CRNT4SBML Documentation

Release 0.0.15

Brandon Reyes

Aug 25, 2021

Contents:

CHAPTER 1

CRNT4SBML

CRNT4SBML is an easily installable Python based package available on MacOS and Windows. 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.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 Citing CRNT4SBML

If you use CRNT4SBML in your research, we would appreciate it if you use the following citation in any works you publish:

> Brandon C Reyes, Irene Otero-Muras, Michael T Shuen, Alexandre M Tartakovsky, Vladislav A Petyuk, CRNT4SBML: a Python package for the detection of bistability in biochemical reaction networks, Bioinformatics.

1.3 Credits

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

CHAPTER 2

Installation

2.1 Base requirements

- Python 3.7 (64-bit)
- networkx==2.3
- python-libsbml==5.18.0
- numpy==1.16.4
- sympy==1.4
- scipy==1.4.1
- matplotlib==3.1.1
- plotnine==0.6.0

2.1.1 MacOS and Windows

- antimony==2.11.0
- rrplugins==1.2.2
- libroadrunner==1.5.2.1

2.2 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 64-bit version of Python 3.7 (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 3.7, we can create the virtual environment crnt4sbml_env in the directory python_environments as follows:

```
$ cd python_environments
$ virtualenv -p /path/to/python/3.7/interpreter crnt4sbml_env
```

The flag "-p" tells virtualenv to create an environment using a specific Python interpreter. If a standard download of Python was followed, then "/path/to/python/3.7/interpreter" can be replaced with "/usr/local/bin/python3.7" on MacOS and Linux, and "C:\Users\your_user_name\AppData\Local \Programs\Python\Python37\python.exe" on Windows. One can now see a directory called "crnt4sbml_env" is created in the directory python_environments.

We can now activate this environment as follows:

On MacOS and Linux:

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

On Windows:

```
$ path\to\crnt4sbml_env\Scripts\activate
```

Note, in case you are using PowerShell, make sure its policy is updated by executing command as administrator `Set-ExecutionPolicy RemoteSigned`. 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.

2.3 Stable Release

Once the environment is activated, one can now install CRNT4SBML as follows:

On MacOS:

```
$ pip install crnt4sbml[MacOS]
```

On Windows:

```
$ pip install crnt4sbml[Windows]
```

On Linux (numerical continuation is unavailable for Linux):

```
$ pip install crnt4sbml[Linux]
```

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.

2.4 Working Version

The current working version of crnt4sbml can be downloaded from the Github repo.

Once the environment is activated, one can now install CRNT4SBML as follows:

On MacOS:

```
$ pip install git+https://github.com/PNNL-Comp-Mass-Spec/CRNT4SBML.git
↪#egg=crnt4sbml[MacOS]
```

On Windows:

```
$ pip install git+https://github.com/PNNL-Comp-Mass-Spec/CRNT4SBML.git
↪#egg=crnt4sbml[Windows]
```

On Linux (numerical continuation is unavailable for Linux):

```
$ pip install git+https://github.com/PNNL-Comp-Mass-Spec/CRNT4SBML.git
↪#egg=crnt4sbml[Linux]
```

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.

CHAPTER 3

Steps for Detecting Bistability

The following are some simple steps to follow for detecting bistability using CRNT4SBML:

1. Construct an SBML file following the guidelines provided in *CellDesigner Walkthrough*.
2. Check if the conditions for the Deficiency Zero or One Theorems are satisfied using the approach outlined in *Low Deficiency Approach*.
3. If the Deficiency Zero or One Theorems are not satisfied, then use `crnt4sbml.Cgraph.get_dim_equilibrium_manifold()` to find λ, the number of mass conservation relationships.
4. If λ is greater than zero one can use the details described in *Mass Conservation Approach Walkthrough* to conduct further analysis of a uniterminal network.
5. If λ is greater than zero one can use the details described in *General Approach Walkthrough* to conduct further analysis of any network.
6. If λ is zero and there is a boundary species present in the SBML file then one can use the details described in *Semi-diffusive Approach Walkthrough* to conduct further analysis of the network.

CHAPTER 4

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] and a general approach for mass conserving systems.

- If you are interested in running the Deficiency Zero and One theorems please consult *Low Deficiency Approach*.
- If one is interested in the general steps to follow in order to detect bistability, one should consult *Steps for Detecting Bistability*.

4.1 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

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

approach = network.get_mass_conservation_approach()

bounds, concentration_bounds = approach.get_optimization_bounds()

params_for_global_min, obj_fun_val_for_params = approach.run_
↪optimization(bounds=bounds,
                                                                                  ␣
↪concentration_bounds=concentration_bounds)

multistable_param_ind, plot_specifications = approach.run_greedy_continuity_
↪analysis(species="s15", parameters=params_for_global_min,
```

(continues on next page)

(continued from previous page)

```
                                                                                ␣
↪auto_parameters={'PrincipalContinuationParameter': 'C3'})

approach.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: 2.0428380000000006

Running feasible point method for 10 iterations ...
Elapsed time for feasible point method: 1.5746338367462158

Running the multistart optimization method ...
Elapsed time for multistart method: 7.010828971862793

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 25.22320318222046

Smallest value achieved by objective function: 0.0
4 point(s) passed the optimization criteria.
Number of multistability plots found: 2
Elements in params_for_global_min that produce multistability:
[0, 1]
```

4.2 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

network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")

approach = network.get_semi_diffusive_approach()

bounds = approach.get_optimization_bounds()

params_for_global_min, obj_fun_val_for_params = approach.run_
↪optimization(bounds=bounds)

multistable_param_ind, plot_specifications = approach.run_greedy_continuity_
↪analysis(species="s7", parameters=params_for_global_min,
                                                                                ␣
↪auto_parameters={'PrincipalContinuationParameter': 're17'})

approach.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: 0.3393716812133789

Running the multistart optimization method ...
Elapsed time for multistart method: 22.361775875091553

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 73.85193490982056

Smallest value achieved by objective function: 0.0
9 point(s) passed the optimization criteria.
Number of multistability plots found: 9
Elements in params_for_global_min that produce multistability:
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

4.3 General Approach Example

In order to run the general 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 general approach with fixed reactions create the following python script:

```
import crnt4sbml

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

approach = network.get_general_approach()
bnds = approach.get_optimization_bounds()

approach.initialize_general_approach(signal="C3", response="s15", fix_reactions=True)

params_for_global_min, obj_fun_vals = approach.run_optimization(bounds=bnds, dual_
↪annealing_iters=100)

multistable_param_ind, plot_specifications = approach.run_greedy_continuity_
↪analysis(species="s15", parameters=params_for_global_min,
                                                                                         ␣
↪auto_parameters={'PrincipalContinuationParameter': "C3"})

approach.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 *General Approach Walkthrough* for a more detailed explanation of running the general approach and the provided output.

```
Running the multistart optimization method ...
Elapsed time for multistart method: 21.040880918502808
```

(continues on next page)

(continued from previous page)

```

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 41.21180701255798

Smallest value achieved by objective function: 0.0
9 point(s) passed the optimization criteria.
Number of multistability plots found: 6
Elements in params_for_global_min that produce multistability:
[1, 2, 4, 5, 7, 8]
```

CHAPTER 5

Parallel CRNT4SBML

Due to the nature of the optimization problem formed, some models can take a long time to complete. In order to improve the user experience, we have developed parallel versions of the optimization routine for all approaches using mpi4py.

5.1 Installing the proper packages

5.1.1 Base Requirements for Parallel Version

- Python 3.7 (64-bit)
- networkx==2.3
- python-libsbml==5.18.0
- numpy==1.16.4
- sympy==1.4
- scipy==1.4.1
- matplotlib==3.1.1
- plotnine==0.6.0
- mpi4py==3.0.3

5.2 MacOS and Windows

- antimony==2.11.0
- rrplugins==1.2.2
- libroadrunner==1.5.2.1

5.2.1 Creating a Virtual Environment

The preferred way to use the parallel version of CRNT4SBML is through a virtual environment. First follow the steps outlined in *Installation* to create a virtual environment with the name mpi_crnt4sbml. Once this is done, we can now activate this environment as follows:

On MacOS and Linux:

```
$ source /path/to/python_environments/mpi_crnt4sbml/bin/activate
```

On Windows:

```
$ path\to\mpi_crnt4sbml\Scripts\activate
```

Note, in case you are using PowerShell, make sure its policy is updated by executing command as administrator `Set-ExecutionPolicy RemoteSigned`. On the command line one should now see "(mpi_crnt4sbml)" on the left side of the command line, which indicates that one is now working in the virtual environment.

One now needs to install mpi4py. Given mpi4py uses mpicc under the covers, we first need to install an MPI compiler onto our system. This is done differently on MacOS, Linux, and Windows.

On MacOS:

The simplest way to install mpicc on MacOS is to use homebrew. To begin, first install homebrew. Then, we need to install open-mpi. This is done in the terminal as follows:

```
$ brew install open-mpi
```

Be sure to take note of the install location of open-mpi. We now need to set the environment variable for the MPI compiler. This is done as follows in the terminal (take note that here we are using version 4.0.2 of open-mpi):

```
$ export MPICC=path/to/open-mpi/4.0.2/bin/mpicc
```

If a standard install was followed, "path/to/" can be replaced with "/usr/local/Cellar/". We are now ready to install mpi4py. With the virtual environment mpi_crnt4sbml activated, mpi4py can be installed as follows:

```
$ pip install mpi4py
```

On Linux:

The simplest way to install an MPI compiler on Linux is to install open-mpi. This is done in the terminal as follows (note that one may need to use sudo):

```
$ apt-get install -y libopenmpi-dev
```

On Windows:

The simplest way to install a proper MPI compiler on Windows is to use Microsoft MPI. If not already installed, one should download Microsoft MPI version 10 or newer. At the time of creating this documentation, this could be done using the following link. Using the link click download and download msmpisetup.exe and run it. After the download, one should have a proper MPI compiler that is compatible with mpi4py.

Note that for some users, one will also need to set the MSMPI path under User Variables. By default the Variable should be set to MSMPI_BIN and the Value should be `C:\Program Files\Microsoft MPI\Bin`. This can be done following the instructions here.

Once the environment is activated, one can now install a parallel CRNT4SBML as follows:

On MacOS:

```
$ pip install crnt4sbml[MPIMacOS]
```

On Windows:

```
$ pip install crnt4sbml[MPIWindows]
```

On Linux (numerical continuation is unavailable for Linux):

```
$ pip install crnt4sbml[MPILinux]
```

note that this will install crnt4sbml in the virtual environment mpi_crnt4sbml. One can only use crnt4sbml within this environment.

5.3 Parallel Mass Conservation Approach

To run the optimization for the mass conservation approach create the following python script named mpi_run.py:

```
import crnt4sbml
import numpy

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

approach = network.get_mass_conservation_approach()

bounds, concentration_bounds = approach.get_optimization_bounds()

params_for_global_min, obj_fun_val_for_params = approach.run_
↪optimization(bounds=bounds, concentration_bounds=concentration_bounds,
                                                                       parallel_
↪flag=True)

if approach.get_my_rank() == 0:
    numpy.save('params.npy', params_for_global_min)

approach.generate_report()
```

You can then run the script from the console using 2 cores using the following command:

```
$ mpiexec -np 2 python mpi_run.py
```

This will provide the following output along with saving the params_for_global_min to the file params.npy in the current directory. You can then load in params.npy and run a serial version of the numerical continuation. Please note that runtimes may vary among different operating systems.

```
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 2.06032
Elapsed time for creating Equilibrium Manifold: 2.0805279999999993

Running feasible point method for 10 iterations ...
Elapsed time for feasible point method: 1.024346

Running the multistart optimization method ...
```

(continues on next page)

(continued from previous page)

```
Elapsed time for multistart method: 3.5696950000000003

Smallest value achieved by objective function: 0.0
4 point(s) passed the optimization criteria.
```

5.4 Parallel Semi-diffusive Approach

To run the optimization for the semi-diffusive approach create the following python script named mpi_run.py:

```
import crnt4sbml
import numpy

network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")

approach = network.get_semi_diffusive_approach()

bounds = approach.get_optimization_bounds()

params_for_global_min, obj_fun_val_for_params = approach.run_
↪optimization(bounds=bounds, parallel_flag=True)

if approach.get_my_rank() == 0:
    numpy.save('params.npy', params_for_global_min)

approach.generate_report()
```

You can then run the script from the console using 2 cores using the following command:

```
$ mpiexec -np 2 python mpi_run.py
```

This will provide the following output along with saving the params_for_global_min to the file params.npy in the current directory. You can then load in params.npy and run a serial version of the numerical continuation. Please note that runtimes may vary among different operating systems.

```
Running feasible point method for 10 iterations ...
Elapsed time for feasible point method: 0.38841

Running the multistart optimization method ...
Elapsed time for multistart method: 17.330986000000003

Smallest value achieved by objective function: 0.0
9 point(s) passed the optimization criteria.
```

5.5 Parallel General Approach

5.5.1 Further libraries required

- plotnine==0.6.0

To run the optimization and direct simulation bistability anaylsis for the general approach create the following python script named mpi_run.py:

```
import crnt4sbml

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

approach = network.get_general_approach()

bnds = approach.get_optimization_bounds()

approach.initialize_general_approach(signal="C3", response="s15", fix_reactions=True)

params_for_global_min, obj_fun_vals = approach.run_optimization(bounds=bnds, dual_
↪annealing_iters=100, confidence_level_flag=True,
                                                                parallel_flag=True)

approach.run_direct_simulation(params_for_global_min, parallel_flag=True)

approach.generate_report()
```

You can then run the script from the console using 4 cores using the following command:

```
$ mpiexec -np 4 python mpi_run.py
```

This will provide the following output along with saving the direct simulation plots in the directory path ./dir_sim_graphs. Please note that runtimes may vary among different operating systems.

```
Running the multistart optimization method ...
Elapsed time for multistart method: 10.842817

Starting direct simulation ...
Elapsed time for direct simulation in seconds: 270.852905
It was found that 0.0 is the minimum objective function value with a confidence level␣
↪of 1.0 .
9 point(s) passed the optimization criteria.
```

CHAPTER 6

Docker and CRNT4SBML

To further the accessibility of CRNT4SBML, we have created a Dockerfile for CRNT4SBML. This allows one to use the full Linux version of CRNT4SBML. Docker is a software platform that uses OS-level virtualization to deliver software in packages called containers. Although there are many reasons to use Docker, our main use case will be to provide our users with a simple install of CRNT4SBML. To begin, first install Docker and then download `Dockerfile` into the directory of your choice.

Once you are in the directory where Dockerfile exists, one can create an image of CRNT4SBML named "crnt4sbml_image" by completing the following in a terminal:

```
$ docker build -t crnt4sbml_image .
```

Using this image, we can then create a basic container named "crnt4sbml_container" using the following command:

```
$ docker create --name crnt4sbml_container -t -i crnt4sbml_image /bin/bash
```

Alternatively, if one would like to mount the folders "sbml_files" and "example_scripts" of the host machine to the container upon creation one can do the following:

```
$ docker create --name crnt4sbml_container --mount type=bind,source=/path/to/sbml_
↪files,target=/home/crnt4sbml-user/sbml_files --mount type=bind,source=/host/path/to/
↪example_scripts,target=/home/crnt4sbml-user/example_scripts -t -i crnt4sbml_image /
↪bin/bash
```

This will allow the user to easily access and add both sbml files and python scripts between Docker and the host machine. To launch the container do the following:

```
$ docker start -i crnt4sbml_container
```

Now that we are in the container, we can run any of the Python scripts for CRNT4SBML that are available for Linux (in particular *`crnt4sbml.GeneralApproach()`* and *`crnt4sbml.MassConservationApproach()`* without numerical continuation).

Useful commands:

Show all containers:

```
$ docker ps -a
```

Show all images:

```
$ docker images -a
```

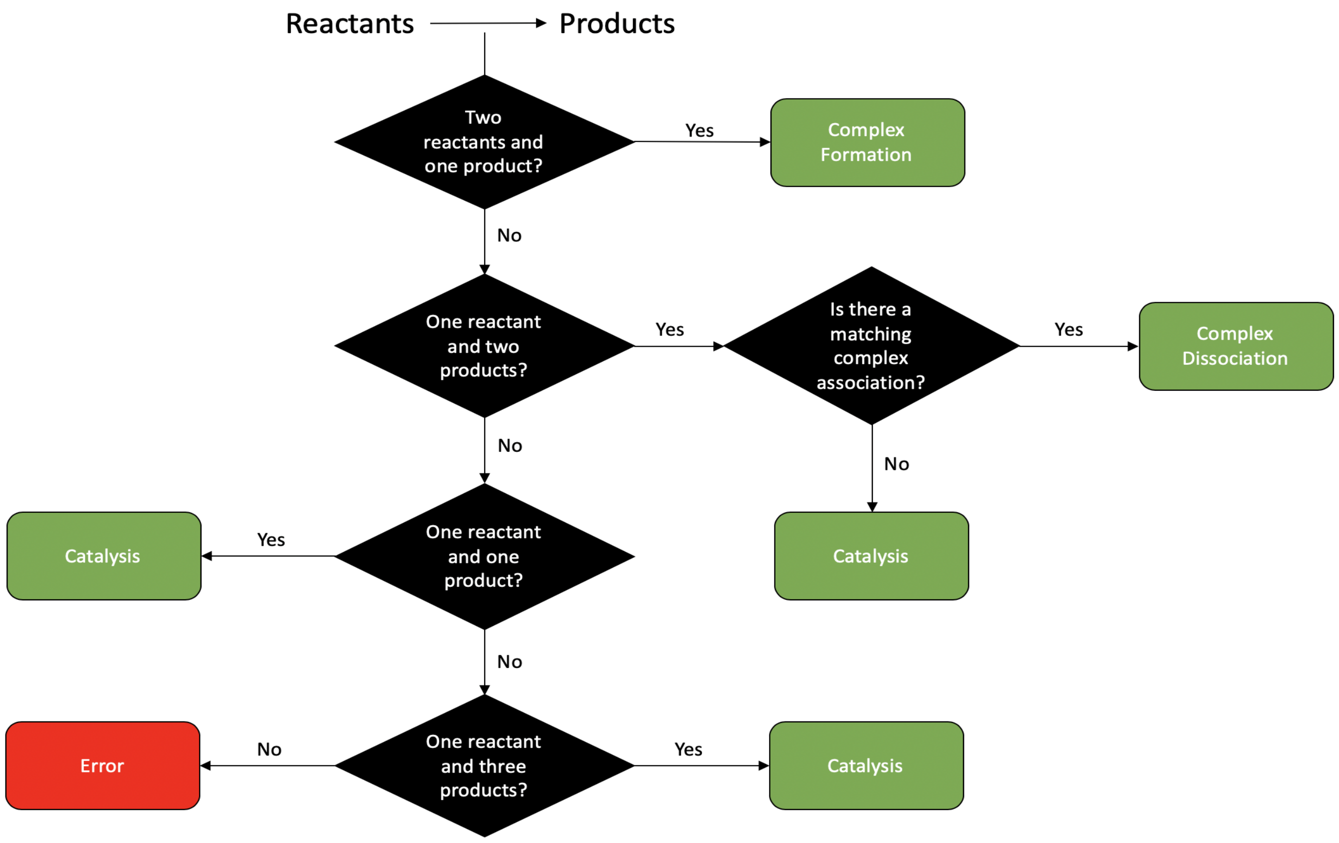
CHAPTER 7

Creating Physiological Bounds

To make crnt4sbml more user friendly and make its search limited to physiological problems, we have constructed the functions *crnt4sbml.MassConservationApproach.get_optimization_bounds()* and *crnt4sbml.SemiDiffusiveApproach.get_optimization_bounds()*, which constructs the appropriate bounds that must be provided to the mass conservation and semi-diffusive optimization routines, respectively. Although this feature can be extremely useful especially if the user is continually changing the SBML file, it should be used with some amount of caution.

7.1 Preprocessing

To provide these physiological bounds, crnt4sbml first identifies the reactions of the network. A reaction can be identified as complex formation, complex dissociation, or catalysis, no other type of reaction is considered. To make this assignment, the reactants and products of the reaction along with the associated stoichiometries are found for the particular reaction. Using the sum of the stoichiometries for the reactants and products the decision tree below is used to determine the type of reaction. If the reaction is not identified as complex formation, complex dissociation, or catalysis, then an error message will be provided and the reaction type will be specified as “None”.


Reactants
Products
Two reactants and one product?
Yes
Complex Formation
No
One reactant and two products?
Yes
Is there a matching complex association?
Yes
Complex Dissociation
No
Catalysis
No
One reactant and one product?
Yes
Catalysis
No
One reactant and three products?
No
Error
Yes
Catalysis

The type of reaction assigned by crnt4sbml can always be found by running the following script where we let `Fig1Ci.xml` be our SBML file

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

this provides the output below:

```
Reaction graph of the form
reaction -- reaction label -- biological reaction type:
s1+s2 -> s3  --  re1 -- complex formation
s3 -> s1+s2  --  re1r -- complex dissociation
s3 -> s6+s2  --  re2 -- catalysis
s6+s7 -> s16  --  re3 -- complex formation
s16 -> s6+s7  --  re3r -- complex dissociation
s16 -> s7+s1  --  re4 -- catalysis
s1+s6 -> s15  --  re5 -- complex formation
s15 -> s1+s6  --  re5r -- complex dissociation
s15 -> 2*s6  --  re6 -- catalysis
```

Creating the proper constraints for the optimization routine for the mass conservation approach differs from that of the semi-diffusive approach. This is because the mass conservation approach requires bounds for the rate constants and species' concentrations while the semi-diffusive approach only requires bounds for the fluxes of the reactions.

7.1.1 Mass Conservation Approach

To construct physiological bounds for the rate constants we first identify the type of the reaction and then we use the function *crnt4sbml.CRNT.get_physiological_range()*, which provides a tuple corresponding to the lower and upper bounds. The values for these bounds are in picomolar (pM). Here we assign pM values rather than molar values because these values are larger and tend to make running the optimization routine much easier. In molar ranges or values close to zero, the optimization becomes difficult because the routine is attempting to minimize an objective function which has a known value of zero. Thus, if the user wishes to assign different bounds, it is suggested that these bounds be scaled such that they are not close to zero.

We now demonstrate the physiological bounds produced for the SBML file `Fig1Ci.xml`

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

approach = network.get_mass_conservation_approach()

bounds, concentration_bounds = approach.get_optimization_bounds()

print(bounds)

print(concentration_bounds)
```

this provides the following output:

```
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 2.060944

[(1e-08, 0.0001), (1e-05, 0.001), (0.001, 1.0), (1e-08, 0.0001), (1e-05, 0.001), (0.
↪001, 1.0), (1e-08, 0.0001), (1e-05, 0.001), (0.001, 1.0), (0.5, 500000.0), (0.5,␣
↪500000.0), (0.5, 500000.0)]
[(0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0)]
```

Where the rate constants and species' concentrations for the list "bounds" can be found by the following command

```
print(approach.get_decision_vector())
```

providing the output:

```
[re1, re1r, re2, re3, re3r, re4, re5, re5r, re6, s2, s6, s15]
```

and the species' concentrations referred to in the list "concentration_bounds" can be determined by the following

```
print(approach.get_concentration_bounds_species())
```

giving the output:

```
[s1, s3, s7, s16]
```

7.1.2 Semi-diffusive Approach

As stated above, the semi-diffusive approach only requires bounds for the fluxes of the reactions. To assign these values, we again use the function *crnt4sbml.CRNT.get_physiological_range()*, which provides a tuple for the lower and upper bounds. However, the values returned by this call are given in molars. The unit of molars is suggested because the ranges produced for fluxes are much smaller than those for pM, making the optimization easier.

To demonstrate the bounds produced for the semi-diffusive approach, we use the SBML file `Fig1Cii.xml`.

```
import crnt4sbml
network = crnt4sbml.CRNT("/path/to/Fig1Cii.xml")

approach = network.get_semi_diffusive_approach()

bounds = approach.get_optimization_bounds()

print(bounds)
```

this provides the following output:

```
[(0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0,␣
↪55), (0, 55), (0, 55)]
```

the elements of which correspond to the fluxes that can be obtained from the following command

```
approach.print_decision_vector()
```

which provides the output:

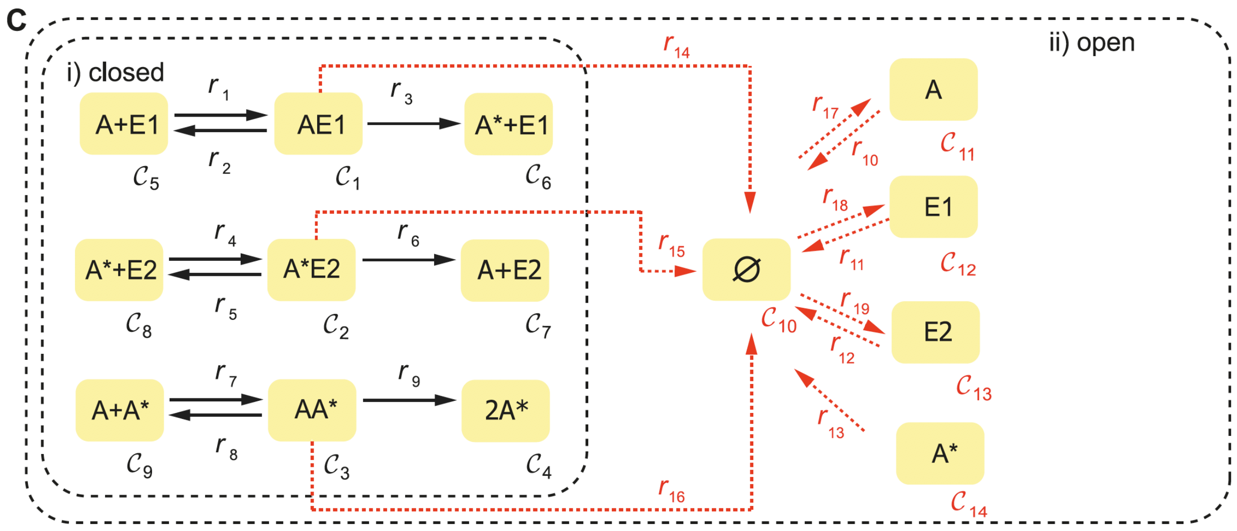
```
Decision vector for optimization:
[v_2, v_3, v_4, v_5, v_6, v_7, v_9, v_11, v_13, v_15, v_17, v_18]

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

Here the decision vector for optimization is defined in terms of fluxes of the reactions. To make identifying which flux we are considering easier, the command above relates the flux to the reaction label. Thus, flux 'v_2' refers to the flux of reaction 're1r'.

CHAPTER 8

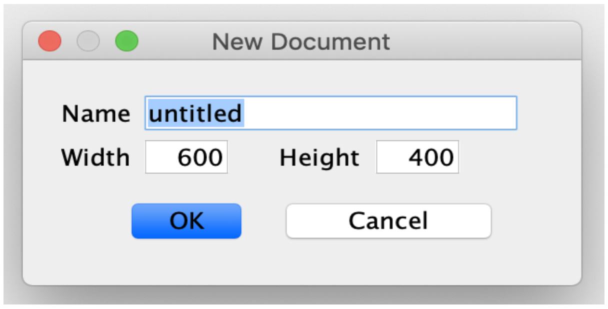
CellDesigner Walkthrough

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.

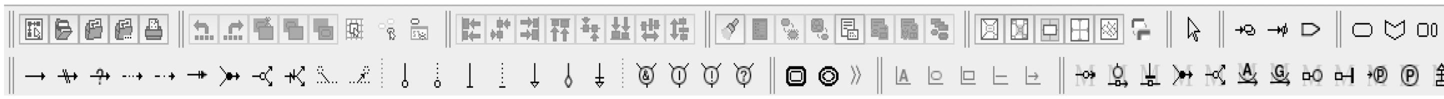


8.1 Creating a Species

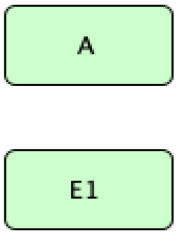
To begin, launch CellDesigner and create a new document. The following new document box will then appear. The name provided in this box can be set to anything the user desires and a specific name is not required. In addition to a name, this box also asks for the dimension of white space available in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



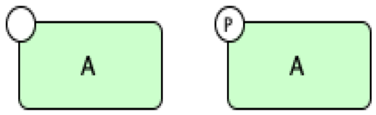
Once the workspace has been created, the species of the network can be represented in CellDesigner by creating a generic protein, which can be found in the top toolbar (as pictured below) by hovering over the symbols.



