PSAMM Documentation

Release 1.2

PSAMM developers

May 10, 2022

Contents

Contents:

CHAPTER 1

Overview

PSAMM is an open source software that is designed for the curation and analysis of metabolic models. It supports model version tracking, model annotation, data integration, data parsing and formatting, consistency checking, automatic gap filling, and model simulations.

PSAMM is developed as an open source project, coordinated through Github. The PSAMM software is being developed in the Zhang Laboratory at the University of Rhode Island.

1.1 Citing PSAMM

If you use PSAMM in a publication, please cite:

Steffensen JL, Dufault-Thompson K, Zhang Y. PSAMM: A Portable System for the Analysis of Metabolic Models. PLOS Comput Biol. Public Library of Science; 2016;12: e1004732. doi:10.1371/journal.pcbi.1004732.

1.2 Software license

CHAPTER 2

PSAMM Tutorials

2.1 Installation and Materials

This tutorial will show you how to get PSAMM up and running on your computer, how to work with the PSAMM YAML format, how to import published models into PSAMM, and how to apply the main tools included with PSAMM to your models.

2.1.1 Downloading the PSAMM Tutorial Data

The PSAMM tutorial materials are available in the psamm-tutorial GitHub repository

These files can be downloaded using the following command:

```
$ git clone https://github.com/zhanglab/psamm-tutorial.git
```

This will create a directory named `psamm-tutorial` in your current working folder. You can then navigate to this directory using the following command:

```
$ cd psamm-tutorial
```

Now you should be in the `psamm-tutorial` folder and should see the following folders:

```
additional_files/
E_coli_sbml/
E_coli_excel/
E_coli_json/
```

These directories include all of the files that will be needed to run the tutorial.

2.1.2 PSAMM Installation

PSAMM can be installed using the Python package installer pip. We recommend that all installations be performed under a virtual Python environment. Major programs and dependencies include: `psamm-model`, which supports model checking, model simulation, and model exports; Linear programming (LP) solvers (e.g. CPLEX, Gurobi, QSopt_ex), which provide the solution of linear programming problems; `psamm-import`, which supports the import of models from SBML, JSON, and Excel formats.

Setting up a Virtual Python Environment (Virtualenv)

It is recommended that the PSAMM software and dependencies should be installed under a virtual Python environment. This can be done by using the Virtualenv software. Virtualenv will set up a Python environment that permits you to install Python packages in a local directory that will not interfere with other programs in the global Python. The virtual environment can be set up at any local directory that you have write permission to. For example, here we will set up the virtual environment under the main directory of this PSAMM tutorial. First, run the following command if you are not in the `psamm-tutorial` folder:

```
$ cd <PATH>/psamm-tutorial
```

In this command, `<PATH>` should be substituted by the directory path to where you created the `psamm-tutorial`. This will change your current directory to the `psamm-tutorial` directory. Then, you can create a virtual environment in the `psamm-tutorial` directory:

```
$ python3 -m venv psamm-env
```

That will set up the virtual environment in a folder called `psamm-env/`. The next step is to activate the virtual environment so that the Python that is being used will be the one that is in the virtualenv. To do this use the following command:

```
$ source psamm-env/bin/activate
```

This will change your command prompt to the following:

```
(psamm-env) $
```

This indicates that the virtual environment is activated, and any installation of Python packages will now be installed in the virtual environment. It is important to note that when you leave the environment and return at a later time, you will have to reactivate the environment (use the `source` command above) to be able to use any packages installed in it.

Note: For Windows users, the virtual environment is installed in a different file structure. The `activate` script on these systems will reside in a `Scripts` folder. To activate the environment on these systems use the command:

```
> psamm-env\Scripts\activate
```

Note: After activating the environment, the command `pip list` can be used to quickly get an overview of the packages installed in the environment and the version of each package.

Setting up a Virtual Python Environment (Anaconda)

Anaconda is an open-source program that allows you to create virtual environments and download Python packages. Unlike VirtualEnv, which is a environment manager for Python, Anaconda is both a package and an environment manager for any programming language. Anaconda manages a list of environments for you, making it easy to work with. Instructions on how to install Anaconda can be found here.

To create a conda environment, you do not have to be in the `psamm-tutorial` directory. You can create the environment from anywhere in your system with a specific version of Python, even if it is not pre-installed:

```
$ conda create --name psamm-env python=<version>
```

Unlike VirtualEnv, there will be no `psamm-env/` folder. A conda environment is not dependent on your current working directory and can be activated from anywhere using the command:

```
$ conda activate psamm-env
```

When you leave the environment and return at a later time, you will have to reactivate the environment (use the `conda activate` command above) to be able to use any packages installed in it.

Note: After activating the environment, the command `conda list` can be used to quickly get an overview of the packages installed in the environment and the version of each package.

Installation of `psamm-model` and `psamm-import`

The next step will be to install `psamm-model` and `psamm-import` as well as their requirements. To do this, you can use the Python Package Installer, *pip*. To install both `psamm-import` and `psamm-model` you can use the following command:

```
(psamm-env) $ pip install git+https://github.com/zhanglab/psamm-import.git
```

This will install `psamm-import` from its Git repository and also install its Python dependencies automatically. One of these dependencies is `psamm-model`, so when `psamm-import` is installed you will also be installing `psamm-model`.

If you only want to install `psamm-model` in the environment you can run the following command:

```
(psamm-env) $ pip install psamm
```

It is important to note that if only `psamm-model` is installed you will be able to apply PSAMM only on models that are represented in the YAML format which will be described later on in the tutorial.

Installation of LP Solvers

The LP (linear programming) solvers are necessary for analysis of metabolic fluxes using the constraint-based modeling approaches.

CPLEX is the recommended solver for PSAMM and is available with an academic license from IBM. Make sure that you use at least **CPLEX version 12.6**. Instructions on how to install CPLEX can be found here.

Once CPLEX is installed, you need to install the Python bindings under the psamm-env virtual environment using the following command:

```
(psamm-env) $ pip install <PATH>/IBM/ILOG/CPLEX_Studio<XXX>/cplex/python/<python_
↪version>/<platform>
```

The directory path in the above command should be replaced with the path to the IBM CPLEX installation in your computer. This will install the Python bindings for CPLEX into the virtual environment.

Note: While the CPLEX software will be installed globally, the Python bindings should be installed specifically under the virtual environment with the PSAMM installation.

PSAMM also supports the use of three other linear programming solvers, Gurobi, QSopt_ex, and GLPK. To install the Gurobi solver, Gurobi will first need to be installed on your computer. Gurobi can be obtained with an academic license from here: Gurobi

Once Gurobi is installed the Python bindings will need to be installed in the virtual environment by running the setup.py script in the package directory. An example of how this could be done on a macOS is (on other platforms the path will be different):

```
(psamm-env) $ cd /Library/gurobi604/mac64/
(psamm-env) $ python setup.py install
```

The QSopt_ex solver can also be used with PSAMM. To install this solver, Python 3.5 or higher is required. You will first need to install Qsopt_ex on your computer and afterwards the Python bindings (*python-qsoptex*) can be installed in the virtual environment:

```
(env) $ pip install cython
(env) $ pip install python-qsoptex
```

Please see the **'python-qsoptex documentation'_** for more information on installing both the library and the Python bindings.

Note: The QSopt_ex solver does not support Integer LP problems and as a result cannot be used to perform flux analysis with thermodynamic constraints. If this solver is used thermodynamic constraints cannot be used during simulation. By default `psamm-model` will not use these constraints.

The GLPK solver is also supported by PSAMM. The GLPK library can be installed in the virtual environment using the following command:

```
(psamm-env) $ pip install swiglpk
```

Once a solver is installed you should now be able to fully use all of the `psamm-model` flux analysis functions. To see a list of the installed solvers the use the `psamm-list-lpsolvers` command:

```
(psamm-env) $ psamm-list-lpsolvers
```

You will see the details on what solvers are installed currently and avaliable to PSAMM. For example if the Gurobi and CPLEX solvers were both installed you would see the following output from `psamm-list-lpsolvers`:

```
Prioritized solvers:
Name: cplex
Priority: 10
MILP (integer) problem support: True
QP (quadratic) problem support: True
Rational solution: False
```

(continues on next page)

(continued from previous page)

```
Class: <class 'psamm.lpsolver.cplex.Solver'>

Name: gurobi
Priority: 9
MILP (integer) problem support: True
QP (quadratic) problem support: False
Rational solution: False
Class: <class 'psamm.lpsolver.gurobi.Solver'>

Name: glpk
Priority: 8
MILP (integer) problem support: True
QP (quadratic) problem support: False
Rational solution: False
Class: <class 'psamm.lpsolver.glpk.Solver'>

Unavailable solvers:
qsoptex: Error loading solver: No module named 'qsoptex'
```

By default the solver with the highest priority (highest priority number) is used in constraint based simulations. If you want to use a solver with a lower priority you will need to specify it by using the `--solver` option. For example to run FBA on a model while using the Gurobi solver the following command would be used:

```
(psamm-env) $ psamm-model fba --solver name=gurobi
```

2.1.3 PSAMM Model Collection

Converted versions of 57 published SBML metabolic models, 9 published Excel models and one MATLAB model are available in the PSAMM Model Collection on GitHub. These models were converted to the YAML format and then manually edited when needed to produce models that can generate non-zero biomass fluxes. The changes to the models are tracked in the Git history of the repository so you can see exactly what changes needed to be made to the models. To download and use these models with *PSAMM* you can clone the Git repository using the following command:

```
$ git clone https://github.com/zhanglab/psamm-model-collection.git
```

This will create the directory `psamm-model-collection` in your current folder that contains one directory named `excel` with the converted Excel models, one directory named `sbml` with the converted SBML models and one named `matlab` with the converted MATLAB model. These models can then be used for simulations with *PSAMM* using the commands detailed in this tutorial.

2.2 Importing, Exporting, and working with Models with PSAMM

This part of the tutorial will focus on how to use PSAMM to convert files between the YAML format and other popular formats. An additional description of the YAML model format and its features is also provided here.

2.2.1 Import Functions in PSAMM

For information on how to install *PSAMM* and the associated requirements, as well how to download the materials required for this tutorial you can reference the Installation and Materials section of the tutorial.

