SAMMIpy Documentation

Release 0.0.12

Andre Schultz

Apr 07, 2020

Contents:

SAMMIpy is a tool for visualizing metabolic networks and metabolic network simulations using SAMMI directly from Python using the COBRApy toolbox. This documentation describes the Pythons wrapper for this visualization. You can view the full documentation for SAMMI here, and the documentation for COBRApy here.

If you use SAMMI for your project, please cite the following publication: Schultz, A., & Akbani, R. (2019). SAMMI: A Semi-Automated Tool for the Visualization of Metabolic Networks. *Bioinformatics*.

CHAPTER 1

Installation and Usage

SAMMIpy can be installed via `pip` by running the following code in the command window:

```
pip install sammi
```

To update the package use:

```
pip install sammi -U
```

To use the package add the following code to your python script:

```
import sammi
```

Some of the functionality available in SAMMI, such as PDF download and data upload, are not directly available though this plugin. To use these functions download the model in a SAMMI format and upload the file in the SAMMI web interface at www.SammiTool.com.

For a full description of this plugin please refer to the remaining sections of this documentation.

CHAPTER 2

Documentation

2.1 Classes

Three classes used to render visualizations are defined in the SAMMIpy package. This section describes these classes. For details on how to use them please refer to the subsequent sections. These classes are the following:

2.1.1 Parser

The class `sammi.parser()` defines an object to be used in parsing the model into subgraphs upon loading. Although there are many ways to partition models using SAMMIpy, one of the options is defining a vector of `sammi.parser()` objects where each object defines one subgraph. The `sammi.parser()` class takes three inputs:

- **reactions**: List. Reaction IDs of reactions to be included in the subgraph.
- **name**: String. Name of the subgraph to be displayed in the visualization.
- **flux**: List. Optional. Values to be mapped as reaction colors. Defaults to all NAs where no data is mapped.

2.1.2 Data

The class `sammi.data()` defines which data to be mapped onto the visualization and how. Similarly to the previous class, a vector of `sammi.data()` objects can be defined to plot multiply data types. The class takes five inputs:

- **group**: String. Has three options: `reactions`, `metabolites`, and `links`. Defines where the data will be mapped.
- **kind**: String. Has two options: `color` and `size`. Defines what kind of data to map onto the defined group. `color` can be user with `reactions` and `metabolites` to define node and link color. `color` cannot be used with `links`, as link color matches the corresponding reaction node color. `size` can be used to define node radius or link width.
- **data**: Numerical array where each row defines a reaction or metabolite and each column defines a condition to be mapped. Should have size `len(ids)` by `len(conditions)`.

- **ids**: List of strings. Reaction or metabolite IDs where the data will be mapped. Should be IDs of the variable defined in *group*.
- **conditions**: List of strings. Names to be used for each data condition mapped.

2.1.3 Options

The class `sammi.options()` defines additional options of how the model is plotted. This class takes three fields:

- **htmlName**: Name of html file where the output will be written. Defaults to `index_load.html`. If this option is not defined, the file `index_load.html` will be continuously overwritten every time a new visualization is generated. If users wish to save a visualization to a different file, or wish to visualize multiple maps at once, this parameter can be changed.
- **load**: Boolean, defaults to `True`. Whether or not to load the visualization on a new browser tab. If this parameter is set to false, new visualizations can be rendered by refreshing a previously loaded tab or by using the `sammi.openmap()` function.
- **jscode**: String. Sequence of JavaScript commands to be run following the rendering of the visualization. This can used, for example, to change coloscales and subgraphs upon loading the model. This options requires familiarity with JavaScript and the SAMMI html layout.

