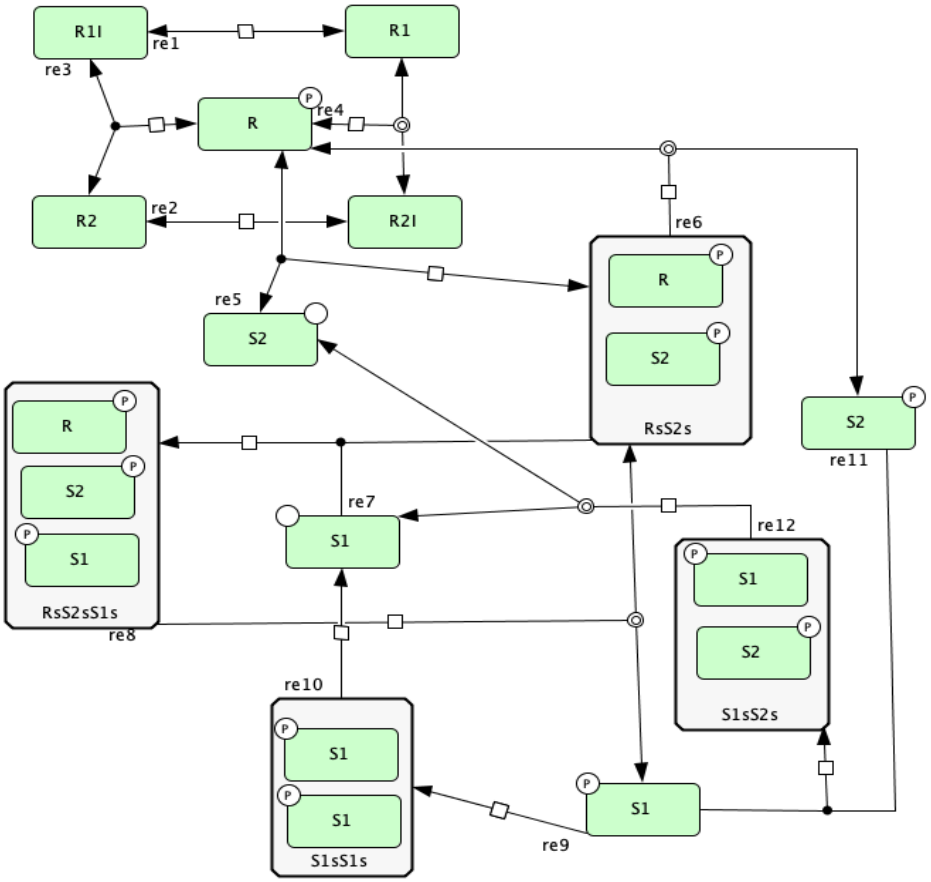
```
Number of reactions: 18
Network deficiency: 5

Reaction graph of the form
reaction -- reaction label:
s1 -> s2  --  re1
s2 -> s1  --  re1r
s3 -> s2  --  re2
s4+s1 -> s5  --  re3
s5 -> s4+s1  --  re3r
s5 -> s4  --  re4
s4+s3 -> s8  --  re5
s8 -> s4+s3  --  re5r
s8 -> s4  --  re6
s5+s4 -> s11  --  re7
s11 -> s5+s4  --  re7r
s11 -> s8+s4  --  re8
s3+s12 -> s13  --  re9
s13 -> s3+s12  --  re9r
s13 -> s1+s12  --  re10
s8+s12 -> s16  --  re11
s16 -> s8+s12  --  re11r
s16 -> s5+s12  --  re12

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 2.713166

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 123.89024500000001

Decision Vector:
[re1, re1r, re2, re3, re3r, re4, re5, re5r, re6, re7, re7r, re8, re9, re9r, re10,␣
↪re11, re11r, re12, s4, s12]

Species for concentration bounds:
[s1, s3, s5, s8, s11, s13, s16]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 89.85705200000001

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 1227.066971

Running continuity analysis ...
Elapsed time for continuity analysis: 18.148615999999947

The number of feasible points that satisfy the constraints: 100
Total feasible points that give F(x) = 0: 13
Total number of points that passed final_check: 13
Number of multistability plots found: 11
Elements in params_for_global_min that produce multistability:
```

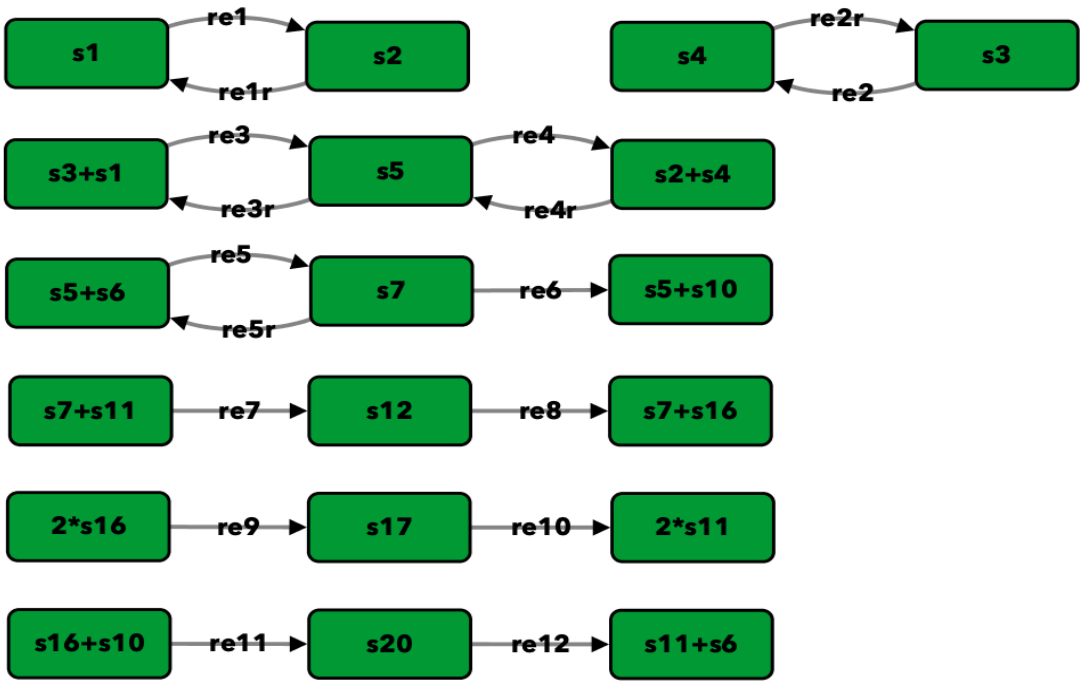
(continues on next page)

(continued from previous page)

```
[1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12]
```

12.2.7 Figure 6A of [irene]





To run this example download the SBML `file` and script `run_Fig6A`. After running this script we obtain the following output:

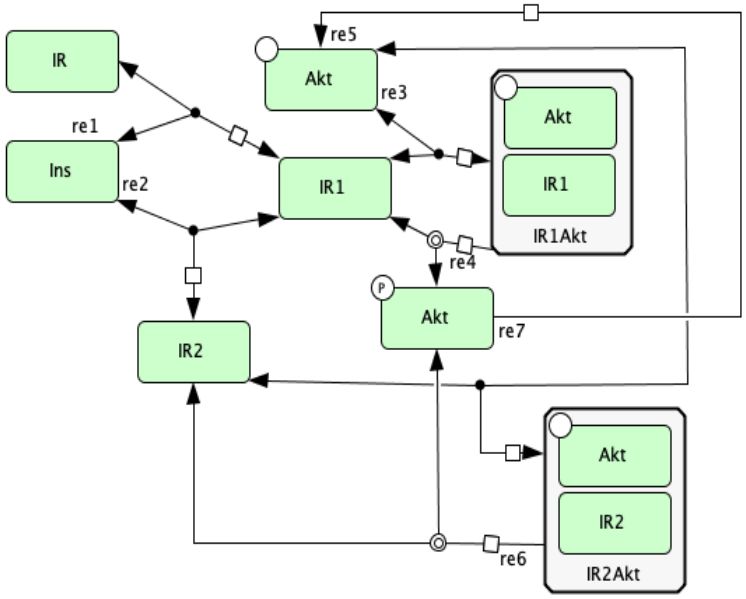
```
Number of species: 13
 Number of complexes: 19
 Number of reactions: 17
 Network deficiency: 3

 Reaction graph of the form
 reaction -- reaction label:
 s1 -> s2  --  re1
 s2 -> s1  --  re1r
 s3 -> s4  --  re2
 s4 -> s3  --  re2r
 s3+s1 -> s5  --  re3
 s5 -> s3+s1  --  re3r
 s5 -> s2+s4  --  re4
 s2+s4 -> s5  --  re4r
 s5+s6 -> s7  --  re5
 s7 -> s5+s6  --  re5r
 s7 -> s5+s10  --  re6
 s7+s11 -> s12  --  re7
 s12 -> s7+s16  --  re8
 2*s16 -> s17  --  re9
 s17 -> 2*s11  --  re10
 s16+s10 -> s20  --  re11
 s20 -> s11+s6  --  re12

 The network does not satisfy Deficiency Zero Theorem.
 The network does not satisfy Deficiency One Theorem.

 Creating Equilibrium Manifold ...
 Elapsed time for creating Equilibrium Manifold: 108.00370000000001
```

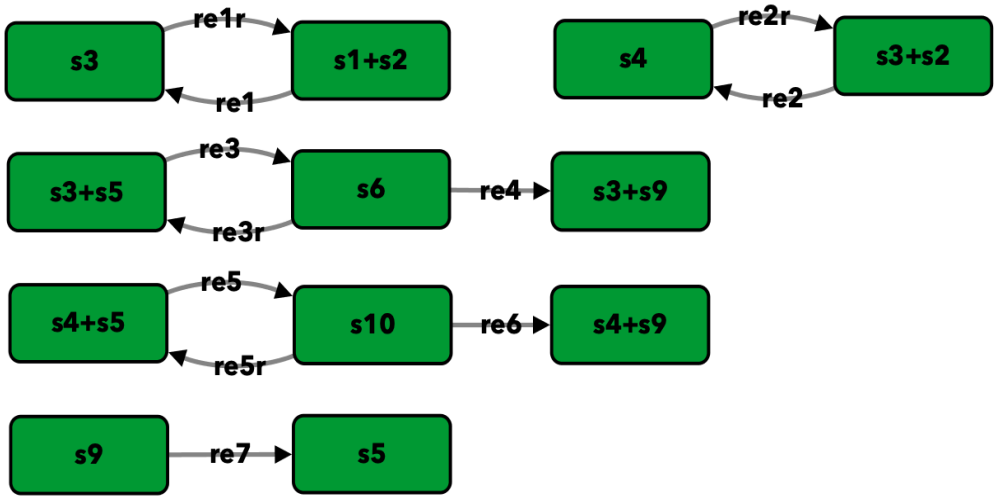
(continues on next page)

(continued from previous page)

```
Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 28.19600299999999

Decision Vector:
[re1, re1r, re2, re2r, re3, re3r, re4, re4r, re5, re5r, re6, re7, re8, re9, re10,␣
↪re11, re12, s4, s6, s11, s16]

Species for concentration bounds:
[s1, s2, s3, s5, s7, s10, s12, s17, s20]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 249.93427100000002

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 278.2530290000001

Running continuity analysis ...
Elapsed time for continuity analysis: 1.983425000000011

The number of feasible points that satisfy the constraints: 49
Total feasible points that give F(x) = 0: 1
Total number of points that passed final_check: 1
Number of multistability plots found: 1
Elements in params_for_global_min that produce multistability:
[0]
```

12.2.8 Double insulin binding





To run this example download the SBML `file` and script `run_double_insulin_binding`. After running this script we obtain the following output:

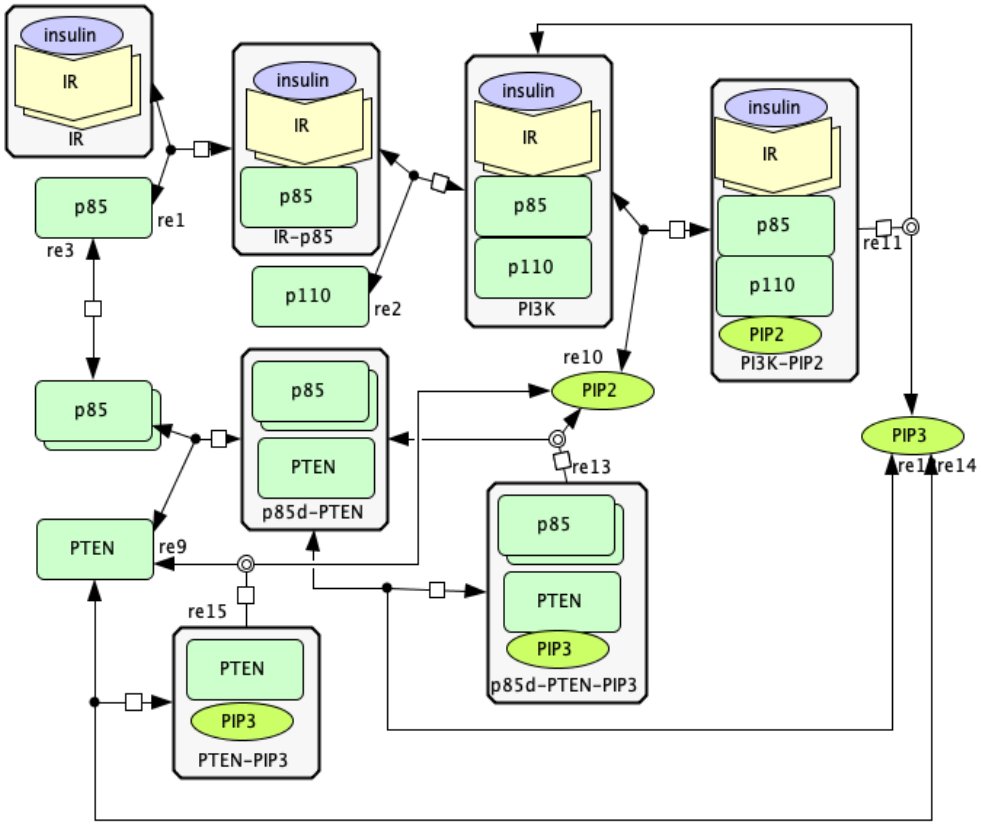
```
Number of species: 8
Number of complexes: 12
Number of reactions: 11
Network deficiency: 2

Reaction graph of the form
reaction -- reaction label:
s1+s2 -> s3  --  re1
s3 -> s1+s2  --  re1r
s3+s2 -> s4  --  re2
s4 -> s3+s2  --  re2r
s3+s5 -> s6  --  re3
s6 -> s3+s5  --  re3r
s6 -> s3+s9  --  re4
s4+s5 -> s10  --  re5
s10 -> s4+s5  --  re5r
s10 -> s4+s9  --  re6
s9 -> s5  --  re7

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.9201350000000001

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.5085129999999998

Decision Vector:
[re1, re1r, re2, re2r, re3, re3r, re4, re5, re5r, re6, re7, s2, s5, s10]

Species for concentration bounds:
[s1, s3, s4, s6, s9]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 30.984338
```