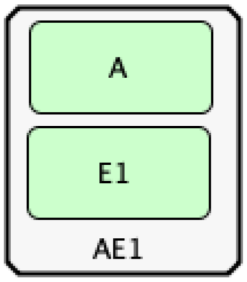
After the generic protein symbol is selected click on the workspace to create a species. A box will then appear and ask for the species name. Although no specific name is required, for visual purposes it is suggested to use a name that is similar to the name used in the C-graph. Below we have created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

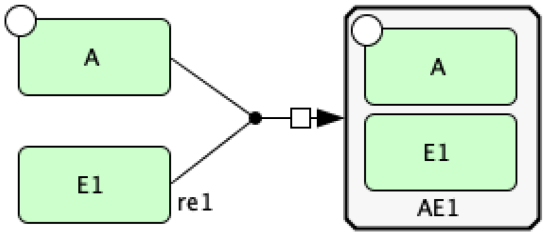


In addition to creating species we can also create chemical complexes as in the C-graph. To do this, in the top toolbox select the symbol "Complex" and click in the workspace, again a specific name is not required, but it is encouraged. One can then place species within this complex using the generic protein approach outlined above. Below is the CellDesigner representation of complex $AE1$.

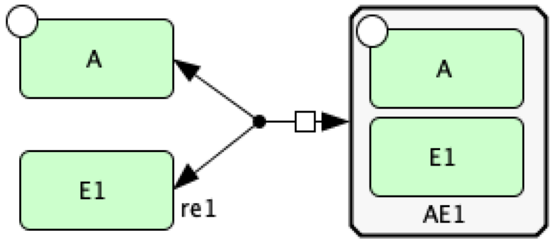


8.2 Creating a Reaction

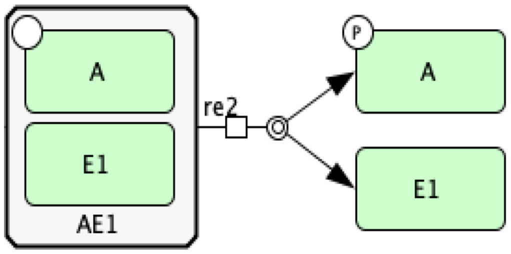
In CellDesigner there are three types of reactions that are important when recreating a C-graph: State Transition $\longrightarrow$, Heterodimer Association, and Dissociation. We will first demonstrate Heterodimer Association and Dissociation reactions by creating reactions r_1, r_2, and r_3 of the C-graph. 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 the CellDesigner depiction should be similar to the picture provided below.



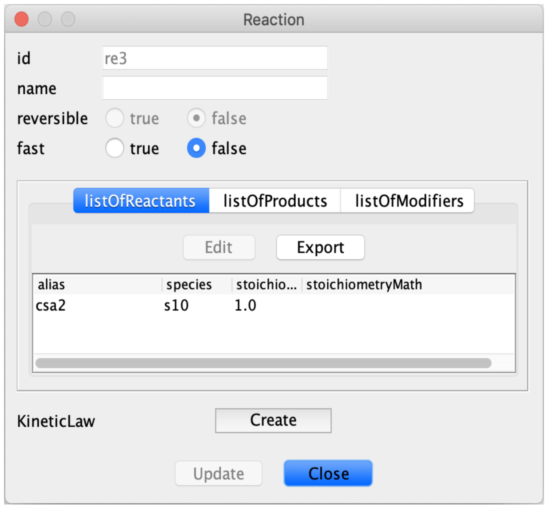
To create r_2 we need to make the heterodimer reaction reversible. To make a reaction reversible right click the reaction and select "Change Identity…", then select True under the reversible category. This provides the CellDesigner representation of r_1 and r_2 as provided below.



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.



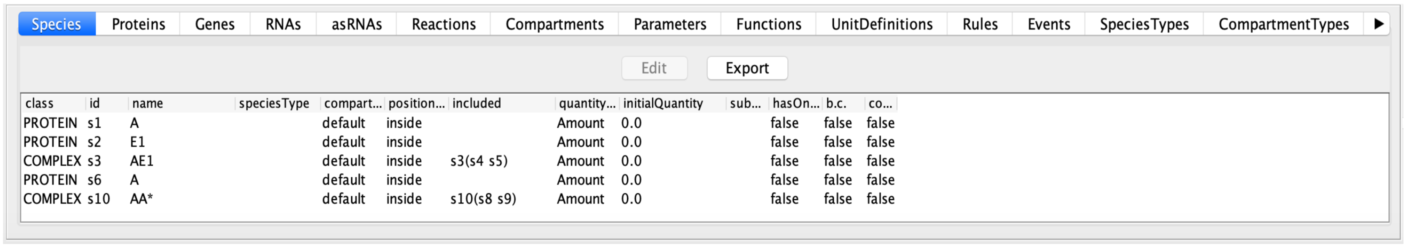
The last type of reaction we will consider is a State Transition, to do this we will produce reaction r_9. After creating complex A^*A, we create reaction r_9 by selecting "State Transition" and first click the complex A^*A and then the phosphorylated species A. Although we have created a reaction we 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.



id: re3
name:
reversible: true [✓ false]
fast: true [✓ false]

alias	species	stoichio...	stoichiometryMath
csa2	s10	1.0	

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.

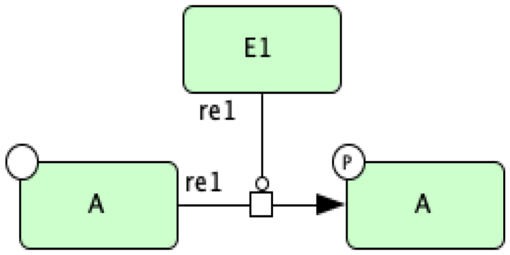


class	id	name	speciesType	compart...	position...	included	quantity...	initialQuantity	sub...	hasOn...	b.c.	co...
PROTEIN	s1	A		default	inside		Amount	0.0		false	false	false
PROTEIN	s2	E1		default	inside		Amount	0.0		false	false	false
COMPLEX	s3	AE1		default	inside	s3(s4 s5)	Amount	0.0		false	false	false
PROTEIN	s6	A		default	inside		Amount	0.0		false	false	false
COMPLEX	s10	AA*		default	inside	s10(s8 s9)	Amount	0.0		false	false	false

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.

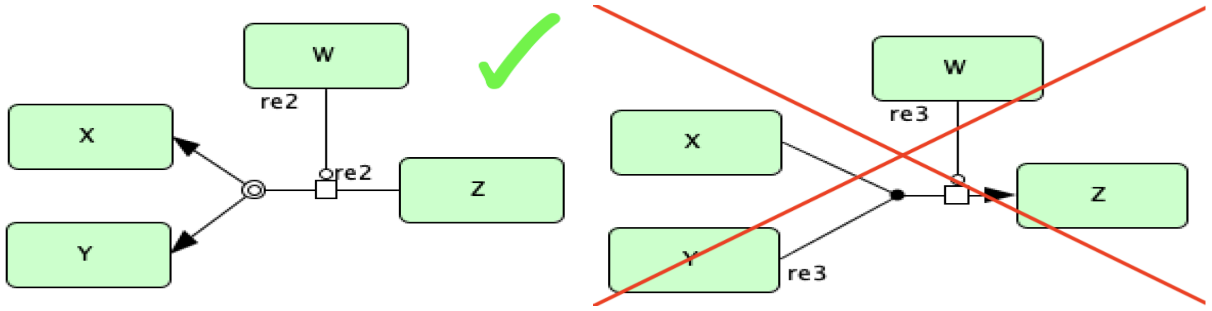
8.3 Representing Catalysis

Another useful feature that has been implemented in crnt4sbml is the ability to represent catalysis. In CellDesigner catalysis is fairly straightforward to implement and can often lead to simpler looking diagrams. If we consider the C-graph provided, one can see that the reactions r_1, r_2, and r_3 depict catalysis, where $E1$ is the catalyst. To represent this in CellDesigner, we first create the species A, $E1$, and phosphorylated species A. Once these species are created, we then construct a State Transition from species A to the phosphorylated species A. Note that the State Transition cannot be reversible. We can now specify catalysis, which is represented in CellDesigner as the symbol , by selecting the symbol for catalysis, selecting species $E1$ and then clicking on the square box of the State Transition. If these steps are followed, the following CellDesigner layout should be produced:



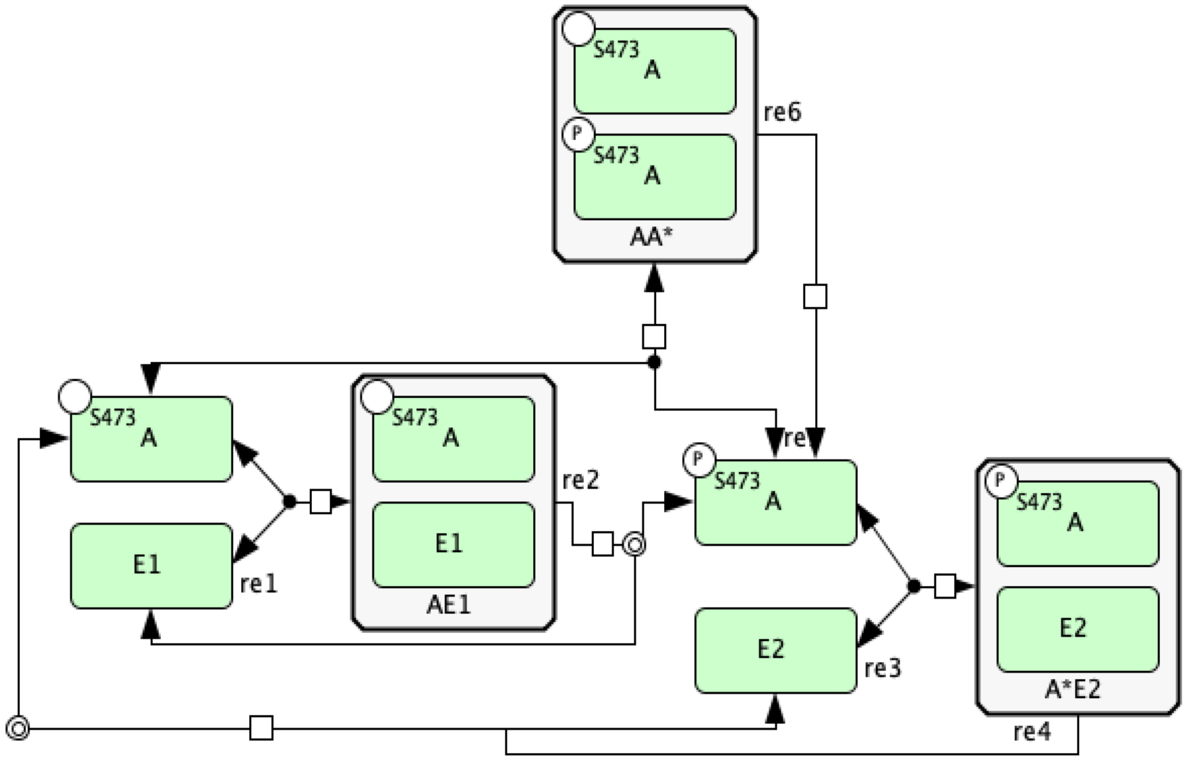
When parsing this type of SBML file, crnt4sbml will construct the underlying C-graph appropriately. For example, if we say the species A is given by species id 's1', phosphorylated species A by species id 's2', and species $E1$ by species id 's3', then crnt4sbml will construct the following reactions s1+s3 -> s3s1, s3s1 -> s1+s3, and s3s1 -> s2+s3. These reactions will then have the reaction labels 're1f', 're1d', and 're1c', respectively, specifying complex formation, complex dissociation, and catalysis, respectively, when referenced in crnt4sbml.

In addition to this type of catalysis, we also allow for catalysis involving a complex dissociation reaction. However, we do not allow for catalysis involving a complex formation reaction. Below we depict these two scenarios.



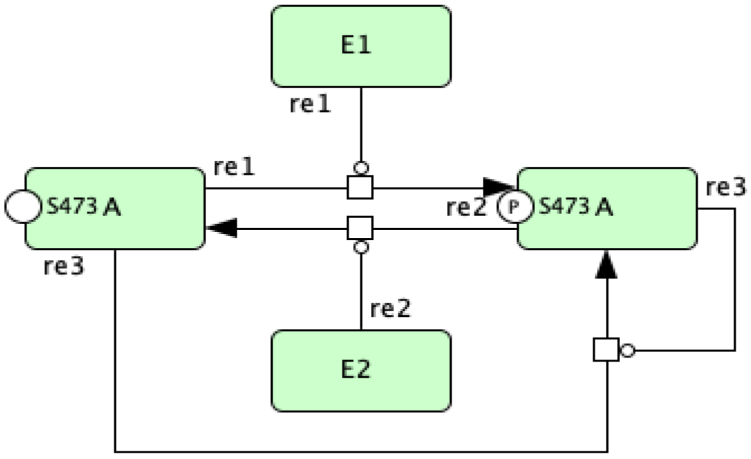
8.4 Basic Mass Conservation SBML File

Using the tools we have outlined so far, we can represent the mass conservation portion of the provided 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.



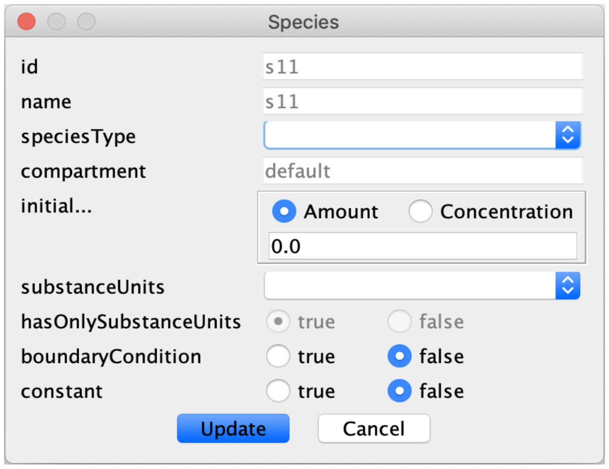
8.5 Catalysis Mass Conservation SBML File

Although the CellDesigner layout produced above is perfectly fine, it may become congested especially if more reactions and species are added. In this case, it may be beneficial to represent particular groups of reactions as catalysis instead. Using the guidelines established in the sections above, we can construct the mass conservation portion of the C-graph as follows in CellDesigner.



8.6 Adding Inflow and Outflow

In a semi-diffusive network we consider the degradation and formation of a species and 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. $\varnothing$). 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 semi-diffusive 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.

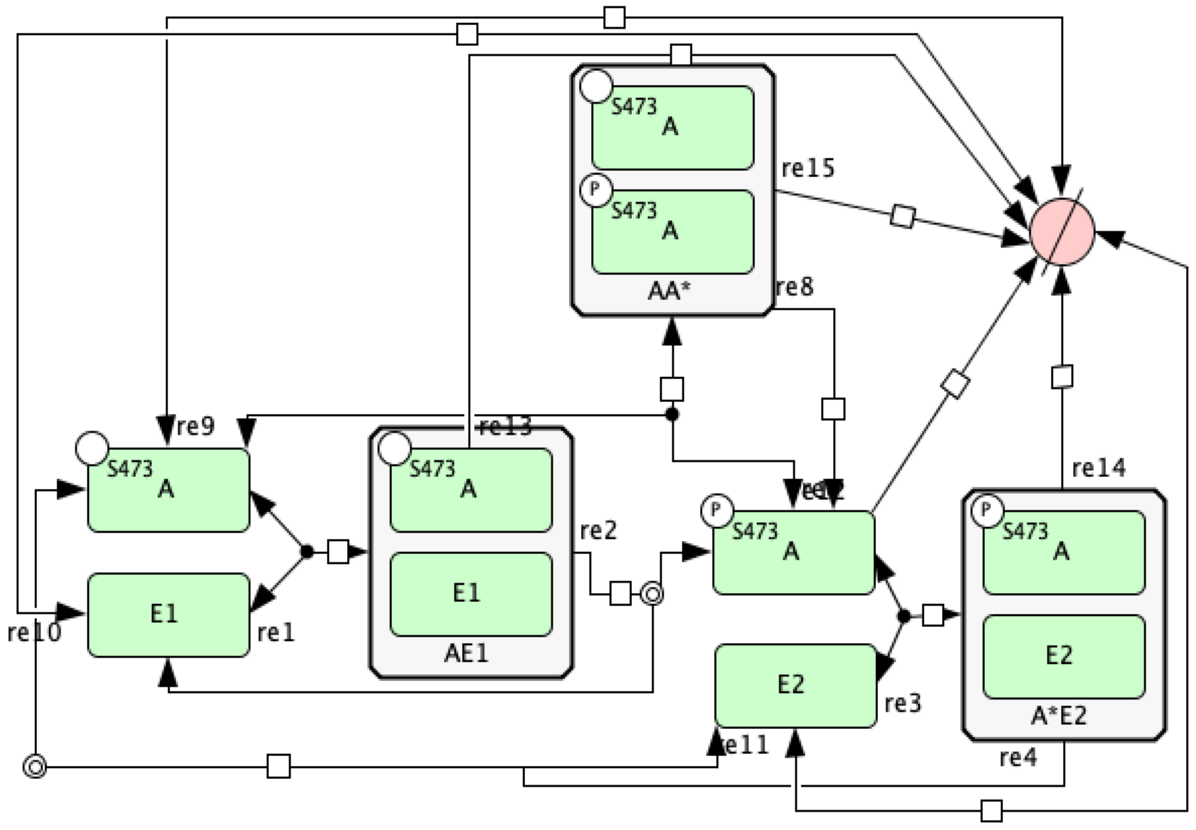

Species

id	s11
name	s11
speciesType	
compartment	default
initial...	[✓ Amount] Concentration 0.0
substanceUnits	
hasOnlySubstanceUnits	[✓ true] false
boundaryCondition	true [✓ false]
constant	true [✓ false]

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.

8.7 Semi-diffusive SBML File

Using the inflow and outflow convention, and the ideas established in the previous subsections, we can recreate the semi-diffusive portion of the provided C-graph using CellDesigner. One possible layout of this C-graph in CellDesigner is provided below.


S473
A
S473
A
AA*
re15
re8
re9
re13
S473
A
S473
A
E1
AE1
re2
re12
S473
A
re14
S473
A
E2
A*E2
E1
re1
re10
E2
re3
re11
re4

CHAPTER 9

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:

```
network = 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.CRNT()`*. To obtain a print out of the number of species, complexes, reactions and deficiency of the network complete the following command:

```
network.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:

```
network.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  --  re5
s15 -> s1+s6  --  re5r
s15 -> 2*s6  --  re6
```

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 're5r'. 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.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 = network.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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.
```

For information on the possible output for this run, please see *crnt4sbml.LowDeficiencyApproach.report_deficiency_one_theorem()* and *crnt4sbml.LowDeficiencyApproach.report_deficiency_zero_theorem()*.

CHAPTER 10

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 uniterminal 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.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
network = crnt4sbml.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:

```
approach = network.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: 1.992364
```

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(approach.get_decision_vector())
```

This provides the following output:

```
[re1, re1r, re2, re3, re3r, re4, re5, re5r, re6, s2, s6, s15]
```

To obtain more available functions that this initialization provides, see `crnt4sbml.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.

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 provide bounds for, we can use the following command:

```
print(approach.get_concentration_bounds_species())
```

This provides the following output:

```
[s1, s3, s7, s16]
```

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 s16, in that order.

As creating these bounds is not initially apparent to novice users or may become cumbersome, we have created a function call that will automatically generate physiological bounds based on the C-graph. To use this functionality one can add the following code:

```
bnds, conc_bnds = approach.get_optimization_bounds()
```

This provides the following values:

```
bnds = [(1e-08, 0.0001), (1e-05, 0.001), (0.001, 1.0), (1e-08, 0.0001), (1e-05, 0.
↪001), (0.001, 1.0),
        (1e-08, 0.0001), (1e-05, 0.001), (0.001, 1.0), (0.5, 500000.0), (0.5, 500000.
↪0), (0.5, 500000.0)]

conc_bnds = [(0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0)]
```

For more information and the correctness on these bounds please refer to *Creating Physiological Bounds*.

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.MassConservationApproach.run_optimization()` for the default values of the routine.

```
import numpy
```

(continues on next page)

(continued from previous page)

```
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 = approach.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: 14.034250974655151

Running the multistart optimization method ...
Elapsed time for multistart method: 67.97679090499878
```

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:

```
approach.generate_report()
```

This will provide the following output:

```
Smallest value achieved by objective function: 0.0
28 point(s) passed the optimization criteria.
```

The first line tells one how the smallest value of the objective function that was found after all points have been ran. The second line describes 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. 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(approach.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.

Here we set the maximum stepsize for numerical continuation, DSMAX to 1e3. 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. In addition to this range, the lower and upper bounds for the measure of the error is also provided as 'A0' and 'A1', 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 next 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, plot_specifications = approach.run_continuity_
↪analysis(species=spcs, parameters=params_for_global_min,
                                                                                 auto_
↪parameters={'PrincipalContinuationParameter': PCP_x,
                                                                                    ␣
↪          'RL0': 1e2, 'RL1': 1e6, 'A0': 0.0, 'A1': 5e6,
                                                                                    ␣
↪          'DSMAX': 1e3},
                                                                                 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 named "multistable_param_ind" above. Along with these indices, the routine will also return the plot specifications for each element in "multistable_param_ind" that specify the range used for the x-axis, y-axis, and the x-y values for each special point in the plot (named "plot_specifications" above). Also note that if multistability plots are produced, the plot names will have the following form: PCP_species id_index of params_for_global.png. The output provided by the numerical continuation run is as follows:

```
Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 26.88336992263794
```

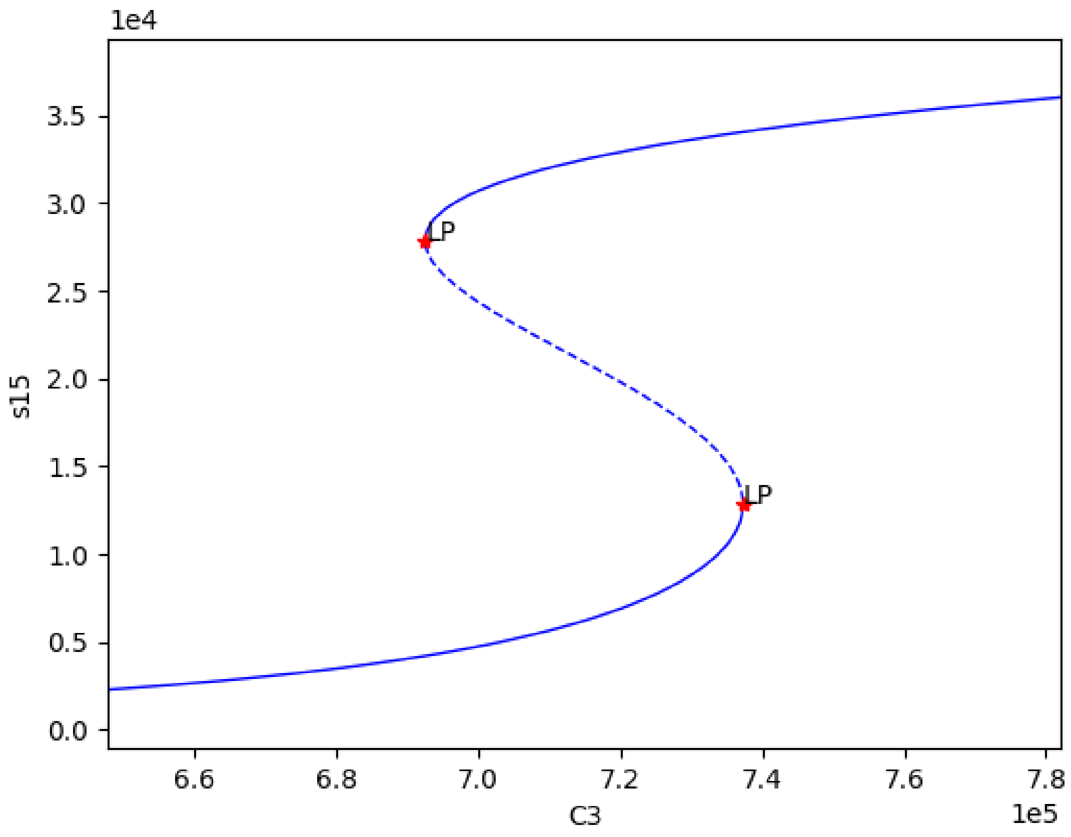
Again, we can generate a report that will contain the numerical optimization routine output and the now added information provided by the numerical continuation run.

```
approach.generate_report()
```

This provides the following output that describes that of the 28 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.

```
Smallest value achieved by objective function: 0.0
28 point(s) passed the optimization criteria.
Number of multistability plots found: 14
Elements in params_for_global_min that produce multistability:
[0, 1, 5, 7, 8, 12, 13, 14, 15, 20, 23, 25, 26, 27]
```

The following is a bistability plot produced by element 27 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, plot_specifications = approach.run_greedy_continuity_
↪analysis(species=spcs, parameters=params_for_global_min, dir_path="./stability_
↪graphs",
                                                                               ␣
↪auto_parameters={'PrincipalContinuationParameter': PCP_x})

approach.generate_report()
```

This provides the following output:

```
Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 144.57969522476196

Smallest value achieved by objective function: 0.0
28 point(s) passed the optimization criteria.
Number of multistability plots found: 19
Elements in params_for_global_min that produce multistability:
[0, 1, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 20, 23, 25, 26, 27]
```

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.

Although numerical continuation can be used by most examples, in some cases, the input vectors found by the optimization method yield an ODE system that has a singular or ill-conditioned Jacobian. For this reason, the numerical continuation method will be unsuccessful. To provide an alternative method to numerical continuation, we have constructed a routine that performs direct simulation in order to construct the bifurcation diagram. See section *Direct Simulation for the General Approach* for further information on the method.

To run bistability analysis using the direct simulation approach, we run the following routine:

```
import crnt4sbml

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

approach = network.get_mass_conservation_approach()

bounds, concentration_bounds = approach.get_optimization_bounds()

params_for_global_min, obj_fun_val_for_params = approach.run_
↪optimization(bounds=bounds, concentration_bounds=concentration_bounds)

approach.run_direct_simulation(response="s15", signal="C3", params_for_global_
↪min=params_for_global_min)

approach.generate_report()
```

This routine will use the input vectors (named params_for_global_min) provided by the optimization and perform the direct simulation approach for bistability analysis, then puts the plots produced in the directory path ./dir_sim_graphs. This provides the following output for the simple_biterminal example:

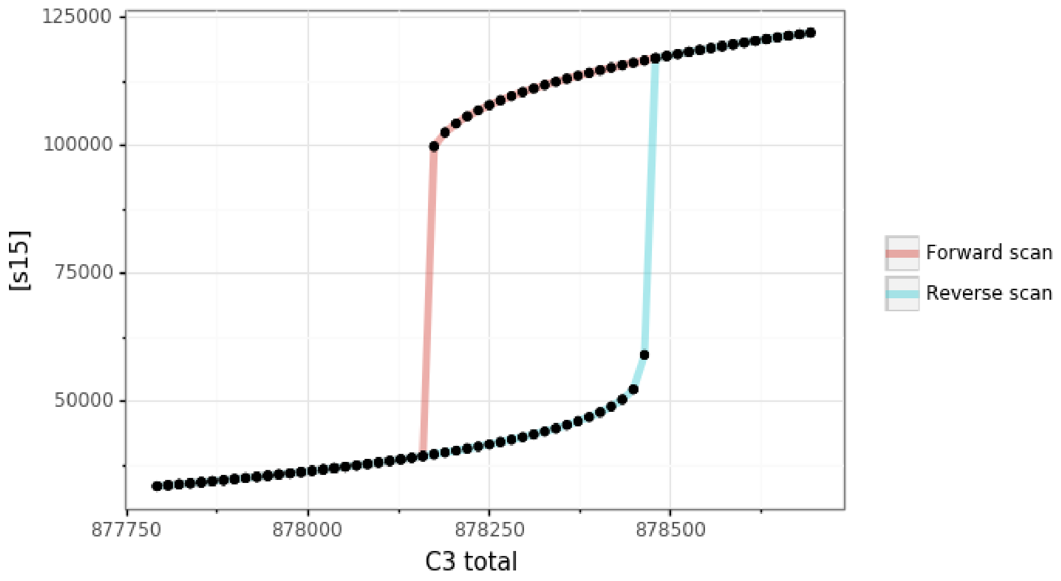
```
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 2.384094