2.2.2 Importing Existing Models (psamm-import)

In order to work with a metabolic model in PSAMM the model must be in the PSAMM-specific YAML format. This format allows for a human readable representation of the model components and allows for enhanced customization with respect to the organization of the metabolic model. This enhanced organization will allow for a more direct interaction with the metabolic model and make the model more accessible to both the modeler and experimental biologists.

Import Formats

The `psamm-import` program supports the import of models in various formats. For the SBML format, it supports the COBRA-compliant SBML specifications, the FBC specifications, and the basic SBML specifications in levels 1, 2, and 3; for the JSON format, it supports the import of JSON files directly from the BiGG database or from locally downloaded versions.

The support for importing from Excel file is model specific and are available for 17 published models. This import requires the installation of the separate psamm-import repository. There is also a generic Excel import for models produced that were produced by older versions of ModelSEED. Models from the current ModelSEED can be imported in the SBML format.

To install the `psamm-import` package for Excel format models use the following command:

```
(psamm-env) $ pip install git+https://github.com/zhanglab/psamm-import.git
```

This install will make the Excel importers available from the command line when the `psamm-import` program is called.

To see a list of the models or model formats that are supported for import, use the command:

```
(psamm-env) $ psamm-import list
```

In the output, you will see a list of specific Excel models that are supported by `psamm-import` as well as the different SBML parsers that are available in PSAMM:

```
Generic importers:
json            COBRApy JSON
modelseed       ModelSEED model (Excel format)
sbml            SBML model (non-strict)
sbml-strict     SBML model (strict)

Model-specific importers:
icce806         Cyanothece sp. ATCC 51142 iCce806 (Excel format), Vu et al., 2012
ecoli_textbook  Escerichia coli Textbook (core) model (Excel format), Orth et al.,␣
↪2010
```

(continues on next page)

(continued from previous page)

```
ijo1366       Escerichia coli iJO1366 (Excel format), Orth et al., 2011
gsmn-tb       Mycobacterium tuberculosis GSMN-TB (Excel format), Beste et al., 2007
inj661        Mycobacterium tuberculosis iNJ661 (Excel format), Jamshidi et al., 2007
inj661m       Mycobacterium tuberculosis iNJ661m (Excel format), Fang et al., 2010
inj661v       Mycobacterium tuberculosis iNJ661v (Excel format), Fang et al., 2010
ijn746        Pseudomonas putida iJN746 (Excel format), Nogales et al., 2011
ijp815        Pseudomonas putida iJP815 (Excel format), Puchalka et al., 2008
stm_v1.0      Salmonella enterica STM_v1.0 (Excel format), Thiele et al., 2011
ima945        Salmonella enterica iMA945 (Excel format), AbuOun et al., 2009
irr1083       Salmonella enterica iRR1083 (Excel format), Raghunathan et al., 2009
ios217_672    Shewanella denitrificans OS217 iOS217_672 (Excel format), Ong et al.,␣
↪2014
imr1_799      Shewanella oneidensis MR-1 iMR1_799 (Excel format), Ong et al., 2014
imr4_812      Shewanella sp. MR-4 iMR4_812 (Excel format), Ong et al., 2014
iw3181_789    Shewanella sp. W3-18-1 iW3181_789 (Excel format), Ong et al., 2014
isyn731       Synechocystis sp. PCC 6803 iSyn731 (Excel format), Saha et al., 2012
```

Now the model can be imported using the `psamm-import` functions. Return to the `psamm-tutorial` folder if you have left it using the following command:

```
(psamm-env) $ cd <PATH>/tutorial-part-1
```

Importing an SBML Model

In this tutorial, we will use the *E. coli* textbook core model [Orth13] as an example to demonstrate these functions in PSAMM. First, we will convert the model from the SBML model. To import the `E_coli_core.xml` model to YAML format run the following command:

```
(psamm-env) $ psamm-import sbml --source E_coli_sbml/ecoli_core_model.xml --dest E_
↪coli_yaml
```

This will convert the SBML file in the `E_coli_sbml` directory into the YAML format that will be stored in the `E_coli_yaml/` directory. The output will give the basic statistics of the model and should look like this:

```
...
WARNING: Species M_pyr_b was converted to boundary condition because of "_b" suffix
WARNING: Species M_succ_b was converted to boundary condition because of "_b" suffix
INFO: Detected biomass reaction: R_Biomass_Ecoli_core_w_GAM
INFO: Removing compound prefix 'M_'
INFO: Removing reaction prefix 'R_'
INFO: Removing compartment prefix 'C_'
Model: Ecoli_core_model
- Biomass reaction: Biomass_Ecoli_core_w_GAM
- Compartments: 2
- Compounds: 72
- Reactions: 95
- Genes: 137
INFO: e is extracellular compartment
INFO: Using default flux limit of 1000.0
INFO: Converting exchange reactions to exchange file
```

`psamm-import` will produce some warnings if there are any aspects of the model that are going to be changed during import. In this case the warnings are notifying you that the metabolites with a _b suffix have been converted to the boundary conditions of the model. There will also be information on what prefixes were removed from the metabolite IDs and if the importer was able to identify the Biomass Reaction in the model. This information is important to check to make sure that the model was imported correctly. After the import the model will be available and ready to use for any other PSAMM functions.

Importing an Excel Model

The process of importing an Excel model is the same as importing an SBML model except that you will need to specify the specific model name in the command. The list of supported models can be seen using the list function above. An example of an Excel model import is below:

```
(psamm-env) $ psamm-import ecoli_textbook --source E_coli_excel/ecoli_core_model.xls -
↪-dest converted_excel_model
```

This will produce a YAML version of the Excel model in the `converted_excel_model/` directory.

Since the Excel models are not in a standardized format these parsers need to be developed on a model-by-model basis in order to parse all of the relevant information out of the model. This means that the parser can only be used for the listed models and not for a general import.

Importing a JSON Model

`psamm-import` also supports the conversion of JSON format models that follows the conventions in COBRApy. If the JSON model is stored locally, it can be converted with the following command:

```
(psamm-env) $ psamm-import json --source E_coli_json/e_coli_core.json --dest␣
↪converted_json_model/
```

Alternatively, an extension of the JSON importer has been provided, `psamm-import-bigg`, which can be applied to convert JSON models from BiGG database. To see the list of available models on the BiGG database the following command can be used:

```
(psamm-env) $ psamm-import-bigg list
```

This will show the available models as well as their names. You can then import any of these models to YAML format. For example, using the following command to import the *E. coli* iJO1366 [Orth11] model from the BiGG database:

```
(psamm-env) $ psamm-import-bigg iJO1366 --dest converted_json_model_bigg/
```

Note: To use `psamm-import-bigg` you must have internet access to download the models remotely.

2.2.3 YAML Format and Model Organization

Now that we have imported the models into the YAML format we can take a look at what the different files are and what information they contain. The PSAMM YAML format stores individual models under a designated directory, in which there will be a number of files that stores the information of the model and specifies the simulation conditions. The entry point of the YAML model is a file named `model.yaml`, which points to additional files that store the information of the model components, including compounds, reactions, flux limits, exchange conditions, etc. While we recommend that you use the name `model.yaml` for the central reference file, the file names for the included files are flexible and can be customized as you prefer. In this tutorial, we simply used the names: `compounds.yaml`, `reactions.yaml`, `limits.yaml`, and `exchange.yaml` for the included files.

First change directory into `E_coli_yaml`:

```
(psamm-env) $ cd E_coli_yaml/
```

The directory contains the main `model.yaml` file as well as the other files that contain the model data:

```
(psamm-env) $ ls
compounds.yaml
exchange.yaml
limits.yaml
model.yaml
reactions.yaml
```

These files can be opened using any standard text editor. We highly recommend using an editor that includes syntax highlighting for the YAML language (one such editor is the Atom editor which includes built-in support for YAML and is available for macOS, Linux and Windows). You can also use commands like `less` and editors like `vi` or `nano` to quickly inspect and edit the files from the command line:

```
(psamm-env) $ less <file_name>.yaml
```

The central file in this organization is the `model.yaml` file. The following is an example of the `model.yaml` file that is obtained from the import of the *E. coli* textbook model. The `model.yaml` file for this imported SBML model should look like the following:

```
name: Ecoli_core_model
biomass: Biomass_Ecoli_core_w_GAM
default_flux_limit: 1000.0
compartments:
- id: c
  adjacent_to: e
  name: Cytoplasm
- id: e
  adjacent_to: c
  name: Extracellular
compounds:
- include: compounds.yaml
reactions:
- include: reactions.yaml
exchange:
- include: exchange.yaml
limits:
- include: limits.yaml
```

The `model.yaml` file defines the basic components of a metabolic model, including the model name (*Ecoli_core_model*), the biomass function (*Biomass_Ecoli_core_w_GAM*), the compound files (`compounds.yaml`), the reaction files (`reactions.yaml`), the flux boundaries (`limits.yaml`), and the exchange conditions (`exchange.yaml`). The additional files are defined using include functions. This organization allows you to easily change aspects of the model like the exchange reactions by simply referencing a different exchange file in the central `model.yaml` definition. In addition to the information on the other components of the model there will also be details on the compartment information for the model. This will provide an overview of how compartments are related to each other and what their abbreviations and names are. For this small model there is only an `e` and a `c` compartment representing the cytoplasm and extracellular space but more complex cells with multiple compartments can also be represented.

This format can also be used to include multiple files in the list of reactions and compounds. This feature can be useful, for example, if you want to name different reaction files based on the subsystem designations or cellular compartments, or if you want to separate the temporary reactions that are used to fill reaction gaps from the main model. An example of how you could designate multiple reaction files is found below. This file can be found in the additional files folder in the file `complex_model.yaml`.

```yaml
name: Ecoli_core_model
biomass: Biomass_Ecoli_core_w_GAM
default_flux_limit: 1000.0
compartments:
- id: c
  adjacent_to: e
  name: Cytoplasm
- id: e
  adjacent_to: c
  name: Extracellular
model:
- include: core_model_definition.tsv
compounds:
- include: compounds.yaml
reactions:
- include: reactions/cytoplasm.yaml
- include: reactions/periplasm.yaml
- include: reactions/transporters.yaml
- include: reactions/extracellular.yaml
exchange:
- include: exchange.yaml
limits:
- include: limits.yaml
```

As can be seen here the modeler chose to distribute their reaction database files into different files representing various cellular compartments and roles. This organization can be customized to suit your preferred workflow.