2.2 Functions

There are three main user functions for rendering and plotting SAMMI visualizations. These are:

2.2.1 Plotting

The function `sammi.plot()` is used to plot SAMMI visualizations in combination with the SAMMI classes. The function inputs are:

- **model**: COBRApy model to be visualized.
- **parsert**: How the model is to be parsed and visualized. There are several options for this parameter:
  - *Empty vector*: Default. Does not parse the model and plots all reactions and metabolites in a single graph. Not recommended for medium to large-size models.
  - *string*: One of two options. (1) A reaction or metabolite field (e.g. `subsystem` or `compartment`), in which case a subgraph will be drawn for each unique value associated with that field. (2) A path to a file specifying a previously drawn SAMMI map, in which case that map will be rendered.
  - *List of strings*: List of reaction IDs to be plotted. A single graph will be plotted containing only the defined reactions.
  - *List of sammi.parser() objects*: A subgraph is plotted for each `sammi.parser()` object defined in the list.
- **datat**: List of `sammi.data()` objects. Each object will be plotted separately in the visualization.
- **secondaries**: List of strings or regular expressions. Any metabolite, in any subgraph, matching any of the regular expressions defined here will be shelved. These metabolites are not deleted and can be returned to the graph through the floating menu window. For details of this functionality please refer to the SAMMI documentation.
- **opts**: `sammi.options()` object. Additional options for loading the map.

2.2.2 Opening a visualization

the function `sammi.openmap()` is used for opening previously drawn visualizations. It takes a single input: a previously drawn html file name. For instance, `sammi.openmap("index_load.html")` or `sammi.openmap("index_load")` open the default file to which maps are exported.

2.2.3 Running SAMMIpy example

Several examples are built into the SAMMIpy package to exemplify and test the package functionalities. These examples are described in the following section as well as the Jupyter Notebook provided. To use this function run `sammi.test(n)` where `n` is a number from zero to eleven describing one of the examples.

2.3 Examples

Here we provide several simple examples for the use of SAMMIpy. Each example is supposed to be more complex than the next, and is intended to exemplify as many different functionalities of SAMMIpy as possible. Each example can be run using `sammi.test(n)`, where `n` refers to the example number provided here.

To start, load the following libraries:

```
import cobra
import cobra.test
import numpy as np
import sammi
```

2.3.1 0. Plot entire model

To plot the entire model simply call `sammi.plot()` on the COBRA model. This is not advisable for medium to large models as the visualization may be too large to render.

```
#Get sample model to plot
model = cobra.test.create_test_model("textbook")
#Plot file to default index_load.html
sammi.plot(model)
```

2.3.2 1-2. Divide the model into subgraphs using model annotation

1. Maps can be divided into subgraphs using model annotation. For instance, users can plot a subgraph for each annotated reaction subsystem:

```
#Get sample model to plot
model = cobra.test.create_test_model("salmonella")
#Plot
sammi.plot(model,'subsystem')
```

2. Or plot a map for each cellular compartment:

```
#Get sample model to plot
model = cobra.test.create_test_model("textbook")
#Plot
sammi.plot(model,'compartment')
```

2.3.3 3. Plot and visualize multiple maps

By default, SAMMI outputs the visualization to a file names `index.load.html` in the package folder. Therefore, by default, every time a new visualization is generated this file is overwritten. The name of the output file can be changed, however, in order to not overwrite files. For instance:

```
#Get sample model to plot
model = cobra.test.create_test_model("salmonella")
#Generate options. This will not load a new tab upon generating the visualization
opts = sammi.options(load = False)
#Plot file to default index_load.html
sammi.plot(model,'subsystem',opts = opts)
#Generate option for new name
opts = sammi.options(htmlName = 'index_load2.html',load = False)
#Plot file to default index_load.html
sammi.plot(model,'compartment',opts = opts)
#Open files in new tabs
sammi.openmap('index_load.html')
sammi.openmap('index_load2.html')
```

2.3.4 4. Plot only user-defined reactions

For a quick visualization of a given group of reactions users can plot only certain reactions in a single graph.

```
#Get sample model to plot
model = cobra.test.create_test_model("textbook")

#Define reactions
tca = ['ACONTa','ACONTb','AKGDH','CS','FUM','ICDHyr','MDH','SUCOAS']
gly = ['ENO','FBA','FBP','GAPD','PDH','PFK','PGI','PGK','PGM','PPS','PYK','TPI']
ppp = ['G6PDH2r','GND','PGL','RPE','RPI','TALA','TKT1','TKT2']
dat = tca + gly + ppp

#Plot
sammi.plot(model,dat)
```