Running feasible point method for 10 iterations ...
Elapsed time for feasible point method: 1.722398281097412

Running the multistart optimization method ...
```

(continues on next page)

(continued from previous page)

```
Elapsed time for multistart method: 8.421388149261475

Starting direct simulation ...
Elapsed time for direct simulation in seconds: 919.151850938797
Smallest value achieved by objective function: 0.0
4 point(s) passed the optimization criteria.
```

Along with this, it also produces the following bifurcation diagram.



Similar to the optimization for the mass conservation approach, we can see that direct simulation can take a long time to complete. For this reason, we have a parallel version of the direct simulation approach and optimization. The parallel version can be ran by setting parallel_flag=True and then running with mpiexec. For further details on running in parallel see section *Parallel CRNT4SBML*.

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

CHAPTER 11

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 the SBML file `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
network = crnt4sbml.CRNT("/path/to/Fig1Cii.xml")
```

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

```
approach = network.get_semi_diffusive_approach()
```

This command creates all the necessary information to construct the optimization problem to be solved. Unlike the mass conservation approach, 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.

```
approach.print_decision_vector()
```

This provides the following output:

```
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, if your are using an SBML file you created yourself, the output may differ. If you would like an explicit list of the decision vector you can use the following command:

```
print(approach.get_decision_vector())
```

Using the decision vector as a reference, we can now provide the bounds for the optimization routine. As creating these bounds is not initially apparent to novice users or may become cumbersome, we have created a function call that will automatically generate physiological bounds based on the C-graph. To use this functionality one can add the following code:

```
bounds = approach.get_optimization_bounds()
```

This provides the following values:

```
bounds = [(0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0,␣
↪55), (0, 55), (0, 55), (0, 55)]
```

For more information and the correctness on these bounds please refer to *Creating Physiological Bounds*. An 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(approach.get_key_species())
print(approach.get_non_key_species())
print(approach.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.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 = approach.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: 1.542820930480957

Running the multistart optimization method ...
Elapsed time for multistart method: 184.3005211353302
```

For a detailed description of the optimization routine see *Numerical Optimization Routine*. 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:

```
approach.generate_report()
```

This provides the following output:

```
Smallest value achieved by objective function: 0.0
76 point(s) passed the optimization criteria.
```

The first line tells the user the smallest value that was achieved after all of the iterations have been completed. The next line tells one 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. 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')
```

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, plot_specifications = approach.run_continuity_analysis(species=
↪'s7', parameters=params_for_global_min,
                                                                              auto_
↪parameters={'PrincipalContinuationParameter': 're17',
                                                                                   ␣
↪          'RL0': 0.1, 'RL1': 100, 'A0': 0.0,
                                                                                   ␣
↪           'A1': 10000})
```

In addition to putting the multistability plots found into the folder num_cont_graphs, this routine will also return the indices of params_for_global_min that correspond to these plots named “multistable_param_ind” above. Along with these indices, the routine will also return the plot specifications for each element in “multistable_param_ind” that specify the range used for the x-axis, y-axis, and the x-y values for each special point in the plot (named “plot_specifications” above). Also note that if multistability plots are produced, the plot names will have the following form: PCP_species id_index of params_for_global.png. For more information on the AUTO parameters provided and the continuation routine itself, refer to *Numerical Continuation Routine*. This provides the following output:

```
Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 126.53627181053162
```

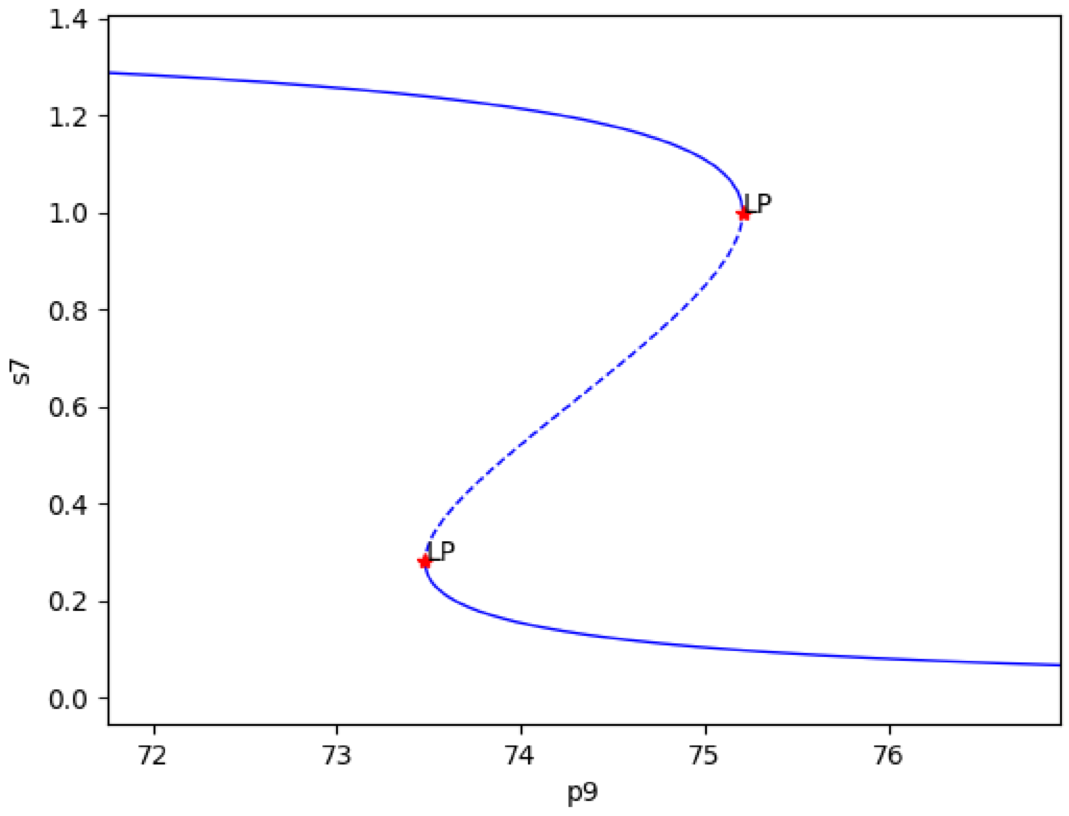
Again we can generate a report that will contain the numerical optimization routine output and the now added information provided by the numerical continuation run:

```
approach.generate_report()
```

This provides the following output:

```
Smallest value achieved by objective function: 0.0
76 point(s) passed the optimization criteria.
Number of multistability plots found: 56
Elements in params_for_global_min that produce multistability:
[2, 3, 4, 5, 6, 9, 10, 12, 13, 14, 16, 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 29, 30,
↪ 31, 32, 33, 35, 36, 37, 38,
 39, 40, 41, 42, 43, 44, 46, 47, 48, 49, 50, 51, 52, 53, 55, 56, 57, 58, 62, 63, 64,␣
↪68, 69, 70, 71, 75]
```

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.png. The following is one multistability plot produced.



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, plot_specifications = approach.run_greedy_continuity_
↪analysis(species="s7", parameters=params_for_global_min,
                                                                              ␣
↪auto_parameters={'PrincipalContinuationParameter': 're17'})

approach.generate_report()
```

This provides the following output:

```
Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 534.1763272285461

Smallest value achieved by objective function: 0.0
76 point(s) passed the optimization criteria.
Number of multistability plots found: 73
Elements in params_for_global_min that produce multistability:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
↪ 25, 26, 27, 29, 30, 31, 32,
 33, 34, 35, 36, 37, 38, 39, 40, 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, 67, 68, 69, 70, 71, 72, 73, 74, 75]
```

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 12

General Approach Walkthrough

Using the SBML file constructed as in *CellDesigner Walkthrough*, we will proceed by completing a more in-depth explanation of running the general approach. Note that the general 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.Cgraph.get_dim_equilibrium_manifold()`* is greater than zero. This tutorial will use `simple_biterminal.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
network = crnt4sbml.CRNT("/path/to/simple_biterminal.xml")
```

If we then want to conduct the general approach, we must first initialize the general_approach, which is done as follows:

```
approach = network.get_general_approach()
```

Now that we have initialized the class, we have to tell the routine the values of the signal (or principal continuation parameter) and response of the bifurcation diagram, as well as whether or not we would like to force a steady state. Just as in *Mass Conservation Approach Walkthrough*, the signal (or PCP for numerical continuation) is a conservation law. To select the signal one needs to know which conservation law to choose. The following command will provide the conservation laws derived by the initialization of the general approach class:

```
print(approach.get_conservation_laws())
```

This provides the following output:

```
C1 = 1.0*s10 + 1.0*s11 + 1.0*s2s10 + 1.0*s2s9 + 1.0*s9
C2 = 1.0*s1 + 1.0*s2 + 1.0*s2s10 + 1.0*s2s9 + 2.0*s3 + 2.0*s6
```

The response of the bifurcation diagram can then be chosen as any species. For this particular example we will choose the following signal and response:

```
signal = "C2"
response = "s11"
```

Now that we have the bifurcation parameters, we should consider whether or not we would like to force a steady state in the ODE system formed by the network by fixing the reactions. Although forcing a steady state by fixing the reactions can provide faster results for some networks when running optimization, it does restrict the solutions found to a particular solution, rather than looking for a general solution. If reactions are fixed, the reactions that are fixed can by found by using *crnt4sbml.GeneralApproach.get_fixed_reactions()*, where the symbolic expressions for these reactions are given by *crnt4sbml.GeneralApproach.get_solutions_to_fixed_reactions()*.

For this particular example, fixing the reactions leads to poor results. Thus, we will choose to not fix the reactions, this is done by setting the fix_reaction variable to False in *crnt4sbml.GeneralApproach.initialize_general_approach()*. Now we can initialize the rest of the general approach as follows:

```
approach.initialize_general_approach(signal=signal, response=response, fix_
↪reactions=False)
```

Now that the approach has been constructed, we can begin to define the specific information needed for the optimization routine for the general approach. One very important value that must be provided to the optimization problem are the bounds for the species and reactions. For this reason, it is useful to see the variables and the order in which they appear. To do this one can add the following command to the script:

```
print(approach.get_input_vector())
```

This provides the following output:

```
[re1, re1r, re2, re2r, re3, re4, re5f, re5d, re5c, re6, re7f, re7d, re7c, re8, s1, s2,
↪ s3, s6, s9, s10, s2s9, s11, s2s10]
```

Using the input 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.

As creating these bounds is not initially apparent to novice users or may become cumbersome, we have created a function call that will automatically generate physiological bounds based on the C-graph. To use this functionality one can add the following code:

```
bnds = approach.get_optimization_bounds()
```

This provides the following values:

```
bnds = [(1e-08, 0.0001), (1e-05, 0.001), (1e-08, 0.0001), (1e-05, 0.001), (0.001, 1.
↪0), (0.001, 1.0), (1e-08, 0.0001),
        (1e-05, 0.001), (0.001, 1.0), (0.001, 1.0), (1e-08, 0.0001), (1e-05, 0.001),␣
↪(0.001, 1.0), (0.001, 1.0),
        (0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0), (0.5,␣
↪500000.0), (0.5, 500000.0),
        (0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0)]
```

For more information and the correctness on these bounds please refer to *Creating Physiological Bounds*.

Although these bounds can be used for this example, they are not ideal. For this reason, we have chosen a particular set of ranges for the species and reactions based on the input vector, which is given as follows (for reference, below we have set the range for re1 to be between 2.4 and 2.42, and set the range for s2 to be between 18.0 and 18.5):

```
bnds = [(2.4, 2.42), (27.5, 28.1), (2.0, 2.15), (48.25, 48.4), (0.5, 1.1), (1.8, 2.1),
↪ (17.0, 17.5), (92.4, 92.6),
        (0.01, 0.025), (0.2, 0.25), (0.78, 0.79), (3.6, 3.7), (0.15, 0.25), (0.06, 0.
↪065)] + [(0.0, 100.0),
        (18.0, 18.5), (0.0, 100.0), (0.0, 100.0), (27.0, 27.1), (8.2, 8.3), (90.0, 90.
↪1), (97.5, 97.9), (30.0, 30.1)]
```

(continues on next page)

(continued from previous page)

The next most important parameter for optimization is the number of initial points for the multi-start optimization. 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). One can even use the built-in confidence level option as described in *Confidence Level Routine* to make an informed decision on whether or not to continue performing more iterations. We state the number of initial points below.

```
iters = 15
```

The last values that can be defined before the optimization portion (as provided below) are the number of iterations allowed for the Dual Annealing optimization method used (provided by Scipy), 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), and the print_flag which tells the program if the objective function value and decision vector for the multi-start method should be printed out (here we set it to False, which means no output will be provided). See `crnt4sbml.GeneralApproach.run_optimization()` for the default values of the routine.

```
d_iters = 1000
sd = 0
prnt_flg = False
```

Using these values, we run the optimization problem using the following command, which returns a list of the parameters (which correspond to the input vector) and corresponding objective function values that produce an objective function value smaller than machine epsilon.

```
params_for_global_min, obj_fun_vals = approach.run_optimization(bounds=bnds,␣
↪iterations=iters, seed=sd, print_flag=prnt_flg,
                                                                dual_annealing_
↪iters=d_iters, confidence_level_flag=True)

approach.generate_report()
```

The following is the output obtained after running the above code:

```
Running the multistart optimization method ...
Elapsed time for multistart method: 2590.524824142456

It was found that 2.1292329042333798e-16 is the minimum objective function value with␣
↪a confidence level of 0.680672268907563 .
1 point(s) passed the optimization criteria.
```

From this output, it is apparent that for some networks the optimization for the general approach can take a long time to complete. For this reason, we have a parallel version of the optimization approach. An example of a parallel general approach can be found in subsection *Parallel General Approach* of section *Parallel CRNT4SBML*.

If the optimization routine returns objective function values smaller than machine epsilon, then bistability analysis can be conducted. As in *Mass Conservation Approach Walkthrough* and *Semi-diffusive Approach Walkthrough* this can be done by using numerical continuation. See the functions `crnt4sbml.GeneralApproach.run_continuity_analysis()` and `crnt4sbml.GeneralApproach.run_greedy_continuity_analysis()` for more information on using numerical continuation with the general approach. Although numerical continuation can be used by most examples, in some cases, the input vectors found by the optimization method yield an ODE system that has a singular or ill-conditioned Jacobian. For this reason, the numerical continuation method will be unsuccessful. In the simple_biterminal example, this is what occurs. To provide an alternative method to numerical continuation, we have constructed a routine that performs direct simulation in order to construct the bifurcation diagram. See section *Direct Simulation for the General Approach* for further information on the method.

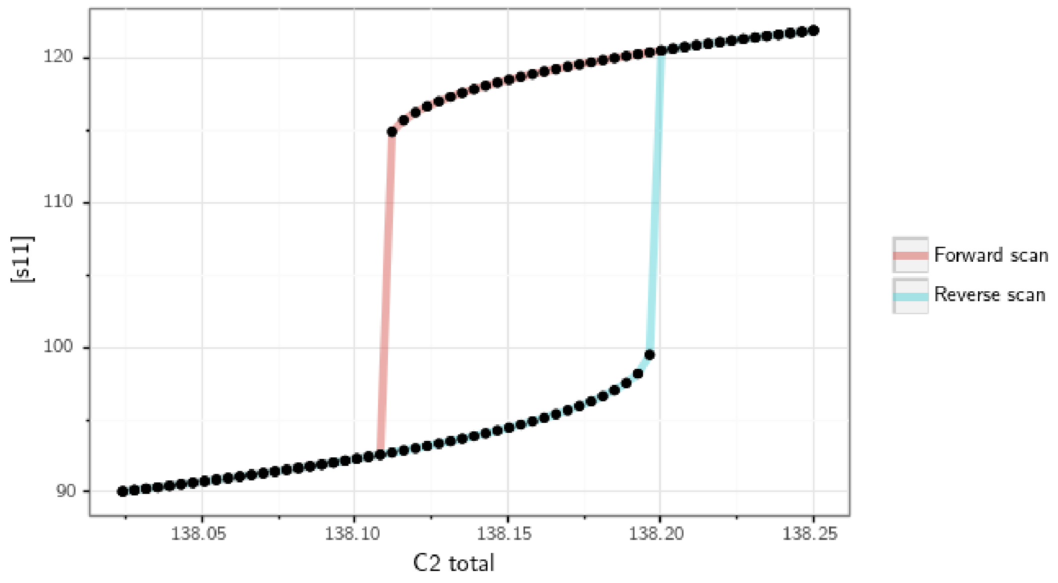
To run bistability analysis using the direct simulation approach, we run the following routine:

```
approach.run_direct_simulation(params_for_global_min)
```

This routine will use the input vectors (named params_for_global_min) provided by the optimization and perform the direct simulation approach for bistability analysis, then puts the plots produced in the directory path ./dir_sim_graphs. This provides the following output for the simple_biterminal example:

```
Starting direct simulation ...
Elapsed time for direct simulation in seconds: 189.25777792930603
```

Along with this, it also produces the following bifurcation diagram.



Similar to the optimization for the general approach, we can see that direct simulation can take a long time to complete. For this reason, we have a parallel version of the direct simulation approach. An example of a parallel direct simulation run for the general approach can be found in subsection *Parallel General Approach* of section *Parallel CRNT4SBML*.

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

CHAPTER 13

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

13.1 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.

13.2 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.

13.3 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 14

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.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 15

Direct Simulation for the General Approach

When using the general approach it is possible that the optimization routine finds kinetic constants that force the Jacobian of the system to be ill-conditioned or even singular, even if species concentrations are varied. If this particular scenario occurs, numerical continuation will not be able to continue as it relies on a well-conditioned Jacobian. To overcome this type of situation we have constructed the function *crnt4sbml.GeneralApproach.run_direct_simulation()* for the general approach. The direct simulation routine strategically chooses the initial conditions for the ODE system and then simulates the ODEs until a steady state occurs. Then based on the user defined signal and optimization values provided, it will vary the signal amount and simulate the ODE system again until a steady state occurs. By varying the signal for several values and different initial conditions, direct simulation is able to construct a bifurcation diagram. Given the direct simulation method is numerically integrating the system of ODEs, this method will often take longer than the numerical continuation routine. Although this is the case, direct simulation may be able to provide a bifurcation diagram when numerical continuation cannot.

15.1 Failure of numerical continuation

In the following example we consider the case where numerical continuation fails to provide the appropriate results. For this example, we will be using the SBML file `simple_biterminal.xml`. We then construct the following script, where we are printing the output of the numerical continuation.

```
import crnt4sbml
network = crnt4sbml.CRNT("/path/to/simple_biterminal.xml")
signal = "C2"
response = "s11"
bnds = [(2.4, 2.42), (27.5, 28.1), (2.0, 2.15), (48.25, 48.4), (0.5, 1.1), (1.8, 2.1),
↪ (17.0, 17.5), (92.4, 92.6),
        (0.01, 0.025), (0.2, 0.25), (0.78, 0.79), (3.6, 3.7), (0.15, 0.25), (0.06, 0.
↪065)] + \ [(0.0, 100.0),
        (18.0, 18.5), (0.0, 100.0), (0.0, 100.0), (27.0, 27.1), (8.2, 8.3), (90.0, 90.
↪1), (97.5, 97.9), (30.0, 30.1)]

approach = network.get_general_approach()
approach.initialize_general_approach(signal=signal, response=response)
```

(continues on next page)

(continued from previous page)

```
params_for_global_min, obj_fun_vals = approach.run_optimization(bounds=bnds,␣
↪iterations=15, dual_annealing_iters=1000,
                                                                confidence_level_
↪flag=True, parallel_flag=False)

multistable_param_ind, plot_specifications = approach.run_greedy_continuity_
↪analysis(species=response, parameters=params_for_global_min, print_lbls_flag=True,
                                                                                          ␣
↪auto_parameters={'PrincipalContinuationParameter': signal})
approach.generate_report()
```

This provides the following output:

```
Running the multistart optimization method ...
Elapsed time for multistart method: 2590.524824142456

It was found that 2.1292329042333798e-16 is the minimum objective function value with␣
↪a confidence level of 0.680672268907563 .
1 point(s) passed the optimization criteria.

Running continuity analysis ...
J0: -> s11; re7c*s2s10 - re8*s11;J1: -> s2; -re1*s2*(C2 - s2 - s2s10 - s2s9 - 2.0*s3 -
↪ 2.0*s6) + re1r*s3 -
re2*s2*(C2 - s2 - s2s10 - s2s9 - 2.0*s3 - 2.0*s6) + re2r*s6 + 2*re4*s6 + re5c*s2s9 +␣
↪re5d*s2s9 -
re5f*s2*(C1 - s10 - s11 - s2s10 - s2s9) + re7c*s2s10 + re7d*s2s10 - re7f*s10*s2;
J2: -> s3; re1*s2*(C2 - s2 - s2s10 - s2s9 - 2.0*s3 - 2.0*s6) - re1r*s3 - re3*s3;
J3: -> s6; re2*s2*(C2 - s2 - s2s10 - s2s9 - 2.0*s3 - 2.0*s6) - re2r*s6 - re4*s6;
J4: -> s10; re5c*s2s9 - re6*s10 + re7d*s2s10 - re7f*s10*s2 + re8*s11;
J5: -> s2s9; -re5c*s2s9 - re5d*s2s9 + re5f*s2*(C1 - s10 - s11 - s2s10 - s2s9);
J6: -> s2s10; -re7c*s2s10 - re7d*s2s10 + re7f*s10*s2;
re1 = 2.4179937298574217;re1r = 27.963833386686552;re2 = 2.1212280827699264;re2r = 48.
↪342142632557824;
re3 = 0.9103403848297675;re4 = 1.8021182302742345;re5f = 17.01982705623611;re5d = 92.
↪47396549104621;
re5c = 0.021611755555125196;re6 = 0.23540156485799416;re7f = 0.7824887292735982;re7d␣
↪= 3.692336204373193;
re7c = 0.20574339517454907;re8 = 0.06329703678602935;s1 = 14.749224746318406;s2 = 18.
↪117522179242442;
s3 = 22.37760479141668;s6 = 11.304051540693258;s9 = 27.001718858136442;s10 = 8.
↪264281271233568;
s2s9 = 90.01696959750683;s11 = 97.69532935525308;s2s10 = 30.05600671002251;C1 = 253.
↪03430579215245;C2 = 220.30303589731005;
Labels from numerical continuation:
['EP', 'MX']
Labels from numerical continuation:
['EP', 'MX']
Labels from numerical continuation:
['EP', 'MX']
Labels from numerical continuation:
['EP', 'MX']
Labels from numerical continuation:
['EP', 'MX']
Labels from numerical continuation:
['EP', 'MX']
Labels from numerical continuation:
```

(continues on next page)

(continued from previous page)

```
['EP', 'MX']
Labels from numerical continuation:
['EP', 'MX']
Labels from numerical continuation:
['EP', 'MX']
Labels from numerical continuation:
['EP', 'MX']

Elapsed time for continuity analysis in seconds: 4.751797914505005

Number of multistability plots found: 0
Elements in params_for_global_min that produce multistability:
[]
```

As we can see, the numerical continuation is unable to find limit points for the example. This is due to the Jacobian being ill-conditioned. In cases where the output of the numerical continuation is consistently "[‘EP’, ‘MX’]" or one of the points is "MX", this often indicates that the Jacobian is ill-conditioned or always singular. If this situation is encountered, it is suggested that the user run the direct simulation routine.

15.2 Outline of direct simulation process

To cover the corner case where numerical continuation is unable to complete because the Jacobian is ill-conditioned, we have constructed a direct simulation approach. This approach directly simulates the full ODE system for the network by numerically integrating the ODE system. Using these results, a bifurcation diagram is then produced. In the following subsections we will provide an overview of the workflow carried out by the direct simulation method.

15.2.1 Finding the appropriate initial conditions

When numerically integrating the full system of ODEs we use the SciPy routine solve_ivp. This routine solves an initial value problem for a system of ODEs. For this reason, we need to provide initial conditions that correspond to the optimization values provided. We need to do this for two cases, one where we obtain a high concentration of the response species and another where we obtain a lower concentration of the response species, at a steady state. To do this we use the first element of the optimization values provided to the routine (which correspond to an input vector consisting of reaction constants and species concentrations) to calculate the conservation laws for the problem.

Once we have the conservation law values, we then construct construct all possible initial conditions for the ODE system. This is done by using the conservation laws of the problem. For our example, we have the following conservation laws:

```
C1 = 1.0*s10 + 1.0*s11 + 1.0*s2s10 + 1.0*s2s9 + 1.0*s9
C2 = 1.0*s1 + 1.0*s2 + 1.0*s2s10 + 1.0*s2s9 + 2.0*s3 + 2.0*s6
```

Thus, we can put the total C1 value in any of the following species: s10, s11, s2s10, s2s9, or s9, in addition to this, we can put the total C2 value in any of the following species: s1, s2, s2s10, s2s9, s3, or s6. For example, we can set the initial condition for the system by setting the initial value of s10 = C1, s1 = C2, and all other species to zero. As one can see, we need to test all possible combinations of these species to see the set that appropriately corresponds to the optimization values provided. The number of combinations tested can be reduced by removing duplicate combinations and repeated species.

To determine the combination that we will use to conduct the bistability analysis, we first find the steady state (using the process outlined in the next subsection) for the corresponding initial condition. Using these steady state values, we then determine the conservation law values at the steady state. If the conservation law values align with the conservation law values calculated using the first element of the optimization values, then we consider this combination as a viable combination. Once we have all of the viable combinations, we then select a set of two of these combinations, where one produces a high concentration of the response species and the other has a lower concentration of the response species, at the steady state. To see the initial conditions that will be used for the bistability analysis, one can set print_flag=True in *crnt4sbml.GeneralApproach.run_direct_simulation()*. This provides the following output for the example:

```
For the forward scan the following initial condition will be used:
s1 = 0.0
s2 = C2
s3 = 0.0
s6 = 0.0
s9 = 0.0
s10 = C1
s2s9 = 0.0
s11 = 0.0
s2s10 = 0.0

For the reverse scan the following initial condition will be used:
s1 = 0.0
s2 = C2
s3 = 0.0
s6 = 0.0
s9 = C1
s10 = 0.0
s2s9 = 0.0
s11 = 0.0
s2s10 = 0.0
```

The process of finding these viable combinations can take a long time depending on the network provided. For this reason, this process can be done in parallel by setting parallel_flag=True in *crnt4sbml.GeneralApproach.run_direct_simulation()*. For more information on parallel runs refer to *Parallel CRNT4SBML*.