There are also situations where you may wish to designate only a subset of the reaction database in a metabolic simulation. In these situations it is possible to use a model definition file to identify which subset of reactions will be used from the larger database. The model definition file is simply a list of reaction IDs that will be included in the simulation.

An example of how to include a model definition file can be found below.

```yaml
name: Ecoli_core_model
biomass: Biomass_Ecoli_core_w_GAM
default_flux_limit: 1000.0
compartments:
- id: c
  adjacent_to: e
  name: Cytoplasm
- id: e
  adjacent_to: c
  name: Extracellular
model:
- include: subset.tsv
compounds:
- include: compounds.yaml
reactions:
- include: reactions.yaml
exchange:
- include: exchange.yaml
limits:
- include: limits.yaml
```

Note: When the model definition file is not identified, PSAMM will include the entire reaction database in the model. However, when it is identified, PSAMM will only include the reactions that are listed in the model definition file in the model. This design can be useful when you want to make targeted tests on a subset of the model or when you want to include a larger database for use with the gap filling functions.

Reactions

The `reactions.yaml` file is where the reaction information is stored in the model. A sample from this file can be seen below:

```
- id: ACALD
  name: acetaldehyde dehydrogenase (acetylating)
  genes: b0351 or b1241
  equation: '|acald[c]| + |coa[c]| + |nad[c]| <=> |accoa[c]| + |h[c]| +
    |nadh[c]|'
  subsystem: Pyruvate Metabolism
- id: ACALDt
  name: acetaldehyde reversible transport
  genes: s0001
  equation: '|acald[e]| <=> |acald[c]|'
  subsystem: Transport, Extracellular
```

Each reaction entry is designated with the reaction ID first. Then the various properties of the reaction can be listed below it. The required properties for a reaction are ID and equation. Along with these required attributes others can be included as needed in a specific project. These can include but are not limited to EC numbers, subsystems, names, and genes associated with the reaction. For example, in a collaborative reconstruction you may want to include a field named `authors` to identify which authors have contributed to the curation of a particular reaction.

Reaction equations can be formatted in multiple ways to allow for more flexibility during the modeling process. The reactions can be formatted in a string format based on the ModelSEED reaction format. In this representation individual compounds in the reaction are represented as compound IDs followed by the cellular compartment in brackets, bordered on both sides by single pipes. For example if a hydrogen compound, `Hydr`, in a `cytosol` compartment was going to be in an equation it would be represented as follows:

```
|Hydr[cytosol]|
```

These individual compounds can be assigned stoichiometric coefficients by adding a number in parentheses before the compound. For example if a reaction contained two hydrogens it could appear as follows:

```
(2) |Hydr[cytosol]|
```

These individual components are separated by + signs in the reaction string. The separation of the reactants and products is through the use of an equal sign with greater than or less than signs designating directionality. These could include => or <= for reactions that can only progress in one direction or <=> for reactions that can progress in both directions. An example of a correctly formatted reaction could be as follows:

```
'|ac[c]| + |atp[c]| <=> |actp[c]| + |adp[c]|'
```

For longer reactions the YAML format provides a way to list each reaction component on a single line. For example a reaction could be represented as follows:

```
- id: ACKr
  name: acetate kinase
  equation:
    compartment: c
    reversible: yes
    left:
```

(continues on next page)

(continued from previous page)

```
      - id: ac_c
        value: 1
      - id: atp_c
        value: 1
    right:
      - id: actp_c
        value: 1
      - id: adp_c
        value: 1
  subsystem: Pyruvate Metabolism
```

This line based format can be especially helpful when dealing with larger equations like biomass reactions where there can be dozens of components in a single reaction.

Gene associations for the reactions in a model can also be included in the reaction definitions so that gene essentiality experiments can be performed with the model. These genes associations are included by adding the `genes` property to the reaction like follows:

```
- id: ACALDt
  name: acetaldehyde reversible transport
  equation: '|acald[e]| <=> |acald[c]|'
  subsystem: Transport, Extracellular
  genes: gene_0001
```

More complex gene associations can also be included by using logical and/or statements in the genes property. When performing gene essentiality simulations this logic will be taken into account. Some examples of using this logic with the genes property can be seen below:

```
genes: gene_0001 or gene_0002

genes: gene_0003 and gene_0004

genes: gene_0003 and gene_0004 or gene_0005 and gene_0006

genes: gene_0001 and (gene_0002 or gene_0003)
```

Compounds

The `compounds.yaml` file is organized in a similar way as the `reactions.yaml`. An example can be seen below.

```
- id: 13dpg_c
  name: 3-Phospho-D-glyceroyl-phosphate
  formula: C3H4O10P2
- id: 2pg_c
  name: D-Glycerate-2-phosphate
  formula: C3H4O7P
- id: 3pg_c
  name: 3-Phospho-D-glycerate
  formula: C3H4O7P
```

The compound entries begin with a compound ID which is then followed by the compound properties. These properties can include a name, chemical formula, and charge of the compound.

Limits

The limits file is used to designate reaction flux limits when it is different from the defaults in PSAMM. By default, PSAMM would assign the lower and upper bounds to reactions based on their reversibility, i.e. the boundary of reversible reactions are $-1000 \leq v_j \leq 1000$, and the boundary for irreversible reactions are $0 \leq v_j \leq 1000$. Therefore, the `limits.yaml` file will consist of only the reaction boundaries that are different from these default values. For example, if you want to force flux through an artificial reaction like the ATP maintenance reaction *ATPM* you can add in a lower limit for the reaction in the limits file like this:

```
- reaction: ATPM
  lower: 8.39
```

Each entry in the limits file includes a reaction ID followed by upper and lower limits. Note that when a model is imported only the non-default flux limits are explicitly stated, so some of the imported models will not contain a predefined limits file. In the *E. coli* core model, only one reaction has a non-default limit. This reaction is an ATP maintenance reaction and the modelers chose to force a certain level of flux through it to simulate the general energy cost of cellular maintenance processes.

Exchange

The exchange file is where you can designate the boundary conditions for the model. The compartment of the exchange compounds can be designated using the `compartment` tag, and if omitted, the extracellular compartment (*e*) will be assumed. An example of the exchange file can be seen below.

```
compounds:
- id: ac_e
  reaction: EX_ac_e
  lower: 0.0
- id: acald_e
  reaction: EX_acald_e
  lower: 0.0
- id: akg_e
  reaction: EX_akg_e
  lower: 0.0
- id: co2_e
  reaction: EX_co2_e
```

Each entry starts with the ID of the boundary compound and followed by lines that defines the lower and upper limits of the compound flux. Internally, PSAMM will translate these boundary compounds into exchange reactions in metabolic models. Additional properties can be designated for the exchange reactions including an ID for the reaction, the compartment for the reaction, and lower and upper flux bounds for the reaction. In the same way that only non-standard limits need to be specified in the limits file, only non-standard exchange limits need to be specified in the exchange file. This can be seen with the example above where the upper limits are not set since they should just be the default limit of 1000.

Model Format Customization

The YAML model format is highly customizable to suit your preferences. File names can be changed according to your own design. These customizations are all allowed by PSAMM as long as the central `model.yaml` file is also updated to reflect the different file names referred. While all the file names can be changed it is recommended that the central `model.yaml` file name does not change. PSAMM will automatically detect and read the information from the file if it is named `model.yaml`. If you *do* wish to also alter the name of this file you will need to specify the path of your model file using the `--model` option whenever any PSAMM commands are run. For example, to run FBA with a different central model file named `ecoli_model.yaml`, you could run the command like this:

```
(psamm-env) $ psamm-model --model ecoli_model.yaml fba
```

2.2.4 Version Control with the YAML Format

The YAML format contains a logical division of the model information and allows for easier modification and interaction with the model. Moreover, the text-based representation of YAML files can enable the tracking of model modifications using version control systems. In this tutorial we will demonstrate the use of the Git version control system during model development to track the changes that have been added to an existing model. This feature will improve the documentation of the model development process and improve collaborative annotations during model curation.

A broad overview of how to use various Git features can be found here: Git

Initiate a Git Repository for the YAML Model

Throughout this tutorial version tracking using Git will be highlighted in various sections. As you follow along with the tutorial you can try to run the Git commands to get a sense of how Git and PSAMM work together. We will also highlight how the features of Git help with model curation and development when using the YAML format.

To start using Git to track the changes in this git model the folder must first be initialized as a Git repository. To do this first enter the YAML model directory and use the following command:

```
(psamm-env) $ git init
Initialized empty Git repository in <...>/psamm-tutorial/E_coli_yaml/.git/
```

After the folder is initialized as a Git repository the files that were initially imported from the SBML version can be added to the repository using the following command:

```
(psamm-env) $ git add *.yaml
```

this will stage all of the files with the `yaml` extension to be committed. Then the addition of these files can be added to the repository to be tracked by using the following command:

```
(psamm-env) $ git commit -m 'Initial import of E. coli Core Model'
```

Now these files will be tracked by Git and any changes that are made will be easily viewable using various Git commands. PSAMM will also print out the Git commit ID when any commands are run. This makes it easier for you to track exactly what version of the model a past simulation was done on.

The next step in the tutorial will be to add in a new carbon utilization pathway to the *E. coli* core model and Git will be used to track these new additions and manage the curation in an easy to track manner. The tutorial will return to the version tracking at various points in order to show how this can be used during model development.