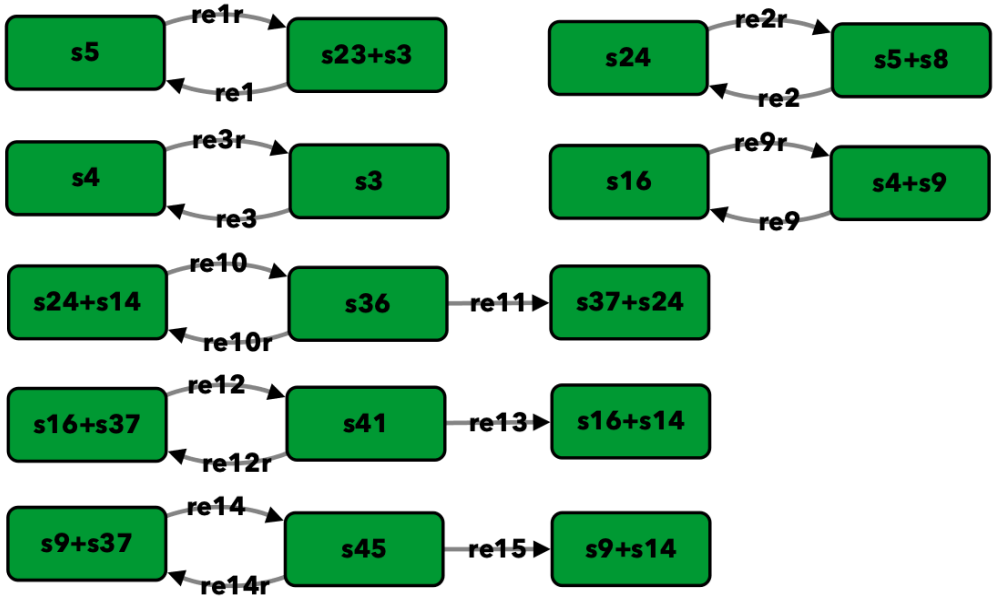
(continues on next page)

(continued from previous page)

```
Running the multistart optimization ...

Smallest value achieved by objective function: 2.3317319454459066e-31

Elapsed time for multistart method: 116.50619499999999

Running continuity analysis ...
Elapsed time for continuity analysis: 102.63304

The number of feasible points that satisfy the constraints: 96
Total feasible points that give F(x) = 0: 67
Total number of points that passed final_check: 67
Number of multistability plots found: 1
Elements in params_for_global_min that produce multistability:
[17]
```

12.2.9 p85-p110-PTEN





To run this example download the SBML `file` and script `run_p85-p110-PTEN`. After running this script we obtain the following output:

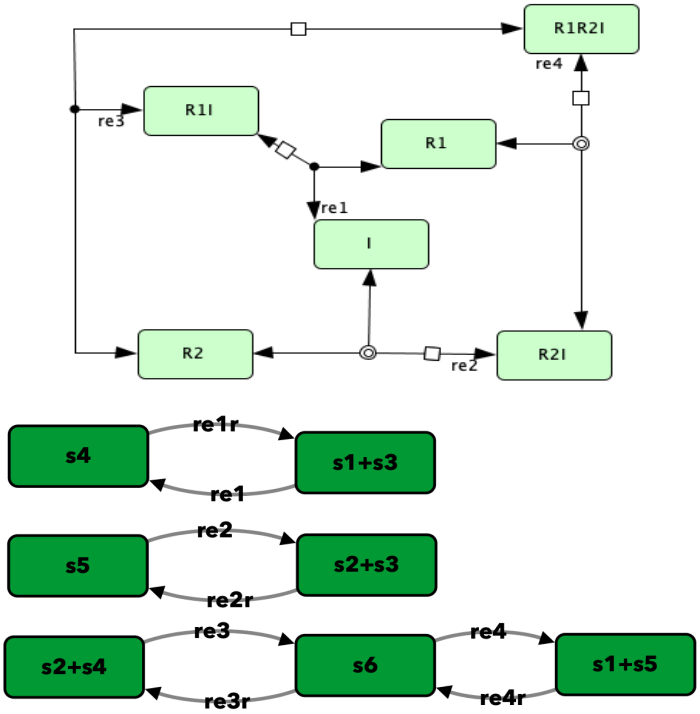
```
Number of species: 13
Number of complexes: 17
Number of reactions: 17
Network deficiency: 2

Reaction graph of the form
reaction -- reaction label:
s23+s3 -> s5  --  re1
s5 -> s23+s3  --  re1r
s5+s8 -> s24  --  re2
s24 -> s5+s8  --  re2r
s3 -> s4  --  re3
s4 -> s3  --  re3r
s4+s9 -> s16  --  re9
s16 -> s4+s9  --  re9r
s24+s14 -> s36  --  re10
s36 -> s24+s14  --  re10r
s36 -> s37+s24  --  re11
s16+s37 -> s41  --  re12
s41 -> s16+s37  --  re12r
s41 -> s16+s14  --  re13
s9+s37 -> s45  --  re14
s45 -> s9+s37  --  re14r
s45 -> s9+s14  --  re15

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 13.720093000000002

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 2.910533000000001
```

(continues on next page)

(continued from previous page)

```
Decision Vector:
[re1, re1r, re2, re2r, re3, re3r, re9, re9r, re10, re10r, re11, re12, re12r, re13,␣
↪re14, re14r, re15, s8, s9, s23, s24, s37]

Species for concentration bounds:
[s3, s4, s5, s14, s16, s36, s41, s45]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 133.578998

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 219.64752199999995

Running continuity analysis ...
Elapsed time for continuity analysis: 667.71624

The number of feasible points that satisfy the constraints: 60
Total feasible points that give F(x) = 0: 43
Total number of points that passed final_check: 43
Number of multistability plots found: 5
Elements in params_for_global_min that produce multistability:
[6, 16, 21, 27, 36]
```

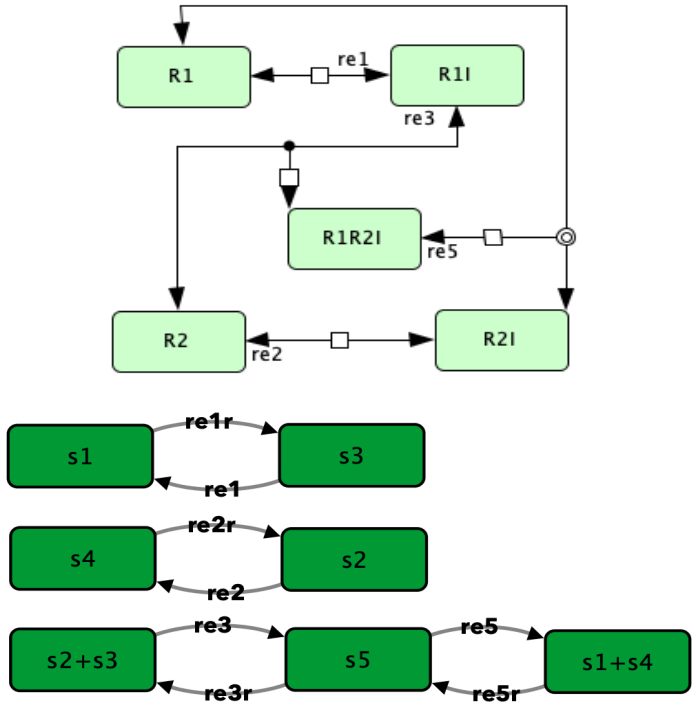
12.2.10 Closed version of Figure 4B from [irene]



To run this example download the SBML `file` and script `run_Fig4B_closed`. After running this script we obtain the following output:

```
Number of species: 6
Number of complexes: 7
Number of reactions: 8
Network deficiency: 1

Reaction graph of the form
reaction -- reaction label:
s1+s3 -> s4  --  re1
s4 -> s1+s3  --  re1r
s5 -> s2+s3  --  re2
s2+s3 -> s5  --  re2r
s2+s4 -> s6  --  re3
s6 -> s2+s4  --  re3r
s6 -> s1+s5  --  re4
s1+s5 -> s6  --  re4r

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.09931699999999966

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.6209340000000001

Decision Vector:
[re1, re1r, re2, re2r, re3, re3r, re4, re4r, s3, s4, s5]

Species for concentration bounds:
[s1, s2, s6]

Running feasible point method for 10000 iterations ...
Elapsed time for feasible point method: 121.99213200000001

Running the multistart optimization ...

Smallest value achieved by objective function: 3.0653012943157734e-09

Elapsed time for multistart method: 10424.801325999999

The number of feasible points that satisfy the constraints: 9996
Total feasible points that give F(x) = 0: 0
Total number of points that passed final_check: 0
```

12.2.11 Closed version of Figure 4C from [irene]



To run this example download the SBML `file` and script `run_Fig4C_closed`. After running this script we obtain the following output:

```
Number of species: 5
Number of complexes: 7
Number of reactions: 8
Network deficiency: 1

Reaction graph of the form
reaction -- reaction label:
s3 -> s1  --  re1
s1 -> s3  --  re1r
s2 -> s4  --  re2
s4 -> s2  --  re2r
s2+s3 -> s5  --  re3
s5 -> s2+s3  --  re3r
s5 -> s1+s4  --  re5
s1+s4 -> s5  --  re5r

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.08830100000000041

Solving for species' concentrations ...
Elapsed time for finding species' concentrations: 0.5211290000000002

Decision Vector:
[re1, re1r, re2, re2r, re3, re3r, re5, re5r, s3, s4]
```

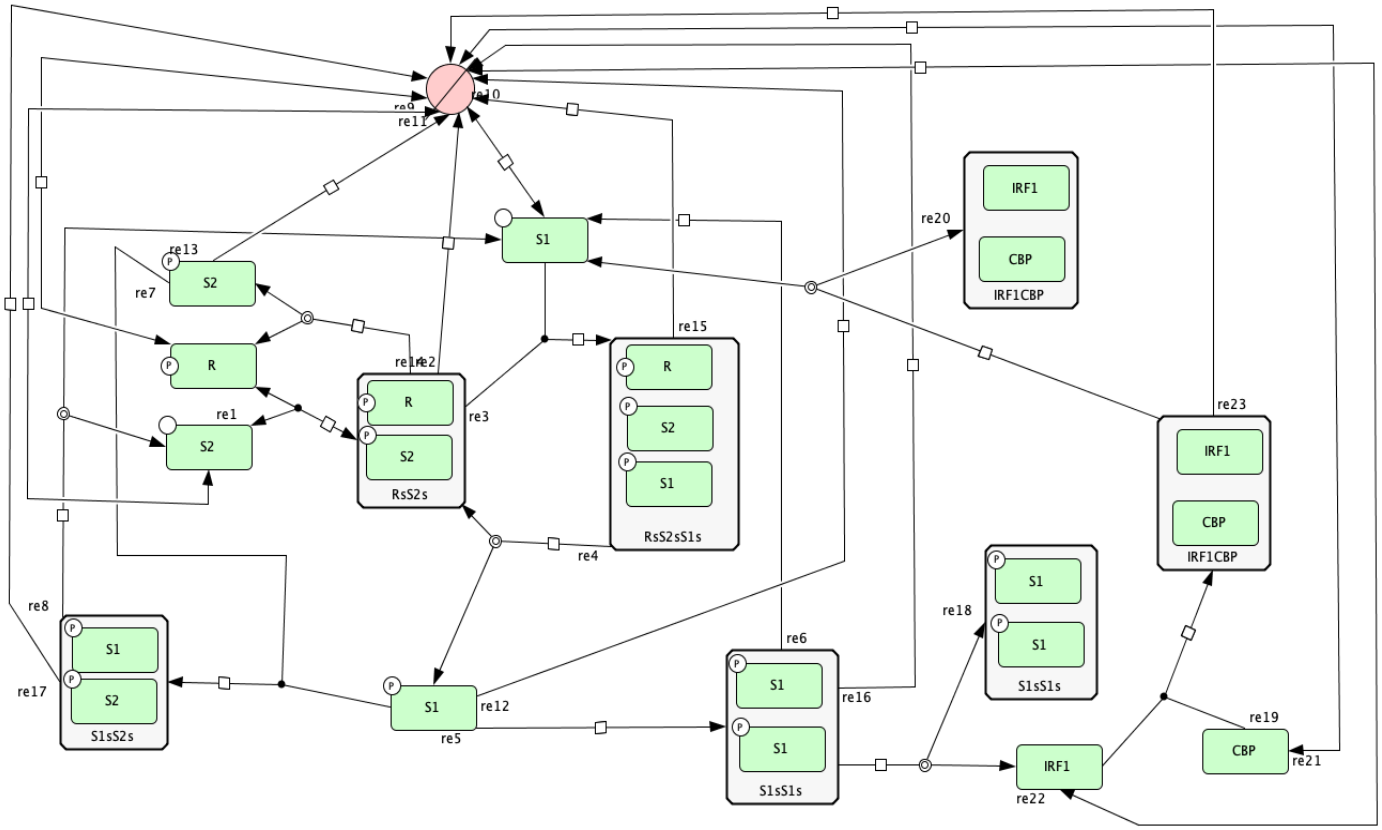
(continues on next page)