2.3.5 5. Shelve secondary metabolites on load

In order to shelve secondary metabolites upon rendering the model, define the `secondaries` input to the plot function. If this argument is defined, any metabolite, matching any of the defined regular expressions, will be shelved. These metabolites can be returned to the graph using the floating menu window.

```
#Get sample model to plot
model = cobra.test.create_test_model("textbook")

#Define reactions
tca = ['ACONTa','ACONTb','AKGDH','CS','FUM','ICDHyr','MDH','SUCOAS']
gly = ['ENO','FBA','FBP','GAPD','PDH','PFK','PGI','PGK','PGM','PPS','PYK','TPI']
ppp = ['G6PDH2r','GND','PGL','RPE','RPI','TALA','TKT1','TKT2']
dat = tca + gly + ppp

#Define secondaries
secondaries = ['^h_.$','^h2o_.$','^atp_.$','^adp_.','^pi_.','^o2_.','^co2_.','^nad_.',
↪'^nadh_.','^nadp_.','^nadph_.']
```

(continues on next page)

(continued from previous page)

```
#Plot
sammi.plot(model,dat,secondaries = secondaries)
```

2.3.6 6. Plot multiple user-defined subgraphs

Users can also plot multiple subgraphs with their defined reactions. To do so, define an instance of `sammi.parser()` for each subgraph:

```
#Get sample model to plot
model = cobra.test.create_test_model("textbook")

#Define reactions
tca = ['ACONTa','ACONTb','AKGDH','CS','FUM','ICDHyr','MDH','SUCOAS']
gly = ['ENO','FBA','FBP','GAPD','PDH','PFK','PGI','PGK','PGM','PPS','PYK','TPI']
ppp = ['G6PDH2r','GND','PGL','RPE','RPI','TALA','TKT1','TKT2']

#Initialize class
dat = [sammi.parser('TCA cycle',tca),
    sammi.parser('Glycolysis/Gluconeogenesis',gly),
    sammi.parser('Pentose Phosphate Pathway',ppp)]
#Plot
sammi.plot(model,dat)
```

2.3.7 7-8. Data mapping

7. Add data to plotted subgraphs. In this example we are generating random data and mapping it onto the desired reactions. Using `sammi.parser()` users can directly map data as reaction colors:

```
#Get sample model to plot
model = cobra.test.create_test_model("textbook")

#Define reactions
tca = ['ACONTa','ACONTb','AKGDH','CS','FUM','ICDHyr','MDH','SUCOAS']
gly = ['ENO','FBA','FBP','GAPD','PDH','PFK','PGI','PGK','PGM','PPS','PYK','TPI']
ppp = ['G6PDH2r','GND','PGL','RPE','RPI','TALA','TKT1','TKT2']

#Initialize class
dat = [sammi.parser('TCA cycle',tca,np.random.rand(len(tca))),
    sammi.parser('Glycolysis/Gluconeogenesis',gly,np.random.rand(len(gly))),
    sammi.parser('Pentose Phosphate Pathway',ppp,np.random.rand(len(ppp)))]
#Plot
sammi.plot(model,dat)
```

8. Alternatively, users can map data onto the map using `sammi.data()`. The following example maps five sets of random data, each in a different way, with three conditions each.

```
#Get sample model to plot
model = cobra.test.create_test_model("salmonella")

#Get reactions and metabolites
rx = [f.id for f in model.reactions]
met = [m.id for m in model.metabolites]
```

(continues on next page)

(continued from previous page)

```
#Generate random data to plot
datat = [sammi.data('reactions','color',np.random.rand(len(rx),3),rx,['c1','c2','c3
↪']),
        sammi.data('reactions','size',np.random.rand(len(rx),3),rx,['c1','c2','c3']),
        sammi.data('metabolites','color',np.random.rand(len(met),3),met,['c1','c2','c3
↪']),
        sammi.data('metabolites','size',np.random.rand(len(met),3),met,['c1','c2','c3
↪']),
        sammi.data('links','size',np.random.rand(len(rx),3),rx,['c1','c2','c3'])]

#Introduce NAs
for k in range(len(datat)):
    for i in range(datat[k].data.shape[0]):
        for j in range(datat[k].data.shape[1]):
            if np.random.rand(1)[0] < 0.1:
                datat[k].data[i,j] = float('nan')

#Define secondaries
secondaries = ['^h_.$','^h2o_.$','^atp_.$','^adp_.','^pi_.','^o2_.','^co2_.','^nad_.',
↪'^nadh_.','^ndap_.','^ndaph_.']

#Plot
sammi.plot(model,'subsystem',datat = datat,secondaries = secondaries,opts = sammi.
↪options(load=True))
```