FBA on Model Before Expansion

Now that the model is imported and being tracked by Git it will be helpful to do a quick simulation to confirm that the model is complete and able to generate flux. To do this you can run the FBA command in the model directory:

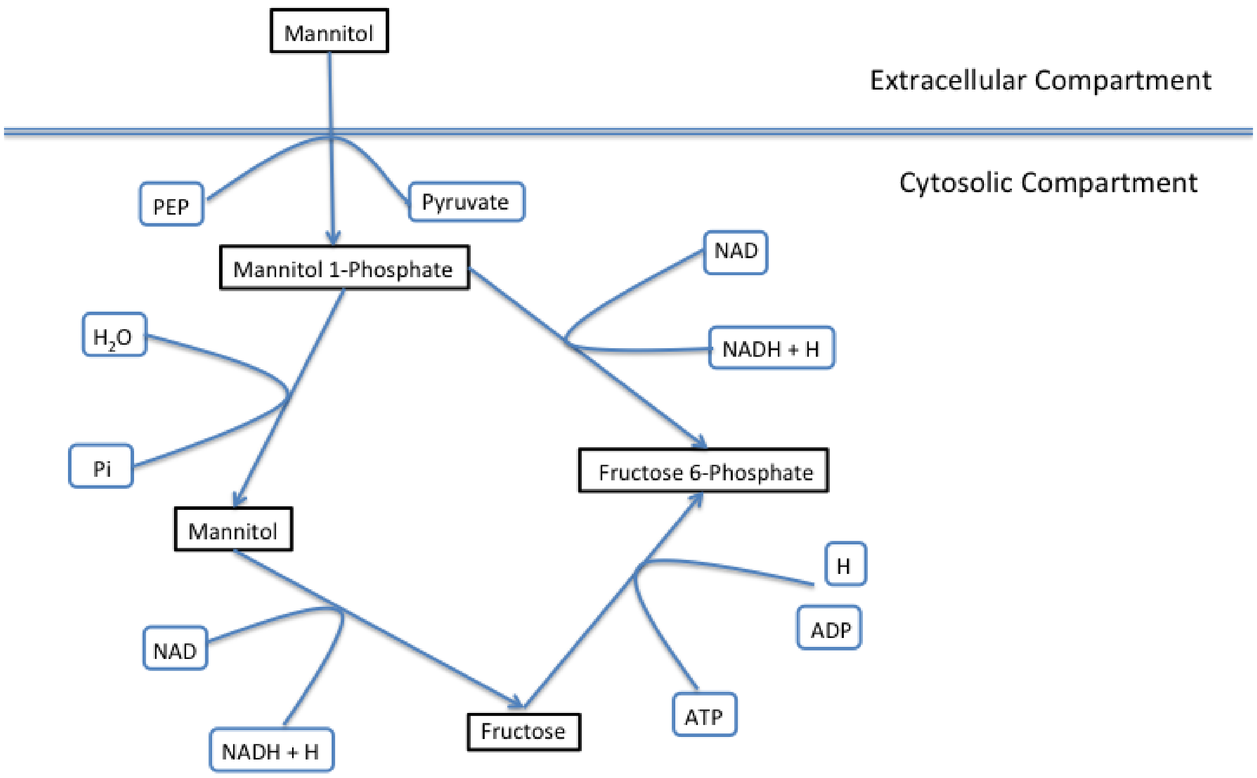
```
(psamm-env) $ psamm-model fba
```

The following is a sample of the output from this initial flux balance analysis. It can be seen that the model is generating flux through the objective function and seems to be a complete working model. Now that this is known any future changes that are made to the model can be made with the knowledge that the unchanged model was able to generate biomass flux.

```
ACONTa       6.00724957535   |Citrate[c]| <=> |cis-Aconitate[c]| + |H2O[c]|  b0118 or␣
↪b1276
ACONTb       6.00724957535   |cis-Aconitate[c]| + |H2O[c]| <=> |Isocitrate[c]|      ␣
↪b0118 or b1276
...
INFO: Objective flux: 0.873921506968
```

Adding a new Pathway to the Model

The *E. coli* textbook model that was imported above is a small model representing the core metabolism of *E. coli*. This model is great for small tests and demonstrations due to its size and excellent curation. For the purposes of this tutorial this textbook model will be modified to include a new metabolic pathway for the utilization of D-Mannitol by *E. coli*. This is a simple pathway which involves the transport of D-Mannitol via the PTS system and then the conversion of D-Mannitol 1-Phosphate to D-Fructose 6-Phosphate. Theoretically the inclusion of this pathway should allow the model to utilize D-Mannitol as a sole carbon source. Along with this direct pathway another set of reactions will be added that remove the phosphate from the mannitol 1-phosphate to create cytoplasmic mannitol which can then be converted to fructose and then to fructose 6-phosphate.



To add these reactions, there will need to be three components added to the model. First the new reactions will be added to the model, then the relevant exchange reactions, and finally the compound information.

The new reactions in the database can be added directly to the already generated reactions file but for this case they will be added to a separate database file that can then be added to the model through the include function in the `model.yaml` file.

A reaction database file named `mannitol_path.yaml` is supplied in `additional_files` folder. This file can be added into the `model.yaml` file by copying it to your working folder using the following command:

```
(psamm-env) $ cp ../additional_files/mannitol_pathway.yaml .
```

And then specifying it in the `model.yaml` file by adding the following line in the reactions section:

```
reactions:
- include: reactions.yaml
- include: mannitol_pathway.yaml
```

Alternatively you can copy an already changed `model.yaml` file from the additional files folder using the following command:

```
(psamm-env) $ cp ../additional_files/model.yaml .
```

This line tells PSAMM that these reactions are also going to be included in the model simulations.

Now you can test the model again to see if there were any effects from these new reactions added in. To run an FBA simulation you can use the following command:

```
(psamm-env) $ psamm-model fba --all-reactions
```

The `--all-reactions` option makes the command write out all reactions in the model even if they have a flux of zero in the simulation result. It can be seen that the newly added reactions are being read into the model since they do appear in the output. For example the *MANNI1DEH* reaction can be seen in the FBA output and it can be seen that this reaction is not carrying any flux. This is because there is no exchange reaction added into the model that would provide mannitol.

```
FRUKIN      0.0     |fru[c]| + |ATP[c]| => |D-Fructose-6-phosphate[c]| + |ADP[c]| +␣
↪|H[c]|
...
MANNI1PDEH  0.0     |Nicotinamide-adenine-dinucleotide[c]| + |manni1p[c]| => |D-
↪Fructose-6-phosphate[c]| + |H[c]| + |Nicotinamide-adenine-dinucleotide-reduced[c]|
MANNI1PPHOS 0.0     |manni1p[c]| + |H2O[c]| => |manni[c]| + |Phosphate[c]|
MANNIDEH    0.0     |Nicotinamide-adenine-dinucleotide[c]| + |manni[c]| =>␣
↪|Nicotinamide-adenine-dinucleotide-reduced[c]| + |fru[c]|
MANNIPTS    0.0     |manni[e]| + |Phosphoenolpyruvate[c]| => |manni1p[c]| +␣
↪|Pyruvate[c]|
...
```

Changing the Boundary Definitions Through the Exchange File

To add new exchange reactions to the model a modified `exchange.yaml` file has been included in the additional files. This new boundary condition could be added by creating a new entry in the existing `exchange.yaml` file but for this tutorial the exchange file can be changed by running the following command:

```
(psamm-env) $ cp ../additional_files/exchange.yaml .
```

This will simulate adding in the new mannitol compound into the exchange file as well as setting the uptake of glucose to be zero.

Now you can track changes to the exchange file using the Git command:

```
(psamm-env) $ git diff exchange.yaml
```

From the output, it can be seen that a new entry was added in the exchange file to add the mannitol exchange reaction and that the lower flux limit for glucose uptake was changed to zero. This will ensure that any future simulations done with the model in these conditions will only have mannitol available as a carbon source.

```
@@ -1,5 +1,7 @@
 name: Default medium
 compounds:
+- id: manni
+  lower: -10
 - id: ac_e
   reaction: EX_ac
   lower: 0.0
@@ -25,7 +27,7 @@
   lower: 0.0
 - id: glc_D_e
   reaction: EX_glc
-  lower: -10.0
+  lower: 0.0
 - id: gln_L_e
   reaction: EX_gln_L
   lower: 0.0
```

In this case the Git output indicates what lines were added or removed from the previous version. Added lines are indicated with a plus sign next to them. These are the new lines in the new version of the file. The lines with a minus sign next to them are the line versions from the old format of the file. This makes it easy to figure out exactly what changed between the new and old version of the file.

Now you can test out if the new reactions are functioning in the model. Since there is no other carbon source, if the model sustains flux through the biomass reaction it must be from the supplied mannitol. The following command can be used to run FBA on the model:

```
(psamm-env) $ psamm-model fba --all-reactions
```

From the output it can be seen that there is flux through the biomass reaction and that the mannitol utilization reactions are being used. In this situation it can also be seen that the pathway that converts mannitol to fructose first is not being used.

```
FRUKIN       0.0     |fru[c]| + |ATP[c]| => |D-Fructose-6-phosphate[c]| + |ADP[c]| +␣
↪|H[c]|
...
MANNI1PDEH   10.0    |Nicotinamide-adenine-dinucleotide[c]| + |manni1p[c]| => |D-
↪Fructose-6-phosphate[c]| + |H[c]| + |Nicotinamide-adenine-dinucleotide-reduced[c]|
MANNI1PPHOS 0.0     |manni1p[c]| + |H2O[c]| => |manni[c]| + |Phosphate[c]|
MANNIDEH     0.0     |Nicotinamide-adenine-dinucleotide[c]| + |manni[c]| =>␣
↪|Nicotinamide-adenine-dinucleotide-reduced[c]| + |fru[c]|
MANNIPTS     10.0    |manni[e]| + |Phosphoenolpyruvate[c]| => |manni1p[c]| +␣
↪|Pyruvate[c]|
```

You can also choose to maximize other reactions in the network. For example this could be used to analyze the network when production of a certain metabolite is maximized or to quickly change between different objective functions that are in the model. To do this you will just need to specify a reaction ID in the command and that will be used as the objective function for that simulation. For example if you wanted to analyze the network when the *FRUKIN* reaction is maximized the following command can be used:

```
(psamm-env) $ psamm-model fba --objective=FRUKIN --all-reactions
```