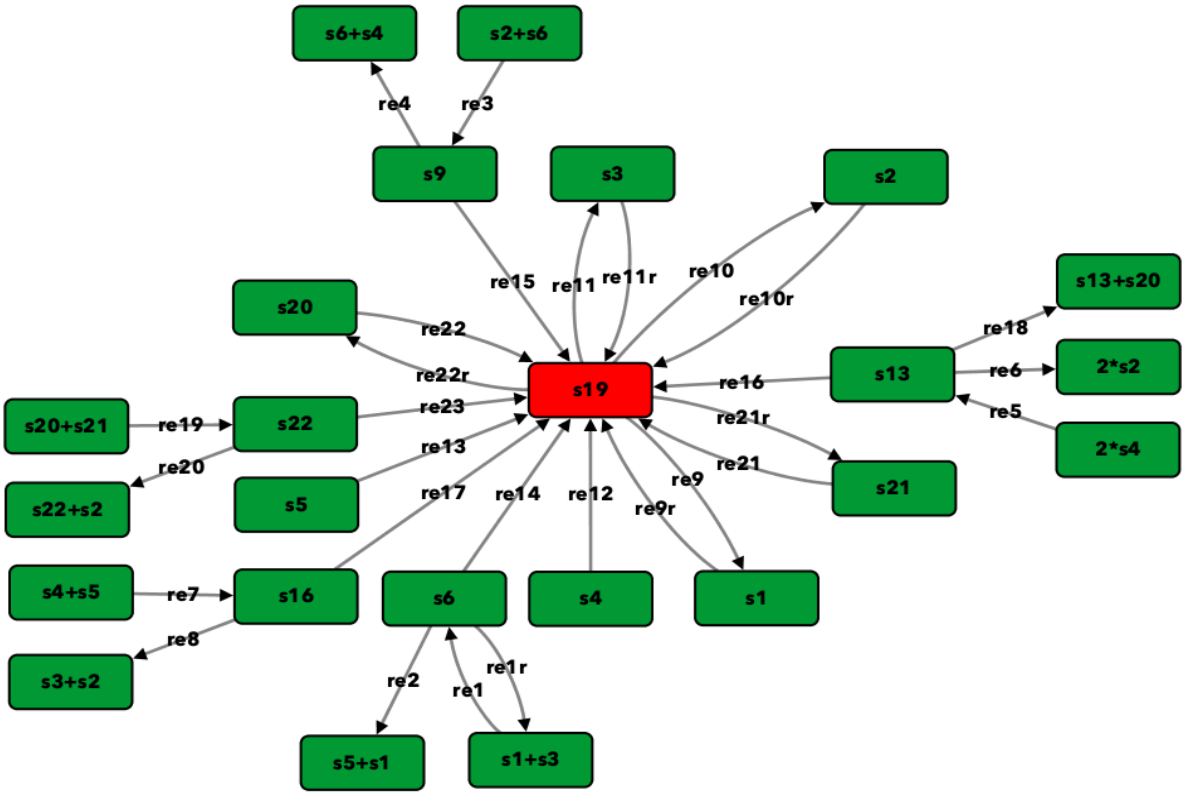
(continued from previous page)

```
Species for concentration bounds:
[s1, s2, s5]

Running feasible point method for 10000 iterations ...
Elapsed time for feasible point method: 699.610803

Running the multistart optimization ...

Smallest value achieved by objective function: 2.2272143587977585e-10

Elapsed time for multistart method: 7437.484507

The number of feasible points that satisfy the constraints: 9961
Total feasible points that give F(x) = 0: 0
Total number of points that passed final_check: 0
```

12.3 Semi-diffusive Approach

12.3.1 Figure 5B of [irene]





To run this example download the SBML `file` and script `run_open_fig5B`. After running this script we obtain the following output:

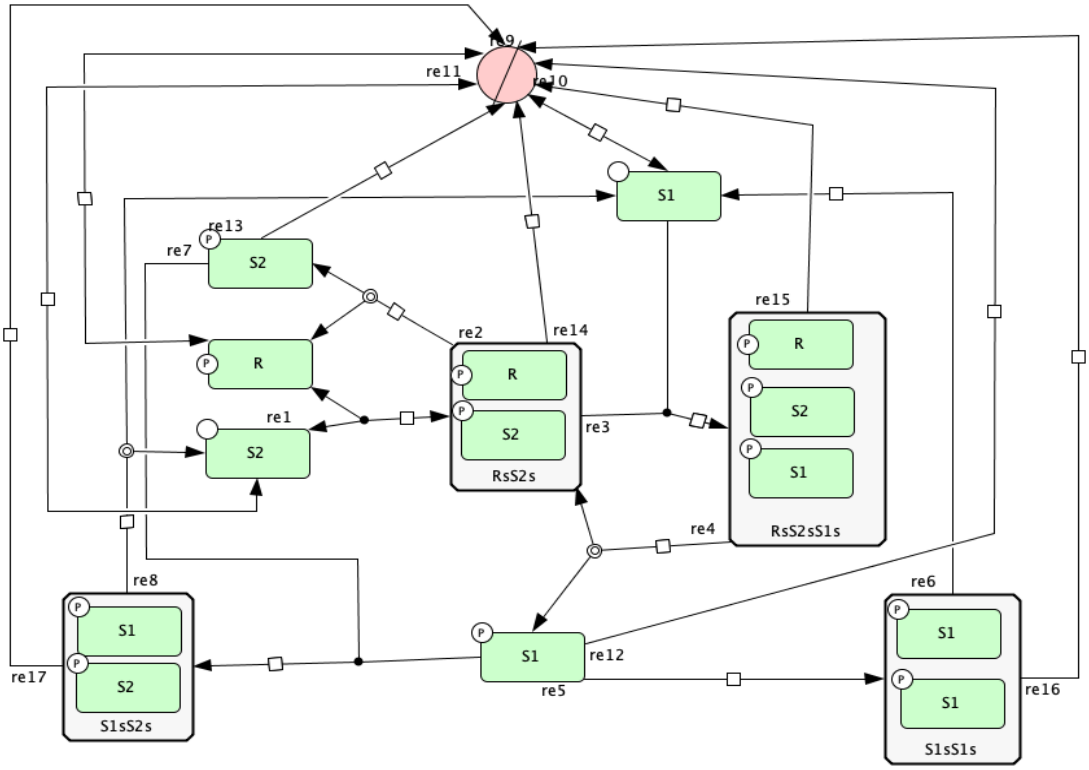
```
Number of species: 12
Number of complexes: 24
Number of reactions: 29
Network deficiency: 11

Reaction graph of the form
reaction -- reaction label:
s1+s3 -> s6  --  re1
s6 -> s1+s3  --  re1r
s6 -> s5+s1  --  re2
s2+s6 -> s9  --  re3
s9 -> s6+s4  --  re4
2*s4 -> s13  --  re5
s13 -> 2*s2  --  re6
s4+s5 -> s16  --  re7
s16 -> s3+s2  --  re8
s19 -> s1  --  re9
s1 -> s19  --  re9r
s19 -> s2  --  re10
s2 -> s19  --  re10r
s19 -> s3  --  re11
s3 -> s19  --  re11r
s4 -> s19  --  re12
s5 -> s19  --  re13
s6 -> s19  --  re14
s9 -> s19  --  re15
s13 -> s19  --  re16
```

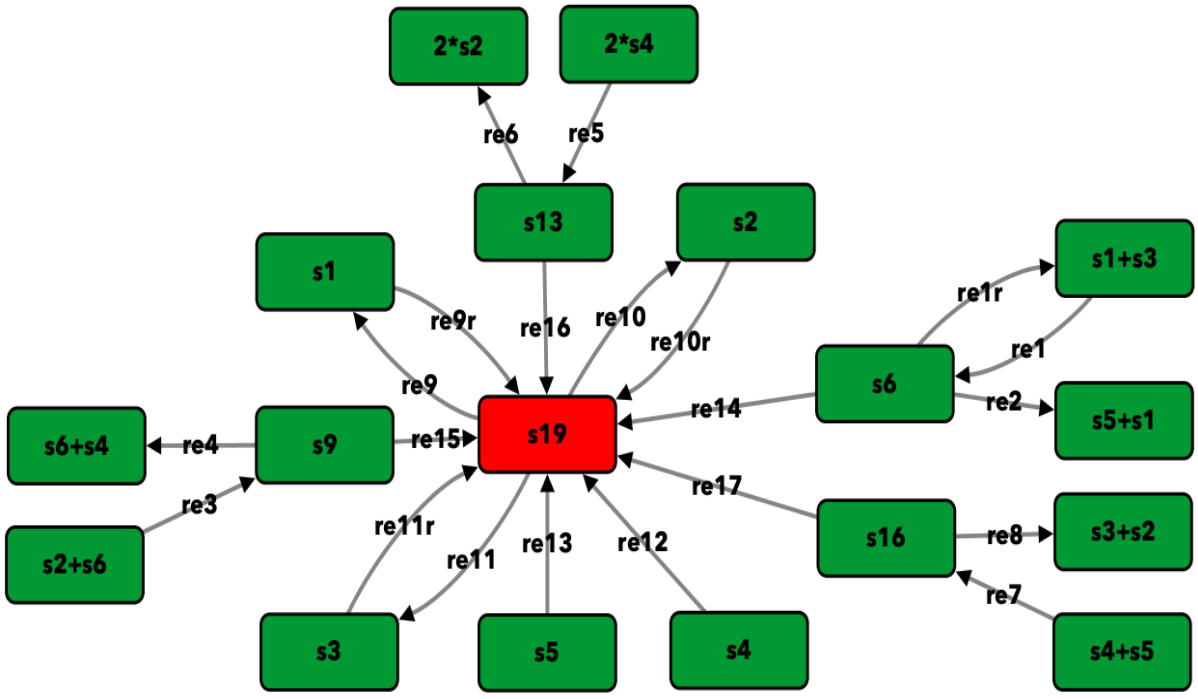
(continues on next page)

(continued from previous page)

```
s16 -> s19  --  re17
s13 -> s13+s20  --  re18
s20+s21 -> s22  --  re19
s22 -> s22+s2  --  re20
s21 -> s19  --  re21
s19 -> s21  --  re21r
s20 -> s19  --  re22
s19 -> s20  --  re22r
s22 -> s19  --  re23

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Decision vector for optimization:
[v_2, v_3, v_4, v_5, v_6, v_8, v_11, v_13, v_15, v_18, v_20, v_21, v_22, v_24, v_25,␣
↪v_27, v_29]

Reaction labels for decision vector:
['re1r', 're2', 're3', 're4', 're5', 're7', 're9r', 're10r', 're11r', 're14', 're16',
↪'re17', 're18', 're20', 're21', 're22', 're23']

Key species:
['s1', 's2', 's3', 's20', 's21']

Non key species:
['s4', 's5', 's6', 's9', 's13', 's16', 's22']

Boundary species:
['s19']

Running feasible point method for 50 iterations ...
Elapsed time for feasible point method: 15.371807999999998

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 278.575646

Running continuity analysis ...
Elapsed time for continuity analysis: 46.75262300000003

The number of feasible points that satisfy the constraints: 50
Total feasible points that give F(x) = 0: 21
Total number of points that passed final_check: 21
Number of multistability plots found: 3
Elements in params_for_global_min that produce multistability:
[2, 12, 20]
```

12.3.2 Open version of Figure 5A from [irene]





To run this example download the SBML `file` and script `run_open_fig5A`. After running this script we obtain the following output:

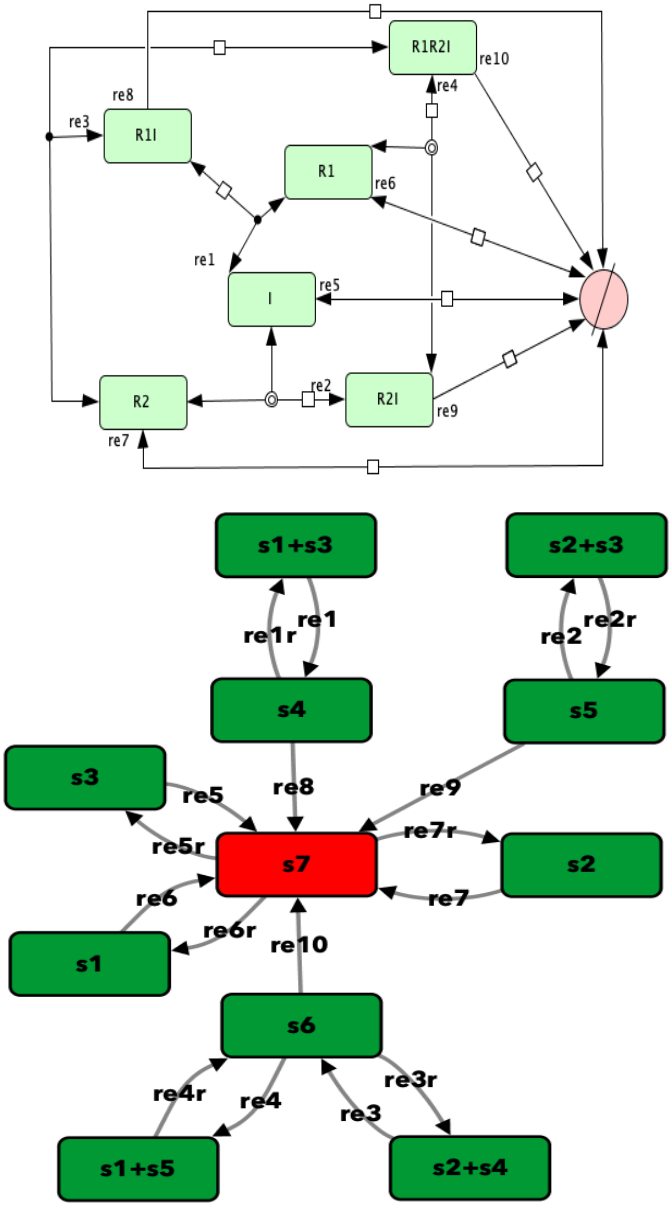
```
Number of species: 9
Number of complexes: 18
Number of reactions: 21
Network deficiency: 8

Reaction graph of the form
reaction -- reaction label:
s1+s3 -> s6  --  re1
s6 -> s1+s3  --  re1r
s6 -> s5+s1  --  re2
s2+s6 -> s9  --  re3
s9 -> s6+s4  --  re4
2*s4 -> s13  --  re5
s13 -> 2*s2  --  re6
s4+s5 -> s16  --  re7
s16 -> s3+s2  --  re8
s19 -> s1  --  re9
s1 -> s19  --  re9r
s19 -> s2  --  re10
s2 -> s19  --  re10r
s19 -> s3  --  re11
s3 -> s19  --  re11r
s4 -> s19  --  re12
s5 -> s19  --  re13
s6 -> s19  --  re14
s9 -> s19  --  re15
s13 -> s19  --  re16
s16 -> s19  --  re17

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Decision vector for optimization:
[v_2, v_3, v_4, v_5, v_6, v_8, v_11, v_13, v_15, v_18, v_20, v_21]

Reaction labels for decision vector:
['re1r', 're2', 're3', 're4', 're5', 're7', 're9r', 're10r', 're11r', 're14', 're16',
↪'re17']

Key species:
['s1', 's2', 's3']

Non key species:
['s4', 's5', 's6', 's9', 's13', 's16']

Boundary species:
['s19']

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 21.670934000000003

Running the multistart optimization ...

Smallest value achieved by objective function: 0.0

Elapsed time for multistart method: 109.49520600000002
```

(continues on next page)

(continued from previous page)

```
Running continuity analysis ...
Elapsed time for continuity analysis: 5.927085000000005

The number of feasible points that satisfy the constraints: 100
Total feasible points that give F(x) = 0: 5
Total number of points that passed final_check: 1
Number of multistability plots found: 1
Elements in params_for_global_min that produce multistability:
[0]
```

12.3.3 Figure 4B from [irene]

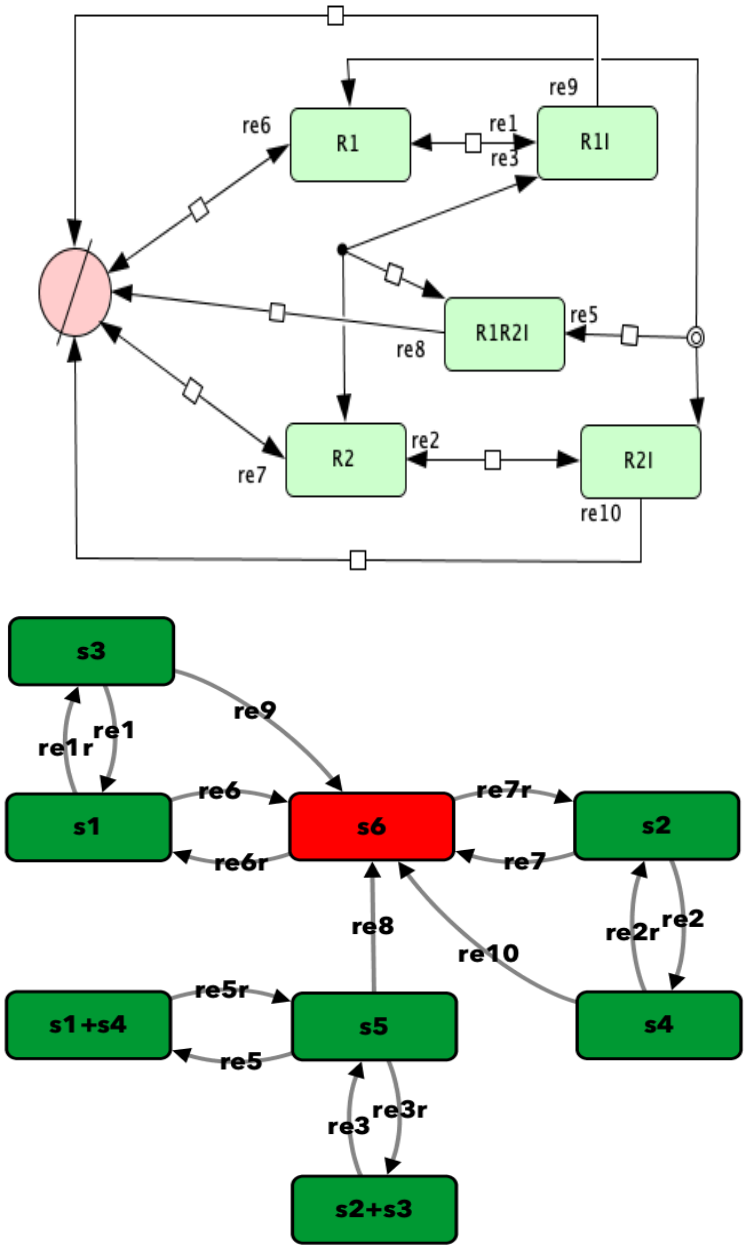


To run this example download the SBML `file` and script `run_Fig4B_open`. After running this script we obtain

the following output:

```
Number of species: 6
Number of complexes: 11
Number of reactions: 17
Network deficiency: 4

Reaction graph of the form
reaction -- reaction label:
s1+s3 -> s4  --  re1
s4 -> s1+s3  --  re1r
s5 -> s2+s3  --  re2
s2+s3 -> s5  --  re2r
s2+s4 -> s6  --  re3
s6 -> s2+s4  --  re3r
s6 -> s1+s5  --  re4
s1+s5 -> s6  --  re4r
s3 -> s7  --  re5
s7 -> s3  --  re5r
s1 -> s7  --  re6
s7 -> s1  --  re6r
s2 -> s7  --  re7
s7 -> s2  --  re7r
s4 -> s7  --  re8
s5 -> s7  --  re9
s6 -> s7  --  re10

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Decision vector for optimization:
[v_2, v_4, v_5, v_6, v_7, v_8, v_9, v_11, v_13, v_15, v_16]

Reaction labels for decision vector:
['re1r', 're2r', 're3', 're3r', 're4', 're4r', 're5', 're6', 're7', 're8', 're9']

Key species:
['s1', 's2', 's3']

Non key species:
['s4', 's5', 's6']

Boundary species:
['s7']

Running feasible point method for 10000 iterations ...
Elapsed time for feasible point method: 268.53081000000003

Running the multistart optimization ...

Smallest value achieved by objective function: 2.304503779693441e-10

Elapsed time for multistart method: 8503.097677999998

The number of feasible points that satisfy the constraints: 10000
Total feasible points that give F(x) = 0: 0
Total number of points that passed final_check: 0
```

12.3.4 Figure 4C from [irene]



To run this example download the SBML `file` and script `run_Fig4C_open`. After running this script we obtain the following output:

```
Number of species: 5
Number of complexes: 8
Number of reactions: 15
Network deficiency: 2

Reaction graph of the form
reaction -- reaction label:
s3 -> s1  --  re1
s1 -> s3  --  re1r
```

(continues on next page)

(continued from previous page)

```
s2 -> s4  --  re2
s4 -> s2  --  re2r
s2+s3 -> s5  --  re3
s5 -> s2+s3  --  re3r
s5 -> s1+s4  --  re5
s1+s4 -> s5  --  re5r
s1 -> s6  --  re6
s6 -> s1  --  re6r
s2 -> s6  --  re7
s6 -> s2  --  re7r
s5 -> s6  --  re8
s3 -> s6  --  re9
s4 -> s6  --  re10

The network does not satisfy Deficiency Zero Theorem.
The network does not satisfy Deficiency One Theorem.

Decision vector for optimization:
[v_2, v_4, v_5, v_6, v_7, v_8, v_9, v_11, v_14, v_15]

Reaction labels for decision vector:
['re1r', 're2r', 're3', 're3r', 're5', 're5r', 're6', 're7', 're9', 're10']

Key species:
['s1', 's2']

Non key species:
['s3', 's4', 's5']

Boundary species:
['s6']

Running feasible point method for 10000 iterations ...
Elapsed time for feasible point method: 215.59860999999998

Running the multistart optimization ...

Smallest value achieved by objective function: 4.5692676949898025e-10

Elapsed time for multistart method: 4489.723483

The number of feasible points that satisfy the constraints: 10000
Total feasible points that give F(x) = 0: 0
Total number of points that passed final_check: 0
```

CHAPTER 13

Reference

CRNT(path)	Class for managing CRNT methods.
Cgraph(model)	Class for constructing core CRNT values and C-graph of the network.
LowDeficiencyApproach(cgraph)	Class for testing the Deficiency Zero and One Theorems.
MassConservationApproach(cgraph)	Class for constructing variables and methods needed for the mass conservation approach.
SemiDiffusiveApproach(cgraph)	Class for constructing variables and methods needed for the semi-diffusive approach.

13.1 crnt4sbml_test.CRNT

class crnt4sbml_test.**CRNT**(*path*)

Class for managing CRNT methods.

__init__(*path*)

Initialization of CRNT class.

Parameters **path** (*string*) – String representation of the path to the XML file.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
```

Methods

__init__(path)	Initialization of CRNT class.
basic_report()	Prints out basic CRNT properties of the network.
get_physiological_range([for_what])	Obtains physiological ranges.
get_low_deficiency_approach()	Initializes and creates an object for the class LowDeficiencyApproach for the CRNT object constructed.
get_mass_conservation_approach()	Initializes and creates an object for the class MassConservationApproach for the CRNT object constructed.
get_semi_diffusive_approach()	Initializes and creates an object for the class SemiDiffusiveApproach for the CRNT object constructed.
get_advanced_deficiency_approach()	Placeholder for Advanced Deficiency Approach.
get_c_graph()	Allows access to the class C-graph for the constructed CRNT object.
print_c_graph()	Prints the reactions and reaction labels for the network.
plot_c_graph()	Generates a matplotlib plot for the C-graph of the network using the networkx.draw function with circular and Kamada Kawai layout.
plot_save_c_graph()	Saves the matplotlib plot for the C-graph of the network using the networkx.draw function with circular and Kamada Kawai layout to the file network_cgraph.png
get_network_graphml()	Writes the NetworkX Digraph to the file network.graphml.

basic_report()
Prints out basic CRNT properties of the network. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> network.basic_report()
    Number of species: 7
    Number of complexes: 9
    Number of reactions: 9
    Network deficiency: 2
```

get_advanced_deficiency_approach()
Placeholder for Advanced Deficiency Approach. Future version of crnt4sbml will include the implementation of the Higher Deficiency Algorithm.

get_c_graph()
Allows access to the class C-graph for the constructed CRNT object. Returns C-graph object for the provided CRNT object.

See also:

crnt4sbml_test.Cgraph()

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> c_graph = network.get_c_graph()
```

get_low_deficiency_approach()
Initializes and creates an object for the class LowDeficiencyApproach for the CRNT object constructed.

See also:

crnt4sbml_test.LowDeficiencyApproach()

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> approach = network.get_low_deficiency_approach()
```

get_mass_conservation_approach()
Initializes and creates an object for the class MassConservationApproach for the CRNT object constructed. Fig1Ci.xml for the provided example.

See also:

crnt4sbml_test.MassConservationApproach()

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.8010799999999998
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 1.1221319999999997
```

get_network_graphml()
Writes the NetworkX Digraph to the file network.graphml. Note that this generation only includes the names of the nodes, edges, and edge reaction names, it does not include other list attributes of the nodes and edges.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_network_graphml()
```

static get_physiological_range(*for_what=None*)
Obtains physiological ranges.