15.2.2 Finding a steady state to the system

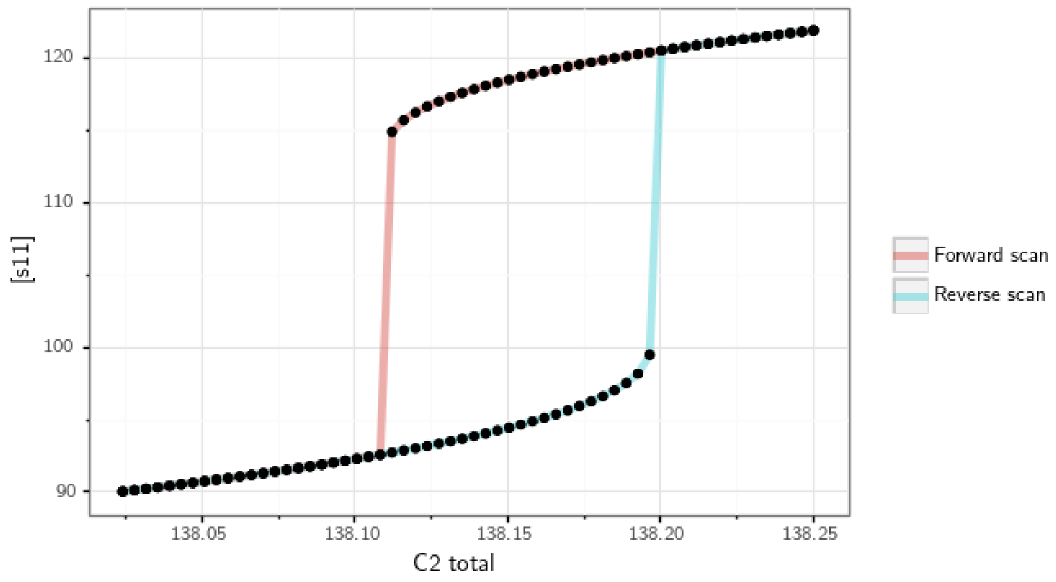
In order to produce a bifurcation diagram, we need to consider the solution of the system of ODEs at a steady state. Due to the nature of the system of ODEs, this solution is often to complex to find analytically. For this reason, we find this solution by numerically integrating the system until we reach a steady state in the system. As mentioned previously, this is done by using the Scipy routine solve_ivp. Specifically, we utilize the BDF method with a rtol of 1e-6 and a atol of 1e-9. To begin, we start with an interval of integration of 0.0 to 100.0, we then continue in increments of 100 until a steady state has been reached or 1000 increments have been completed. A system of ODEs is considered to be at a steady state when the relative error (of the last and current time step of the concentration of the response species) is less than or equal to the user defined variable change_in_relative_error of *crnt4sbml.GeneralApproach.run_direct_simulation()*. It should be noted that a smaller value of change_in_relative_error will run faster, but may produce an ODE system that is not at a steady state.

15.2.3 Constructing the bifurcation diagram

Once the appropriate initial conditions have been given, the direct simulation routine then attempts to construct a bifurcation diagram. Note that this process does not guarantee that a bifurcation diagram with bistability will be provided, rather it will produce a plot of the long-term behavior of the ODEs in a particular interval for the user defined signal. The first step in this process is defining the search radius of the signal. This search radius can be defined by the user by modifying the variables left_multiplier and right_multiplier of *crnt4sbml.GeneralApproach.run_direct_simulation()*, which provide a lower and upper -bound for the signal value. Specifically, when considering different values of the signal, the range for these different values will be in the interval [signal_value -

signal_value*left_multiplier, signal_value - signal_value*right_multiplier], where the signal value is the beginning value of the signal as provided by the input vectors produced by optimization.

Using this range, the routine then splits the range into 100 evenly spaced numbers. The signal is then set equal to each of these numbers and the ODE system is simulated until a steady state occurs, using the initial conditions of both the forward and reverse scan values established in the previous subsection. Using all 200 values, the minimum and maximum value of the response species' concentration is found. This process is then repeated using 60 evenly spaced numbers between the signal values that correspond to the minimum and maximum values of the response species' concentration. Using the 120 values produced, the minimum and maximum values of the response species are found. This process is repeated for 5 iterations or until there are 10 or more signal values between the signal values that correspond to the minimum and maximum values of the response species' concentration of the current iteration. This process effectively detects and "zooms in" on the region where bistability is thought to exist. Although this process can be very effective, it can take a long time to complete. Thus, it is suggested that this be done in parallel by setting parallel_flag=True in `crnt4sbml.GeneralApproach.run_direct_simulation()`. For more information on parallel runs refer to *Parallel CRNT4SBML*. For the example we have been considering, we obtain the following bifurcation diagram.



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.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.

CHAPTER 17

Confidence Level Routine

Although the general, mass conservation, and semi-diffusive approaches can quickly provide confirmation of bistability for most examples, this may not always be the case. In fact, an important item of discussion is that these approaches cannot exclude bistability, even if a large amount of random decision vectors are explored. It is this uncertainty that we wish to address. This is done by assigning a probability that the minimum objective function value achieved is equal to the true global minimum. We achieve this probability by considering a slightly modified version of the unified Bayesian stopping rule in [BGS04] and Theorem 4.1 of [SF87], where the rule was first established.

Let α_k and α^* denote the probability that the optimization routine has converged to the local minimum objective function value, say f_k, and global minimum objective function value, say f^*. Assuming that $\alpha^* \geq \alpha_k$ for all local minimum values f_k we may then state that the probability that $\tilde{f} = f^*$ is as follows:

$$Pr[\tilde{f} = f^*] \geq q(n,r) = 1 - \frac{(n+a+b-1)!(2n+b-r-1)!}{(2n+a+b-1)!(n+b-r-1)!},$$

where n is the number of initial decision vectors that are considered, $\tilde{f} = min\{f_1, \dots, f_n\}$, a and b are parameters of the Beta distribution $\beta(a,b)$, and $q(n,r)$ is the confidence level. We then let r be the number of f_k for $k = 1, \dots, n$ that are in the neighborhood of $\tilde{f}$.

Given our minimum objective function value is zero, for some networks it may be the case that the f_k are nearly zero with respect to machine precision. For this reason, we say that f_k is in the neighborhood of $\tilde{f}$ if

$$\frac{|\tilde{f} - f_k|}{\tilde{f}} \leq 10^{-2}.$$

This means that f_k is in the neighborhood of $\tilde{f}$ if the relative error of f_k and $\tilde{f}$ is less than 1%. If $\tilde{f}$ is considered zero with respect to the system's minimum positive normalized float, then we consider this value zero and provide $q(n,r) = 1.0$, skipping the computation of $q(n,r)$. Thus, we can state that the probability that the obtained $\tilde{f}$ is the global minimum (for the prescribed bounds of the decision vector) is greater than or equal to the confidence level $q(n,r)$. Using the standard practice in statistics, it should be noted that $q(n,r) \geq 0.95$ is often considered an acceptable confidence level to make the conclusion that $\tilde{f}$ is the global minimum of the objective function.

For information on how to enable the construction of a confidence level for each of the approaches, please refer to the following for each approach:

- **Mass conservation approach:**

  - If using *crnt4sbml.MassConservationApproach.run_optimization()* set confidence_level_flag = True and and prescribe a value to change_in_rel_error (if applicable)
  - If using `crnt4sbml.MassConservationApproach.run_mpi_optimization()` set confidence_level_flag = True and and prescribe a value to change_in_rel_error (if applicable)

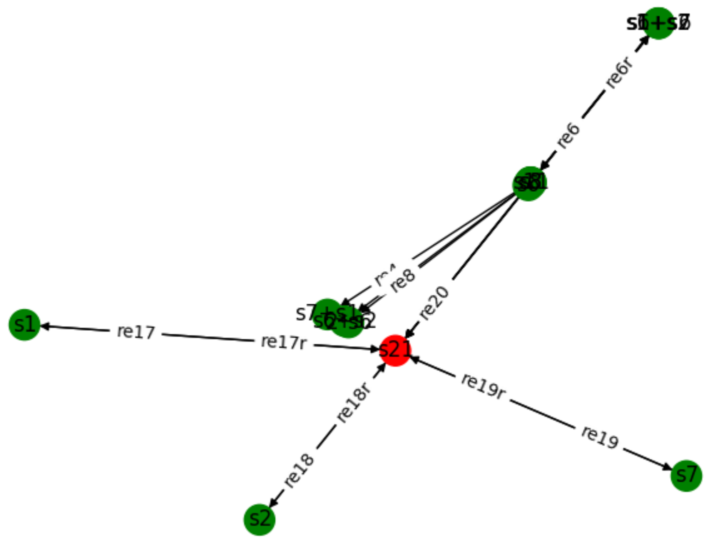
- **Semi-diffusive approach:**
  - If using *crnt4sbml.SemiDiffusiveApproach.run_optimization()* set confidence_level_flag = True and prescribe a value to change_in_rel_error (if applicable)
  - If using `crnt4sbml.SemiDiffusiveApproach.run_mpi_optimization()` set confidence_level_flag = True and and prescribe a value to change_in_rel_error (if applicable)

- **General approach:**
  - If using *crnt4sbml.GeneralApproach.run_optimization()* set confidence_level_flag = True and prescribe a value to change_in_rel_error (if applicable)

CHAPTER 18

Generating Presentable C-graphs

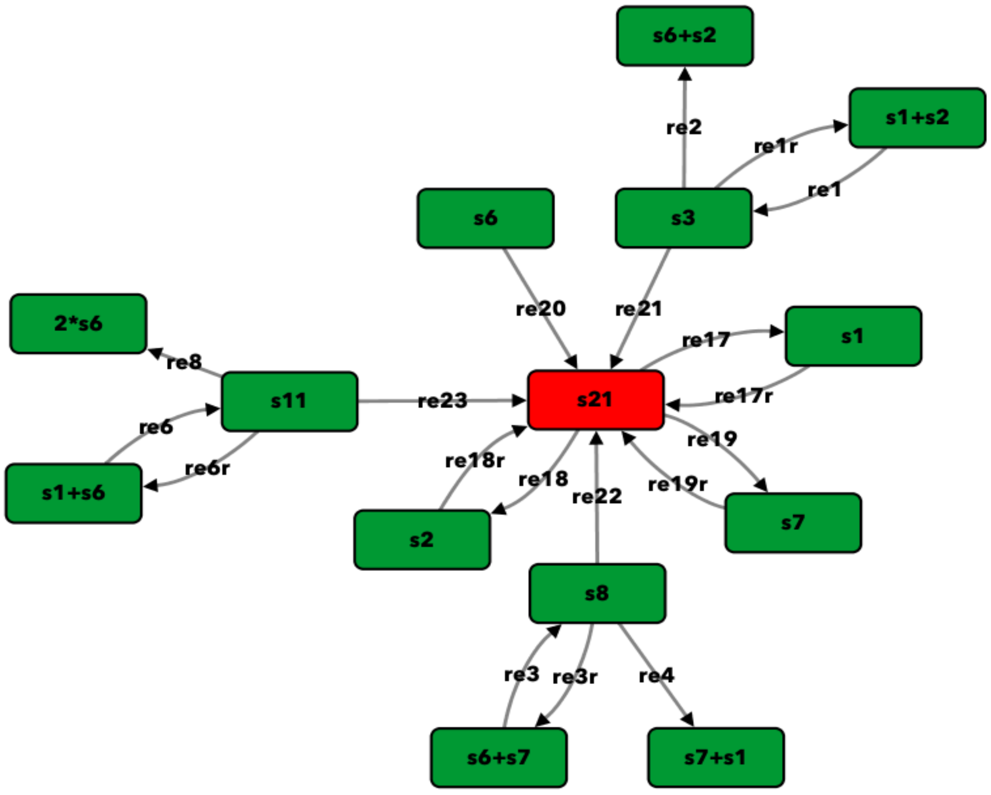
In practice complex networks can be difficult to display in terms of the CellDesigner format. For this reason, it is usually simpler to present networks in terms of C-graphs. Although CRNT4SBML provides the functions *crnt4sbml.CRNT.plot_c_graph()* and *crnt4sbml.CRNT.plot_save_c_graph()* to plot and 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.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

c = crnt4sbml.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 and navigating to the menu bar selecting File -> Import -> Network from File… then selecting network.graphml from the appropriate directory. We can then import the `CRNT4SBML Cytoscape Style` by navigating to the menu bar selecting File -> Import -> Styles from File … then selecting crnt4sbml_cytoscape_style.xml from the appropriate directory. Once the style has been imported, we can use this style by selecting "Style" in the Control Panel and selecting "CRNT4SBML Style" in the Current Style drop down box. Using the CRNT4SBML Style leads to the following C-graph.



CHAPTER 19

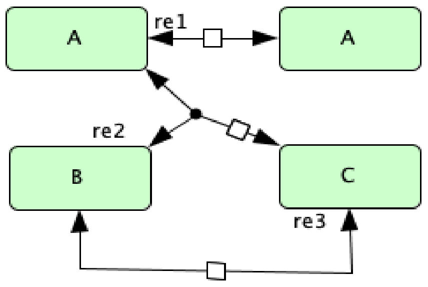
Further Examples

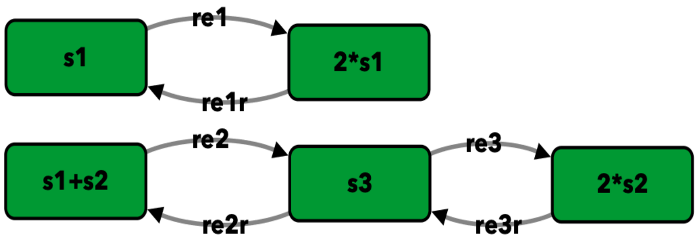
In this section we present multiple examples for the mass conservation, semi-diffusive, and general approaches. In addition to this, we provide some examples satisfying the deficiency theorems. Before each example we depict the CellDesigner layout and C-graph generated using the instructions in *Generating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

Contents

19.1 Low Deficiency Approach

19.1.1 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:

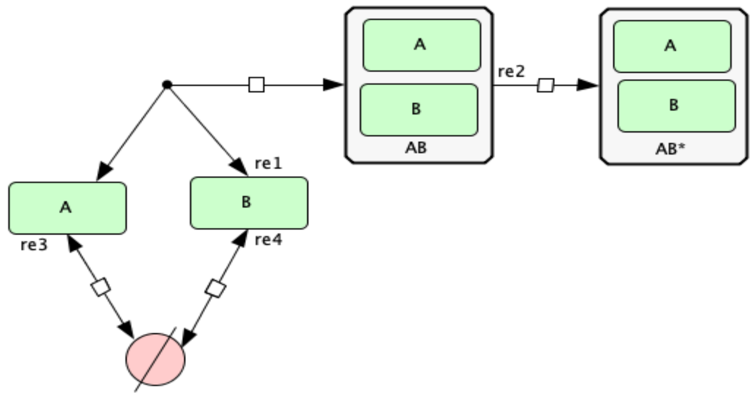
```
Number of species: 3
Number of complexes: 5
Number of reactions: 6
Network deficiency: 0


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
```

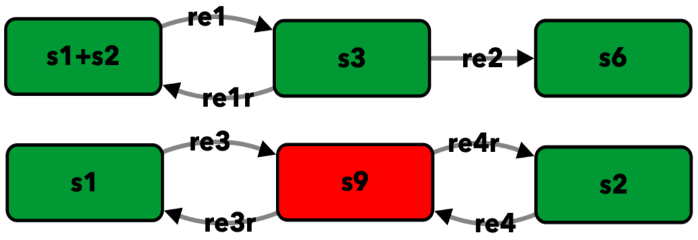
(continues on next page)

(continued from previous page)

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

The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.
Network satisfies one of the low deficiency theorems.
One should not run the optimization-based methods.
```

19.1.2 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:

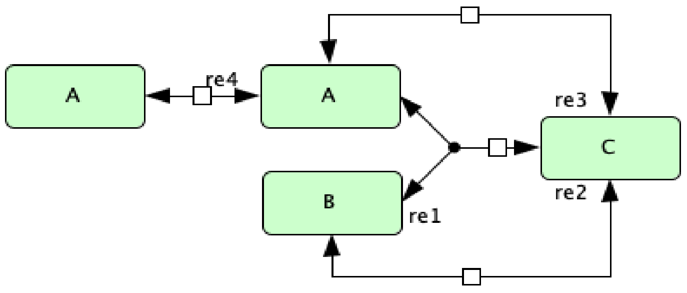
```
Number of species: 4
Number of complexes: 6
Number of reactions: 7
Network deficiency: 0


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
```

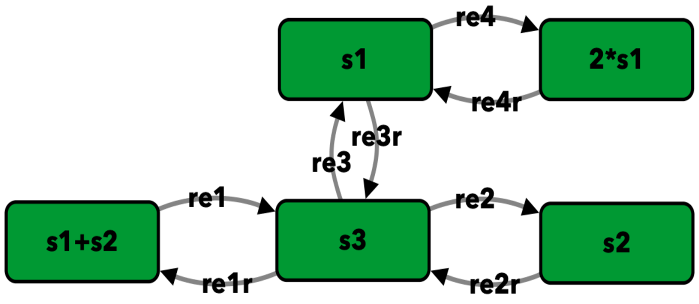
(continues on next page)

(continued from previous page)

```
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 the Deficiency One Theorem, multistability cannot be␣
↪excluded.
Network satisfies one of the low deficiency theorems.
One should not run the optimization-based methods.
```

19.1.3 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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
```

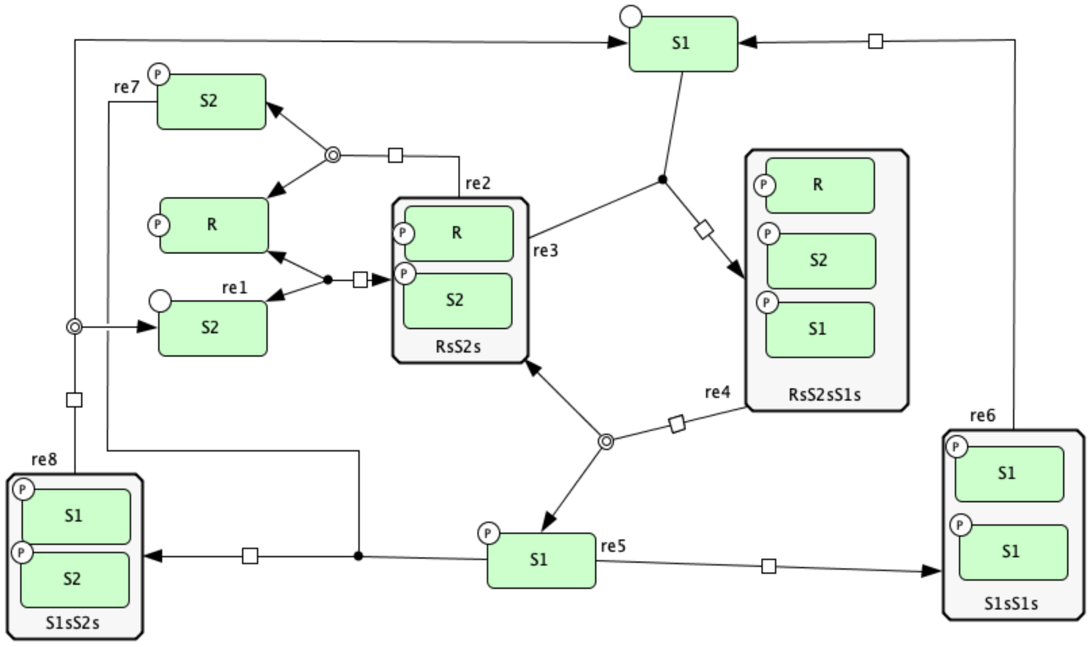
(continues on next page)

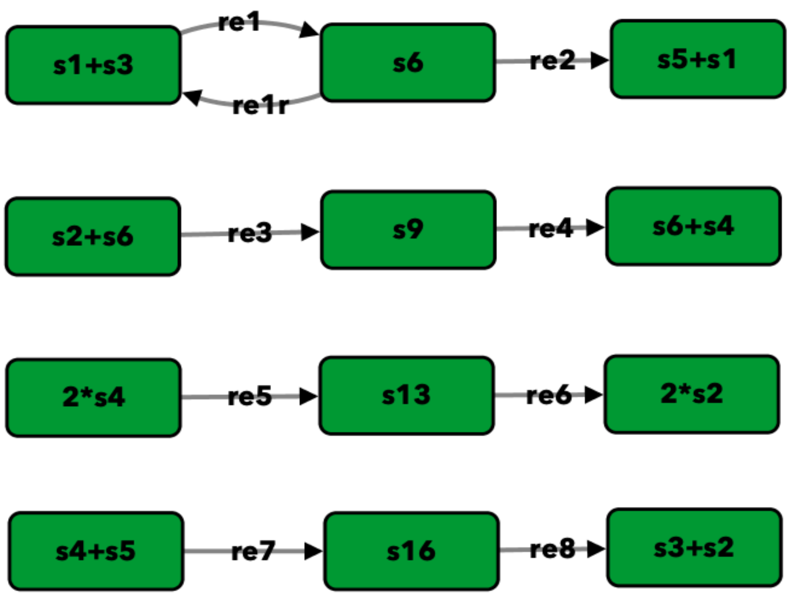
(continued from previous page)

```
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.
```

19.2 Mass Conservation Approach

19.2.1 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:

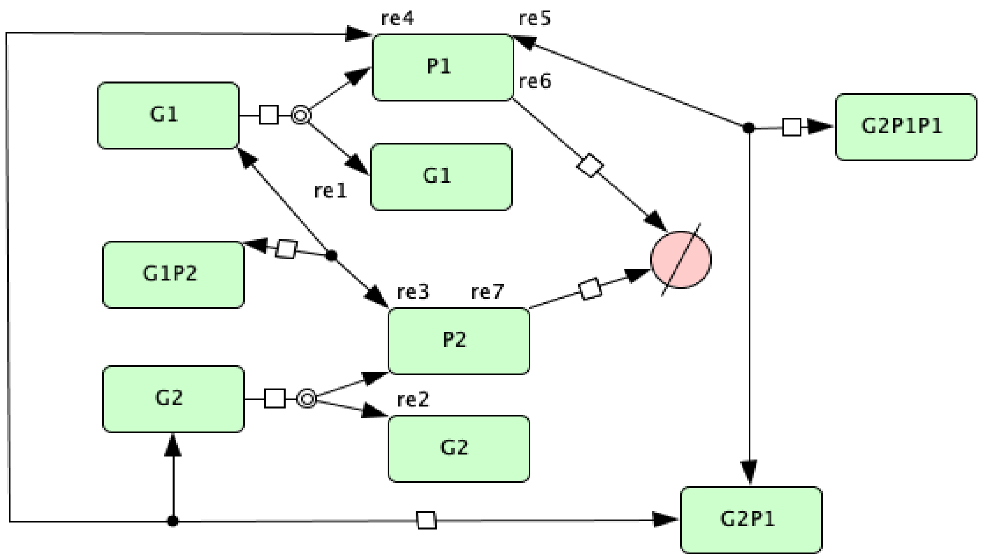
```
Number of species: 9
Number of complexes: 12
Number of reactions: 9
Network deficiency: 2


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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 3.3645559999999994
Decision Vector:
[re1, re1r, re2, re3, re4, re5, re6, re7, re8, s3, s2, s4]

Species for concentration bounds:
[s1, s6, s5, s9, s13, s16]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 42.63995385169983
```

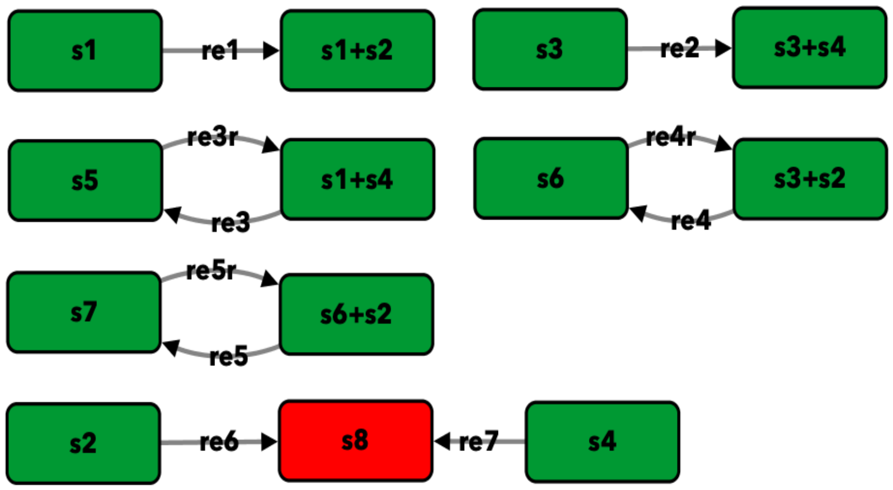
(continues on next page)

(continued from previous page)

```
Running the multistart optimization method ...
Elapsed time for multistart method: 109.29019284248352

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 16.06424617767334

Smallest value achieved by objective function: 0.0
15 point(s) passed the optimization criteria.
Number of multistability plots found: 2
Elements in params_for_global_min that produce multistability:
[0, 12]
```

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

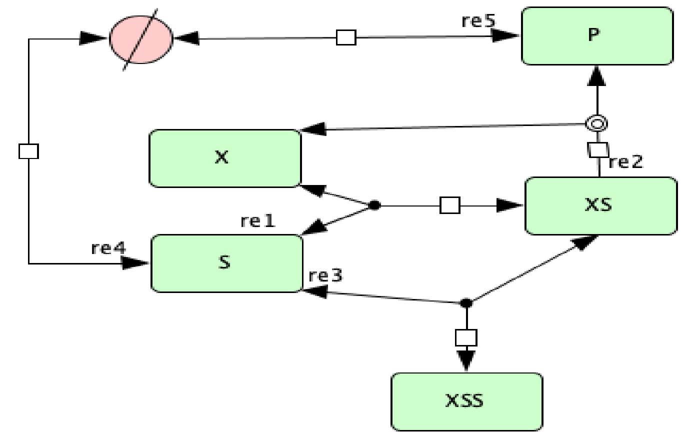


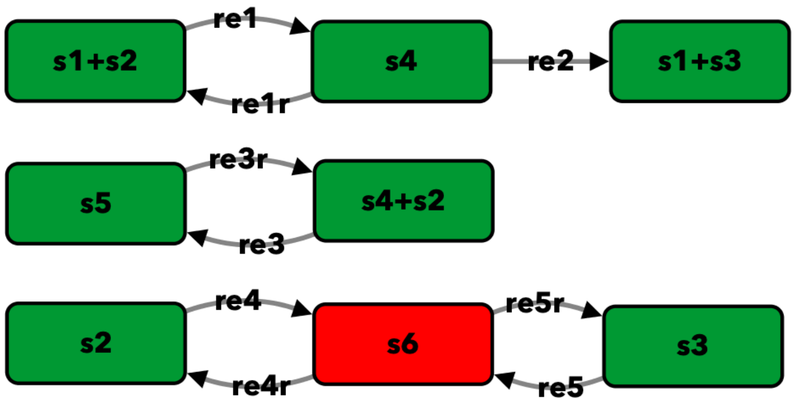


To run this example download the SBML `file` and script `run_irene2014`. After running this script we obtain the following output:

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

The network does not satisfy the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 1.772672
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: 25.66311025619507

Running the multistart optimization method ...
Elapsed time for multistart method: 119.89791989326477

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 100.14113593101501

Smallest value achieved by objective function: 0.0
93 point(s) passed the optimization criteria.
Number of multistability plots found: 21
Elements in params_for_global_min that produce multistability:
[1, 3, 9, 11, 15, 21, 24, 27, 32, 35, 40, 45, 56, 62, 70, 79, 80, 83, 84, 85, 88]
```

19.2.3 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:

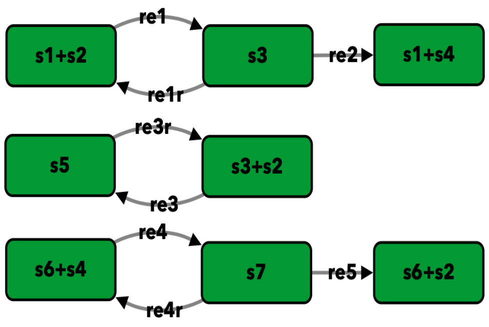
```
Number of species: 5
Number of complexes: 8
Number of reactions: 9
Network deficiency: 1


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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.
```

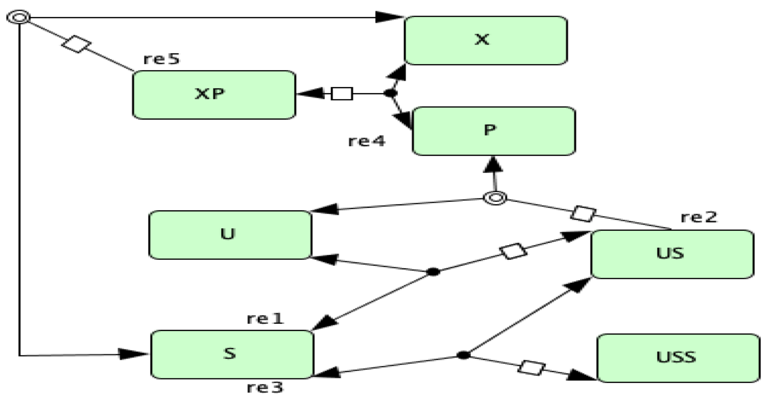
(continues on next page)

(continued from previous page)