It can be seen from this simulation that the *FRUKIN* reaction is now being used and that the fluxes through the network have changed from when the biomass function was used as the objective function.

```
...
EX_lac_D_e  20.0    |D-Lactate[e]| <=>
EX_manni_e  -10.0   |manni[e]| <=>
EX_o2_e     -5.0    |O2[e]| <=>
EX_pi_e     0.0     |Phosphate[e]| <=>
EX_pyr_e    0.0     |Pyruvate[e]| <=>
EX_succ_e   0.0     |Succinate[e]| <=>
FBA 10.0    |D-Fructose-1-6-bisphosphate[c]| <=> |Dihydroxyacetone-phosphate[c]| +␣
↪|Glyceraldehyde-3-phosphate[c]|  b2097 or b1773 or b2925
FBP 0.0     |D-Fructose-1-6-bisphosphate[c]| + |H2O[c]| => |D-Fructose-6-
↪phosphate[c]| + |Phosphate[c]|     b3925 or b4232
FORt2       0.0     |Formate[e]| + |H[e]| => |Formate[c]| + |H[c]|  b0904 or b2492
FORti       0.0     |Formate[c]| => |Formate[e]|    b0904 or b2492
FRD7        0.0     |Fumarate[c]| + |Ubiquinol-8[c]| => |Ubiquinone-8[c]| +␣
↪|Succinate[c]|  b4151 and b4152 and b4153 and b4154
FRUKIN      10.0    |fru[c]| + |ATP[c]| => |D-Fructose-6-phosphate[c]| + |ADP[c]| +␣
↪|H[c]|
...
```

Adding new Compounds to the Model

In the previous two steps the reactions and boundary conditions were added into the model. There was no information added in about what the compounds in these reactions actually are but PSAMM is still able to treat them as metabolites in the network and utilize them accordingly. It will be helpful if there is information on these compounds in the model. This will allow you to use the various curation tools and will allow PSAMM to use the new compound names in the output of these various simulations. To add the new compounds to the model a modified `compounds.yaml` file has been provided in the `additional_files` folder. These compounds can be entered into the existing `compounds.yaml` file but for this tutorial the new version can be copied over by running the following command.

```
(psamm-env) $ cp ../additional_files/compounds.yaml .
```

Using the diff command in Git, you will be able to identify changes in the new `compounds.yaml` file:

```
(psamm-env) $ git diff compounds.yaml
```

It can be seen that the new compound entries added to the model were the various new compounds involved in this new pathway.

```
@@ -1,3 +1,12 @@
+- id: fru_c
+  name: Fructose
+  formula: C6H12O6
+- id: manni
+  name: Mannitol
+  formula: C6H14O6
+- id: manni1p
+  name: Mannitol 1-phosphate
+  formula: C6H13O9P
 - id: 13dpg_c
   name: 3-Phospho-D-glyceroyl-phosphate
   formula: C3H4O10P2
```

This will simulate adding in the new compounds to the existing database. Now you can run another FBA simulation to check if these new compound properties are being incorporated into the model. To do this run the following command:

```
(psamm-env) $ psamm-model fba --all-reactions
```

It can be seen that the reactions are no longer represented with compound IDs but are now represented with the compound names. This is because the new compound features are now being added to the model.

```
EX_manni_e  -10.0    |Mannitol[e]| <=>
...
FRUKIN      0.0      |Fructose[c]| + |ATP[c]| => |D-Fructose-6-phosphate[c]| +␣
↪|ADP[c]| + |H[c]|
...
MANNI1PDEH  10.0     |Nicotinamide-adenine-dinucleotide[c]| + |Mannitol 1-
↪phosphate[c]| => |D-Fructose-6-phosphate[c]| + |H[c]| + |Nicotinamide-adenine-
↪dinucleotide-reduced[c]|
MANNI1PPHOS 0.0      |Mannitol 1-phosphate[c]| + |H2O[c]| => |Mannitol[c]| +␣
↪|Phosphate[c]|
MANNIDEH    0.0      |Nicotinamide-adenine-dinucleotide[c]| + |Mannitol[c]| =>␣
↪|Nicotinamide-adenine-dinucleotide-reduced[c]| + |Fructose[c]|
MANNIPTS    10.0     |Mannitol[e]| + |Phosphoenolpyruvate[c]| => |Mannitol 1-
↪phosphate[c]| + |Pyruvate[c]|
```

Checking File Changes with Git

Now that the model has been updated it will be useful to track the changes that have been made.

First it will be helpful to get a summary of all the files have been modified in the model. Since the changes have been tracked with Git the files that have changed can be viewed by using the following Git command:

```
(psamm-env) $ git status
```

The output of this command should show that the exchange, compound, and `model.yaml` files have changed and that there is a new file that is not being tracked named `mannitol_pathway.yaml`. First the new mannitol pathway file can be added to the Git repository so that future changes can be tracked using the following commands:

```
(psmam-env) $ git add mannitol_pathway.yaml
```

Then specific changes in individual files can be viewed by using the `git diff` command followed by the file name. For example to view the changes in the `compounds.yaml` file the following command can be run.

```
(psamm-env) $ git diff model.yaml
```

The output should look like the following:

```
@@ -5,6 +5,7 @@ compounds:
   - include: compounds.yaml
   reactions:
   - include: reactions.yaml
+  - include: mannitol_pathway.yaml
   exchange:
   - include: exchange.yaml
   limits:
```

This can be done with any file that had changes to make sure that no accidental changes are added in along with whatever the desired changes are. In this example there should be one line added in the `model.yaml` file, three compounds added into the `compounds.yaml` file, and one exchange reaction added into the `exchange.yaml` file along with one change that removed glucose from the list of carbon sources in the exchange settings (by changing the lower bound of its exchange reaction to zero).

Once the changes are confirmed these files can be added with the Git add command.

```
(psamm-env) $ git add compounds.yaml
(psamm-env) $ git add exchange.yaml
(psamm-env) $ git add model.yaml
```

These changes can then be committed to the repository using the following command:

```
(psamm-env) $ git commit -m 'Addition of mannitol utilization pathway and associated␣
↪compounds'
```

Now the model has been updated and the changes have been committed. The Git log command can be used to view what commits have been made in the repository. This allows you to track the overall progress as well as examine what specific changes have been made. More detailed information between the commits can be viewed using the `git diff` command along with the commit ID that you want to compare the current version to. This will tell you specifically what changes occurred between that commit and the current version.

You can also view a log of the commits in the model by using the following command:

```
(psamm-env) $ git log
```

This can be helpful for getting an overall view of what changes have been made to a repository.

The Git version tracking can also be used with GitHub, BitBucket, GitLab or any other Git hosting provider to share repositories with other people. This can enable you to collaborate on different aspects of the modeling process while still tracking the changes made by different groups and maintaining a functional model.

2.2.5 Using PSAMM to export the model to other Software

If you want to export the model in a format to use with other software, that is also possible using PSAMM. The YAML formatted model can be easily exported as an SBML file using the following command:

```
(psamm-env) $ psamm-model sbmlexport Modified_core_ecoli.xml
```

This will export the model in SBML level 3 version 1 format which can then be used in other software that support this format.

2.3 Model Curation

This tutorial will go over how to utilize the curation functions in PSAMM to correct common errors and ensure that metabolic reconstructions are accurate representations of the metabolism of an organism.

2.3.1 Materials

For information on how to install *PSAMM* and the associated requirements, as well how to download the materials required for this tutorial you can reference the Installation and Materials section of the tutorial.

For this part of the tutorial we will be using a modified version of the E. coli core metabolic models that has been used in the other sections of the tutorial. This model has been modified to add in a new pathways for the utilization of mannitol as a carbon source. To access this model and the other files needed you will need to go into the tutorial-part-2 folder located in the psamm-tutorial folder.

```
(psamm-env) $ cd <PATH>/tutorial-part-2/
```

Once in this folder you should see two directories. One is the E_coli_yaml folder which contains the version of the model we will use. The other is called additional_files, which contains some files we will use during the tutorial.

2.3.2 Common Errors in Metabolic Reconstructions

Many types of errors can be introduced into metabolic models. Some errors can be introduced during manual editing of model files while others can result from inconsistent representations of the biology of the system. Various features in PSAMM are designed ot help identify and fix these problems to ensure that the reconstruction does not contain these kinds of errors.

Some errors cannot be easily identified without extensive manual inspection of the model data files. These PSAMM functions are designed to help identify these errors and make the correction process easier.

2.3.3 PSAMM Warnings

The most basic way to identify possible errors in a model will be through reading the warning messages printed out by PSAMM when any functions are run on a model. These warning messages can be an easy way to identify if something in the reconstruction is not set up the way that was intended. The following are examples of the types of warnings that PSAMM will provide and what kinds of errors they might indicate.

The first type of warning that PSAMM can provide is a waning that there is a compound that is in a reaction but is not defined in the compound information of the model. While PSAMM doesn't necessarily know if this is an error, these warning can help identify compound ids in the reconstruction that may have typos in them or that need to be defined in the compounds data for the reconstruction. For example in the warning below it would appear that the compound id for ATP had been mistyped and included two extra t's in it. These types of errors can make reactions in a model inconsistent and may lead to incorrect conclusions from the model if they are not corrected.

```
WARNING: The compound cpd_atttp was not defined in the list of compounds
```

The second type of warning will similarly help identify if there was an error introduced in one of the reconstruction's reactions. This warning will indicate that there is a compound present in the reconstruction that has a compartment that is not defined elsewhere in the model. In the example below a compound was added in a reaction as being in the compartment 'X'. Since this compartment was not used in the model the reaction involving this instance of the compound would become flux inconsistent.

```
WARNING: The compartment X was not defined in the list of compartments.
```

The third and fourth types of warnings can be useful in identify that the exchange file is set up correctly for the reconstruction. These two kinds of errors will help identify if there are compounds that are present in the extracellular compartment but do not have a corresponding exchange reaction in the boundary conditions. This can be problematic for some models that require certain sinks for overproduced compounds in the boundary. The other kind of warning will indicate if there are compounds in the exchange reactions that cannot be utilized by any reactions in the model.

This could indicate that a transport reaction is missing from the model or that the compound could be removed from the exchange file.

```
WARNING: The compound cpd_chitob was in the extracellular compartment but not defined␣
↪in the medium
WARNING: The compound cpd_etoh was defined in the medium but is not in the␣
↪extracellular compartment
```

2.3.4 Reaction Consistency in PSAMM

The previous examples of warning messages produced by PSAMM can be helpful as a first step in identifying possible errors in a model but there are various other types of errors that may be present in models that specific PSAMM functions can help identify. The first kind of errors are ones related to the balancing of reactions in model. It is important that metabolic models be balanced in terms of elements, charge, and stoichiometry. PSAMM has three functions available to identify reactions that are not balanced in these properties which can help correct them and lead to more accurate and true representations of metabolism.