Parameters **for_what** (*string*) – Accepted values: “concentration”, “complex formation”, “complex dissociation”, or “catalysis”

Returns

- *concentration* – 5e-1,5e5 pM
- *complex formation* – 1e-8,1e-4 pM^-1s^-1
- *complex dissociation* – 1e-5,1e-3 s^-1
- *catalysis* – 1e-3,1 s^-1

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_physiological_range("concentration")
```

get_semi_diffusive_approach()
Initializes and creates an object for the class SemiDiffusiveApproach for the CRNT object constructed.

See also:

crnt4sbml_test.SemiDiffusiveApproach()

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/semi_diffusive_sbml_file.xml")
>>> approach = network.get_semi_diffusive_approach()
```

plot_c_graph()
Generates a matplotlib plot for the C-graph of the network using the networkx.draw function with circular and Kamada Kawai layout.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.plot_c_graph()
```

plot_save_c_graph()
Saves the matplotlib plot for the C-graph of the network using the networkx.draw function with circular and Kamada Kawai layout to the file network_cgraph.png

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.plot_save_c_graph()
```

print_c_graph()
Prints the reactions and reaction labels for the network. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> network.print_c_graph()
    Reaction graph of the form
    reaction -- reaction label:
    s1+s2 -> s3  --  re1
    s3 -> s1+s2  --  re1r
    s3 -> s6+s2  --  re2
    s6+s7 -> s16  --  re3
    s16 -> s6+s7  --  re3r
    s16 -> s7+s1  --  re4
    s1+s6 -> s15  --  re6
    s15 -> s1+s6  --  re6r
    s15 -> 2*s6  --  re8
```

13.2 crnt4sbml_test.Cgraph

class crnt4sbml_test.**Cgraph**(*model*)

Class for constructing core CRNT values and C-graph of the network.

__init__(*model*)

Initialization of Cgraph class.

See also:

crnt4sbml_test.CRNT.get_c_graph()

Methods

__init__(model)	Initialization of Cgraph class.
get_species()	Returns Python list of strings representing the species of the network.
get_complexes()	Returns Python list of strings representing the complexes of the network.
get_reactions()	Returns Python list of strings representing the reactions of the network.
get_deficiency()	Returns integer value representing the deficiency of the network, δ.
get_dim_equilibrium_manifold()	Returns integer value representing the dimension of the equilibrium manifold, λ.
get_ode_system()	Returns SymPy matrix representing the ODE system.
get_a()	Returns SymPy matrix representing the kinetic constant matrix, A.
get_b()	Returns SymPy matrix representing the mass conservation matrix, B.
get_s()	Returns SymPy matrix representing the stoichiometric matrix, S.

Continued on next page

Table 3 – continued from previous page

get_y()	Returns SymPy matrix representing the molecularity matrix, Y.
get_lambda()	Returns SymPy matrix representing the linkage class matrix, Λ.
get_psi()	Returns SymPy matrix representing the mass action monomials, ψ.
get_graph()	Returns the NetworkX DiGraph representation of the network.
get_g_nodes()	Returns a list of strings that represent the order of the nodes of the NetworkX DiGraph.
get_g_edges()	Returns a list of tuples of strings that represent the order of the edges of the NetworkX DiGraph.
get_network_dimensionality_classification()	Returns a two element list specifying the dimensionality of the network.
get_linkage_classes()	Returns list of NetworkX subgraphs representing the linkage classes.
get_linkage_classes_deficiencies()	Returns an interger list of each linkage class deficiency.
get_if_cgraph_weakly_reversible()	Returns weak reversibility of the network.
get_weak_reversibility_of_linkage_classes()	Returns list of Python boolean types for the weak reversibility of each linkage class.
get_number_of_terminal_strong_lc_per_lc()	Returns an integer list stating the number of terminally strong linkage classes per linkage class.
print()	Prints edges and nodes of NetworkX DiGraph.
plot()	Plots NetworkX DiGraph.
plot_save()	Saves the plot of the NetworkX DiGraph.

get_a()

Returns SymPy matrix representing the kinetic constant matrix, A. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> sympy.pprint(network.get_c_graph().get_a())
⎡   -re₁      re1r       0    0          0        0    0          0        0⎤
⎢                                                                         ⎥
⎢   re₁    -re1r - re₂   0    0          0        0    0          0        0⎥
⎢                                                                         ⎥
⎢    0         re₂       0    0          0        0    0          0        0⎥
⎢                                                                         ⎥
⎢    0          0        0  -re₃      re3r       0    0          0        0⎥
⎢                                                                         ⎥
⎢    0          0        0   re₃    -re3r - re₄  0    0          0        0⎥
⎢                                                                         ⎥
⎢    0          0        0    0         re₄       0    0          0        0⎥
⎢                                                                         ⎥
⎢    0          0        0    0          0        0  -re₆      re6r       0⎥
⎢                                                                         ⎥
⎢    0          0        0    0          0        0   re₆    -re6r - re₈  0⎥
⎢                                                                         ⎥
⎣    0          0        0    0          0        0    0         re₈       0⎦
```

get_b()
Returns SymPy matrix representing the mass conservation matrix, B. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> sympy.pprint(network.get_c_graph().get_b())
 0    0    0    0   1.0   0   1.0

 0   1.0  1.0   0    0    0    0

1.0   0   1.0  1.0   0   2.0  1.0
```

get_complexes()
Returns Python list of strings representing the complexes of the network. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> print(network.get_c_graph().get_complexes())
    ['s1+s2', 's3', 's6+s2', 's6+s7', 's16', 's7+s1', 's1+s6', 's15', '2*s6']
```

get_deficiency()
Returns integer value representing the deficiency of the network, δ. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> print(network.get_c_graph().get_deficiency())
    2
```

get_dim_equilibrium_manifold()
Returns integer value representing the dimension of the equilibrium manifold, λ. This value is the number of mass conservation relationships. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> print(network.get_c_graph().get_dim_equilibrium_manifold())
    3
```

get_g_edges()
Returns a list of tuples of strings that represent the order of the edges of the NetworkX DiGraph.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_g_edges()
```

get_g_nodes()
Returns a list of strings that represent the order of the nodes of the NetworkX DiGraph.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_g_nodes()
```

get_graph()
Returns the NetworkX DiGraph representation of the network.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_graph()
```

get_if_cgraph_weakly_reversible()
Returns weak reversibility of the network. If the network is weakly reversible True is returned, False otherwise.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_if_cgraph_weakly_reversible()
```

get_lambda()
Returns SymPy matrix representing the linkage class matrix, Λ. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> sympy.pprint(network.get_c_graph().get_lambda())
    1  0  0

    1  0  0
```

(continues on next page)

(continued from previous page)

```
    1   0   0

    0   1   0

    0   1   0

    0   1   0

    0   0   1

    0   0   1

    0   0   1
```

get_linkage_classes()
Returns list of NetworkX subgraphs representing the linkage classes.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_linkage_classes()
```

get_linkage_classes_deficiencies()
Returns an interger list of each linkage class deficiency. Here, the first element corresponds to the first linkage class with order as defined by *crnt4sbml_test.Cgraph.get_linkage_classes()*.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_linkage_classes_deficiencies()
```

get_network_dimensionality_classification()
Returns a two element list specifying the dimensionality of the network. Possible output: [“over-dimensioned”,0]

or

[“proper”,1]

or

[“under-dimensioned”,2]

or

[“NOT DEFINED!”,3]

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_network_dimensionality_classification()
```

get_number_of_terminal_strong_lc_per_lc()
Returns an integer list stating the number of terminally strong linkage classes per linkage class. Here, the first element corresponds to the first linkage class with order as defined by *crnt4sbml_test.Cgraph.get_linkage_classes()*.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_number_of_terminal_strong_lc_per_lc()
```

get_ode_system()
Returns SymPy matrix representing the ODE system. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> sympy.pprint(network.get_c_graph().get_ode_system())
        -re1*s1*s2 + re1r*s3 + re4*s16 - re6*s1*s6 + re6r*s15

                        -re1*s1*s2 + s3*(re1r + re2)

                        re1*s1*s2 + s3*(-re1r - re2)

    re2*s3 - re3*s6*s7 + re3r*s16 - re6*s1*s6 + s15*(re6r + 2*re8)

                       -re3*s6*s7 + s16*(re3r + re4)

                       re6*s1*s6 + s15*(-re6r - re8)

                       re3*s6*s7 + s16*(-re3r - re4)
```

get_psi()
Returns SymPy matrix representing the mass action monomials, ψ. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> sympy.pprint(network.get_c_graph().get_psi())
    s1*s2

     s3

    s2*s6

    s6*s7

     s16
```

(continues on next page)

(continued from previous page)

```
    S1S7

    S1S6

     S15

       2
     S6
```

get_reactions()
Returns Python list of strings representing the reactions of the network. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> print(network.get_c_graph().get_reactions())
    ['re1', 're1r', 're2', 're3', 're3r', 're4', 're6', 're6r', 're8']
```

get_s()
Returns SymPy matrix representing the stoichiometric matrix, S. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> sympy.pprint(network.get_c_graph().get_s())
    -1  1   0   0   0   1   -1  1   0

    -1  1   1   0   0   0   0   0   0

    1   -1  -1  0   0   0   0   0   0

    0   0   1   -1  1   0   -1  1   2

    0   0   0   -1  1   1   0   0   0

    0   0   0   0   0   0   1   -1  -1

    0   0   0   1   -1  -1  0   0   0
```

get_species()
Returns Python list of strings representing the species of the network. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> print(network.get_c_graph().get_species())
    ['s1', 's2', 's3', 's6', 's7', 's15', 's16']
```

get_weak_reversibility_of_linkage_classes()

Returns list of Python boolean types for the weak reversibility of each linkage class. If the linkage class is weakly reversible then the entry in the list is True, False otherwise with order as defined by *crnt4sbml_test.Cgraph.get_linkage_classes()*.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> network.get_c_graph().get_weak_reversibility_of_linkage_classes()
```

get_y()

Returns SymPy matrix representing the molecularity matrix, Y. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> sympy.pprint(network.get_c_graph().get_y())
    1  0  0  0  0  1  1  0  0

    1  0  1  0  0  0  0  0  0

    0  1  0  0  0  0  0  0  0

    0  0  1  1  0  0  1  0  2

    0  0  0  1  0  1  0  0  0

    0  0  0  0  0  0  0  1  0

    0  0  0  0  1  0  0  0  0
```

plot()

Plots NetworkX DiGraph.

See also:

crnt4sbml_test.CRNT.plot_c_graph()

plot_save()

Saves the plot of the NetworkX DiGraph.

See also:

crnt4sbml_test.CRNT.plot_save_c_graph()

print()

Prints edges and nodes of NetworkX DiGraph.

See also:

crnt4sbml_test.CRNT.print_c_graph()

13.3 crnt4sbml_test.LowDeficiencyApproach

class crnt4sbml_test.**LowDeficiencyApproach**(*cgraph*)

Class for testing the Deficiency Zero and One Theorems.

__init__(*cgraph*)

Initialization of LowDeficiency Approach class.

See also:

crnt4sbml_test.CRNT.get_low_deficiency_approach()

Methods

__init__(cgraph)	Initialization of LowDeficiency Approach class.
does_satisfy_deficiency_zero_theorem()	Function to see if the network satisfies the Deficiency Zero Theorem.
does_satisfy_deficiency_one_theorem()	Function to see if the network satisfies the Deficiency One Theorem.
does_satisfy_any_low_deficiency_theorem()	Function to see if the network satisfies the Deficiency Zero or One Theorem.
report_deficiency_zero_theorem()	Prints out the applicability of the Deficiency Zero Theorem for the provided network.
report_deficiency_one_theorem()	Prints out the applicability of the Deficiency One Theorem for the provided network.

does_satisfy_any_low_deficiency_theorem()

Function to see if the network satisfies the Deficiency Zero or One Theorem. Returns True if the network satisfies the Deficiency Zero or One Theorem, False otherwise. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_low_deficiency_approach()
>>> print(approach.does_satisfy_any_low_deficiency_theorem())
    False
```

does_satisfy_deficiency_one_theorem()

Function to see if the network satisfies the Deficiency One Theorem. Returns True if the network satisfies the Deficiency One Theorem, False otherwise. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
```

(continues on next page)

(continued from previous page)

```
>>> approach = network.get_low_deficiency_approach()
>>> print(approach.does_satisfy_deficiency_one_theorem())
    False
```

does_satisfy_deficiency_zero_theorem()

Function to see if the network satisfies the Deficiency Zero Theorem. Returns True if the network satisfies the Deficiency Zero Theorem, False otherwise. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_low_deficiency_approach()
>>> print(approach.does_satisfy_deficiency_zero_theorem())
    False
```

report_deficiency_one_theorem()

Prints out the applicability of the Deficiency One Theorem for the provided network. Possible output:

"By the Deficiency One Theorem, the differential equations admit precisely one equilibrium in each positive stoichiometric compatibility class. Thus, multiple equilibria cannot exist for the network."

or

"The network satisfies relaxed Deficiency One Theorem. That is it is not weakly reversable, but each linkage class contains no more than one terminal linkage class. There can exist within a positive stoichiometric compatibility class at most one equilibrium. Thus, multiple equilibria cannot exist for the network."

or

"The network does not satisfy Deficiency One Theorem."

Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_low_deficiency_approach()
>>> print(approach.report_deficiency_zero_theorem())
    The network does not satisfy Deficiency One Theorem.
```

report_deficiency_zero_theorem()

Prints out the applicability of the Deficiency Zero Theorem for the provided network. Possible output:

"By the Deficiency Zero Theorem, the differential equations cannot admit a positive equilibrium or a cyclic composition trajectory containing a positive composition. Thus, multiple equilibria cannot exist for the network."

or

"By the Deficiency Zero Theorem, there exists within each positive stoichiometric compatibility class precisely one equilibrium. Thus, multiple equilibria cannot exist for the network."

or

"The network does not satisfy Deficiency Zero Theorem."

Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_low_deficiency_approach()
>>> print(approach.report_deficiency_zero_theorem())
    The network does not satisfy Deficiency Zero Theorem.
```

13.4 crnt4sbml_test.MassConservationApproach

class crnt4sbml_test.**MassConservationApproach**(*cgraph*)

Class for constructing variables and methods needed for the mass conservation approach.

__init__(*cgraph*)

Initialization of the MassConservationApproach class.

See also:

crnt4sbml_test.CRNT.get_mass_conservation_approach()

Methods

__init__(cgraph)	Initialization of the MassConservationApproach class.
generate_report()	Prints out helpful details constructed by *crnt4sbml_test.MassConservationApproach.run_optimization()* and *crnt4sbml_test.MassConservationApproach.run_continuity_analysis()*.
get_conservation_laws()	Returns a string representation of the conservation laws.
get_decision_vector()	Returns a list of SymPy variables that represent the decision vector of the optimization problem.
get_objective_fun_params()	Returns a list of SymPy variables that represent those variables that may be contained in the G matrix, Jacobian of the equilibrium manifold with respect to the species, or objective function.
get_concentration_vals()	Returns a list of SymPy expressions representing the species in terms of those variables present in the decision vector.
get_concentration_solutions()	Returns a more readable string representation of the species defined in terms of the decision vector.
get_concentration_funs()	Returns a list of lambda functions representing each of the species.

Continued on next page

Table 5 – continued from previous page

get_concentration_bounds_species()	Returns a list of SymPy variables that represents the order of species for the concentration bounds provided to *crnt4sbml_test.MassConservationApproach.run_optimization()*.
get_w_nullspace()	Returns a list of SymPy column vectors representing $Null([Y, \Lambda^T]^T)$.
get_w_matrix()	Returns SymPy matrix $[Y, \Lambda^T]^T$, which we call the W matrix.
get_dch_matrix()	Returns a SymPy matrix representing the Jacobian of the equilibrium manifold with respect to the species.
get_lambda_dch_matrix()	Returns a lambda function representation of the Jacobian of the equilibrium manifold matrix.
get_h_vector()	Returns a SymPy matrix representing the equilibrium manifold.
get_g_matrix()	Returns a SymPy matrix representing the G matrix of the defined optimization problem.
get_lambda_g_matrix()	Returns a lambda function representation of the G matrix.
get_symbolic_objective_fun()	Returns SymPy expression for the objective function of the optimization problem.
get_lambda_objective_fun()	Returns a lambda function representation of the objective function of the optimization problem.
run_optimization([bounds, iterations, ...])	Function for running the optimization problem for the mass conservation approach.
run_continuity_analysis([species, ...])	Function for running the numerical continuation and bistability analysis portions of the mass conservation approach.
run_greedy_continuity_analysis([species, ...])	Function for running the greedy numerical continuation and bistability analysis portions of the mass conservation approach.

generate_report()
Prints out helpful details constructed by *crnt4sbml_test.MassConservationApproach.run_optimization()* and *crnt4sbml_test.MassConservationApproach.run_continuity_analysis()*.

Example

See *Mass Conservation Approach Example* and *Mass Conservation Approach Walkthrough*.

get_concentration_bounds_species()
Returns a list of SymPy variables that represents the order of species for the concentration bounds provided to *crnt4sbml_test.MassConservationApproach.run_optimization()*. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
```

(continues on next page)

(continued from previous page)

```
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_concentration_bounds_species())
    [s1, s3, s7, s15]
```

get_concentration_funs()

Returns a list of lambda functions representing each of the species. Here the species are those expressions provided by *crnt4sbml_test.MassConservationApproach.get_concentration_vals()* where the arguments of each lambda function is provided by *crnt4sbml_test.MassConservationApproach.get_decision_vector()*. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_concentration_funs())
    [<function _lambdifygenerated at 0x12b673f28>, <function _
↪lambdifygenerated at 0x12b7a6ea0>,
    <function _lambdifygenerated at 0x12b7a6d08>, <function _
↪lambdifygenerated at 0x12b7a67b8>,
    <function _lambdifygenerated at 0x12b7a6d90>, <function _
↪lambdifygenerated at 0x12b7a6488>,
    <function _lambdifygenerated at 0x12b7a6950>]
```

get_concentration_solutions()

Returns a more readable string representation of the species defined in terms of the decision vector. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_concentration_solutions())
    s1 = re4*s16*(re1r*re6r + re1r*re8 + re2*re6r + re2*re8)/(re1*re2*re6r*s2␣
↪+ re1*re2*re8*s2 + re1r*re6*re8*s6 + re2*re6*re8*s6)
    s2 = s2
    s3 = re1*re4*s16*s2*(re6r + re8)/(re1*re2*re6r*s2 + re1*re2*re8*s2 +␣
↪re1r*re6*re8*s6 + re2*re6*re8*s6)
```

(continues on next page)

(continued from previous page)

```
    s6 = s6
    s7 = s16*(re3r + re4)/(re3*s6)
    s15 = re4*re6*s16*s6*(re1r + re2)/(re1*re2*re6r*s2 + re1*re2*re8*s2 +␣
↪re1r*re6*re8*s6 + re2*re6*re8*s6)
    s16 = s16
```

get_concentration_vals()

Returns a list of SymPy expressions representing the species in terms of those variables present in the decision vector. The order is that established in *crnt4sbml_test.Cgraph.get_species()*. Note that if only a single species is provided as an element in the list, this means the species is a free variable.

See also:

crnt4sbml_test.MassConservationApproach.get_concentration_solutions()

Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_concentration_vals())
    [re4*s16*(re1r*re6r + re1r*re8 + re2*re6r + re2*re8)/(re1*re2*re6r*s2 +␣
↪re1*re2*re8*s2 + re1r*re6*re8*s6 +
    re2*re6*re8*s6), s2, re1*re4*s16*s2*(re6r + re8)/(re1*re2*re6r*s2 +␣
↪re1*re2*re8*s2 + re1r*re6*re8*s6 +
    re2*re6*re8*s6), s6, s16*(re3r + re4)/(re3*s6), re4*re6*s16*s6*(re1r +␣
↪re2)/(re1*re2*re6r*s2 +
    re1*re2*re8*s2 + re1r*re6*re8*s6 + re2*re6*re8*s6), s16]
```

get_conservation_laws()

Returns a string representation of the conservation laws. Here the values on the left hand side of each equation are the constants of the conservation laws. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_conservation_laws())
    C1 = 1.0*s16 + 1.0*s7
    C2 = 1.0*s2 + 1.0*s3
    C3 = 1.0*s1 + 2.0*s15 + 1.0*s16 + 1.0*s3 + 1.0*s6
```

get_dch_matrix()
Returns a SymPy matrix representing the Jacobian of the equilibrium manifold with respect to the species. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> sympy.pprint(approach.get_dch_matrix())
    -re1s2  -re1s1     re1r          0        0          0            0

    re1s2   re1s1   -re1r - re2      0        0          0            0

       0       0          0        -re3s7  -re3s6       0          re3r

       0       0          0        re3s7   re3s6        0       -re3r - re4

    -re6s6     0          0       -re6s1     0        re6r           0

    re6s6      0          0        re6s1     0     -re6r - re8       0
```

get_decision_vector()
Returns a list of SymPy variables that represent the decision vector of the optimization problem. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_decision_vector())
    [re1, re1r, re2, re3, re3r, re4, re6, re6r, re8, s2, s6, s16]
```

get_g_matrix()
Returns a SymPy matrix representing the G matrix of the defined optimization problem. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
```

(continues on next page)

(continued from previous page)

```
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> sympy.pprint(approach.get_g_matrix())
    -re₁s₂  -re₁s₁     re1r          0         0          0             0        1␣
↪ -1
                                                                                  ␣
↪
    re₁s₂   re₁s₁   -re1r - re₂      0         0          0             0        0␣
↪ 0
                                                                                  ␣
↪
      0         0          0        -re₃s₇   -re₃s₆        0           re3r       1␣
↪ 0
                                                                                  ␣
↪
      0         0          0        re₃s₇    re₃s₆         0       -re3r - re₄    0␣
↪ 0
                                                                                  ␣
↪
    -re₆s₆      0          0       -re₆s₁      0         re6r            0        0␣
↪ 1
                                                                                  ␣
↪
    re₆s₆       0          0       re₆s₁       0     -re6r - re₈         0        0␣
↪ 0
                                                                                  ␣
↪
      0         0          0          0        1.0         0           1.0         ␣
↪ 0   0
                                                                                  ␣
↪
      0        1.0        1.0         0         0          0            0          ␣
↪ 0   0
                                                                                  ␣
↪
     1.0        0         1.0        1.0        0         2.0          1.0         ␣
↪ 0   0
```

get_h_vector()

Returns a SymPy matrix representing the equilibrium manifold. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> sympy.pprint(approach.get_h_vector())
    a₁ - a₂ - re₁s₁s₂ + re1rs₃
```

(continues on next page)

(continued from previous page)

```
    re1s1s2 + s3(-re1r - re2)

      a1 - re3s6s7 + re3rs16

    re3s6s7 + s16(-re3r - re4)

      a2 - re6s1s6 + re6rs15

    re6s1s6 + s15(-re6r - re8)
```

get_lambda_dch_matrix()

Returns a lambda function representation of the Jacobian of the equilibrium manifold matrix. Here the arguments of the lambda function are given by the values provided by *crnt4sbml_test.MassConservationApproach.get_objective_fun_params()*. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_lambda_dch_matrix())
    <function _lambdifygenerated at 0x131a06ea0>
```

get_lambda_g_matrix()

Returns a lambda function representation of the G matrix. Here the arguments of the lambda function are given by the values provided by *crnt4sbml_test.MassConservationApproach.get_objective_fun_params()*. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_lambda_g_matrix())
    <function _lambdifygenerated at 0x13248ac80>
```

get_lambda_objective_fun()

Returns a lambda function representation of the objective function of the optimization problem. Here the arguments of the lambda function are given by the values provided by *crnt4sbml_test.MassConservationApproach.get_objective_fun_params()*. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_lambda_objective_fun())
    <function _lambdifygenerated at 0x12f6f7ea0>
```

get_objective_fun_params()

Returns a list of SymPy variables that represent those variables that may be contained in the G matrix, Jacobian of the equilibrium manifold with respect to the species, or objective function. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_objective_fun_params())
    [re1, re1r, re2, re3, re3r, re4, re6, re6r, re8, s1, s2, s3, s6, s7, s15,␣
↪s16]
```

get_symbolic_objective_fun()

Returns SymPy expression for the objective function of the optimization problem. This is the determinant of the G matrix squared. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> print(approach.get_symbolic_objective_fun())
    1.0*re1**2*re2**2*re3**2*re4**2*re6**2*re8**2*s1**2*s6**2*s7**2*((1.0*s2/
↪s7 - 1.0*s2*(-re3r - re4)/
    (re3*s6*s7))/re4 + (1.0 + 1.0*re1r/(re1*s1))/re2 + 1.0/(re1*s1))**2*(-((1.
↪0*s6*(-1.0*s1/s6 + 1.0)/s7 + 1.0
    - (-re3r - re4)*(-1.0*s1/s6 + 1.0)/(re3*s7))/re4 + 1.0/re2)*(-1.0*re6r*s2/
↪(re6*re8*s1*s6) - 1.0*s2*(1 +
    re6*s6/(re1*s2))/(re6*s1*s6) - (1.0 + 1.0*re1r/(re1*s1))/re2)/((1.0*s2/s7␣
↪- 1.0*s2*(-re3r - re4)/
```

(continues on next page)

(continued from previous page)

```
    (re3*s6*s7))/re4 + (1.0 + 1.0*re1r/(re1*s1))/re2 + 1.0/(re1*s1)) + (2.0 +␣
↪1.0*re6r/(re6*s6))/re8 + 1.0*(1 +
    re6*s6/(re1*s2))/(re6*s6) - 1.0/re2 - 1.0/(re1*s2))**2
```

get_w_matrix()

Returns SymPy matrix $[Y, \Lambda^T]^T$, which we call the W matrix. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> sympy.pprint(approach.get_w_matrix())
    1  0  0  0  0  1  1  0  0

    1  0  1  0  0  0  0  0  0

    0  1  0  0  0  0  0  0  0

    0  0  1  1  0  0  1  0  2

    0  0  0  1  0  1  0  0  0

    0  0  0  0  0  0  0  1  0

    0  0  0  0  1  0  0  0  0

    1  1  1  0  0  0  0  0  0

    0  0  0  1  1  1  0  0  0

    0  0  0  0  0  0  1  1  1
```

get_w_nullspace()

Returns a list of SymPy column vectors representing $Null([Y, \Lambda^T]^T)$. Fig1Ci.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 0.7020819999999999
    Solving for species' concentrations ...
    Elapsed time for finding species' concentrations: 0.888776
>>> sympy.pprint(approach.get_w_nullspace())
```

(continues on next page)

(continued from previous page)