```
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.715592
Decision Vector:
[re1, re1r, re2, re3, re3r, re4, re4r, re5, re5r, s2]

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

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 15.607332229614258

Running the multistart optimization method ...
Elapsed time for multistart method: 66.42637610435486

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 72.26282095909119

Smallest value achieved by objective function: 0.0
84 point(s) passed the optimization criteria.
Number of multistability plots found: 48
Elements in params_for_global_min that produce multistability:
[3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 19, 21, 22, 23, 27, 30, 31, 34, 35, 36, 37,␣
↪38, 39, 41, 42, 47, 48, 50, 51, 54, 55, 56, 57, 59, 60, 61, 64, 65, 66, 68, 69, 72,␣
↪73, 74, 75, 77, 83]
```

19.2.4 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:

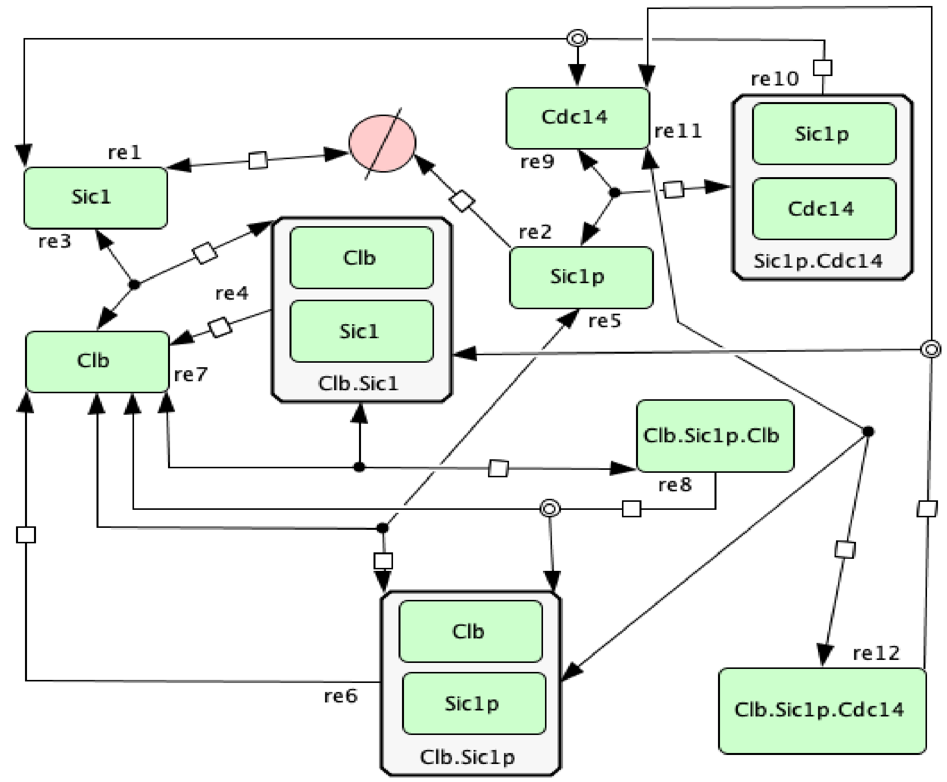
```
Number of species: 7
Number of complexes: 8
Number of reactions: 8
Network deficiency: 1


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

The network does not satisfy the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.7393859999999997
Decision Vector:
[re1, re1r, re2, re3, re3r, re4, re4r, re5, s2, s6, s7]

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

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 13.359651803970337

Running the multistart optimization method ...
Elapsed time for multistart method: 103.19853806495667

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 90.50077891349792

Smallest value achieved by objective function: 0.0
96 point(s) passed the optimization criteria.
```

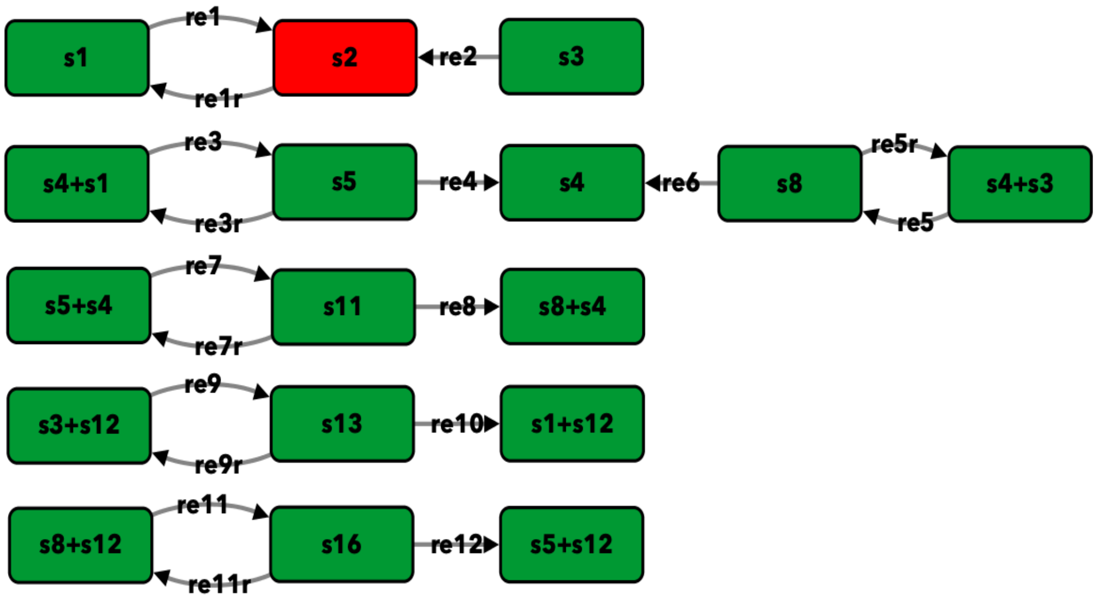
(continues on next page)

(continued from previous page)

```
Number of multistability plots found: 14
Elements in params_for_global_min that produce multistability:
[1, 22, 25, 33, 37, 42, 51, 53, 57, 58, 59, 64, 74, 87]
```

19.2.5 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:

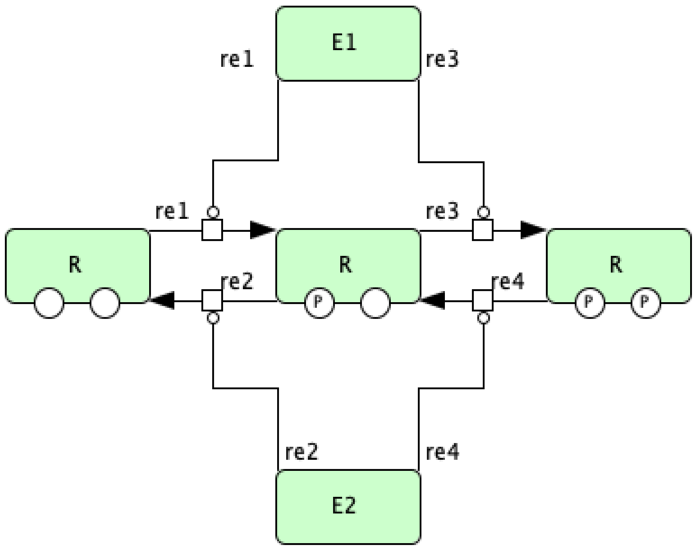
```
Number of species: 9
Number of complexes: 17
Number of reactions: 18
Network deficiency: 5


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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 260.415536
```

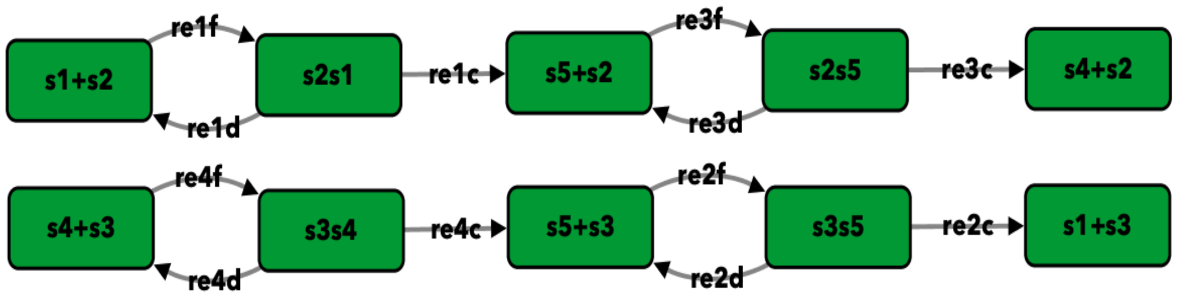
(continues on next page)

(continued from previous page)

```
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: 73.16450190544128

Running the multistart optimization method ...
Elapsed time for multistart method: 800.0220079421997

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 15.878800868988037

Smallest value achieved by objective function: 0.0
13 point(s) passed the optimization criteria.
Number of multistability plots found: 11
Elements in params_for_global_min that produce multistability:
[0, 1, 2, 3, 5, 6, 7, 8, 10, 11, 12]
```

19.2.6 Double phosphorylation in signal transduction of [double_phos]





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

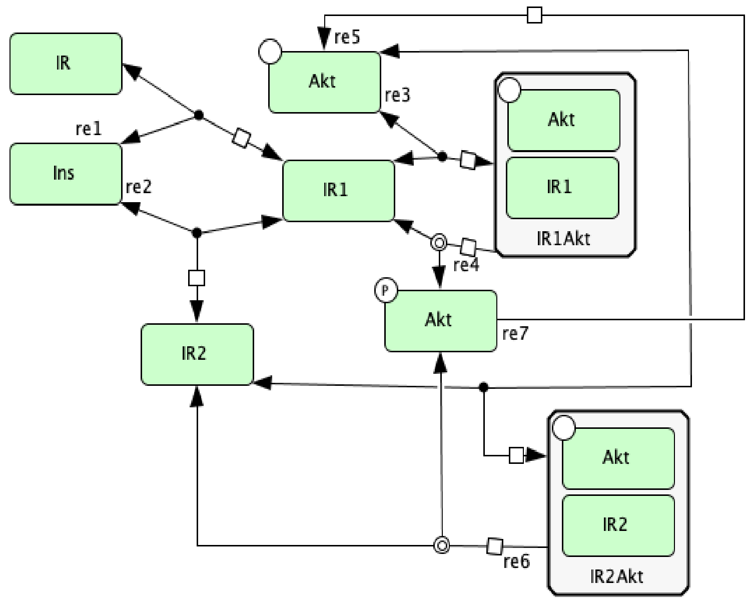
```
Number of species: 9
Number of complexes: 10
Number of reactions: 12
Network deficiency: 2


Reaction graph of the form
reaction -- reaction label:
s1+s2 -> s2s1  --  re1f
s2s1 -> s1+s2  --  re1d
s2s1 -> s5+s2  --  re1c
s5+s3 -> s3s5  --  re2f
s3s5 -> s5+s3  --  re2d
s3s5 -> s1+s3  --  re2c
s5+s2 -> s2s5  --  re3f
s2s5 -> s5+s2  --  re3d
s2s5 -> s4+s2  --  re3c
s4+s3 -> s3s4  --  re4f
s3s4 -> s4+s3  --  re4d
s3s4 -> s5+s3  --  re4c

The network does not satisfy the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 5.184272
Decision Vector:
[re1f, re1d, re1c, re2f, re2d, re2c, re3f, re3d, re3c, re4f, re4d, re4c, s2, s3, s3s4]

Species for concentration bounds:
[s1, s5, s2s1, s3s5, s4, s2s5]

Running feasible point method for 100 iterations ...
Elapsed time for feasible point method: 18.401470184326172

Running the multistart optimization method ...
Elapsed time for multistart method: 95.46931576728821

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 372.1889531612396

Smallest value achieved by objective function: 0.0
```

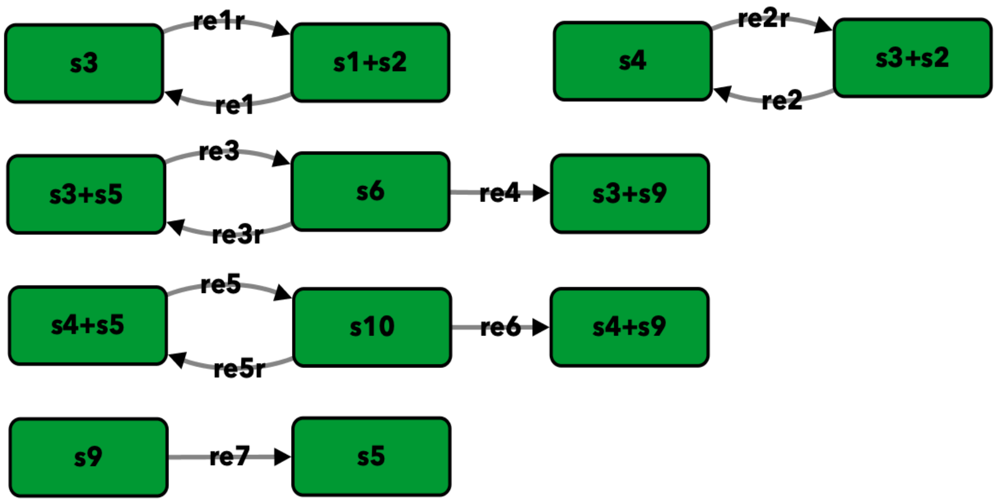
(continues on next page)

(continued from previous page)

```
97 point(s) passed the optimization criteria.
Number of multistability plots found: 89
Elements in params_for_global_min that produce multistability:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
↪ 25, 26, 27, 28, 29, 30, 31,
 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,␣
↪54, 55, 56, 57, 58, 59, 60,
 61, 62, 64, 65, 66, 67, 69, 70, 71, 72, 73, 74, 75, 76, 77, 79, 80, 81, 82, 83, 84,␣
↪87, 88, 90, 91, 92, 93, 94, 95, 96]
```

19.2.7 Double insulin binding

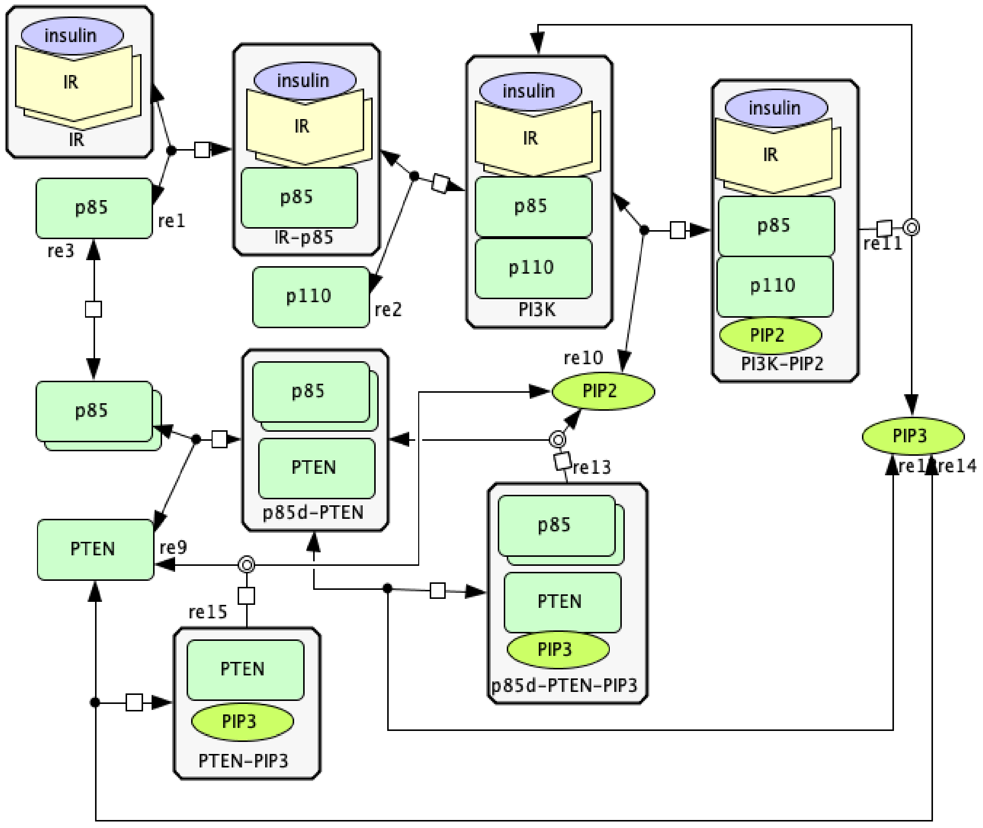


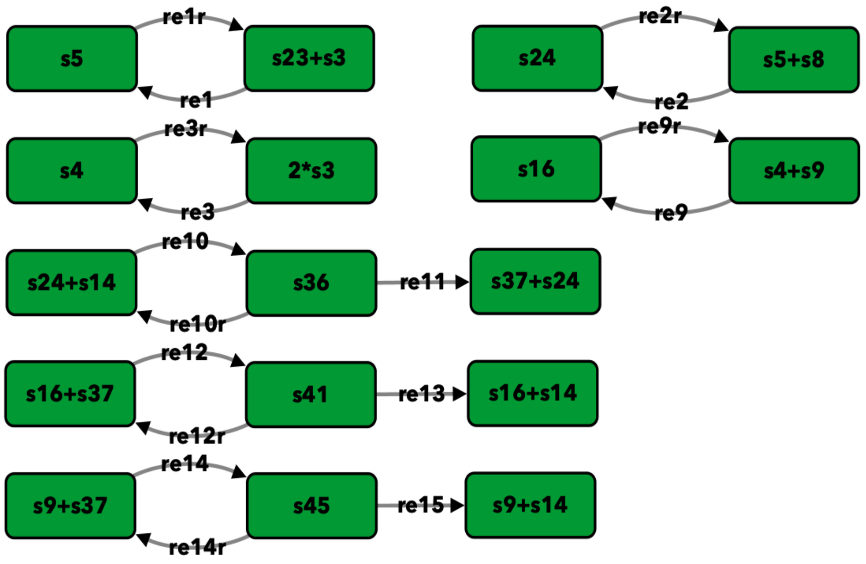


To run this example download the SBML `file` and script `run_double_insulin_binding`. After running this script we obtain the following output:

```
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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 2.2847300000000006
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: 25.920205116271973

Running the multistart optimization method ...
Elapsed time for multistart method: 94.97992706298828

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 652.6215398311615

Smallest value achieved by objective function: 2.3317319454459066e-31
67 point(s) passed the optimization criteria.
Number of multistability plots found: 2
Elements in params_for_global_min that produce multistability:
[8, 38]
```

19.2.8 p85-p110-PTEN





To run this example download the SBML `file` and script `run_p85-p110-PTEN`. After running this script using four cores, we obtain the following output (for more information on running this script in parallel see *Parallel CRNT4SBML*):

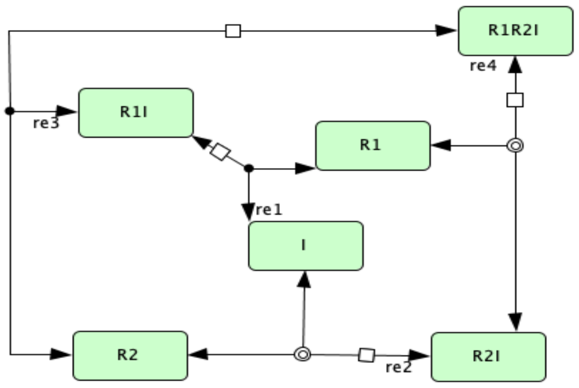
```
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 107.71943200000001
Elapsed time for creating Equilibrium Manifold: 108.786772
Elapsed time for creating Equilibrium Manifold: 108.861678
Elapsed time for creating Equilibrium Manifold: 109.171994

Running feasible point method for 5000 iterations ...
Elapsed time for feasible point method: 2519.281478

Running the multistart optimization method ...
Elapsed time for multistart method: 403.41574900000023


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
2*s3 -> s4  --  re3
s4 -> 2*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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

Decision Vector:
[re1, re1r, re2, re2r, re3, re3r, re9, re9r, re10, re10r, re11, re12, re12r, re13,␣
↪re14, re14r, re15, s3, s8, s9, s14, s37]

Species for concentration bounds:
[s23, s5, s24, s4, s16, s36, s41, s45]

A parallel version of numerical continuation is not available.
Numerical continuation will be ran using only one core.
For your convenience, the provided parameters have been saved in the current␣
↪directory under the name params.npy.
```

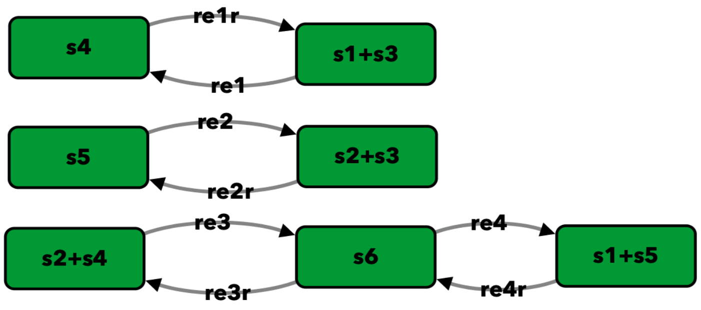
(continues on next page)

(continued from previous page)

```
Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 5766.086745023727

Smallest value achieved by objective function: 0.0
429 point(s) passed the optimization criteria.
Number of multistability plots found: 5
Elements in params_for_global_min that produce multistability:
[171, 191, 213, 272, 296]
```

19.2.9 Closed version of Figure 4B from [irene]





To run this example download the SBML `file` and script `run_Fig4B_closed`. After running this script using four cores, we obtain the following output (for more information on running this script in parallel see *Parallel CRNT4SBML*):

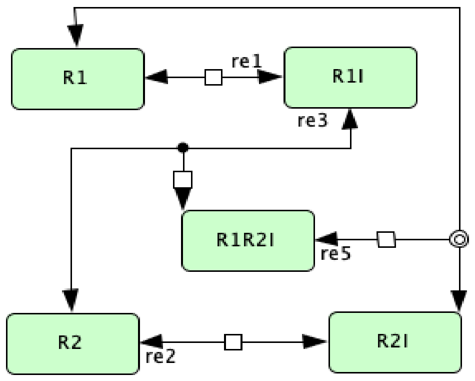
```
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 1.2114520000000004
Elapsed time for creating Equilibrium Manifold: 1.2372060000000005
Elapsed time for creating Equilibrium Manifold: 1.229298
Elapsed time for creating Equilibrium Manifold: 1.2412400000000003

Running feasible point method for 10000 iterations ...
Elapsed time for feasible point method: 518.759626

Running the multistart optimization method ...
Elapsed time for multistart method: 2561.635341
```

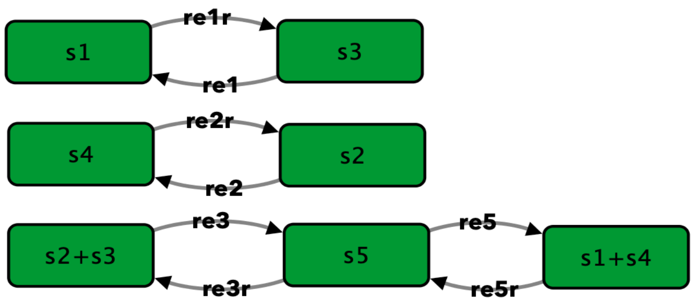
(continues on next page)

(continued from previous page)

```
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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

Decision Vector:
[re1, re1r, re2, re2r, re3, re3r, re4, re4r, s3, s5, s2]

Species for concentration bounds:
[s1, s4, s6]
Smallest value achieved by objective function: 2.454796889817468e-10
0 point(s) passed the optimization criteria.
```

19.2.10 Closed version of Figure 4C from [irene]





To run this example download the SBML `file` and script `run_Fig4C_closed`. After running this script using four cores, we obtain the following output (for more information on running this script in parallel see *Parallel CRNT4SBML*):

```
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Creating Equilibrium Manifold ...
Elapsed time for creating Equilibrium Manifold: 0.9796280000000004
Elapsed time for creating Equilibrium Manifold: 0.9905299999999997
Elapsed time for creating Equilibrium Manifold: 0.997398
Elapsed time for creating Equilibrium Manifold: 0.9981960000000001

Running feasible point method for 10000 iterations ...
Elapsed time for feasible point method: 236.957728

Running the multistart optimization method ...
Elapsed time for multistart method: 2115.088291


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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

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

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

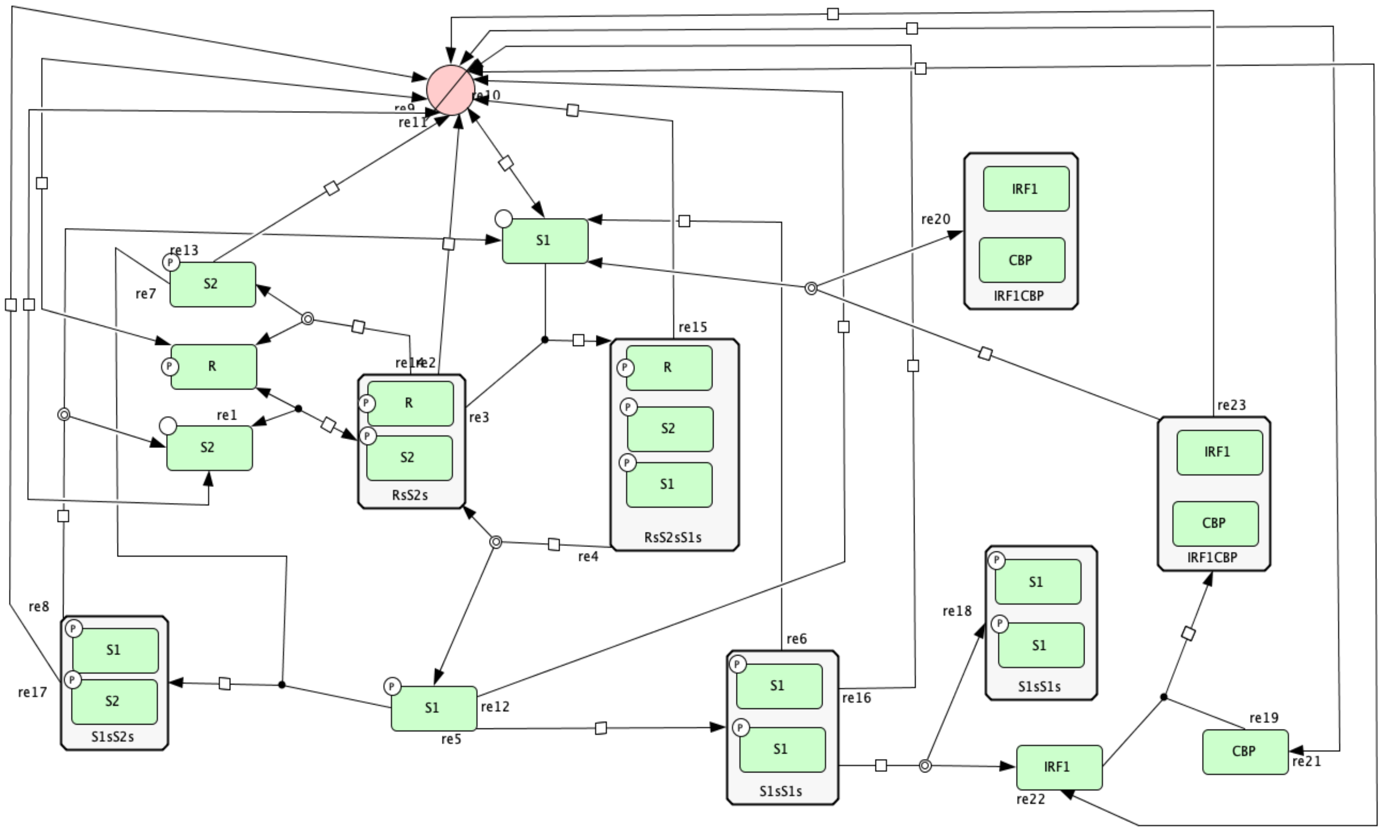
(continues on next page)