Stoichiometric Checking

PSAMM's masscheck tool can be used to check if the reactions in the model are stoichiometrically consistent and the compounds that are causing the imbalance. This can be useful when curating the model because it can assist in easily identify missing compounds in reactions. A common problem that can be identified using this tool is a loss of hydrogen atoms during a metabolic reaction. This can occur due to modeling choices or incomplete reaction equations but is generally easy to identify using masscheck.

To report on the compounds that are not balanced use the following masscheck command:

```
(psamm-env) $ psamm-model masscheck
```

This command will produce an output like the following:

```
...
accoa_c      1.0      Acetyl-CoA
acald_e      1.0      Acetaldehyde
acald_c      1.0      Acetaldehyde
h_e 0.0      H
h_c 0.0      H
INFO: Consistent compounds: 73/75
```

The `masscheck` command will first try to assign a positive mass to all of the compounds in the model while balancing the masses such that the left-hand side and right-hand side add up in every model reaction. All the compound masses are reported, and the compounds that have been assigned a zero value for the mass are the ones causing imbalances.

In certain cases a metabolic model can contain compounds that represent electrons, photons, or some other artificial compound. These compounds can cause problems with the stoichiometric balance of a reaction because of their unique functions. In order to deal with this an additional property can be added to the compound entry that will designate it as a compound with zero mass. This designation will tell PSAMM to consider these compounds to have no mass during the stoichiometric checking which will prevent them from causing imbalances in the reactions. An example of how to add that property to a compound entry can be seen below:

```
- id: phot
  name: Photon
  zeromass: yes
```

To report on the specific reactions that may be causing the imbalance, the following command can be used:

```
(psamm-env) $ psamm-model masscheck --type=reaction
...
FRUKIN      1.0     |Fructose[c]| + |ATP[c]| => |D-Fructose-6-phosphate[c]| +␣
↪|ADP[c]| + |H[c]|
INFO: Consistent reactions: 100/101
```

This check is performed similarly to the compound check. In addition, mass residual values are introduced for each metabolic reaction in the network. These mass residuals are then minimized and any reactions that result in a non-zero mass residual value after minimization are reported as being stoichiometrically inconsistent. A non-zero residual value after minimization tells you that the reaction in question may be unbalanced and missing some mass from it.

Sometimes the residue minimization problem may have multiple solutions. In these cases the residue value may be reallocated among a few connected reactions. In this example the unbalanced reaction is the MANNIDEH reaction:

```
MANNIDEH    |manni[c]| + |nad[c]| => |fru[c]| + |nadh[c]|
```

In this reaction equation the right hand side is missing a proton. However minimization problem can result in the residue being placed on either the *fru_c* or the *nadh_c* compounds in an attempt to balance the reaction. Because *nadh_c* occurs in thirteen other reactions in the network, the program has already determined that that compound is stoichiometrically consistent. On the other hand *fru_c* only occurs one other time. Since this compound is less connected the minimization problem will assign the non-zero residual to this compound. This process results in the FRUKIN reaction which contains this compound as being identified as being stoichiometrically inconsistent.

In these cases you will need to manually check the reaction and then use the `--checked` option for the `masscheck` command to force the non-zero residual to be placed on a different reaction. This will rerun the consistency check and force the residual to be placed on a different reaction. To do this we would run the following command.

```
(psamm-env) $ psamm-model masscheck --type=reaction --checked FRUKIN
```

Now, the output should report the *MANNIDEH* reaction and it can be seen that the reaction equation of *MANNIDEH* is specified incorrectly. It appears that a hydrogen compound was left out of the reaction for *MANNIDEH*. This would be an easy problem to correct by simply adding in a hydrogen compound to correct the lost atom in the equation.

The stoichiometric consistency checking allows for the easy identification of stoichiometrically inconsistent compounds while providing a more targeted subset of reactions to check to fix the problem. This allows you to quickly identify problematic reactions rather than having to manually go through the whole reaction database in an attempt to find the problem.

In some cases there are reactions that are going to be inherently unbalanced and might cause problems with using these methods. If you know that this is the case for a specific reaction they can specify that the reaction be excluded from the mass check so that the rest of the network can be analyzed. To do this the `--exclude` option can be used. For example if you wanted to exclude the reaction *FRUKIN* from the mass check they could use the following command:

```
(psamm-env) $ psamm-model masscheck --exclude FRUKIN
```

This exclude option can be helpful in removing inherently unbalanced reactions like macromolecule synthesis reations or incomplete reactions that would be identified as being stoichiometrically inconsistent. It is also possible to create a file that lists multiple reactions to exclude. Put each reaction identifier on a separate line in the file and refer to the file be prefixing the file name with a `@`:

```
(psamm-env) $ psamm-model masscheck --exclude @excluded_reactions.txt
```

Before we fix the model with the correction to the *MANNIDEH* reaction, let us first check the model for formula inconsistencies to show how this can also be used in conjunction with mass checking and other methods to correct model inconsistencies.

Formula Consistency Checking

Formula checking will check that each reaction in the model is balanced with respect to the chemical formulas of each compound. To check the model for formula consistencies run the formula check command:

```
(psamm-env) $ psamm-model formulacheck
```

The output should appear as follows:

```
INFO: Model: Ecoli_core_model
INFO: Model Git version: 9812080
MANNIDEH    C27H40N7O20P2   C27H39N7O20P2           H
Biomass_Ecoli_core_w_GAM    C1088.0232H1471.1810N446.7617O1236.7018P240.5298S3.7478␣
↪C1045.4677H1395.2089N441.3089O1189.0281P236.8511S3.7478         C42.5555H75.9721N5.
↪4528O47.6737P3.6787
INFO: Unbalanced reactions: 2/80
INFO: Unchecked reactions due to missing formula: 0/80
```

In this case two reactions are identified in the model as being unbalanced. The biomass objective function, *Biomass_Ecoli_core_w_GAM*, and the reaction that was previously identified through masscheck as being unbalanced, *MANNIDEH*. In the case of the objective function this is imbalanced due to the formulation of the objective function. The reaction functions as a sink for the compounds required for growth and only outputs depleted energy compounds. This leads to it being inherently formula imbalanced but it is a necessary feature of the model. The other reaction is *MANNIDEH*. It can be seen that the total number of atoms on each side does not match up. PSAMM also outputs what atoms would be needed to balance the reaction on both sides. In this case there is a missing hydrogen atom on the right side of the equation. This can be easily rectified by adding in the missing hydrogen. To do this correction in this tutorial, you can copy a fixed version of the mannitol pathway from the additional files folder using the following command:

```
(psamm-env) $ cp ../additional_files/mannitol_pathway_v2.yaml mannitol_pathway.yaml
```

Once that problem with the new reaction is fixed the model will pass both the formula check and mass check.

Charge Consistency Checking

The charge consistency function is similar to the formula consistency function but instead of using the chemical formulas for the compounds, PSAMM will use the assigned charges that are designated in the compounds file and check that these charges are balanced on both sides of the reaction.

To run a charge consistency check on the model use the chargecheck command:

```
(psamm-env) $ psamm-model chargecheck
```

This *E. coli* SBML model does not contain charge information for the compounds. A sample output is provided below to show what the results would look like for a charge imbalanced model. The output from the charge check will display any reactions that are charge imbalanced and show what the imbalance is and then show the reaction equation. This can be used to quickly check for any missed inconsistencies and identify reactions and compounds that should be looked at more closely to confirm their correctness.

```
...
rxn12510    1.0     |ATP[c]| + |Pantothenate[c]| => |4-phosphopantothenate[c]| +␣
↪|H+[c]| + |ADP[c]|
rxn12825    4.0     |hemeO[c]| + |H2O[c]| => |Heme[c]| + (4) |H+[c]|
rxn13643    1.0     |ADP-glucose[c]| => |Glycogen[c]| + |H+[c]| + |ADP[c]|
rxn13710    6.0     (5) |D-Glucose[c]| + (4) |ATP[c]| => |Glycogen[c]| + (4) |H+[c]|␣
↪+ (4) |Phosphate[c]| + (4) |H2O[c]| + |ADP[c]|
```

(continues on next page)

(continued from previous page)

```
INFO: Unbalanced reactions: 94/1093
INFO: Unchecked reactions due to missing charge: 0/1093
```

Flux Consistency Checking

The flux consistency checking function can be used to identify reactions that cannot carry flux in the model. This tool can be used as a curation tool as well as an analysis tool. In this tutorial it will be highlighted for the curation aspects and later its use in flux analysis will be demonstrated.

To run a flux consistency check on the model use the `fluxcheck` command:

```
(psamm-env) $ psamm-model fluxcheck --unrestricted
```

The unrestricted option with the command will tell PSAMM to remove any limits on the exchange reactions. This will tell you which reactions in the model can carry flux if the model is given all compounds in the media freely. This can be helpful for identifying which reactions may not be linked to other parts of the metabolism and can be helpful in identifying gaps in the model. In this case it can be seen that no reactions were identified as being inconsistent.

In some situations there are pathways that might be modeled but not necessarily connected to the other aspects of metabolism. A common occurrence of this is with vitamin biosynthesis pathways that are not incorporated into the biomass in the model. `fluxcheck` will identify these as being flux inconsistent but the modeler will need to identify if this is due to incomplete information on the pathways or if it is due to some error in the formulation of the reactions.

PSAMM will tell you how many exchange reactions cannot be used as well as how many internal model reactions cannot carry flux. PSAMM will also list the reactions and the equations for the reactions to make curation of these reactions easier.

Above the `fluxcheck` command was used with the –unrestricted option which allowed the exchange reactions to all be active. This command can also be used to see what reactions cannot carry flux when specific media are supplied. To run this command on the network with the media that is specified in the media file run the following command:

```
(psamm-env) $ psamm-model fluxcheck
INFO: Model: Ecoli_core_model
INFO: Model Git version: 9812080
INFO: Using flux bounds to determine consistency.
...
EX_fru_e     |D-Fructose[e]| <=>
EX_fum_e     |Fumarate[e]| <=>
EX_glc_e     |D-Glucose[e]| <=>
EX_gln_L_e   |L-Glutamine[e]| <=>
EX_mal_L_e   |L-Malate[e]| <=>
FRUpts2      |D-Fructose[e]| + |Phosphoenolpyruvate[c]| => |D-Fructose-6-phosphate[c]|␣
↪+ |Pyruvate[c]|
FUMt2_2      (2) |H[e]| + |Fumarate[e]| => (2) |H[c]| + |Fumarate[c]|
GLCpts       |Phosphoenolpyruvate[c]| + |D-Glucose[e]| => |Pyruvate[c]| + |D-Glucose-6-
↪phosphate[c]|
GLNabc       |ATP[c]| + |L-Glutamine[e]| + |H2O[c]| => |L-Glutamine[c]| + |ADP[c]| +␣
↪|H[c]| + |Phosphate[c]|
MALt2_2      |L-Malate[e]| + (2) |H[e]| => |L-Malate[c]| + (2) |H[c]|
INFO: Model has 5/80 inconsistent internal reactions (0 disabled by user)
INFO: Model has 5/21 inconsistent exchange reactions (0 disabled by user)
```

In this case it can be seen that there are various exchange reactions blocked as well as various internal reactions related to other carbon metabolic pathways. The current model should only be supplying mannitol as a carbon source and this would mean that these other carbon pathways would be blocked in this condition. In this way, you can use the `fluxcheck` command to see what reactions are specific to certain metabolic pathways and environmental conditions.