```
    -1  1

    0   0

    1   -1

    -1  0

    0 , 0

    1   0

    0   -1

    0   0

    0   1
```

run_continuity_analysis(*species=None*, *parameters=None*, *dir_path='./num_cont_graphs'*, *print_lbls_flag=False*, *auto_parameters=None*)

Function for running the numerical continuation and bistability analysis portions of the mass conservation approach.

Parameters

- **species** (*string*) – A string stating the species that is the y-axis of the bifurcation diagram.
- **parameters** (*list of numpy arrays*) – A list of numpy arrays corresponding to the decision vectors that produce a small objective function value.
- **dir_path** (*string*) – A string stating the path where the bifurcation diagrams should be saved.
- **print_lbls_flag** (*bool*) – If True the routine will print the special points found by AUTO 2000 and False will not print any special points.
- **auto_parameters** (*dict*) – Dictionary defining the parameters for the AUTO 2000 run. Please note that one should **not** set 'SBML' or 'ScanDirection' in these parameters as these are automatically assigned. It is absolutely necessary to set PrincipalContinuationParameter in this dictionary. For more information on these parameters refer to AUTO parameters. 'NMX' will default to 10000 and 'ITMX' to 100.

Returns **multistable_param_ind** – A list of those indices in 'parameters' that produce multistable plots.

Return type list of integers

Example

See *Mass Conservation Approach Example* and *Mass Conservation Approach Walkthrough*.

run_greedy_continuity_analysis(*species=None*, *parameters=None*, *dir_path='./num_cont_graphs'*, *print_lbls_flag=False*, *auto_parameters=None*)

Function for running the greedy numerical continuation and bistability analysis portions of the mass conservation approach. This routine uses the initial value of the principal continuation parameter to construct AUTO parameters and then tests varying fixed step sizes for the continuation problem.

Note that this routine may produce jagged or missing sections in the plots provided. To produce better plots one should use the information provided by this routine to run *crnt4sbml_test.MassConservationApproach.run_continuity_analysis()*.

Parameters

- **species** (*string*) – A string stating the species that is the y-axis of the bifurcation diagram.
- **parameters** (*list of numpy arrays*) – A list of numpy arrays corresponding to the decision vectors that produce a small objective function value.
- **dir_path** (*string*) – A string stating the path where the bifurcation diagrams should be saved.
- **print_lbls_flag** (*bool*) – If True the routine will print the special points found by AUTO 2000 and False will not print any special points.
- **auto_parameters** (*dict*) – Dictionary defining the parameters for the AUTO 2000 run. Please note that only the PrincipalContinuationParameter in this dictionary should be defined, no other AUTO parameters should be set. For more information on these parameters refer to AUTO parameters.

Returns **multistable_param_ind** – A list of those indices in 'parameters' that produce multistable plots.

Return type list of integers

Example

See *Mass Conservation Approach Walkthrough*.

run_optimization(*bounds=None*, *iterations=10*, *sys_min_val=2.220446049250313e-16*, *seed=0*, *print_flag=False*, *numpy_dtype=<class 'numpy.float64'>*, *concentration_bounds=None*)

Function for running the optimization problem for the mass conservation approach.

Parameters

- **bounds** (*list of tuples*) – A list defining the lower and upper bounds for each variable in the decision vector. Here the reactions are allowed to be set to a single value.
- **iterations** (*int*) – The number of iterations to run the feasible point method.
- **sys_min_val** (*float*) – The value that should be considered zero for the optimization problem.
- **seed** (*int*) – Seed for the random number generator. None should be used if a random generation is desired.
- **print_flag** (*bool*) – Should be set to True if the user wants the objective function values found in the optimization problem and False otherwise.
- **numpy_dtype** – The numpy data type used within the optimization routine. All variables in the optimization routine will be converted to this data type.
- **concentration_bounds** (*list of tuples*) – A list defining the lower and upper bounds for those species' concentrations not in the decision vector. The user is not allowed to set the species' concentration to a single value. See also: *crnt4sbml_test.MassConservationApproach.get_concentration_bounds_species()*.

Returns

- **params_for_global_min** (*list of numpy arrays*) – A list of numpy arrays that correspond to the decision vectors of the problem.
- **obj_fun_val_for_params** (*list of floats*) – A list of objective function values produced by the corresponding decision vectors in params_for_global_min.

Examples

See *Mass Conservation Approach Example* and *Mass Conservation Approach Walkthrough*.

13.5 crnt4sbml_test.SemiDiffusiveApproach

class crnt4sbml_test.**SemiDiffusiveApproach**(*cgraph*)

Class for constructing variables and methods needed for the semi-diffusive approach.

__init__(*cgraph*)

Initialization of the SemiDiffusiveApproach class.

See also:

crnt4sbml_test.CRNT.get_semi_diffusive_approach()

Methods

__init__(cgraph)	Initialization of the SemiDiffusiveApproach class.
generate_report()	Prints out helpful details constructed by *crnt4sbml_test.SemiDiffusiveApproach.run_optimization()* and *crnt4sbml_test.SemiDiffusiveApproach.run_continuity_analysis()*.
get_key_species()	Returns a list of string variables corresponding to the key species.
get_non_key_species()	Returns a list of string variables corresponding to those species that are not key species.
get_boundary_species()	Returns a list of string variables corresponding to those species that are defined as boundary species.
get_decision_vector()	Returns a list of SymPy variables corresponding to the decision vector for the optimization problem.
print_decision_vector()	Prints an easily readable form of the decision vector.
get_mu_vector()	Returns a list of SymPy variables corresponding to the vector of fluxes, μ .
get_s_to_matrix()	Returns SymPy matrix representing the S_{to} matrix.
get_y_r_matrix()	Returns SymPy matrix representing the Y_r matrix.
get_symbolic_polynomial_fun()	Returns SymPy matrix representing the vector of polynomial functions, $-S_{to}\mu$.
get_lambda_polynomial_fun()	Returns a list of lambda functions for the vector of polynomial functions.
get_symbolic_objective_fun()	Returns SymPy expression for the objective function of the optimization problem.

Continued on next page

Table 6 – continued from previous page

get_lambda_objective_fun()	Returns a lambda function representation of the objective function of the optimization problem.
run_optimization([bounds, iterations, ...])	Function for running the optimization problem for the semi-diffusive approach.
run_continuity_analysis([species, ...])	Function for running the numerical continuation and bistability analysis portions of the semi-diffusive approach.
run_greedy_continuity_analysis([species, ...])	Function for running the greedy numerical continuation and bistability analysis portions of the semi-diffusive approach.

generate_report()
Prints out helpful details constructed by *crnt4sbml_test.SemiDiffusiveApproach.run_optimization()* and *crnt4sbml_test.SemiDiffusiveApproach.run_continuity_analysis()*.

Example

See *Semi-diffusive Approach Example* and *Semi-diffusive Approach Walkthrough*.

get_boundary_species()
Returns a list of string variables corresponding to those species that are defined as boundary species. Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> print(approach.get_boundary_species())
    ['s21']
```

get_decision_vector()
Returns a list of SymPy variables corresponding to the decision vector for the optimization problem. Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> print(approach.get_decision_vector())
    [v_2, v_3, v_4, v_5, v_6, v_7, v_9, v_11, v_13, v_15, v_17, v_18]
```

See also:

crnt4sbml_test.SemiDiffusiveApproach.print_decision_vector()

get_key_species()
Returns a list of string variables corresponding to the key species. Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> print(approach.get_key_species())
    ['s1', 's2', 's7']
```

get_lambda_objective_fun()

Returns a lambda function representation of the objective function of the optimization problem. Here the arguments of the lambda function are given by the values provided by *crnt4sbml_test.SemiDiffusiveApproach.get_mu_vector()*.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> approach.get_lambda_objective_fun()
```

get_lambda_polynomial_fun()

Returns a list of lambda functions for the vector of polynomial functions. The index of the list corresponds to the row in the vector of polynomial functions. Here the arguments of the lambda function are given by the values provided by *crnt4sbml_test.SemiDiffusiveApproach.get_mu_vector()*.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> approach.get_lambda_polynomial_fun()
```

get_mu_vector()

Returns a list of SymPy variables corresponding to the vector of fluxes, μ . Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> print(approach.get_mu_vector())
    [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_11, v_13, v_15, v_16, v_
↪17, v_18, v_19]
```

get_non_key_species()

Returns a list of string variables corresponding to those species that are not key species. Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> print(approach.get_non_key_species())
    ['s3', 's6', 's8', 's11']
```

get_s_to_matrix()

Returns SymPy matrix representing the S_{to} matrix. The columns of which correspond to the true and outflow reactions of the stoichiometric matrix. Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> sympy.pprint(approach.get_s_to_matrix())
    -1  1   0   0   -1  1   0   1   0   -1  0   0   0   0   0   0

    -1  1   0   0   0   0   1   0   0   0   -1  0   0   0   0   0

    1   -1  0   0   0   0   -1  0   0   0   0   0   0   -1  0   0

    0   0   -1  1   -1  1   1   0   2   0   0   0   -1  0   0   0

    0   0   -1  1   0   0   0   1   0   0   0   -1  0   0   0   0

    0   0   1   -1  0   0   0   -1  0   0   0   0   0   0   -1  0

    0   0   0   0   1   -1  0   0   -1  0   0   0   0   0   0   -1
```

get_symbolic_objective_fun()

Returns SymPy expression for the objective function of the optimization problem. This is the determinant of $S_{to}diag(\mu)Y_r^T$ squared.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> approach.get_symbolic_objective_fun()
```

get_symbolic_polynomial_fun()

Returns SymPy matrix representing the vector of polynomial functions, $-S_{to}\mu$. Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> sympy.pprint(approach.get_symbolic_polynomial_fun())
       v₁ + v₁₁ - v₂ + v₅ - v₆ - v₈

             v₁ + v₁₃ - v₂ - v₇

             -v₁ + v₁₇ + v₂ + v₇

    v₁₆ + v₃ - v₄ + v₅ - v₆ - v₇ - 2v₉

             v₁₅ + v₃ - v₄ - v₈

             v₁₈ - v₃ + v₄ + v₈

             v₁₉ - v₅ + v₆ + v₉
```

get_y_r_matrix()

Returns SymPy matrix representing the Y_r matrix. The columns of which correspond to the true and outflow reactions of the molecularity matrix. Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> sympy.pprint(approach.get_y_r_matrix())
    1  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0

    1  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0

    0  1  0  0  0  0  1  0  0  0  0  0  0  1  0  0

    0  0  1  0  1  0  0  0  0  0  0  0  1  0  0  0

    0  0  1  0  0  0  0  0  0  0  0  1  0  0  0  0

    0  0  0  1  0  0  0  1  0  0  0  0  0  0  1  0

    0  0  0  0  0  1  0  0  1  0  0  0  0  0  0  1
```

print_decision_vector()

Prints an easily readable form of the decision vector. It first prints the decision vector and then the corresponding reaction labels. Fig1Cii.xml for the provided example.

Example

```
>>> import crnt4sbml
>>> import sympy
```

(continues on next page)

(continued from previous page)

```
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> approach.print_decision_vector()
    Decision vector for optimization:
    [v_2, v_3, v_4, v_5, v_6, v_7, v_9, v_11, v_13, v_15, v_17, v_18]
    Reaction labels for decision vector:
    ['re1r', 're3', 're3r', 're6', 're6r', 're2', 're8', 're17r', 're18r',
↪'re19r', 're21', 're22']
```

run_continuity_analysis(*species=None*, *parameters=None*, *dir_path='./num_cont_graphs'*, *print_lbls_flag=False*, *auto_parameters=None*)

Function for running the numerical continuation and bistability analysis portions of the semi-diffusive approach.

Parameters

- **species** (*string*) – A string stating the species that is the y-axis of the bifurcation diagram.
- **parameters** (*list of numpy arrays*) – A list of numpy arrays corresponding to the decision vectors that produce a small objective function value.
- **dir_path** (*string*) – A string stating the path where the bifurcation diagrams should be saved.
- **print_lbls_flag** (*bool*) – If True the routine will print the special points found by AUTO 2000 and False will not print any special points.
- **auto_parameters** (*dict*) – Dictionary defining the parameters for the AUTO 2000 run. Please note that one should **not** set 'SBML' or 'ScanDirection' in these parameters as these are automatically assigned. It is absolutely necessary to set PrincipalContinuationParameter in this dictionary. For more information on these parameters refer to AUTO parameters. 'NMX' will default to 10000 and 'ITMX' to 100.

Returns **multistable_param_ind** – A list of those indices in 'parameters' that produce multistable plots.

Return type list of integers

Example

See *Semi-diffusive Approach Example* and *Semi-diffusive Approach Walkthrough*.

run_greedy_continuity_analysis(*species=None*, *parameters=None*, *dir_path='./num_cont_graphs'*, *print_lbls_flag=False*, *auto_parameters=None*)

Function for running the greedy numerical continuation and bistability analysis portions of the semi-diffusive approach. This routine uses the initial value of the principal continuation parameter to construct AUTO parameters and then tests varying fixed step sizes for the continuation problem. Note that this routine may produce jagged or missing sections in the plots provided. To produce better plots one should use the information provided by this routine to run *crnt4sbml_test.SemiDiffusiveApproach.run_continuity_analysis()*.

Parameters

- **species** (*string*) – A string stating the species that is the y-axis of the bifurcation diagram.

- **parameters** (*list of numpy arrays*) – A list of numpy arrays corresponding to the decision vectors that produce a small objective function value.
- **dir_path** (*string*) – A string stating the path where the bifurcation diagrams should be saved.
- **print_lbls_flag** (*bool*) – If True the routine will print the special points found by AUTO 2000 and False will not print any special points.
- **auto_parameters** (*dict*) – Dictionary defining the parameters for the AUTO 2000 run. Please note that only the PrincipalContinuationParameter in this dictionary should be defined, no other AUTO parameters should be set. For more information on these parameters refer to AUTO parameters.

Returns **multistable_param_ind** – A list of those indices in 'parameters' that produce multistable plots.

Return type list of integers

Example

See *Semi-diffusive Approach Walkthrough*.

run_optimization(*bounds=None*, *iterations=10*, *sys_min_val=2.220446049250313e-16*, *seed=0*, *print_flag=False*, *numpy_dtype=<class 'numpy.float64'>*)

Function for running the optimization problem for the semi-diffusive approach. Note that there are no bounds enforced on species' concentrations as they are automatically restricted to be greater than zero by the theory.

Parameters

- **bounds** (*list of tuples*) – A list defining the lower and upper bounds for each variable in the decision vector. Here the reactions are allowed to be set to a single value.
- **iterations** (*int*) – The number of iterations to run the feasible point method.
- **sys_min_val** (*float*) – The value that should be considered zero for the optimization problem.
- **seed** (*int*) – Seed for the random number generator. None should be used if a random generation is desired.
- **print_flag** (*bool*) – Should be set to True if the user wants the objective function values found in the optimization problem and False otherwise.
- **numpy_dtype** – The numpy data type used within the optimization routine. All variables in the optimization routine will be converted to this data type.