2.3.8 9. Change map upon load

SAMMI options also allow users to change visualization parameters upon loading the model. This can be done by adding JavaScript code to the end of the visualization. To use this advanced feature users need to be familiar with JavaScript and need to familiarize themselves with the SAMMI visualization html layout. The following code loads the previous map, changes the visualization to the Citric Acid Cycle subgraph, and changes the colorscale upon loading.

```
#Get sample model to plot
model = cobra.test.create_test_model("salmonella")

#Get reactions and metabolites
rx = [f.id for f in model.reactions]
met = [m.id for m in model.metabolites]

#Generate random data to plot
datat = [sammi.data('reactions','color',np.random.rand(len(rx),3),rx,['c1','c2','c3
↪']),
        sammi.data('reactions','size',np.random.rand(len(rx),3),rx,['c1','c2','c3']),
        sammi.data('metabolites','color',np.random.rand(len(met),3),met,['c1','c2','c3
↪']),
        sammi.data('metabolites','size',np.random.rand(len(met),3),met,['c1','c2','c3
↪']),
        sammi.data('links','size',np.random.rand(len(rx),3),rx,['c1','c2','c3'])]

#Introduce NAs
for k in range(len(datat)):
    for i in range(datat[k].data.shape[0]):
        for j in range(datat[k].data.shape[1]):
```

(continues on next page)

(continued from previous page)

```
            if np.random.rand(1)[0] < 0.1:
                datat[k].data[i,j] = float('nan')

#Define secondaries
secondaries = ['^h_.$','^h2o_.$','^atp_.$','^adp_.','^pi_.','^o2_.','^co2_.','^nad_.',
↪'^nadh_.','^ndap_.','^ndaph_.']

#Generate javascript
jscode = 'x = document.getElementById("onloadf1");' + \
'x.value = "Citric Acid Cycle";' + \
'onLoadSwitch(x);' + \
'document.getElementById("fluxmin").valueAsNumber = -0.1;' + \
'document.getElementById("fluxmax").valueAsNumber = 0.1;' + \
'fluxmin = -0.1; fluxmax = 0.1;' + \
'document.getElementById("edgemin").value = "#ff0000";' + \
'document.getElementById("edgemax").value = "#0000ff";' + \
'document.getElementById("addrxnbreak").click();' + \
'document.getElementsByClassName("rxnbreakval")[0].value = 0;' + \
'document.getElementsByClassName("rxnbreakcol")[0].value = "#c0c0c0";' + \
'defineFluxColorVectors();'

#Plot
sammi.plot(model,'subsystem',datat = datat,secondaries = secondaries,opts = sammi.
↪options(load=True,jscode=jscode))
```

2.3.9 10. Type-III Pathways

Type-III pathways are thermodynamically infeasible loops within the model that do not involve exchange reactions. Here we visualize some of these pathways. We first block all exchange reactions and perform FVA to determine reactions still able to carry flux. Next, we optimize each of these reactions using pFBA to determine the smallest possible Type-III pathway involving the reaction. This example might take a couple of minutes to run.

```
#Import
from cobra.flux_analysis import flux_variability_analysis
from cobra.flux_analysis.loopless import add_loopless, loopless_solution
#Get model and tailor
model = cobra.test.create_test_model("salmonella")
model.reactions.get_by_id('ATPM').lower_bound = 0
model.reactions.get_by_id('ATPM').upper_bound = 1000
rxns = [r.id for r in model.reactions]
#Close exchange reactions
medium = model.medium
for i in model.medium:
    medium[i] = 0.0
model.medium = medium
#Perform FVA on the model
fva = flux_variability_analysis(model,fraction_of_optimum = 0)
fva.maximum[fva.maximum < 1e-03] = 0
fva.minimum[fva.minimum > -1e-03] = 0
#Initialize
dat = []
#Parse through positive reactions
for i in range(len(fva.maximum)):
    if fva.maximum[i] != 0:
        model.objective = model.reactions[i]
```