2.3.5 Gap Identification in PSAMM

In addition to inconsistencies found within individual reactions there can also be global inconsistencies for the reactions within a metabolic network. These include metabolites that can be produced but not consumed, ones that can be consumed by reactions but are not produced, and reactions that cannot carry flux in a model. PSAMM includes various functions for the identification of these features in a network including the functions `gapcheck` and `fluxcheck`. Additionally the functions `gapfill` and `fastgapfill` can be used to help fill these gaps that are present through the introduction of additional reactions into the network.

Gapcheck in PSAMM

The `gapcheck` function in *PSAMM* can be used to identify dead end metabolites in a metabolic network. These dead end metabolites are compounds in the metabolic model that can either be produced but not consumed or ones that can be consumed but not produced. Reactions that contain these compounds cannot carry flux within a model and are often the result of knowledge gaps in our understanding of metabolic networks.

The `gapcheck` function allows the use of three methods for the identification of these dead end metabolites within a metabolic network. These are the `prodcheck`, `sinkcheck`, and `gapfind` methods.

The `prodcheck` method is the most straightforward of these methods and can be used to identify any compounds that cannot be produced in the metabolic network. It will iterate through the reactions in a network and maximize each one. If the reaction can carry a flux then the metabolites involved in the reaction are not considered to be blocked.

To use this function the following command can be run:

```
(psamm-env) $ psamm-model gapcheck --method prodcheck
```

The function will produce output like the following that lists out any metabolites in the model that cannot be produced in this condition:

```
fru[e]      D-Fructose
fum[e]      Fumarate
glc_D[e]    D-Glucose
gln_L[e]    L-Glutamine
mal_L[e]    L-Malate
INFO: Blocked compounds: 5
```

This result indicates that the following metabolites currently cannot be produced in the model. This only tells part of the story though, as this function was run with the defined media that was set for the model. As a result there are gaps identified like, ‘D-Glucose’, that will not be considered gaps in other conditions. To do a global check using this function on the model without restrictions on the media the following command can be used:

```
(psamm-env) $ psamm-model gapcheck --method prodcheck --unrestricted-exchange
```

The unrestricted tag in this function will temporarily set all of the exchange reaction bounds to be -1000 to 1000 allowing all nutrients to be either taken up or produced. Gap-checking in this condition will allow for the identification of gaps that are not media dependent and may instead be the result of incomplete pathways and knowledge gaps.

The second method implemented in the `gapcheck` function is the `sinkcheck` method. This method is similar to `prodcheck` but is implemented in a way where the flux through each introduced sink for a compound is maximized. This ensures that the metabolite can be produced in excess from the network for it to not be considered a dead end metabolite.

```
(psamm-env) $ psamm-model gapcheck --method sinkcheck --unrestricted-exchange
```

The last method implemented in the `gapcheck` function is the `gapfind` method. This method is an implementation of a previously published method to identify gaps in metabolic networks [Kumar07]. This method will use a network based optimization to identify metabolites with no production pathways present.

```
(psamm-env) $ psamm-model gapcheck --method gapfind --unrestricted-exchange
```

These methods included in the `gapcheck` function can be used to identify various kinds of 'gaps' in a metabolic model network. *PSAMM* also includes three functions for filling these gaps through the addition of artificial reactions or reactions from a supplied database. The functions `gapfill`, `fastgapfill`, and `completepath` can be used to perform these gapfilling procedures during the process of generating and curating a model.

GapFill

The `gapfill` function in PSAMM can be used to apply a GapFill algorithm based on [Kumar07] to a metabolic model to search for and identify reactions that can be added into a model to unblock the production of a specific compound or set of compounds. To provide an example of how to utilize this `gapfill` function a version of the E. coli core model has been provided in the *tutorial-part-2/Gapfilling_Model/* directory. In this directory is the E. coli core model with a small additional, incomplete pathway, added that contains the following reactions:

```
- id: rxn1
  equation: succ_c[c] => a[c]
- id: rxn3
  equation: b[c] => c[c] + d[c]
```

This small additional pathway converts succinate to an artificial compound 'a'. The other reaction can convert compound 'b' to 'c' and 'd'. There is no reaction to convert 'a' to 'b' though, and this can be considered a metabolic gap. In an additional reaction database, but not included in the model itself, is an additional reaction:

- id: rxn2 equation: a[c] => b[c]

This reaction, if added would be capable of unblocking the production of 'c' or 'd', by allowing for the conversion of compound 'a' to 'b'. In most cases when performing gap-filling on a model a larger database of non-model reactions could be used. For this test case the production of compound 'd[c]' could be unblocked by running the following command:

```
(psamm-env) psamm-model gapfill --compound d[c]
```

This would produce an output that first lists all of the reactions from the original metabolic model. Then lists the included gap-filling reactions with their associated penalty values. And lastly will list any reactions where the gap-filling result suggests that the flux bounds of the reaction be changed. A sample of the reaction is shown below:

```
....
TPI Model    0       Dihydroxyacetone-phosphate[c] <=> Glyceraldehyde-3-phosphate[c]
rxn1         Model   0       Succinate[c] => a[c]
rxn3         Model   0       b[c] => c[c] + d[c]
rxn2         Add     1       a[c] => b[c]
```

Some additional options can be used to refine the gap-filling. The first of these options is `--no-implicit-sinks` option that can be added to the command. If this option is used then the gap-filling will be performed with no implicit sinks for compounds, meaning that all compounds produced need to be consumed by other reactions in the metabolic model. By default, if this option is not used with the command, then implicit sinks are added for all compounds in the model meaning that any compound that is produced in excess can be removed through the added sinks.

The other way to refine the gap-filling procedure is through defining specific penalty values for the addition of reactions from different sources. Penalties can be set for specific reactions in a gap-filling database through a tab separated file provided in the command using the `--penalty` option. Additionally penalty values for all database reactions can be set using the `--db-penalty` option followed by a penalty value. Similarly penalty values can be assigned to added transport reactions using the `--tp-penalty` option and to added exchange reactions using the `--ex-penalty` option. An example of a command that applies these penalties to a gap-filling simulation would be like follows:

```
(psamm-env) $ psamm-model gapfill --compound d[c] --ex-penalty 100 --tp-penalty 10 --
↪db-penalty 1
```

The `gapfill` function in PSAMM can be used through the model construction process to help identify potential new reactions to add to a model and to explore how metabolic gaps effect the capabilities of a metabolic network.

FastGapFill

The `fastgapfill` function in *PSAMM* is different gap-filling method that uses the FastGapFill algorithm to attempt to generate a gap-filled model that is entirely flux consistent [Thiele14]. The implementation of this algorithm in *PSAMM* can be utilized for unblocking an entire metabolic model or for unblocking specific reactions in a network. Often times unblocking all of the reactions in a model at the same time will not produce the most meaningful and easy to understand results so only performing this function on a subset of reactions is preferable. To do this the `--subset` option can be used to provide a file that contains a list of reactions to unblock. In this example that list would look like this:

```
rxn1
rxn3
```

This file can be provided to the command to unblock the small artificial pathway that was added to the E. coli core model:

```
(psamm-env) $ psamm-model fastgapfill --subset subset.tsv
```

In this case the output from this command will show the following:

```
....
TPI Model    0       Dihydroxyacetone-phosphate[c] <=> Glyceraldehyde-3-phosphate[c]
rxn1         Model   0       Succinate[c] => a[c]
rxn3         Model   0       b[c] => c[c] + d[c]
EX_c[e]      Add     1       c[e] <=>
EX_d[e]      Add     1       d[e] <=>
EX_succ_c[e]         Add     1       Succinate[e] <=>
TP_c[c]_c[e]         Add     1       c[c] <=> c[e]
TP_d[c]_d[e]         Add     1       d[c] <=> d[e]
TP_succ_c[c]_succ_c[e]       Add     1       Succinate[c] <=> Succinate[e]
rxn2         Add     1       a[c] => b[c]
```

The output will first list the model reactions which are labeled with the 'Model' tag in the second column of the output. *PSAMM* will list out any artificial exchange and transporters, as well as any gap reactions included from the larger database. These will be labeled with the *Add* tag in the second column. When compared to the `gapfill` results from the previous section it can be seen that the `fastgapfill` result suggests some artificial transporters and exchange reactions for certain compounds. This is due to this method trying to find a flux consistent gap-filling solution.

Penalty values can be assigned for different types of reactions in the same way that they are in the `gapfill` command. With `--ex-penalty` for artificial exchange reactions, `--tp-penalty` for artificial transporters, `--db-penalty` for new database reactions, and penalties on specific reactions through a penalty file provided with the `--penalty` option.