Returns

- **params_for_global_min** (*list of numpy arrays*) – A list of numpy arrays that correspond to the decision vectors of the problem.
- **obj_fun_val_for_params** (*list of floats*) – A list of objective function values produced by the corresponding decision vectors in params_for_global_min.

Examples

See *Semi-diffusive Approach Example* and *Semi-diffusive Approach Walkthrough*.

CHAPTER 14

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

14.1 Types of Contributions

14.1.1 Report Bugs

Report bugs at https://github.com/breye12/crnt4sbml_test/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

14.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

14.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

14.1.4 Write Documentation

CRNT4SBML_Test could always use more documentation, whether as part of the official CRNT4SBML_Test docs, in docstrings, or even on the web in blog posts, articles, and such.

14.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/breye12/crnt4sbml_test/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

14.2 Get Started!

Ready to contribute? Here's how to set up *crnt4sbml_test* for local development.

1. Fork the *crnt4sbml_test* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/crnt4sbml_test.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv crnt4sbml_test
$ cd crnt4sbml_test/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 crnt4sbml_test tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

14.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/breye12/crnt4sbml_test/pull_requests and make sure that the tests pass for all supported Python versions.

14.4 Tips

To run a subset of tests:

```
$ py.test tests.test_crnt4sbml_test
```

14.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 15

Credits

15.1 Development Lead

- Brandon C Reyes <brandon@example.com>

15.2 Contributors

None yet. Why not be the first?

CHAPTER 16

History

16.1 0.0.1 (2019-06-18)

- First release on PyPI.

16.2 0.0.2 (2019-06-26)

- Added dictionary input to continuation.
- Added a new routine to create stability graphs.

16.3 0.0.3 (2019-06-29)

- switched to networkX for graph theory

16.4 0.0.4 (2019-07-12)

- Added changes to Equilibrium Manifold creation
- Added new way to plot stability directly from AUTO

16.5 0.0.5 (2019-07-12)

- Changed python version requirement

16.6 0.0.6 (2019-07-16)

- Changed class names and corresponding file names

16.7 0.0.7 (2019-07-19)

- Updated all code to satisfy PEP 8 standards

16.8 0.0.8 (2019-07-23)

- Updated Cgraph documentation

16.9 0.0.9 (2019-07-23)

- Adding strict requirment version to libroadrunner

16.10 0.0.10 (2019-07-23)

- Reverting back to the newest version of libroadrunner

16.11 0.0.11 (2019-07-24)

- Allowing for the use of Python version 3.6 and above

16.12 0.0.12 (2019-07-27)

- Changes in write_graphml and in bistability finder class

16.13 0.0.13 (2019-07-29)

- Changes in bistability_finder to account for unions in stability analysis

16.14 0.0.14 (2019-08-5)

- Added greedy numerical continuation routine

16.15 0.0.15 (2019-08-6)

- Changes to greedy numerical continuation routine

16.16 0.0.16 (2019-08-6)

- Changes to get_decision_vector

16.17 0.0.17 (2019-08-7)

- Took out deficiency parameters in linear check for equilibrium manifold to reduce runtime.

16.18 0.0.18 (2019-08-7)

- Took out requirement of inputting concentration bounds twice.

16.19 0.0.19 (2019-08-22)

- Added testpypi to requirements file

CHAPTER 17

Bibliography

Bibliography

[Chi14] John W. Chinneck. *Practical Optimization: a Gentle Introduction*. 2014.

[CSRGR05] C. Conradi, J. Saez-Rodriguez, E. Gilles, and J. Raisch. Using chemical reaction network theory to discard a kinetic mechanism hypothesis. *IEE Proceedings - Systems Biology*, 152(4):243–248, Dec 2005. doi:10.1049/ip-syb:20050045.

[CFRS07] Carsten Conradi, Dietrich Flockerzi, Jörg Raisch, and Jörg Stelling. Subnetwork analysis reveals dynamic features of complex (bio)chemical networks. *Proceedings of the National Academy of Sciences*, 104(49):19175–19180, 2007. URL: https://www.pnas.org/content/104/49/19175, arXiv:https://www.pnas.org/content/104/49/19175.full.pdf, doi:10.1073/pnas.0705731104.

[Fei79] Martin Feinberg. Lectures on chemical reaction networks. notes of lectures given at the mathematics research center, university of wisconsin. *https://crnt.osu.edu/LecturesOnReactionNetworks*, 1979.

[HC87] Jean-François Hervagault and Stéphane Canu. Bistability and irreversible transitions in a simple substrate cycle. *Journal of Theoretical Biology*, 127(4):439 – 449, 1987. URL: http://www.sciencedirect.com/science/article/pii/S0022519387801418, doi:https://doi.org/10.1016/S0022-5193(87)80141-8.

[OMBA09] Irene Otero-Muras, Julio R. Banga, and Antonio A. Alonso. Exploring multiplicity conditions in enzymatic reaction networks. *Biotechnology Progress*, 25(3):619–631, 2009. URL: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/btpr.112, arXiv:https://aiche.onlinelibrary.wiley.com/doi/pdf/10.1002/btpr.112, doi:10.1002/btpr.112.

[OMYS14] Irene Otero-Muras, Pencho Yordanov, and Joerg Stelling. A method for inverse bifurcation of biochemical switches: inferring parameters from dose response curves. *BMC Systems Biology*, 8(1):114, 2014. URL: https://doi.org/10.1186/s12918-014-0114-2, doi:10.1186/s12918-014-0114-2.

[OMYS17] Irene Otero-Muras, Pencho Yordanov, and Joerg Stelling. Chemical reaction network theory elucidates sources of multistability in interferon signaling. *PLos computational biology*, 2017.

[SBG+15] Endre T. Somogyi, Jean-Marie Bouteiller, James A. Glazier, Matthias König, J. Kyle Medley, Maciej H. Swat, and Herbert M. Sauro. libRoadRunner: a high performance SBML simulation and analysis library. *Bioinformatics*, 31(20):3315–3321, 06 2015. URL: https://doi.org/10.1093/bioinformatics/btv363, arXiv:http://oup.prod.sis.lan/bioinformatics/article-pdf/31/20/3315/17087875/btv363.pdf, doi:10.1093/bioinformatics/btv363.

Index

Symbols

__init__() (*crnt4sbml_test.CRNT method*), 73
__init__() (*crnt4sbml_test.Cgraph method*), 77
__init__() (*crnt4sbml_test.LowDeficiencyApproach method*), 85
__init__() (*crnt4sbml_test.MassConservationApproach method*), 87
__init__() (*crnt4sbml_test.SemiDiffusiveApproach method*), 98

B

basic_report() (*crnt4sbml_test.CRNT method*), 74

C

Cgraph (*class in crnt4sbml_test*), 77
CRNT (*class in crnt4sbml_test*), 73

D

does_satisfy_any_low_deficiency_theorem() (*crnt4sbml_test.LowDeficiencyApproach method*), 85
does_satisfy_deficiency_one_theorem() (*crnt4sbml_test.LowDeficiencyApproach method*), 85
does_satisfy_deficiency_zero_theorem() (*crnt4sbml_test.LowDeficiencyApproach method*), 86

G

generate_report() (*crnt4sbml_test.MassConservationApproach method*), 88
generate_report() (*crnt4sbml_test.SemiDiffusiveApproach method*), 99
get_a() (*crnt4sbml_test.Cgraph method*), 78
get_advanced_deficiency_approach() (*crnt4sbml_test.CRNT method*), 74
get_b() (*crnt4sbml_test.Cgraph method*), 78
get_boundary_species() (*crnt4sbml_test.SemiDiffusiveApproach method*), 99
get_c_graph() (*crnt4sbml_test.CRNT method*), 74
get_complexes() (*crnt4sbml_test.Cgraph method*), 79
get_concentration_bounds_species() (*crnt4sbml_test.MassConservationApproach method*), 88
get_concentration_funs() (*crnt4sbml_test.MassConservationApproach method*), 89
get_concentration_solutions() (*crnt4sbml_test.MassConservationApproach method*), 89
get_concentration_vals() (*crnt4sbml_test.MassConservationApproach method*), 90
get_conservation_laws() (*crnt4sbml_test.MassConservationApproach method*), 90
get_dch_matrix() (*crnt4sbml_test.MassConservationApproach method*), 90
get_decision_vector() (*crnt4sbml_test.MassConservationApproach method*), 91
get_decision_vector() (*crnt4sbml_test.SemiDiffusiveApproach method*), 99
get_deficiency() (*crnt4sbml_test.Cgraph method*), 79
get_dim_equilibrium_manifold() (*crnt4sbml_test.Cgraph method*), 79
get_g_edges() (*crnt4sbml_test.Cgraph method*), 79
get_g_matrix() (*crnt4sbml_test.MassConservationApproach method*), 91
get_g_nodes() (*crnt4sbml_test.Cgraph method*), 80
get_graph() (*crnt4sbml_test.Cgraph method*), 80
get_h_vector() (*crnt4sbml_test.MassConservationApproach method*), 92

get_if_cgraph_weakly_reversible() (*crnt4sbml_test.Cgraph method*), 80
get_key_species() (*crnt4sbml_test.SemiDiffusiveApproach method*), 99
get_lambda() (*crnt4sbml_test.Cgraph method*), 80
get_lambda_dch_matrix() (*crnt4sbml_test.MassConservationApproach method*), 93
get_lambda_g_matrix() (*crnt4sbml_test.MassConservationApproach method*), 93
get_lambda_objective_fun() (*crnt4sbml_test.MassConservationApproach method*), 93
get_lambda_objective_fun() (*crnt4sbml_test.SemiDiffusiveApproach method*), 100
get_lambda_polynomial_fun() (*crnt4sbml_test.SemiDiffusiveApproach method*), 100
get_linkage_classes() (*crnt4sbml_test.Cgraph method*), 81
get_linkage_classes_deficiencies() (*crnt4sbml_test.Cgraph method*), 81
get_low_deficiency_approach() (*crnt4sbml_test.CRNT method*), 75
get_mass_conservation_approach() (*crnt4sbml_test.CRNT method*), 75
get_mu_vector() (*crnt4sbml_test.SemiDiffusiveApproach method*), 100
get_network_dimensionality_classification() (*crnt4sbml_test.Cgraph method*), 81
get_network_graphml() (*crnt4sbml_test.CRNT method*), 75
get_non_key_species() (*crnt4sbml_test.SemiDiffusiveApproach method*), 100
get_number_of_terminal_strong_lc_per_lc() (*crnt4sbml_test.Cgraph method*), 81
get_objective_fun_params() (*crnt4sbml_test.MassConservationApproach method*), 94
get_ode_system() (*crnt4sbml_test.Cgraph method*), 82
get_physiological_range() (*crnt4sbml_test.CRNT static method*), 75
get_psi() (*crnt4sbml_test.Cgraph method*), 82
get_reactions() (*crnt4sbml_test.Cgraph method*), 83
get_s() (*crnt4sbml_test.Cgraph method*), 83
get_s_to_matrix() (*crnt4sbml_test.SemiDiffusiveApproach method*), 101
get_semi_diffusive_approach() (*crnt4sbml_test.CRNT method*), 76
get_species() (*crnt4sbml_test.Cgraph method*), 83
get_symbolic_objective_fun() (*crnt4sbml_test.MassConservationApproach method*), 94
get_symbolic_objective_fun() (*crnt4sbml_test.SemiDiffusiveApproach method*), 101
get_symbolic_polynomial_fun() (*crnt4sbml_test.SemiDiffusiveApproach method*), 101
get_w_matrix() (*crnt4sbml_test.MassConservationApproach method*), 95
get_w_nullspace() (*crnt4sbml_test.MassConservationApproach method*), 95
get_weak_reversibility_of_linkage_classes() (*crnt4sbml_test.Cgraph method*), 84
get_y() (*crnt4sbml_test.Cgraph method*), 84
get_y_r_matrix() (*crnt4sbml_test.SemiDiffusiveApproach method*), 102

L

LowDeficiencyApproach (*class in crnt4sbml_test*), 85

M

MassConservationApproach (*class in crnt4sbml_test*), 87

P

plot() (*crnt4sbml_test.Cgraph method*), 84
plot_c_graph() (*crnt4sbml_test.CRNT method*), 76
plot_save() (*crnt4sbml_test.Cgraph method*), 84
plot_save_c_graph() (*crnt4sbml_test.CRNT method*), 76
print() (*crnt4sbml_test.Cgraph method*), 84
print_c_graph() (*crnt4sbml_test.CRNT method*), 76
print_decision_vector() (*crnt4sbml_test.SemiDiffusiveApproach method*), 102

R

report_deficiency_one_theorem() (*crnt4sbml_test.LowDeficiencyApproach method*), 86
report_deficiency_zero_theorem() (*crnt4sbml_test.LowDeficiencyApproach method*), 86
run_continuity_analysis() (*crnt4sbml_test.MassConservationApproach method*), 96

run_continuity_analysis() (*crnt4sbml_test.SemiDiffusiveApproach method*), 103
run_greedy_continuity_analysis() (*crnt4sbml_test.MassConservationApproach method*), 96
run_greedy_continuity_analysis() (*crnt4sbml_test.SemiDiffusiveApproach method*), 103
run_optimization() (*crnt4sbml_test.MassConservationApproach method*), 97
run_optimization() (*crnt4sbml_test.SemiDiffusiveApproach method*), 104

S

SemiDiffusiveApproach (*class in crnt4sbml_test*), 98