(continues on next page)

(continued from previous page)

```
        model.optimize()
        flux = cobra.flux_analysis.pfba(model)
        flux.fluxes[abs(flux.fluxes) < 1e-3] = 0
        tmp = abs(flux.fluxes) >= 1e-3
        dat.append(sammi.parser(model.reactions[i].id + ' positive',list(flux.
↪fluxes[tmp].index),list(flux.fluxes[tmp].values)))
#Parse through negative reactions
for i in range(len(fva.minimum)):
    if fva.minimum[i] != 0:
        model.objective = model.reactions[i]
        model.reactions[i].objective_coefficient = -1
        flux = model.optimize()
        flux = cobra.flux_analysis.pfba(model)
        flux.fluxes[abs(flux.fluxes) < 1e-3] = 0
        tmp = abs(flux.fluxes) >= 1e-3
        dat.append(sammi.parser(model.reactions[i].id + ' negative',list(flux.
↪fluxes[tmp].index),list(flux.fluxes[tmp].values)))
#Plot
sammi.plot(model,dat)
```

2.3.10 11. Metabolic Adaptation

Visualize adaptation to gene knockout. In short, the following code performs the following steps for each reaction we wish to simulate.

1. Simulate reaction knockout and get maximum growth rate on KO model.
2. Set upper and lower bound growth rate on wild-type model to KO growth rate and calculate a loopless flux distribution.
3. Using MOMA, calculate a flux distribution in the knockout strain that closely matches the flux distribution in the previous step.
4. Find the difference in flux distributions in steps two and three and plot them.

This process allows users to visualize how the flux was rewired in the knockout strain. This example may take a couple of minutes to run.

```
from cobra.flux_analysis import single_reaction_deletion, moma
from cobra.flux_analysis.loopless import add_loopless, loopless_solution

#Get model
model = cobra.test.create_test_model("ecoli")
#Set objective
model.objective = "Ec_biomass_iJO1366_core_53p95M"
#Initialize parsing list
dat = []
#Define reactions to simulate knockout
korxns = ['ENO','FBA','TKT2','TALA','FUM','MDH','GAPD','TPI']
#Simulate reaction knockout
for r in korxns:
    with model:
        #Save original bounds
        lb = model.reactions.get_by_id(r).lower_bound
        ub = model.reactions.get_by_id(r).upper_bound
        #Set objective to KO
```

(continues on next page)

(continued from previous page)

```
        model.reactions.get_by_id(r).knock_out()
        objval = model.optimize().objective_value
        model.reactions.get_by_id("Ec_biomass_iJO1366_core_53p95M").upper_bound =␣
↪objval
        model.reactions.get_by_id("Ec_biomass_iJO1366_core_53p95M").lower_bound =␣
↪objval
        #Restore bounds
        model.reactions.get_by_id(r).lower_bound = lb
        model.reactions.get_by_id(r).upper_bound = ub
        #Calculate objective
        model.optimize()
        flux = loopless_solution(model)
        #Calculate adaptation
        model.reactions.get_by_id(r).knock_out()
        koflux = cobra.flux_analysis.moma(model,solution=flux)
        #Save
        tmp = flux.fluxes - koflux.fluxes
        bol = abs(tmp) > 1e-7
        x = tmp[bol]
        dat.append(sammi.parser(r + ' - ' + str(round(objval,4)),list(x.index),
↪list(x)))
        #Restore bounds again
        model.reactions.get_by_id(r).lower_bound = lb
        model.reactions.get_by_id(r).upper_bound = ub
        model.reactions.get_by_id("Ec_biomass_iJO1366_core_53p95M").upper_bound = 1000
        model.reactions.get_by_id("Ec_biomass_iJO1366_core_53p95M").lower_bound = 0
#Define secondaries
secondaries = ['^h_.$','^h2o_.$','^atp_.$','^adp_.','^pi_.','^o2_.','^co2_.','^nad_.',
↪'^nadh_.','^ndap_.','^ndaph_.',\
            '^q8_.$','^q8h2_.$','^nadp_.','^nadph_.']
#Plot difference in scatterplot
sammi.plot(model,dat,secondaries = secondaries)
```