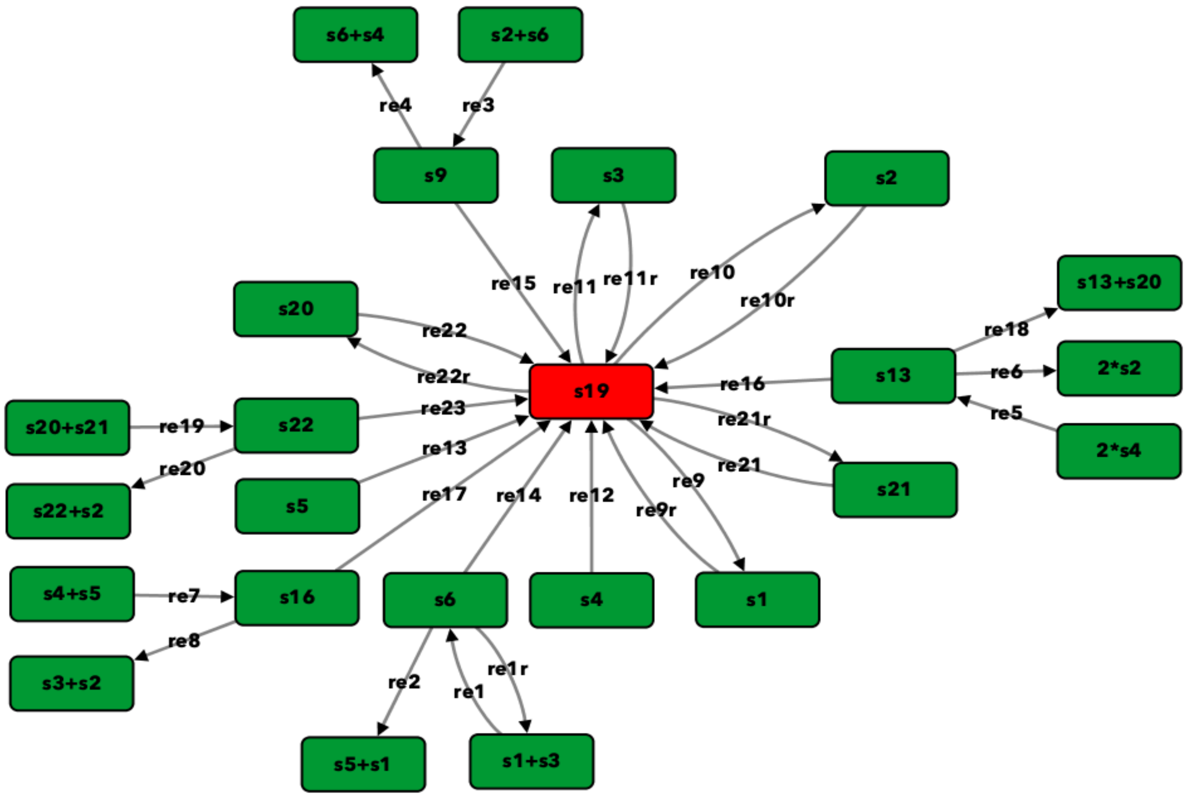
(continued from previous page)

```
Smallest value achieved by objective function: 1.2913762450176939e-09
0 point(s) passed the optimization criteria.
```

19.3 Semi-diffusive Approach

19.3.1 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:

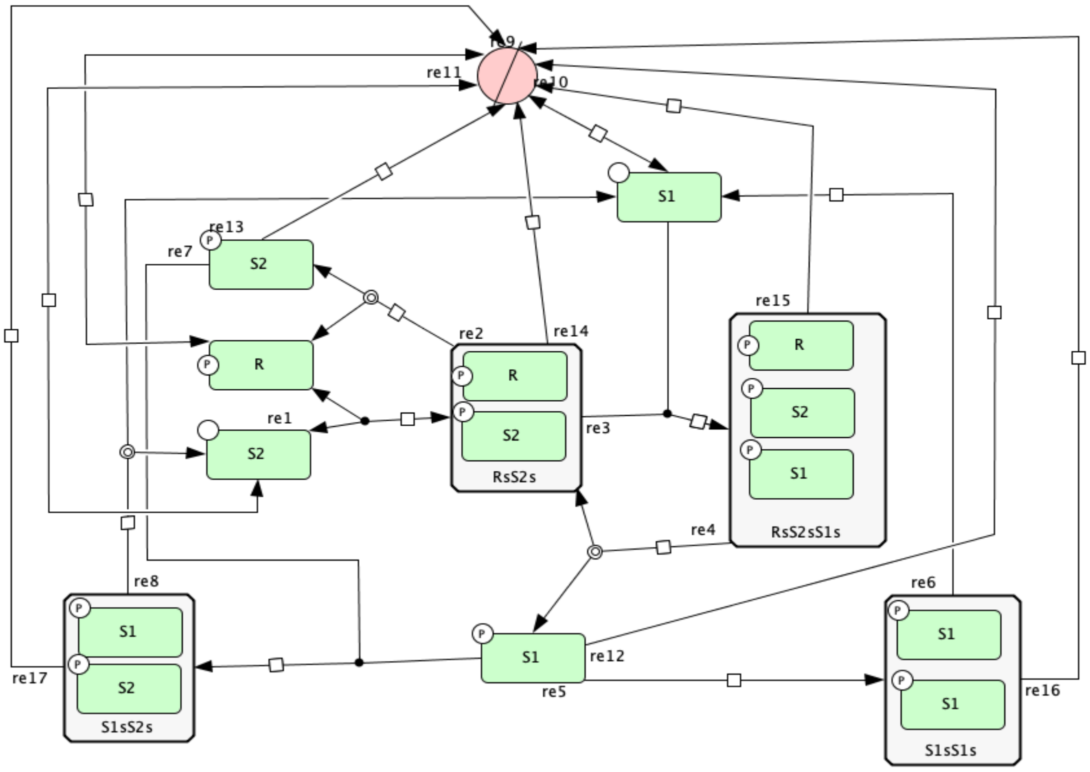
```
Number of species: 12
Number of complexes: 24
Number of reactions: 29
Network deficiency: 11


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 -> s25  --  re5
s25 -> 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
```

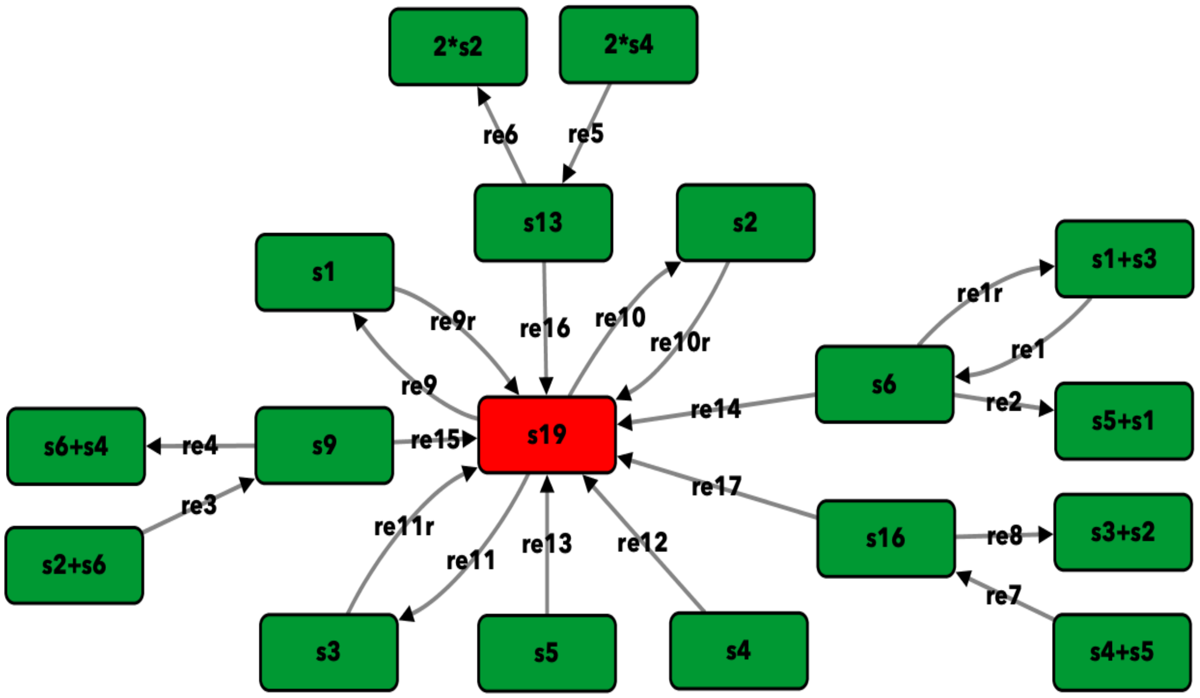
(continues on next page)

(continued from previous page)

```
s25 -> s19  --  re16
s16 -> s19  --  re17
s25 -> s25+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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

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', 's3', 's2', 's20', 's21']

Non key species:
['s6', 's5', 's9', 's4', 's25', 's16', 's22']

Boundary species:
['s19']

Running feasible point method for 50 iterations ...
Elapsed time for feasible point method: 14.352675676345825

Running the multistart optimization method ...
Elapsed time for multistart method: 352.3979892730713

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 42.74703788757324

Smallest value achieved by objective function: 0.0
22 point(s) passed the optimization criteria.
Number of multistability plots found: 4
Elements in params_for_global_min that produce multistability:
[0, 1, 3, 16]
```

19.3.2 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:

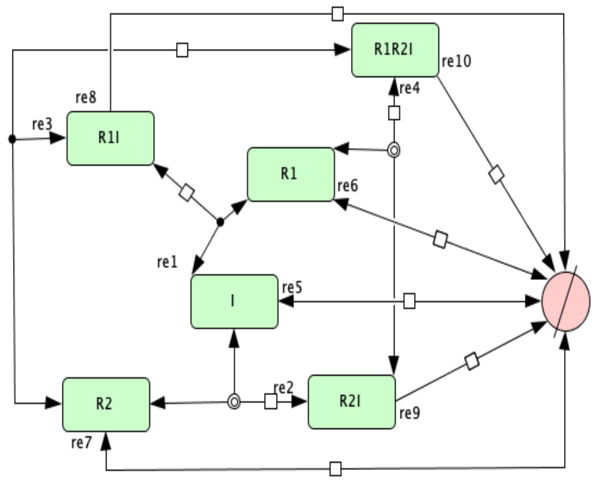
```
Number of species: 9
Number of complexes: 18
Number of reactions: 21
Network deficiency: 8


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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

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', 's3', 's2']

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

Boundary species:
['s19']

Running feasible point method for 500 iterations ...
Elapsed time for feasible point method: 40.84808683395386

Running the multistart optimization method ...
Elapsed time for multistart method: 597.4433598518372
```

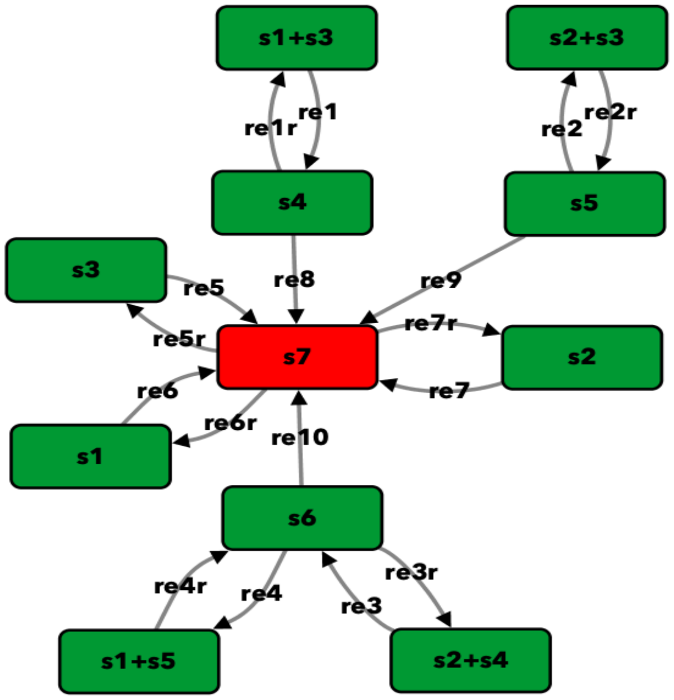
(continues on next page)

(continued from previous page)

```
Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 1777.679843902588

Smallest value achieved by objective function: 0.0
108 point(s) passed the optimization criteria.
Number of multistability plots found: 1
Elements in params_for_global_min that produce multistability:
[85]
```

19.3.3 Figure 4B from [irene]



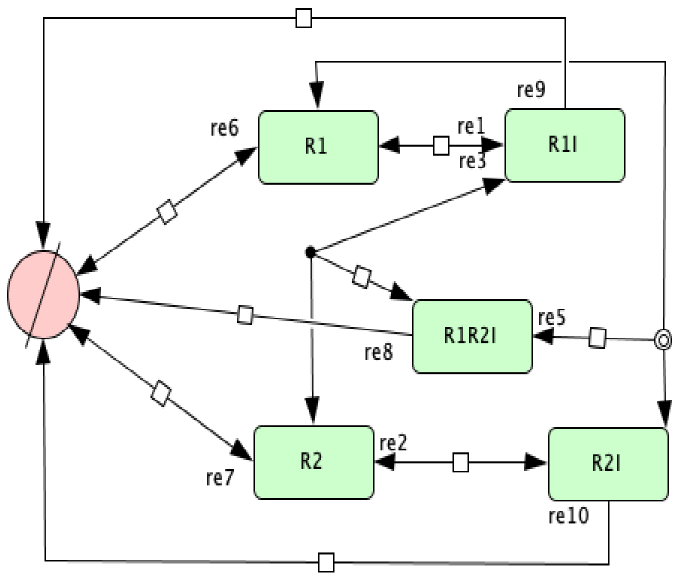


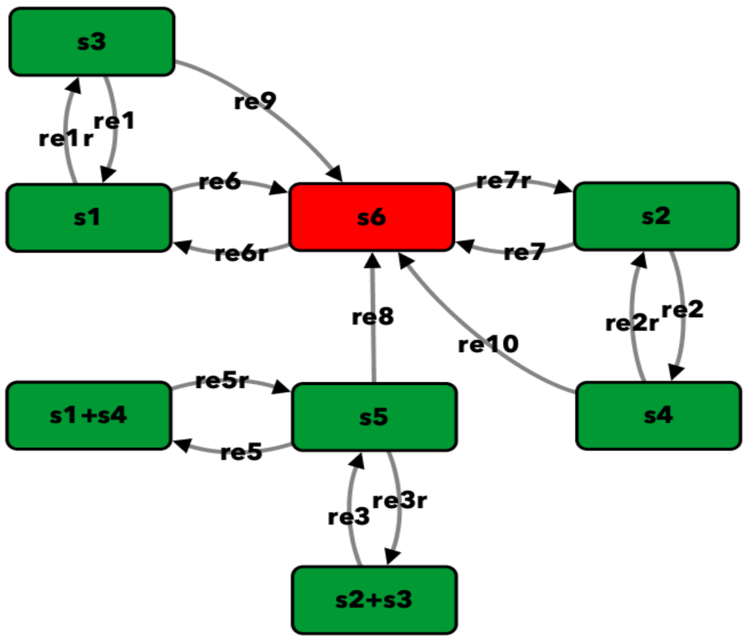
To run this example download the SBML `file` and script `run_Fig4B_open`. After running this script using four cores, we obtain the following output (for more information on running this script in parallel see *Parallel*

CRNT4SBML):

```
Running feasible point method for 10000 iterations ...
Elapsed time for feasible point method: 73.587205

Running the multistart optimization method ...
Elapsed time for multistart method: 3675.938109


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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

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', 's3', 's2']

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

Boundary species:
['s7']
Smallest value achieved by objective function: 2.3045037796933692e-10
0 point(s) passed the optimization criteria.
```

19.3.4 Figure 4C from [irene]





To run this example download the SBML `file` and script `run_Fig4C_open`. After running this script using four cores, we obtain the following output (for more information on running this script in parallel see *Parallel CRNT4SBML*):

```
Running feasible point method for 10000 iterations ...
Elapsed time for feasible point method: 57.548688

Running the multistart optimization method ...
Elapsed time for multistart method: 1432.020307


Number of species: 5
```

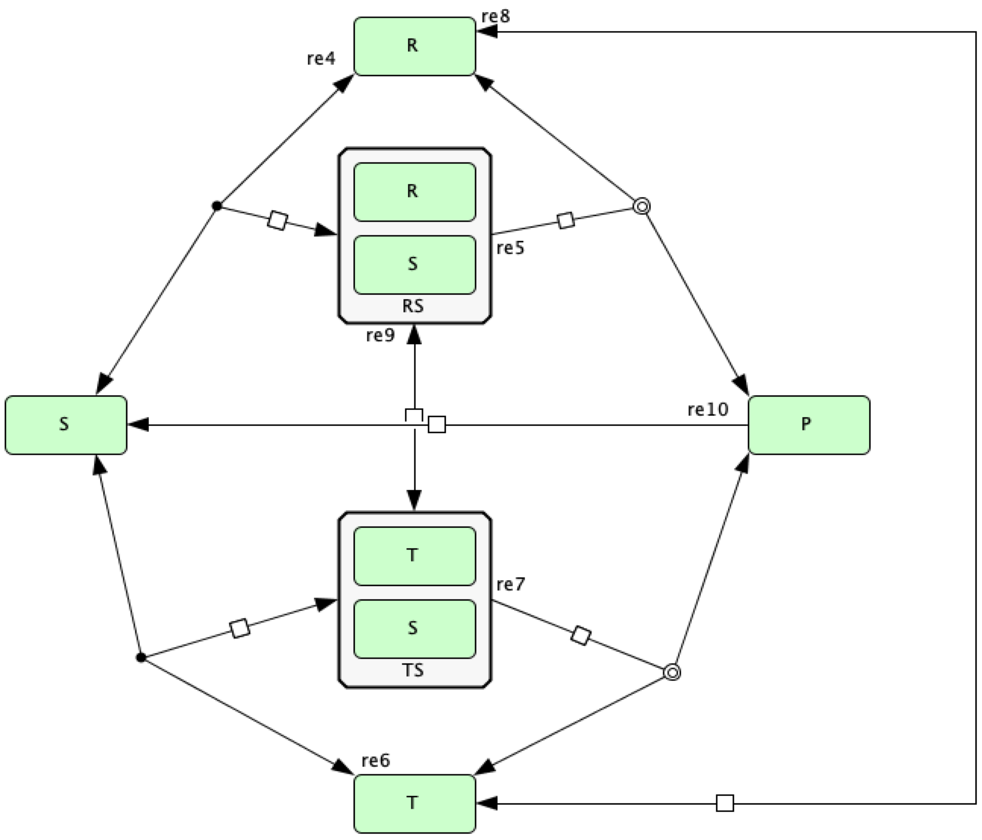
(continues on next page)

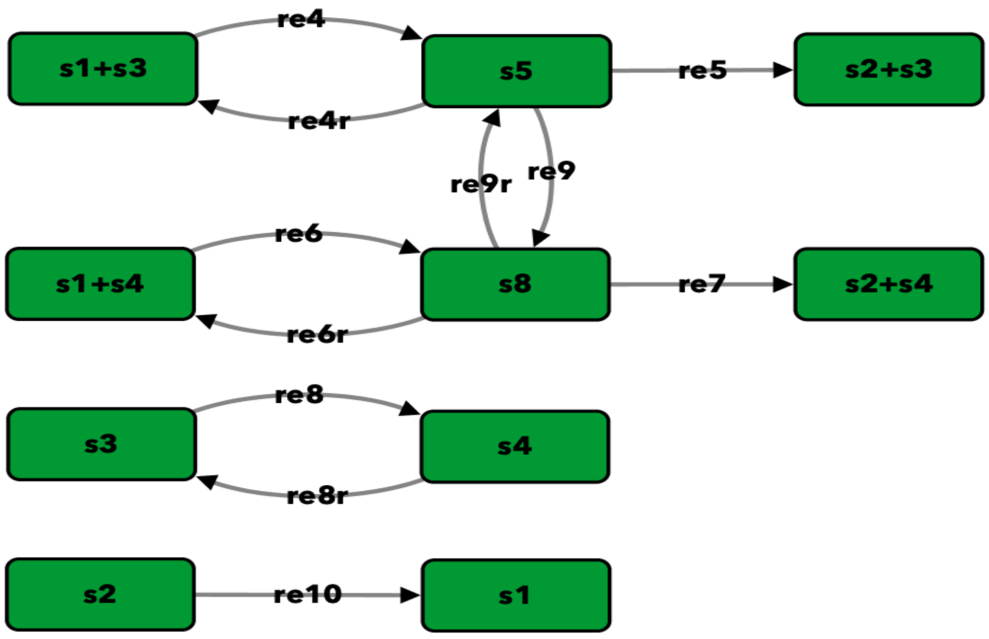
(continued from previous page)

```
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
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 the Deficiency Zero Theorem, multistability cannot be␣
↪excluded.
The network does not satisfy the Deficiency One Theorem, multistability cannot be␣
↪excluded.

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']
Smallest value achieved by objective function: 4.5692676949897973e-10
0 point(s) passed the optimization criteria.
```

19.4 General Approach

19.4.1 Song model of [song_paper]





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

```
Number of species: 6
Number of complexes: 10
Number of reactions: 11
Network deficiency: 3


Reaction graph of the form
reaction -- reaction label:
s1+s3 -> s5  --  re4
s5 -> s1+s3  --  re4r
s5 -> s2+s3  --  re5
s1+s4 -> s8  --  re6
s8 -> s1+s4  --  re6r
s8 -> s2+s4  --  re7
s3 -> s4  --  re8
s4 -> s3  --  re8r
s5 -> s8  --  re9
s8 -> s5  --  re9r
s2 -> s1  --  re10

[re4, re4r, re5, re6, re6r, re7, re8, re8r, re9, re9r, re10, s1, s3, s5, s2, s4, s8]

Running the multistart optimization method ...
Elapsed time for multistart method: 1228.3208582401276

Running continuity analysis ...
Elapsed time for continuity analysis in seconds: 28.140807151794434

Smallest value achieved by objective function: 0.0
5 point(s) passed the optimization criteria.
Number of multistability plots found: 2
Elements in params_for_global_min that produce multistability:
[1, 4]
```

CHAPTER 20

An Example User Case Scenario

In this section we describe a general process that can be followed if one would like to simulate the ODE system or conduct stability analysis of those reaction networks that produce bistability, as determined by the mass conservation approach.

20.1 Serializing Important Information

20.1.1 Storing Important Information

Given the act of performing the numerical optimization and continuation routines can take a significant amount of time for highly complex networks, we will describe how to store the necessary information needed to simulate the ODEs. To complete this process one will need to install dill, a Python library that extends Python's pickle module for serializing and de-serializing Python objects. A simple way to do this is by using pip:

```
$ pip install dill
```

Using dill and Numpy, we can now save the parameter sets produced by optimization and the variables and values constructed by continuation that will be necessary when simulating the ODE system of the network. This is done as follows:

```
import crnt4sbml
import numpy
import sympy
import dill

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

opt = network.get_mass_conservation_approach()

bounds, concentration_bounds = opt.get_optimization_bounds()

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

(continues on next page)

(continued from previous page)

```
                                                                                  concentration_
↪bounds=concentration_bounds)

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

multistable_param_ind, plot_specifications = opt.run_greedy_continuity_
↪analysis(species="s15",
                                                                                             ␣
↪parameters=params_for_global_min,
                                                                                             auto_
↪parameters={'PrincipalContinuationParameter': 'C3'})

odes = network.get_c_graph().get_ode_system()
sympy_reactions = [sympy.Symbol(i, positive=True) for i in network.get_c_graph().get_
↪reactions()]
sympy_species = [sympy.Symbol(i, positive=True) for i in network.get_c_graph().get_
↪species()]
concentration_funs = opt.get_concentration_funs()
BT_matrix = network.get_c_graph().get_b()

important_variables = [odes, sympy_reactions, sympy_species, concentration_funs, BT_
↪matrix, multistable_param_ind,
                       plot_specifications]

dill.settings['recurse'] = True # allows us to pickle the lambdified functions

with open("important_variables.dill", 'wb') as f:
    dill.dump(important_variables, f)
```

Once this code is ran, one will obtain the files “params.npy” and “important_variables.dill”. Here, “params.npy” is a special numpy file that holds the array of decision vectors produced by the optimization routine. The file “important_variables.dill” is a dill file that contains the rest of the information necessary to simulate the ODE system.

20.1.2 Importing Important Information

Once the section above is completed, one can then import the information in the files params.npy and important_variables.dill into a new Python session by creating the following script:

```
import dill
import sympy
import numpy

with open("important_variables.dill", 'rb') as f:
    out = dill.load(f)

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

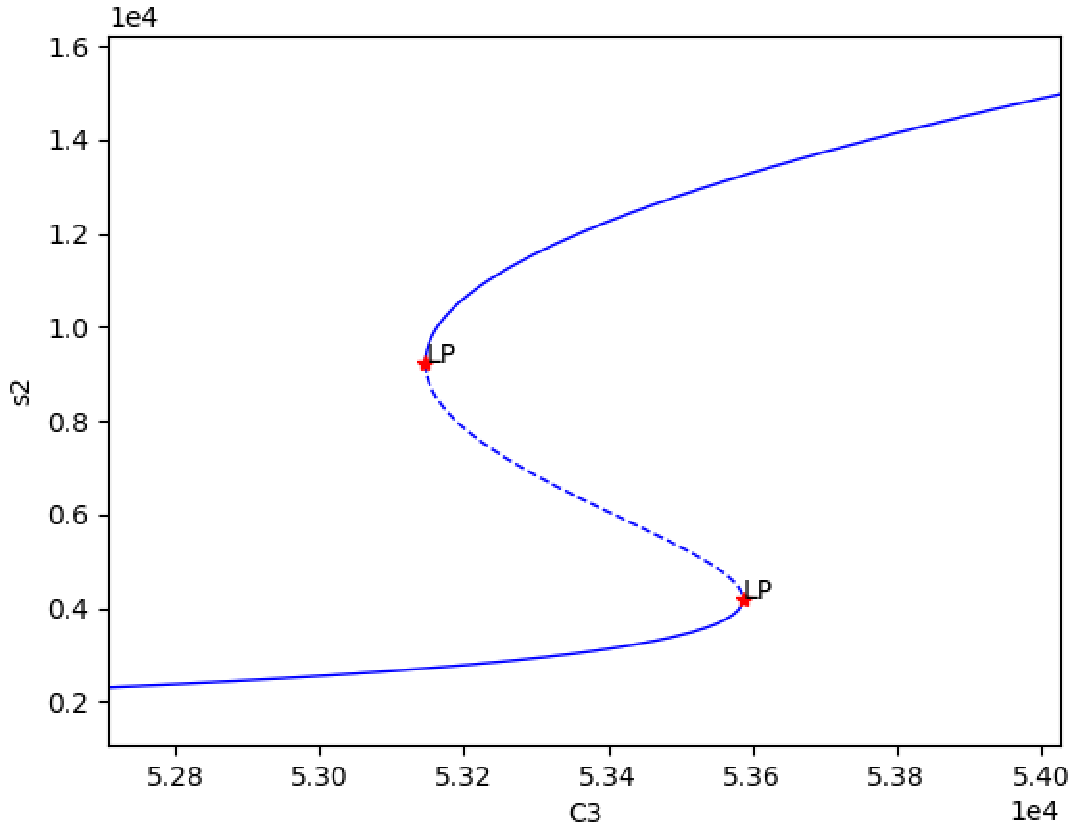
20.2 Simulating the ODE system

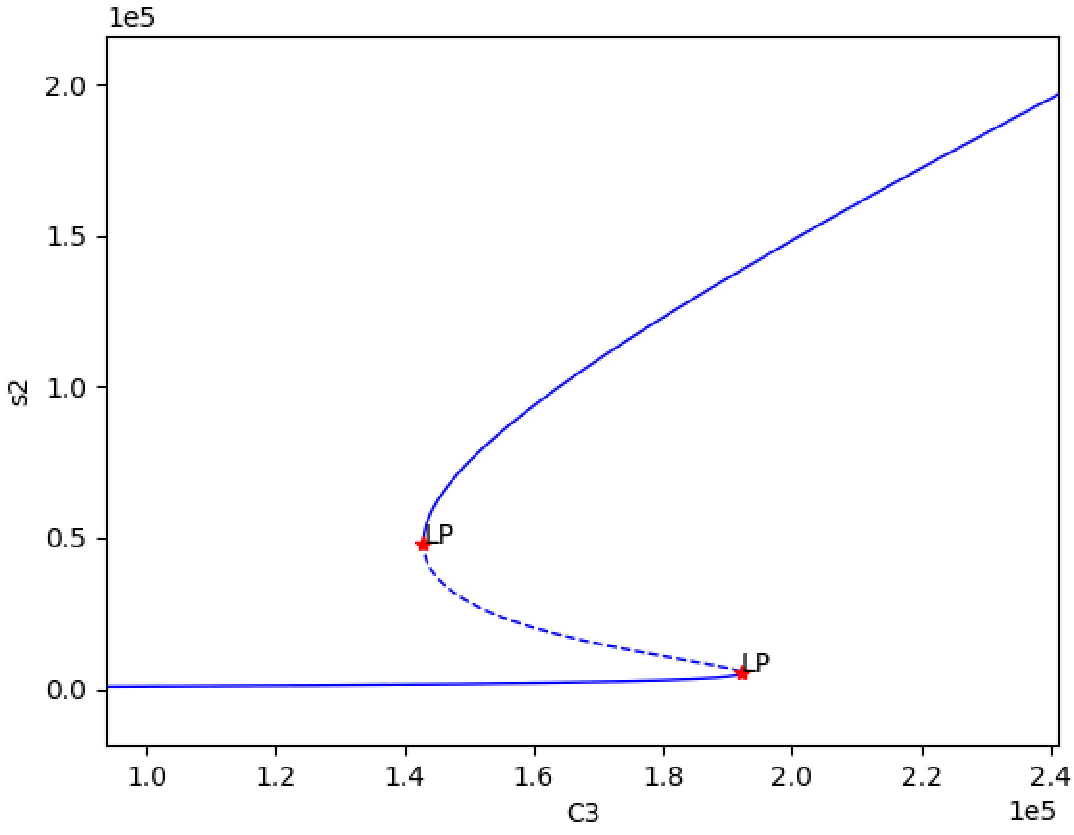
For this section we will be using the files `Script a_full_use_case_scenario.py` and `Model basic_example_1.xml` to demonstrate how one can create nice looking plots that depict the simulation of the ODE system.

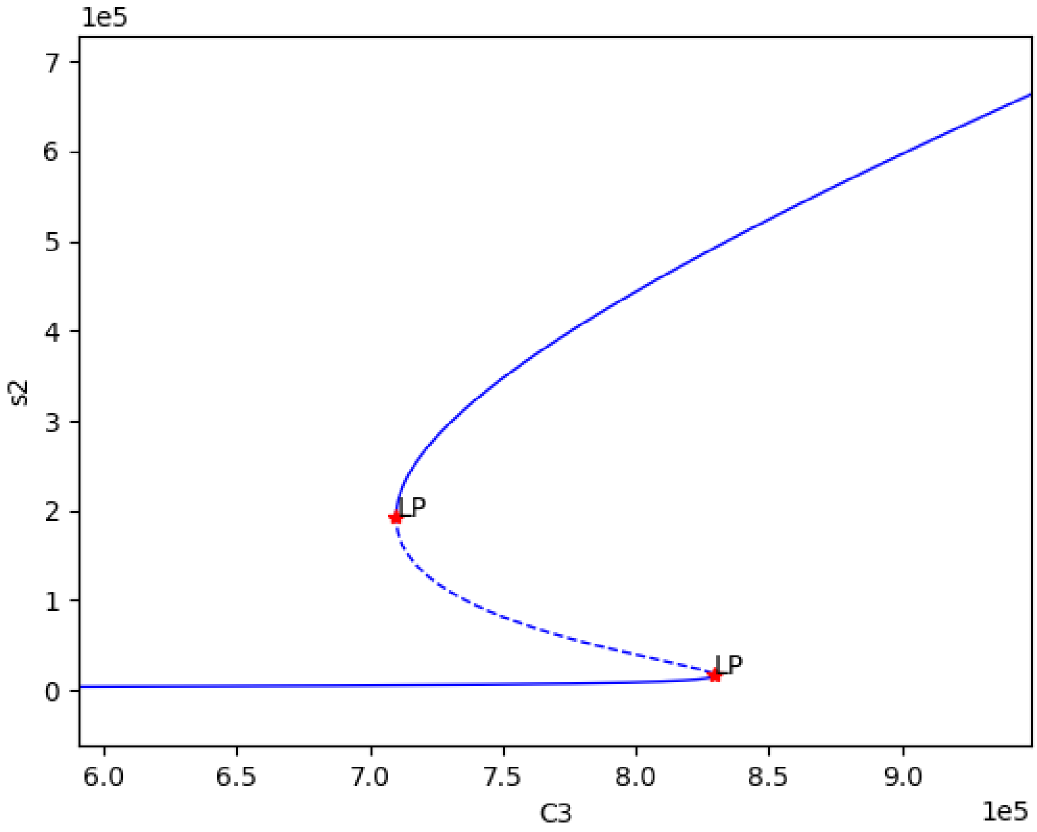
Below we solve for those points that satisfy *det(Jacobian) = 0* using the optimization routine followed by continuation analysis:

```
import crnt4sbml
import numpy
import pandas
import sympy
import scipy.integrate as itg
from plotnine import ggplot, aes, geom_line, ylim, scale_color_distiller, facet_wrap,␣
↪theme_bw, geom_path, geom_point


network = crnt4sbml.CRNT("path/to/basic_example_1.xml")
network.print_biological_reaction_types()

ldt = network.get_low_deficiency_approach()
ldt.report_deficiency_zero_theorem()
ldt.report_deficiency_one_theorem()

# optimization approach
opt = network.get_mass_conservation_approach()
opt.generate_report()

# the decision vector
opt.get_decision_vector()

# this function suggests physiological bounds
bounds, concentration_bounds = opt.get_optimization_bounds()

# overwriting with a narrower or wider range. In this case we are setting narrow␣
↪range for re1c.
bounds[2] = (0.001, 0.01)

# overwriting specie concentration bounds for s4. Concentrations are in pM.
opt.get_concentration_bounds_species()
concentration_bounds[2] = (0.5, 5e2)

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

# The reponse-related specie should be picked based on CellDesigner IDs. In our case␣
↪phoshorylated A is s2.
# How to pick continuation parameter? In our case it is the amount of A protein, thus␣
↪the conservation law 3.
print(opt.get_conservation_laws())
multistable_param_ind, plot_specifications = opt.run_greedy_continuity_
↪analysis(species="s2", parameters=params_for_global_min,
                                                                                 auto_
↪parameters={'PrincipalContinuationParameter': 'C3'})

opt.generate_report()
```

Using the above code we find three set of values for which bistability exists, providing the following plots:


1e4
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
s2
LP
LP
5.28
5.30
5.32
5.34
5.36
5.38
5.40
C3
1e4


1e5
2.0
1.5
1.0
0.5
0.0
s2
LP
LP
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
1e5
C3



We can now select one of these sets of kinetic constants and species' concentrations to conduct ODE simulation:

```
# Parameters that produced bistability.
# re* are kinetic constants. Units can be found here help(network.get_physiological_
↪range).
df = pandas.DataFrame(numpy.vstack([params_for_global_min[i] for i in multistable_
↪param_ind]).T,
                      columns=["set" + str(i + 1) for i in multistable_param_ind],
                      index=[str(i) for i in opt.get_decision_vector()])

################## selected parameter set #########################
decision_vector_values = numpy.array(df['set1'])
# alternative declaration (for the sake of reference)
decision_vector_values = params_for_global_min[0]
plot_specifications = plot_specifications[0]  # warning, overwriting variable!!!

################ ODEs ###################################
print("Original ODEs")
odes = network.get_c_graph().get_ode_system()
sympy.pprint(odes)

# why we need this? String -> Sympy objects
# construct sympy form of reactions and species
sympy_reactions = [sympy.Symbol(i, positive=True) for i in network.get_c_graph().get_
↪reactions()]
sympy_species = [sympy.Symbol(i, positive=True) for i in network.get_c_graph().get_
↪species()]
```

(continues on next page)

(continued from previous page)

```
# joining together
lambda_inputs = sympy_reactions + sympy_species
# creating a lambda function for each ODE to
ode_lambda_functions = [sympy.utilities.lambdify(lambda_inputs, odes[i]) for i in␣
↪range(len(odes))]

################################ kinetic constants ####################################
↪####################
# Does this work for over, proper and under-dimensioned networks
kinetic_constants = numpy.array([decision_vector_values[i] for i in range(len(network.
↪get_c_graph().get_reactions()))])

################################## Computing material conservation values ############
↪###############
# equilibrium species concentrations
species_concentrations = [i(*tuple(decision_vector_values)) for i in opt.get_
↪concentration_funs()]
print(network.get_c_graph().get_species())
print(species_concentrations)
print(opt.get_conservation_laws())
# combine equilibrium specie concentrations according to conservation relationships
conservation_values = network.get_c_graph().get_b()*sympy.Matrix([species_
↪concentrations]).T

################################## starting concentrations ###########################
↪################
# this assumes that a chemical moiety in one state (specie) and other species␣
↪containing this moiety are zero
# assignment of conservation values to species requires exploring the model in␣
↪CellDesigner
# C1 is in s4, free enzyme E2
# C2 is in s3, free enzyme E1
# C3 is in s1, free unphosphorylated specie A
# ['s1', 's2', 's3', 's3s1', 's4', 's4s2', 's2s1']
# ['C3',    0, 'C2',      0, 'C1',      0,      0]
y_fwd = [conservation_values[2], 0.0, conservation_values[1], 0.0, conservation_
↪values[0], 0.0, 0.0]
y_rev = [0.0, conservation_values[2], conservation_values[1], 0.0, conservation_
↪values[0], 0.0, 0.0]
# Note, the continuation parameter C3 (first position) will be varied during␣
↪simulations

############ simulation ###################
# computing dy/dt increments
def f(cs, t, ks, ode_lambda_func, start_ind):
    return [i(*tuple(ks), *tuple(cs)) for i in ode_lambda_func]  # dy/dt

def sim_fun_fwd(x):
    y_fwd[0] = x  # updating s1 concentration or C3
    return itg.odeint(f, y_fwd, t, args=(kinetic_constants, ode_lambda_functions,␣
↪len(ode_lambda_functions)))

def sim_fun_rev(x):
    y_rev[1] = x  # updating s2 concentration
    return itg.odeint(f, y_rev, t, args=(kinetic_constants, ode_lambda_functions,␣
↪len(sympy_reactions)))
```

(continues on next page)

(continued from previous page)

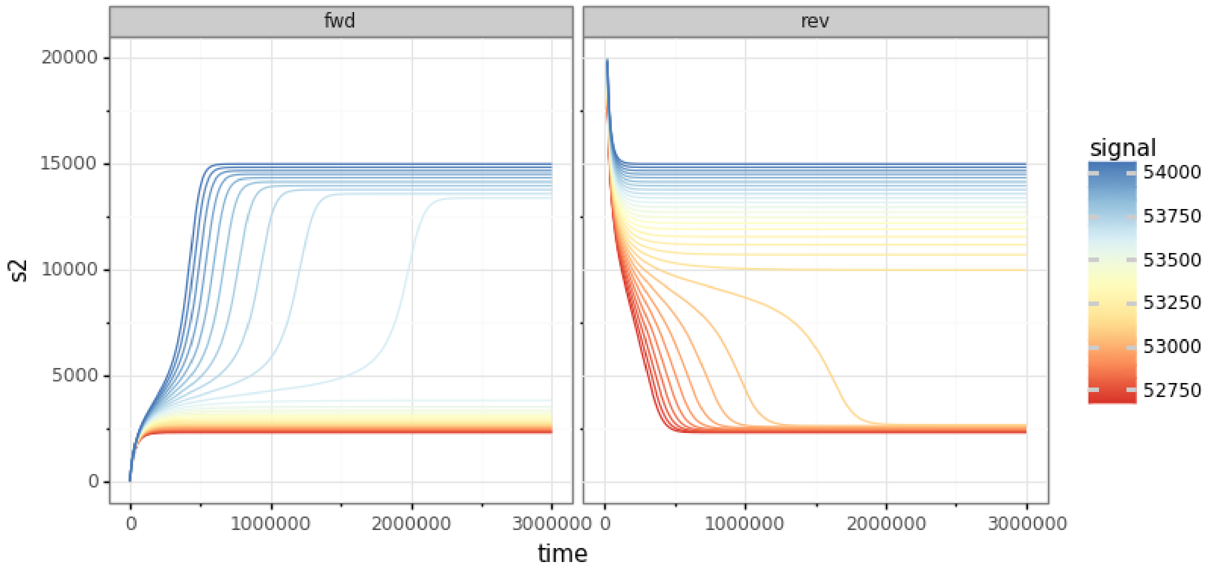
```
# starting and ending time in seconds, number of data points
t = numpy.linspace(0.0, 3000000.0, 3000)
# signal parameter scanning range and data points. Forward scan.
# C3_scan = numpy.linspace(5.3e4, 5.4e4, 60)
# alternatively can be taken from plot_specifications
C3_scan = numpy.linspace(*plot_specifications[0], 30)
sim_res_fwd = [sim_fun_fwd(i) for i in C3_scan]  # occupies sys.getsizeof(sim_res_
↪rev[0])*len(sim_res_rev)/2**20 Mb
# Reverse C3_scan. Reverse means that s2 is already high and signal is decreasing.
sim_res_rev = [sim_fun_rev(i) for i in numpy.flip(C3_scan)]
```

Exporting the results for interrogation using 3rd party tools

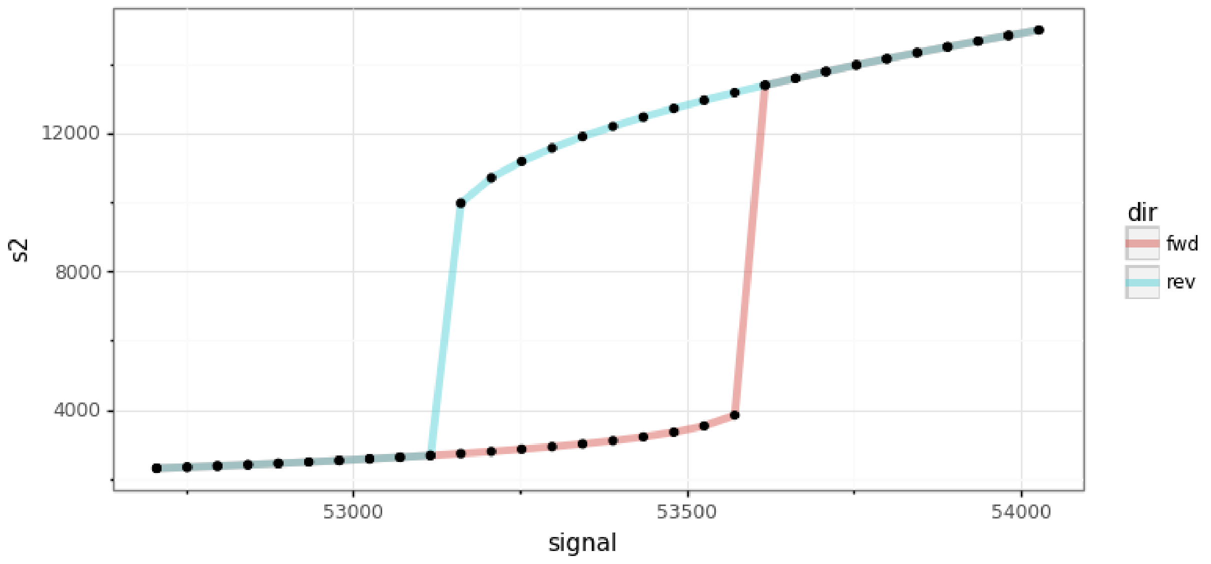
```
################## exporting to text #####################################
out = pandas.DataFrame(columns=['dir','signal','time'] + network.get_c_graph().get_
↪species())
for i in range(len(sim_res_fwd)):
    out_i = pandas.DataFrame(sim_res_fwd[i], columns=out.columns[3:])
    out_i['time'] = t
    out_i['signal'] = C3_scan[i]
    out_i['dir'] = 'fwd'
    out = pandas.concat([out, out_i[out.columns]])
for i in range(len(sim_res_rev)):
    out_i = pandas.DataFrame(sim_res_rev[i], columns=out.columns[3:])
    out_i['time'] = t
    out_i['signal'] = numpy.flip(C3_scan)[i]
    out_i['dir'] = 'rev'
    out = pandas.concat([out, out_i[out.columns]])
out.to_csv("sim.txt", sep="\t", index=False)
```

Visualising the results using plotnine:

```
###################### plotting ###################################
g = (ggplot(out, aes('time', 's2', group='signal', color='signal'))
     + geom_line(size=0.5)
     + ylim(0, 20000)
     + scale_color_distiller(palette='RdYlBu', type="diverging")
     + facet_wrap('~dir')
     + theme_bw())
g.save(filename="./num_cont_graphs/sim_fwd_rev.png", format="png", width=8, height=4,␣
↪units='in', verbose=False)
```



```
eq = out[out.time == max(out.time)]
g = (ggplot(eq)
     + aes(x='signal', y='s2', color='dir')
     + geom_path(size=2, alpha=0.5)
     + geom_point(color="black")
     + theme_bw())
g.save(filename="./num_cont_graphs/sim_bif_diag.png", format="png", width=8, height=4,
↪ units='in', verbose=False)
```



CHAPTER 21

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.
GeneralApproach(cgraph, get_physiological_range)	Class for constructing a more general approach to bistability detection for systems with mass action kinetics.

21.1 crnt4sbml.CRNT

class crnt4sbml.**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_general_approach()	Initializes and creates an object for the class GeneralApproach 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.
print_biological_reaction_types()	Prints the reactions, reaction labels, and biological reaction type 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.Cgraph()

Example

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

get_general_approach()

Initializes and creates an object for the class GeneralApproach for the CRNT object constructed.

See also:

crnt4sbml.GeneralApproach()

Example

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

get_low_deficiency_approach()

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

See also:

crnt4sbml.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.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: 2.060944
```

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”, “catalysis”, or “flux”

Returns

- **concentration** (*tuple*) – (5e-1,5e5) pM
- **complex formation** (*tuple*) – (1e-8,1e-4) pM^-1s^-1
- **complex dissociation** (*tuple*) – (1e-5,1e-3) s^-1
- **catalysis** (*tuple*) – (1e-3,1) s^-1
- **flux** (*tuple*) – (0, 55) M 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.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_biological_reaction_types()

Prints the reactions, reaction labels, and biological reaction type for the network. `Fig1Ci.xml` for the provided example.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> network.print_biological_reaction_types()
    Reaction graph of the form
    reaction -- reaction label -- biological reaction type:
    s1+s2 -> s3  --  re1 -- complex formation
    s3 -> s1+s2  --  re1r -- complex dissociation
    s3 -> s6+s2  --  re2 -- catalysis
    s6+s7 -> s16  --  re3 -- complex formation
    s16 -> s6+s7  --  re3r -- complex dissociation
    s16 -> s7+s1  --  re4 -- catalysis
    s1+s6 -> s15  --  re5 -- complex formation
    s15 -> s1+s6  --  re5r -- complex dissociation
    s15 -> 2*s6  --  re6 -- catalysis
```

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  --  re5
    s15 -> s1+s6  --  re5r
    s15 -> 2*s6  --  re6
```

21.2 crnt4sbml.Cgraph

class `crnt4sbml.`**Cgraph**(*model*)

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

`__init__`(*model*)
Initialization of Cgraph class.

See also:

`crnt4sbml.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.
`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_classifica`()	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_cla`()	Returns list of Python boolean types for the weak reversibility of each linkage class.
`get_number_of_terminal_strong_lc_per_`()	Returns an integer list stating the number of terminally strong linkage classes per linkage class.
`print`()	Prints edges and nodes of NetworkX DiGraph.

Continued on next page

Table 3 – continued from previous page

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  -re₅       re5r       0

     0          0        0   0          0         0  re₅    -re5r - re₆  0

     0          0        0   0          0         0   0          re₆        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  1.0   0

     0   1.0  1.0   0    0    0    0

    1.0   0   1.0  1.0   0   1.0  2.0
```

get_complexes()

Returns Python list of strings representing the complexes of the network. `Fig1Ci.xml` for the provided example.

Example

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

    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.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.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())
        -re₁s₁s₂ + re1rs₃ + re₄s₁₆ - re₅s₁s₆ + re5rs₁₅
```

(continues on next page)

(continued from previous page)

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

                      re₁s₁s₂ + s₃(−re1r − re₂)

re₂s₃ − re₃s₆s₇ + re3rs₁₆ − re₅s₁s₆ + s₁₅(re5r + 2re₆)

                      −re₃s₆s₇ + s₁₆(re3r + re₄)

                      re₃s₆s₇ + s₁₆(−re3r − re₄)

                      re₅s₁s₆ + s₁₅(−re5r − re₆)
```

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())
    s₁s₂

     s₃

    s₂s₆

    s₆s₇

     s₁₆

    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
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> print(network.get_c_graph().get_reactions())
    ['re1', 're1r', 're2', 're3', 're3r', 're4', 're5', 're5r', 're6']
```

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   1   -1  -1  0   0   0

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

get_species()

Returns Python list of strings representing the species of the network. `Fig1Ci.xml` for the provided example.

Example

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

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.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  1  0  0  0  0

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

plot()
Plots NetworkX DiGraph.

See also:

crnt4sbml.CRNT.plot_c_graph()

plot_save()
Saves the plot of the NetworkX DiGraph.

See also:

crnt4sbml.CRNT.plot_save_c_graph()

print()
Prints edges and nodes of NetworkX DiGraph.

See also:

crnt4sbml.CRNT.print_c_graph()

21.3 crnt4sbml.LowDeficiencyApproach

class crnt4sbml.**LowDeficiencyApproach**(*cgraph*)
Class for testing the Deficiency Zero and One Theorems.

__init__(*cgraph*)
Initialization of LowDeficiency Approach class.

See also:

crnt4sbml.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.

Continued on next page

Table 4 – continued from previous page

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")
>>> 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 the Deficiency One Theorem, multistability cannot be excluded."

`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 the Deficiency One Theorem, multistability␣
↪cannot be excluded.
```

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 the Deficiency Zero Theorem, multistability cannot be excluded."

`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 the Deficiency Zero Theorem, multistability␣
↪cannot be excluded.
```

21.4 crnt4sbml.MassConservationApproach

class crnt4sbml.**MassConservationApproach**(*cgraph*, *get_physiological_range*)

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

__init__(*cgraph*, *get_physiological_range*)

Initialization of the MassConservationApproach class.

See also:

crnt4sbml.CRNT.get_mass_conservation_approach()

Methods

`__init__`(cgraph, get_physiological_range)	Initialization of the MassConservationApproach class.
`generate_report`()	Prints out helpful details constructed by `crnt4sbml.MassConservationApproach.run_optimization()` and `crnt4sbml.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.
`get_concentration_bounds_species`()	Returns a list of SymPy variables that represents the order of species for the concentration bounds provided to `crnt4sbml.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.
`get_independent_odes`()	Returns a SymPy Matrix where the rows represent the independent ODEs used in the numerical continuation routine.

Continued on next page

Table 5 – continued from previous page

get_independent_species()	Returns a list of SymPy representations of the independent species used in the numerical continuation routine.
get_optimization_bounds()	Builds all of the necessary physiological bounds for the optimization routine.
get_my_rank()	Returns the rank assigned by mpi4py if it is initialized, otherwise None will be returned.
get_comm()	Returns a mpi4py communicator if it has been initialized and None otherwise.
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.
run_direct_simulation([response, signal, ...])	Function for running direct simulation to conduct bistability analysis of the mass conservation approach.

generate_report()

Prints out helpful details constructed by *crnt4sbml.MassConservationApproach.run_optimization()* and *crnt4sbml.MassConservationApproach.run_continuity_analysis()*.

Example

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

get_comm()

Returns a mpi4py communicator if it has been initialized and None otherwise.

get_concentration_bounds_species()

Returns a list of SymPy variables that represents the order of species for the concentration bounds provided to *crnt4sbml.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()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 2.060944
```

```
>>> print(approach.get_concentration_bounds_species())
    [s1, s3, s7, s16]
```

get_concentration_funs()

Returns a list of lambda functions representing each of the species. Here the species are those expressions provided by *crnt4sbml.MassConservationApproach.get_concentration_vals()* where the arguments of each lambda function is provided by *crnt4sbml.MassConservationApproach.get_decision_vector()*. `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: 2.060944
```

```
>>> print(approach.get_concentration_funs())
    [<function _lambdifygenerated at 0x135f8b4d0>, <function _
↪lambdifygenerated at 0x135f72050>,
    <function _lambdifygenerated at 0x135f728c0>, <function _
↪lambdifygenerated at 0x135f725f0>,
    <function _lambdifygenerated at 0x135f5f830>, <function _
↪lambdifygenerated at 0x135fa0170>,
    <function _lambdifygenerated at 0x135fa04d0>]
```

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
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 2.060944
```

```
>>> print(approach.get_concentration_solutions())
    s1 = s15*(re5r + re6)/(re5*s6)
    s2 = s2
    s3 = re1*s15*s2*(re5r + re6)/(re5*s6*(re1r + re2))
    s6 = s6
    s7 = -s15*(re5*re5r*s6*(re1r + re2)*(re3r + re4) - (re5r + re6)*(-
↪re1*re1r*re3r*s2 - re1*re1r*re4*s2 + re1*re3r*s2*(re1r + re2) +␣
↪re1*re4*s2*(re1r + re2) + re5*s6*(re1r + re2)*(re3r + re4)))/
↪(re3*re4*re5*s6**2*(re1r + re2))
    s16 = s15*(re1*re2*re5r*s2 + re1*re2*re6*s2 + re1r*re5*re6*s6 +␣
↪re2*re5*re6*s6)/(re4*re5*s6*(re1r + re2))
    s15 = s15
```

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.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.MassConservationApproach.get_concentration_solutions()

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: 2.060944
```

```
>>> print(approach.get_concentration_vals())
    [s15*(re5r + re6)/(re5*s6), s2, re1*s15*s2*(re5r + re6)/(re5*s6*(re1r +␣
↪re2)), s6,
    -s15*(re5*re5r*s6*(re1r + re2)*(re3r + re4) - (re5r + re6)*(-
↪re1*re1r*re3r*s2 - re1*re1r*re4*s2 +
    re1*re3r*s2*(re1r + re2) + re1*re4*s2*(re1r + re2) + re5*s6*(re1r +␣
↪re2)*(re3r + re4)))/(re3*re4*re5*s6**2*
    (re1r + re2)), s15*(re1*re2*re5r*s2 + re1*re2*re6*s2 + re1r*re5*re6*s6 +␣
↪re2*re5*re6*s6)/(re4*re5*s6*(re1r + re2)), s15]
```

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
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 2.060944
```

```
>>> 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: 2.060944
```

```
>>> 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      re3r           0

       0        0          0        re₃s₇    re₃s₆    -re3r - re₄      0

    -re₅s₆      0          0       -re₅s₁      0          0          re5r

    re₅s₆       0          0        re₅s₁      0          0       -re5r - re₆
```

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: 2.060944
```

```
>>> print(approach.get_decision_vector())
    [re1, re1r, re2, re3, re3r, re4, re5, re5r, re6, s2, s6, s15]
```

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()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 2.060944
```

```
>>> 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      re3r           0       1␣
↪ 0
                                                                                ␣
↪
```

(continues on next page)

(continued from previous page)

```
        0          0           0         re₃s₇    re₃s₆    -re3r - re₄         0          0␣
↪ 0
                                                                                            ␣
↪
      -re₅s₆      0           0        -re₅s₁      0          0           re5r         0␣
↪ 1
                                                                                            ␣
↪
      re₅s₆       0           0         re₅s₁      0          0       -re5r - re₆    0␣
↪ 0
                                                                                            ␣
↪
        0          0           0           0        1.0        1.0            0          ␣
↪ 0   0
                                                                                            ␣
↪
        0         1.0         1.0          0         0          0             0          ␣
↪ 0   0
                                                                                            ␣
↪
       1.0         0          1.0         1.0        0         1.0           2.0         ␣
↪ 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: 2.060944
```

```
>>> sympy.pprint(approach.get_h_vector())
    a₁ - a₂ - re₁s₁s₂ + re1rs₃

    re₁s₁s₂ + s₃(-re1r - re₂)

      a₁ - re₃s₆s₇ + re3rs₁₆

    re₃s₆s₇ + s₁₆(-re3r - re₄)

      a₂ - re₅s₁s₆ + re5rs₁₅

    re₅s₁s₆ + s₁₅(-re5r - re₆)
```

get_independent_odes()

Returns a SymPy Matrix where the rows represent the independent ODEs used in the numerical continuation routine. Here the entries of the list correspond to the time derivatives of the corresponding species provided by *crnt4sbml.MassConservationApproach.get_independent_species()*. Note that the independent ODEs created are based on the species chosen for the numerical continuation. Thus, the continuation routine needs to be ran first. If this function is called before the numerical continuation routine then None will be returned.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> approach = network.get_mass_conservation_approach()
>>> multistable_param_ind = approach.run_greedy_continuity_analysis(species=
↪"species", parameters=params_for_global_min,
                                                                           auto_
↪parameters={'PrincipalContinuationParameter': "PCP"})
>>> odes = approach.get_independent_odes()
```

get_independent_species()

Returns a list of SymPy representations of the independent species used in the numerical continuation routine. Note that the independent species created are based on the species chosen for the numerical continuation. Thus, the continuation routine needs to be ran first. If this function is called before the numerical continuation routine then None will be returned.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> approach = network.get_mass_conservation_approach()
>>> multistable_param_ind = approach.run_greedy_continuity_analysis(species=
↪"species", parameters=params_for_global_min,
                                                                           auto_
↪parameters={'PrincipalContinuationParameter': "PCP"})
>>> species = approach.get_independent_species()
```

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.MassConservationApproach.get_objective_fun_params()*. `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: 2.060944
```

```
>>> 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.MassConservationApproach.get_objective_fun_params()*. `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: 2.060944
```

```
>>> 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.MassConservationApproach.get_objective_fun_params()*. `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: 2.060944
```

```
>>> print(approach.get_lambda_objective_fun())
    <function _lambdifygenerated at 0x12f6f7ea0>
```

get_my_rank()

Returns the rank assigned by mpi4py if it is initialized, otherwise None will be returned.

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
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 2.060944
```

```
>>> print(approach.get_objective_fun_params())
    [re1, re1r, re2, re3, re3r, re4, re5, re5r, re6, s1, s2, s3, s6, s7, s16,␣
↪s15]
```

get_optimization_bounds()

Builds all of the necessary physiological bounds for the optimization routine. `Fig1Ci.xml` for the provided example.

Returns

- **bounds** (*list of tuples*) – List of tuples defining the upper and lower bounds for the decision vector variables based on physiological ranges.
- **concentration_bounds** (*list of tuples*) – List of tuples defining the upper and lower bounds for those concentrations not in the decision vector based on physiological ranges.

Examples

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 2.060944
```

```
>>> bounds, concentration_bounds = approach.get_optimization_bounds()
>>> print(bounds)
    [(1e-08, 0.0001), (1e-05, 0.001), (0.001, 1.0), (1e-08, 0.0001), (1e-05,␣
↪0.001), (0.001, 1.0),
    (1e-08, 0.0001), (1e-05, 0.001), (0.001, 1.0), (0.5, 500000.0), (0.5,␣
↪500000.0), (0.5, 500000.0)]
```

```
>>> print(concentration_bounds)
    [(0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0), (0.5, 500000.0)]
```

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
>>> network = crnt4sbml.CRNT("path/to/Fig1Ci.xml")
>>> approach = network.get_mass_conservation_approach()
    Creating Equilibrium Manifold ...
    Elapsed time for creating Equilibrium Manifold: 2.060944
```

```
>>> print(approach.get_symbolic_objective_fun())
    1.0*re1**2*re2**2*re3**2*re4**2*re5**2*re6**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*re5r*s2/(re5*re6*s1*s6) -
    1.0*s2*(1 + re5*s6/(re1*s2))/(re5*s1*s6) - (1.0 + 1.0*re1r/(re1*s1))/re2)/
↪((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)) + (2.0 + 1.0*re5r/(re5*s6))/
    re6 + 1.0*(1 + re5*s6/(re1*s2))/(re5*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: 2.060944
```

```
>>> 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  1  0  0  0  0

    0  0  0  0  0  0  0  1  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: 2.060944
```

```
>>> sympy.pprint(approach.get_w_nullspace())
    -1  1

    0   0

    1   -1

    -1  0

    0 , 0

    1   0
```

(continues on next page)

(continued from previous page)

```

    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*, *plot_labels=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.
- **plot_labels** (*list of strings*) – A list of strings defining the labels for the x-axis, y-axis, and title. Where the first element is the label for x-axis, second is the y-axis label, and the last element is the title label. If you would like to use the default settings for some of the labels, simply provide None for that element.

Returns

- **multistable_param_ind** (*list of integers*) – A list of those indices in 'parameters' that produce multistable plots.
- **plot_specifications** (*list of lists*) – A list whose elements correspond to the plot specifications of each element in multistable_param_ind. Each element is a list where the first element specifies the range used for the x-axis, the second element is the range for the y-axis, and the last element provides the x-y values and special point label for each special point in the plot.

Example

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

run_direct_simulation(*response=None*, *signal=None*, *params_for_global_min=None*, *dir_path='./dir_sim_graphs'*, *change_in_relative_error=1e-06*, *parallel_flag=False*, *print_flag=False*, *left_multiplier=0.5*, *right_multiplier=0.5*)

Function for running direct simulation to conduct bistability analysis of the mass conservation approach.

Note: This routine is more expensive than the numerical continuation routines, but can provide solutions when the Jacobian of the ODE system is always singular. A parallel version of this routine is available. The routine automatically produces plots of the direct simulation runs and puts them in the user specified dir_path.

Parameters

- **`response`** (*`string`*) – A string stating the response species of the bifurcation analysis.
- **`signal`** (*`string`*) – A string stating the signal of the bifurcation analysis. Can be any of the of the conservation laws.
- **`params_for_global_min`** (*`list of numpy arrays`*) – A list of numpy arrays corresponding to the input vectors that produce a small objective function value.
- **`dir_path`** (*`string`*) – A string stating the path where the bifurcation diagrams should be saved.
- **`change_in_relative_error`** (*`float`*) – A float value that determines how small the relative error should be in order for the solution of the ODE system to be considered at a steady state. Note: a smaller value will run faster, but may produce an ODE system that is not at a steady state.
- **`parallel_flag`** (*`bool`*) – If set to True a parallel version of direct simulation is ran. If False, a serial version of the routine is ran. See *Parallel General Approach* for further information.
- **`print_flag`** (*`bool`*) – If set to True information about the direct simulation routine will be printed. If False, no output will be provided.
- **`left_multiplier`** (*`float`*) – A float value that determines the percentage of the signal that will be searched to the left of the signal value. For example, the lowerbound for the signal range will be signal_value - signal_value*left_multiplier.
- **`right_multiplier`** (*`float`*) – A float value that determines the percentage of the signal that will be searched to the right of the signal value. For example, the upperbound for the signal range will be signal_value + signal_value*right_multiplier.

Returns list_of_ggplots

Return type list of ggplots produced by plotnine

Example

See *Mass Conservation Approach Walkthrough*.

`run_greedy_continuity_analysis`(*species=None*, *parameters=None*, *dir_path='./num_cont_graphs'*, *print_lbls_flag=False*, *auto_parameters=None*, *plot_labels=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.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`.
- **plot_labels** (*list of strings*) – A list of strings defining the labels for the x-axis, y-axis, and title. Where the first element is the label for x-axis, second is the y-axis label, and the last element is the title label. If you would like to use the default settings for some of the labels, simply provide None for that element.

Returns

- **multistable_param_ind** (*list of integers*) – A list of those indices in 'parameters' that produce multistable plots.
- **plot_specifications** (*list of lists*) – A list whose elements correspond to the plot specifications of each element in multistable_param_ind. Each element is a list where the first element specifies the range used for the x-axis, the second element is the range for the y-axis, and the last element provides the x-y values and special point label for each special point in the plot.

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*, *confidence_level_flag=False*, *change_in_rel_error=0.1*, *parallel_flag=False*)

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.MassConservationApproach.get_concentration_bounds_species()*.

- **confidence_level_flag** (*bool*) – If True a confidence level for the objective function will be given.
- **change_in_rel_error** (*float*) – The maximum relative error that should be allowed to consider f_k in the neighborhood of $\widetilde{f}$.
- **parallel_flag** (*bool*) – If set to True a parallel version of the optimization routine is ran. If False, a serial version of the optimization routine is ran. See *Parallel General Approach*.

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*.

21.5 crnt4sbml.SemiDiffusiveApproach

class crnt4sbml.**SemiDiffusiveApproach**(*cgraph*, *get_physiological_range*)
Class for constructing variables and methods needed for the semi-diffusive approach.

__init__(*cgraph*, *get_physiological_range*)
Initialization of the SemiDiffusiveApproach class.

See also:

crnt4sbml.CRNT.get_semi_diffusive_approach()

Methods

__init__(cgraph, get_physiological_range)	Initialization of the SemiDiffusiveApproach class.
generate_report()	Prints out helpful details constructed by *crnt4sbml.SemiDiffusiveApproach.run_optimization()* and *crnt4sbml.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.

Continued on next page

Table 6 – continued from previous page

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.
get_lambda_objective_fun()	Returns a lambda function representation of the objective function of the optimization problem.
get_optimization_bounds()	Returns a list of tuples defining the upper and lower bounds for the decision vector variables based on physiological ranges.
get_my_rank()	Returns the rank assigned by mpi4py if it is initialized, otherwise None will be returned.
get_comm()	Returns a mpi4py communicator if it has been initialized and None otherwise.
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.SemiDiffusiveApproach.run_optimization()* and *crnt4sbml.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
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> print(approach.get_boundary_species())
    ['s21']
```

get_comm()

Returns a mpi4py communicator if it has been initialized and None otherwise.

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
>>> 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.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
>>> 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.SemiDiffusiveApproach.get_mu_vector()*.

Example

```
>>> import crnt4sbml
>>> 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.SemiDiffusiveApproach.get_mu_vector()*.

Example

```
>>> import crnt4sbml
>>> 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
>>> 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_my_rank()
Returns the rank assigned by mpi4py if it is initialized, otherwise None will be returned.

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

get_optimization_bounds()
Returns a list of tuples defining the upper and lower bounds for the decision vector variables based on physiological ranges. `Fig1Cii.xml` for the provided example.

Examples

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/Fig1Cii.xml")
>>> approach = network.get_semi_diffusive_approach()
>>> bounds = approach.get_optimization_bounds()
>>> print(bounds)
    [(0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55), (0, 55),␣
↪(0, 55), (0, 55), (0, 55), (0, 55)]
```

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")
```

(continues on next page)

(continued from previous page)

```
>>> 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
>>> 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
>>> 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*, *plot_labels=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.
- **plot_labels** (*list of strings*) – A list of strings defining the labels for the x-axis, y-axis, and title. Where the first element is the label for x-axis, second is the y-axis label, and the last element is the title label. If you would like to use the default settings for some of the labels, simply provide None for that element.

Returns

- **multistable_param_ind** (*list of integers*) – A list of those indices in 'parameters' that produce multistable plots.
- **plot_specifications** (*list of lists*) – A list whose elements correspond to the plot specifications of each element in multistable_param_ind. Each element is a list where the first element specifies the range used for the x-axis, the second element is the range for the y-axis, and the last element provides the x-y values and special point label for each special point in the plot.

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*, *plot_labels=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.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`.
- **plot_labels** (*list of strings*) – A list of strings defining the labels for the x-axis, y-axis, and title. Where the first element is the label for x-axis, second is the y-axis label, and the last element is the title label. If you would like to use the default settings for some of the labels, simply provide None for that element.

Returns

- **multistable_param_ind** (*list of integers*) – A list of those indices in 'parameters' that produce multistable plots.
- **plot_specifications** (*list of lists*) – A list whose elements correspond to the plot specifications of each element in multistable_param_ind. Each element is a list where the first element specifies the range used for the x-axis, the second element is the range for the y-axis, and the last element provides the x-y values and special point label for each special point in the plot.

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'>*, *confidence_level_flag=False*, *change_in_rel_error=0.1*, *parallel_flag=False*)

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.
- **confidence_level_flag** (*bool*) – If True a confidence level for the objective function will be given.
- **change_in_rel_error** (*float*) – The maximum relative error that should be allowed to consider f_k in the neighborhood of $\widetilde{f}$.
- **parallel_flag** (*bool*) – If set to True a parallel version of the optimization routine is ran. If False, a serial version of the optimization routine is ran. See *Parallel General Approach*.

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*.

21.6 crnt4sbml.GeneralApproach

class `crnt4sbml.`**`GeneralApproach`**(*cgraph*, *get_physiological_range*)

Class for constructing a more general approach to bistability detection for systems with mass action kinetics.

`__init__`(*cgraph*, *get_physiological_range*)

Initialization of GeneralApproach class.

See also:

`crnt4sbml.CRNT.get_general_approach()`

Methods

`__init__`(cgraph, get_physiological_range)	Initialization of GeneralApproach class.
`initialize_general_approach`([signal, …])	Function for initializing the necessary variables for the general approach.
`generate_report`()	Prints out helpful details constructed by `crnt4sbml.GeneralApproach.run_optimization()`, `crnt4sbml.GeneralApproach.run_continuity_analysis()`, and `crnt4sbml.GeneralApproach.run_greedy_continuity_analysis()`.
`get_conservation_laws`()	Returns a string representation of the conservation laws.
`get_input_vector`()	Returns a list of SymPy variables that specifies the ordering of the reactions and species for which bounds need to be provided.
`get_decision_vector`()	Returns a list of SymPy variables that specifies the ordering of the reactions and species of the decision vector used in optimization.
`get_optimization_bounds`()	Returns a list of tuples that corresponds to the determined physiological bounds chosen for the problem.
`get_ode_lambda_functions`()	Returns a list of lambda functions where each index corresponds to the lambda function for the corresponding ODE, where the species corresponds to the list of species of the network.
`get_independent_odes`()	Returns a Sympy Matrix representing the independent ODE system without conservation laws substituted in.
`get_independent_odes_subs`()	Returns a Sympy Matrix representing the independent ODE system with conservation laws substituted in.
`get_independent_species`()	Returns a list of SymPy variables that reflects the independent species chosen for the general approach.

Continued on next page

Table 7 – continued from previous page

`get_fixed_reactions`()	Returns a list of SymPy variables that describe the reactions that were chosen to be fixed when ensuring a steady-state solution exists.
`get_solutions_to_fixed_reactions`()	Returns a list of SymPy expressions corresponding to the fixed reactions.
`get_jacobian`()	Returns a Sympy expression of the Jacobian, where the Jacobian is with respect to the independent species.
`get_jac_lambda_function`()	Returns a lambda function of the Jacobian, where the Jacobian is with respect to full system and species.
`get_determinant_of_jacobian`()	Returns a Sympy expression of the determinant of the Jacobian, where the Jacobian is with respect to the independent species.
`get_comm`()	Returns a mpi4py communicator if it has been initialized and None otherwise.
`get_my_rank`()	Returns the rank assigned by mpi4py if it is initialized, otherwise None will be returned.
`run_optimization`([bounds, iterations, seed, . . .])	Function for running the optimization problem for the general approach.
`run_continuity_analysis`([species, . . .])	Function for running the numerical continuation and bistability analysis portions of the general approach.
`run_greedy_continuity_analysis`([species, . . .])	Function for running the greedy numerical continuation and bistability analysis portions of the general approach.
`run_direct_simulation`([. . .])	Function for running direct simulation to conduct bistability analysis of the general approach.

generate_report()

Prints out helpful details constructed by `crnt4sbml.GeneralApproach.run_optimization()`, `crnt4sbml.GeneralApproach.run_continuity_analysis()`, and `crnt4sbml.GeneralApproach.run_greedy_continuity_analysis()`.

Example

See also *General Approach Example* and *General Approach Walkthrough*

get_comm()

Returns a mpi4py communicator if it has been initialized and None otherwise.

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.

Example

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

get_decision_vector()

Returns a list of SymPy variables that specifies the ordering of the reactions and species of the decision vector used in optimization. Note: this method should not be used to create bounds for the optimization routine, rather *crnt4sbml.GeneralApproach.get_input_vector()* should be used.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
>>> print(GA.get_decision_vector())
```

get_determinant_of_jacobian()

Returns a Sympy expression of the determinant of the Jacobian, where the Jacobian is with respect to the independent species.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
>>> GA.get_determinant_of_jacobian()
```

get_fixed_reactions()

Returns a list of SymPy variables that describe the reactions that were chosen to be fixed when ensuring a steady-state solution exists. Note that fixed_reactions must be set to True in *crnt4sbml.GeneralApproach.initialize_general_approach()*.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response, fix_
↪reactions=True)
>>> GA.get_fixed_reactions()
```

get_independent_odes()

Returns a Sympy Matrix representing the independent ODE system without conservation laws substituted in. Each row corresponds to the ODE for the species corresponding to the list provided by *crnt4sbml.GeneralApproach.get_independent_species()*.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
>>> GA.get_independent_odes()
```

get_independent_odes_subs()

Returns a Sympy Matrix representing the independent ODE system with conservation laws substituted in. Each row corresponds to the ODE for the species corresponding to the list provided by *crnt4sbml.GeneralApproach.get_independent_species()*.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
>>> GA.get_independent_odes_subs()
```

get_independent_species()

Returns a list of SymPy variables that reflects the independent species chosen for the general approach.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
>>> GA.get_independent_species()
```

get_input_vector()

Returns a list of SymPy variables that specifies the ordering of the reactions and species for which bounds need to be provided.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
>>> print(GA.get_input_vector())
```

get_jac_lambda_function()

Returns a lambda function of the Jacobian, where the Jacobian is with respect to full system and species.

get_jacobian()
Returns a Sympy expression of the Jacobian, where the Jacobian is with respect to the independent species.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
>>> GA.get_jacobian()
```

get_my_rank()
Returns the rank assigned by mpi4py if it is initialized, otherwise None will be returned.

get_ode_lambda_functions()
Returns a list of lambda functions where each index corresponds to the lambda function for the corresponding ODE, where the species corresponds to the list of species of the network.

get_optimization_bounds()
Returns a list of tuples that corresponds to the determined physiological bounds chosen for the problem. Each entry corresponds to the list provided by *crnt4sbml.GeneralApproach.get_input_vector()*.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
>>> GA.get_optimization_bounds()
```

get_solutions_to_fixed_reactions()
Returns a list of SymPy expressions corresponding to the fixed reactions. The ordering of the elements corresponds to the list returned by *crnt4sbml.GeneralApproach.get_fixed_reactions()*. Note that fixed_reactions must be set to True in *crnt4sbml.GeneralApproach.initialize_general_approach()*.

Example

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response, fix_
↪reactions=True)
>>> GA.get_solutions_to_fixed_reactions()
```

initialize_general_approach(*signal=None*, *response=None*, *fix_reactions=False*)
Function for initializing the necessary variables for the general approach.

Parameters

- **signal** (*String*) – A string stating the conservation law that is the x-axis of the bifurcation diagram.
- **response** (*String*) – A string stating the species that is the y-axis of the bifurcation diagram.
- **fix_reactions** (*bool*) – A bool that determines if a steady state is enforced by fixing the reactions. See *General Approach Walkthrough* for specific details.

Examples

```
>>> import crnt4sbml
>>> network = crnt4sbml.CRNT("path/to/sbml_file.xml")
>>> GA = network.get_general_approach()
>>> signal = "C1"
>>> response = "s1"
>>> GA.initialize_general_approach(signal=signal, response=response)
```

See also *General Approach Example* and *General Approach Walkthrough*.

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

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

Note: A parallel version of this routine is not available.

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.
- **plot_labels** (*list of strings*) – A list of strings defining the labels for the x-axis, y-axis, and title. Where the first element is the label for x-axis, second is the y-axis label, and the last element is the title label. If you would like to use the default settings for some of the labels, simply provide None for that element.

Returns

- **multistable_param_ind** (*list of integers*) – A list of those indices in 'parameters' that produce multistable plots.
- **plot_specifications** (*list of lists*) – A list whose elements correspond to the plot specifications of each element in multistable_param_ind. Each element is a list where the first element specifies the range used for the x-axis, the second element is the range for the

y-axis, and the last element provides the x-y values and special point label for each special point in the plot.

Example

See *General Approach Example* and *General Approach Walkthrough*..

`run_direct_simulation`(*params_for_global_min=None*, *dir_path='./dir_sim_graphs'*, *change_in_relative_error=1e-06*, *parallel_flag=False*, *print_flag=False*, *left_multiplier=0.5*, *right_multiplier=0.5*)

Function for running direct simulation to conduct bistability analysis of the general approach.

Note: This routine is more expensive than the numerical continuation routines, but can provide solutions when the Jacobian of the ODE system is always singular. A parallel version of this routine is available. The routine automatically produces plots of the direct simulation runs and puts them in the user specified dir_path.

Parameters

- **`params_for_global_min`** (*`list of numpy arrays`*) – A list of numpy arrays corresponding to the input vectors that produce a small objective function value.
- **`dir_path`** (*`string`*) – A string stating the path where the bifurcation diagrams should be saved.
- **`change_in_relative_error`** (*`float`*) – A float value that determines how small the relative error should be in order for the solution of the ODE system to be considered at a steady state. Note: a smaller value will run faster, but may produce an ODE system that is not at a steady state.
- **`parallel_flag`** (*`bool`*) – If set to True a parallel version of direct simulation is ran. If False, a serial version of the routine is ran. See *Parallel General Approach* for further information.
- **`print_flag`** (*`bool`*) – If set to True information about the direct simulation routine will be printed. If False, no output will be provided.
- **`left_multiplier`** (*`float`*) – A float value that determines the percentage of the signal that will be searched to the left of the signal value. For example, the lowerbound for the signal range will be signal_value - signal_value*left_multiplier.
- **`right_multiplier`** (*`float`*) – A float value that determines the percentage of the signal that will be searched to the right of the signal value. For example, the upperbound for the signal range will be signal_value + signal_value*right_multiplier.

Returns list_of_ggplots

Return type list of ggplots produced by plotnine

Example

See *General Approach Walkthrough*.

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

Function for running the greedy numerical continuation and bistability analysis portions of the general 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.GeneralApproach.run_continuity_analysis()*.

Note: A parallel version of this routine is not available.

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`.
- **plot_labels** (*list of strings*) – A list of strings defining the labels for the x-axis, y-axis, and title. Where the first element is the label for x-axis, second is the y-axis label, and the last element is the title label. If you would like to use the default settings for some of the labels, simply provide None for that element.

Returns

- **multistable_param_ind** (*list of integers*) – A list of those indices in 'parameters' that produce multistable plots.
- **plot_specifications** (*list of lists*) – A list whose elements correspond to the plot specifications of each element in multistable_param_ind. Each element is a list where the first element specifies the range used for the x-axis, the second element is the range for the y-axis, and the last element provides the x-y values and special point label for each special point in the plot.

Example

See *General Approach Example* and *General Approach Walkthrough*.

run_optimization(*bounds=None*, *iterations=10*, *seed=0*, *print_flag=False*, *dual_annealing_iters=1000*, *confidence_level_flag=False*, *change_in_rel_error=0.1*, *constraints=None*, *parallel_flag=False*)

Function for running the optimization problem for the general approach.

Parameters

- **bounds** (*list of tuples*) – A list defining the lower and upper bounds for each variable in the input vector. See *crnt4sbml.GeneralApproach.get_input_vector()*.
- **iterations** (*int*) – The number of iterations to run the multistart method.
- **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.
- **`dual_annealing_iters`** (*`integer`*) – The number of iterations that should be ran for dual annealing routine in optimization.
- **`confidence_level_flag`** (*`bool`*) – If True a confidence level for the objective function will be given.
- **`change_in_rel_error`** (*`float`*) – The maximum relative error that should be allowed to consider f_k in the neighborhood of $\widetilde{f}$.
- **`constraints`** (*`list of dictionaries`*) – Each dictionary is of the form {'type': '…', 'fun': lambda x: … }, where 'type' can be set to 'ineq' or 'eq' and the lambda function to be defined by the user. The 'ineq' refers to an inequality constraint c(x) with c(x) <= 0. For the lambda function the input x refers to the input vector of the optimization routine. See *General Approach Walkthrough* for further details.
- **`parallel_flag`** (*`bool`*) – If set to True a parallel version of the optimization routine is ran. If False, a serial version of the optimization routine is ran. See *Parallel General Approach*.

Returns

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

Examples

See *General Approach Example* and *General Approach Walkthrough*.

CHAPTER 22

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:

22.1 Types of Contributions

22.1.1 Report Bugs

Report bugs at https://github.com/PNNL-Comp-Mass-Spec/CRNT4SBML/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.

22.1.2 Fix Bugs

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

22.1.3 Implement Features

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

22.1.4 Write Documentation

crnt4sbml could always use more documentation, whether as part of the official crnt4sbml docs, in docstrings, or even on the web in blog posts, articles, and such.

22.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/PNNL-Comp-Mass-Spec/CRNT4SBML/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 :)

CHAPTER 23

Credits

23.1 Development Lead

- Brandon Reyes <reyesb123@gmail.com>

23.2 Contributors

None yet. Why not be the first?

CHAPTER 24

History

24.1 0.0.1 (08-22-2019)

- First release on PyPI.

24.2 0.0.2 (08-23-2019)

- Addition of Cytoscape functionality.

24.3 0.0.3 (08-26-2019)

- Improvements to the plots produced by the continuity analysis.

24.4 0.0.4 (09-5-2019)

- Addition of safety precautions for numerical continuation.

24.5 0.0.5 (09-16-2019)

- Addition of routines to parse catalysis in SBML.
- Addition of routines to automatically generate physiological bounds.

24.6 0.0.6 (09-23-2019)

- Updating safety wrapper to smoothly work on Windows and Mac.

24.7 0.0.7 (10-11-2019)

- Adding output to continuity routine to make post-processing simpler.

24.8 0.0.8 (10-23-2019)

- Adding functionality to allow use of Jupyter notebooks.

24.9 0.0.9 (11-6-2019)

- Improving stability when creating the equilibrium manifold in the mass conservation approach.

24.10 0.0.10 (11-6-2019)

- Adding routine for an exhaustive equilibrium manifold creation in the mass conservation approach.

24.11 0.0.11 (4-23-2020)

- Adding a general approach for mass conserving systems.

24.12 0.0.12 (5-21-2020)

- Adding a different layout for installation of package.

24.13 0.0.13 (8-10-2020)

- Improving usability on Windows machines.

24.14 0.0.14 (1-28-2021)

- Improving the output of plots produced by direct simulation.

24.15 0.0.15 (8-25-2021)

- Updated tolerances for Bayesian stopping rule to 0.1.

CHAPTER 25

Bibliography

Bibliography

[BGS04] H.P.J. Bolton, A.A. Groenwold, and J.A. Snyman. The application of a unified bayesian stopping criterion in competing parallel algorithms for global optimization. *Computers and Mathematics with Applications*, 48(3):549 – 560, 2004. URL: http://www.sciencedirect.com/science/article/pii/S0898122104840768, doi:https://doi.org/10.1016/j.camwa.2003.09.030.

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

[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.

[FSW+16] Song Feng, Meritxell Sáez, Carsten Wiuf, Elisenda Feliu, and Orkun Soyer. Core signalling motif displaying multistability through multi-state enzymes. *Journal of the Royal Society Interface*, 13:, 10 2016. doi:10.1098/rsif.2016.0524.

[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.

[OGM+06] Fernando Ortega, José L. Garcés, Francesc Mas, Boris N. Kholodenko, and Marta Cascante. Bistability from double phosphorylation in signal transduction. *The FEBS Journal*, 273(17):3915–3926, 2006. doi:10.1111/j.1742-4658.2006.05394.x.

[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.

[SF87] J. A. Snyman and L. P. Fatti. A multi-start global minimization algorithm with dynamic search trajectories. *Journal of Optimization Theory and Applications*, 54(1):121–141, 1987. URL: https://doi.org/10.1007/BF00940408, doi:10.1007/BF00940408.

[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