2.3.6 Search Functions in PSAMM

`psamm-model` includes a search function that can be used to search the model information for specific compounds or reactions. To do this the search function can be used. This can be used for various search methods. For example to search for the compound named fructose the following command can be used:

```
(psamm-env) $ psamm-model search compound --name 'Fructose'
INFO: Model: Ecoli_core_model
INFO: Model Git version: db22229
id: fru_c
formula: C6H12O6
name: Fructose
Defined in ./compounds.yaml:?:?
```

To do the same search but instead use the compound ID the following command can be used:

```
(psamm-env) $ psamm-model search compound --id 'fru_c'
```

These searches will result in a printout of the relevant information contained within the model about these compounds. In a similar way reactions can also be searched. For example to search for a reaction by a specific ID the following command can be used:

```
(psamm-env) $ psamm-model search reaction --id 'FRUKIN'
```

Or to search for all reactions that include a specific compound the following command can be used:

```
(psamm-env) $ psamm-model search reaction --compound 'manni[c]'
```

2.3.7 Duplicate Reaction Checks

An additional searching function called `Dupcheck` is also included in *PSAMM*. This function will search through a model and compare all of the reactions in the network to each other. Any reactions that have all of the same metabolites consumed and produced will then be reported. This can be a helpful function to use if there a multiple people working on the construction of a model as it allows for an automated checking that two individuals did not add the same reaction to the reconstruction. The `dupcheck` function can be run through the following command:

```
(psamm-env) $ psamm-model dupcheck
```

The additional tags `--compare-direction` and `--compare-stoichiometry` can be added to the command to take into account the reaction directionality and metabolite stoichiometry when comparing two different reactions.

2.4 Constraint Based Analysis with PSAMM

This tutorial will go over how to use the constraint based analysis methods that are included in PSAMM. These methods can be used to perform various simulations of growth with metabolic models. These simulations can be used to explore growth phenotypes, nutrient utilization, and gene essentiality.

2.4.1 Tutorial Materials

The materials used in the part of the tutorial can be found in the *tutorial-part-3* directory in the psamm-tutorial repository. This directory contains a copy of the E. coli core metabolic model that has been used in the other tutorials. This model can be used to run all of the simulations in this part of the tutorial. In addition to the model the virtual environment where PSAMM has been installed will need to be activated to run the *psamm-model* commands. For instructions on how to install or activate *PSAMM* in a virtual environment reference the Installation and Materials section of the tutorial.

To access the materials needed to run the following commands go to the E_coli_yaml folder in the tutorial-part-3 folder.

```
(psamm-env) $ cd <PATH>/tutorial-part-3/E_coli_yaml
```

2.4.2 Constraint-based Flux Analysis with PSAMM

Along with the various curation tools that are included with PSAMM there are also various flux analysis tools that can be used to perform simulations on the model. This allows for a seamless integration of the model development, curation, and simulation processes.

There are various options that you can change in these different flux analysis commands. Before introducing the specific commands these options will be detailed here.

Loop Minimization in PSAMM

First, you can choose the options for loop minimization when running constraint-based analyses. This can be done by using the `--loop-removal` option. There are three options for loop removal when performing constraint based analysis:

none No removal of loops

tfba Removes loops by applying thermodynamic constraints

l1min Removes loops by minimizing the L1 norm (the sum of absolute flux values)

For example, you could run flux balance analysis with thermodynamic constraints:

```
(psamm-env) $ psamm-model fba --loop-removal=tfba
```

or without:

```
(psamm-env) $ psamm-model fba --loop-removal=none
```

Choosing Linear Programming Solvers

You also have the option to set which solver you want to use for the linear programming problems. To view the solvers that are currently installed the following command can be used:

```
(psamm-env) $ psamm-list-lpsolvers
```

By default PSAMM will use CPLEX if it available but if you want to specify a different solver you can do so using the `--solver` option. For example to select the Gurobi solver during an FBA simulation you can use the following command:

```
(psamm-env) $ psamm-model fba --solver name=gurobi
```

If multiple solvers are installed and you do not want to use the default solver, you will need to set this option for every simulation run.

Note: The QSopt_ex solver does not support integer linear programming problems. This solver can be used with any commands but you will not be able to run the simulation with thermodynamic constraints.

Other Global Options

Another option that can be used with the various flux analysis commands is the `--epsilon` option. This option can be used to set the minimum value that a flux needs to be above to be considered non-zero. By default PSAMM will consider any number above 10^{-5} to be non-zero. An example of changing the epsilon value with this option during an FBA simulation is:

```
(psamm-env) $ psamm-model fba --epsilon 0.0001
```

These various options can be used for any of the flux analysis functions in PSAMM by adding them to the command that is being run. A list of the functions available in PSAMM can be viewed by using the command:

```
(psamm-env) $ psamm-model --help
```

The options for a specific function can be viewed by using the command:

```
(psamm-env) $ psamm-model <command> --help
```

2.4.3 FBA in PSAMM

PSAMM allows for the integration of the model development and curation process with the simulation process. In this way changes to a metabolic model can be immediately tested using the various flux analysis tools that are present in PSAMM. In this tutorial, aspects of the *E. coli* core model [Orth11] will be expanded to demonstrate the various functions available in PSAMM and throughout these changes the model will be analyzed with PSAMM's simulation functions to make sure that these changes are resulting in a functional model.

Flux Balance Analysis

Flux Balance Analysis (FBA) is one of the basic methods that allows you to quickly examine if the model is viable (i.e. can produce biomass). PSAMM provides the `fba` function in the `psamm-model` command to perform FBA on metabolic models. For example, to run FBA on the *E. coli* core model first make sure that the current directory is the `E_coli_yaml/` directory using the following command:

```
(psamm-env) $ cd <PATH>/psamm-tutorial/E_coli_yaml/
```

Then run FBA on the model with the following command.

```
(psamm-env) $ psamm-model fba
```

Note that the command above should be executed within the folder that stores the `model.yaml` file. Alternatively, you could run the following command anywhere in your file system:

```
(psamm-env) $ psamm-model --model <PATH-TO-MODEL.YAML> fba
```

The following is a sample of some output from the FBA command:

```
INFO: Model: Ecoli_core_model
INFO: Model Git version: 9812080
INFO: Using Biomass_Ecoli_core_w_GAM as objective
INFO: Loop removal disabled; spurious loops are allowed
INFO: Setting feasibility tolerance to 1e-09
INFO: Setting optimality tolerance to 1e-09
INFO: Solving took 0.05 seconds
ACONTa      6.00724957535   |Citrate[c]| <=> |cis-Aconitate[c]| + |H2O[c]|  b0118 or␣
↪b1276
ACONTb      6.00724957535   |cis-Aconitate[c]| + |H2O[c]| <=> |Isocitrate[c]|      ␣
↪b0118 or b1276
AKGDH       5.06437566148   |2-Oxoglutarate[c]| + |Coenzyme-A[c]|...
...
INFO: Objective flux: 0.873921506968
INFO: Reactions at zero flux: 47/95
```

At the beginning of the output of `psamm-model` commands information about the model as well as information about simulation settings will be printed. At the end of the output PSAMM will print the maximized flux of the designated objective function. The rest of the output is a list of the reaction IDs in the model along with their fluxes, and the reaction equations represented with the compound names. This output is human readable because the reactions equations are represented with the full names of compound. It can be saved as a tab separated file that can be sorted and analyzed quickly allowing for easy analysis and comparison between FBA in different conditions.

By default, PSAMM fba will use the biomass function designated in the central model file as the objective function. If the biomass tag is not defined in a `model.yaml` file or if you want to use a different reaction as the objective function, you can simply specify it using the `--objective` option. For example to maximize the citrate synthase reactions, *CS*, the command would be as follows:

```
(psamm-env) $ psamm-model fba --objective=CS
```

Flux balance analysis will be used throughout this tutorial as both a checking tool during model curation and an analysis tool. PSAMM allows you to easily integrate analysis tools like this into the various steps during model development.

Flux Variability Analysis

Another flux analysis tool that can be used in PSAMM is flux variability analysis. This analysis will maximize the objective function that is designated and provide a lower and upper bound of the various reactions in the model that would still allow the model to sustain the same objective function flux. This can provide insights into alternative pathways in the model and allow the identification of reactions that can vary in use.

To run FVA on the model use the following command:

```
(psamm-env) $ psamm-model fva
...
EX_pi_e     -3.44906664664  -3.44906664664  |Phosphate[e]| <=>
EX_pyr_e    -0.0    -0.0    |Pyruvate[e]| <=>
EX_succ_e   -0.0    -0.0    |Succinate[e]| <=>
FBA 7.00227721609   7.00227721609   |D-Fructose-1-6-bisphosphate[c]| <=>␣
↪|Dihydroxyacetone-phosphate[c]| + |Glyceraldehyde-3-phosphate[c]|
FBP 0.0     0.0     |D-Fructose-1-6-bisphosphate[c]| + |H2O[c]| => |D-Fructose-6-
↪phosphate[c]| + |Phosphate[c]|
```

(continues on next page)

(continued from previous page)

```
FORt2        0.0      0.0      |Formate[e]| + |H[e]| => |Formate[c]| + |H[c]|
...
```

The output shows the reaction IDs in the first column and then shows the lower bound of the flux, the upper bound of the flux, and the reaction equations. With the current conditions the flux is not variable through the equations in the model. It can be seen that the upper and lower bounds of each reaction are the same. If another carbon source was added in though it would allow for more reactions to be variable. For example if glucose was added into the media along with mannitol then the results might appear as follows:

```
EX_glc_e     -10.0    -2.0     |D-Glucose[e]| <=>
EX_manni_e   -9.0     -3.0     |Mannitol[e]| <=>
MANNIPTS     3.0      9.0      |Mannitol[e]| + |Phosphoenolpyruvate[c]| => |Mannitol 1-
↪phosphate[c]| + |Pyruvate[c]|
GLCpts       2.0      10.0     |D-Glucose[e]| + |Phosphoenolpyruvate[c]| =>␣
↪|Pyruvate[c]| + |D-Glucose-6-phosphate[c]|
```

It can be seen that in this situation the lower and upper bounds of some reactions are different indicating that their flux can be variable. This indicates that there is some variability in the model as to how certain reactions can be used while still maintaining the same objective function flux.

Robustness Analysis

Robustness analysis can be used to analyze the model under varying conditions. Robustness analysis will maximize a designated reaction while varying the flux through another designated reaction. For example, you could vary the amount of oxygen present while trying to maximize the biomass production to see how the model responds to different oxygen supply. You can specify the number of steps that will be performed in the robustness as well as the reaction that will be varied during the steps.

By default, the reaction that is maximized will be the biomass reaction defined in the `model.yaml` file but a different reaction can be designated with the optional `--objective` option. The flux bounds of this reaction will then be obtained to determine the lower and upper value for the robustness analysis. These values will then be used as the starting and stopping points for the robustness analysis. You can also set a customized upper and lower flux value of the varying reaction using the `--lower` and `--upper` options.

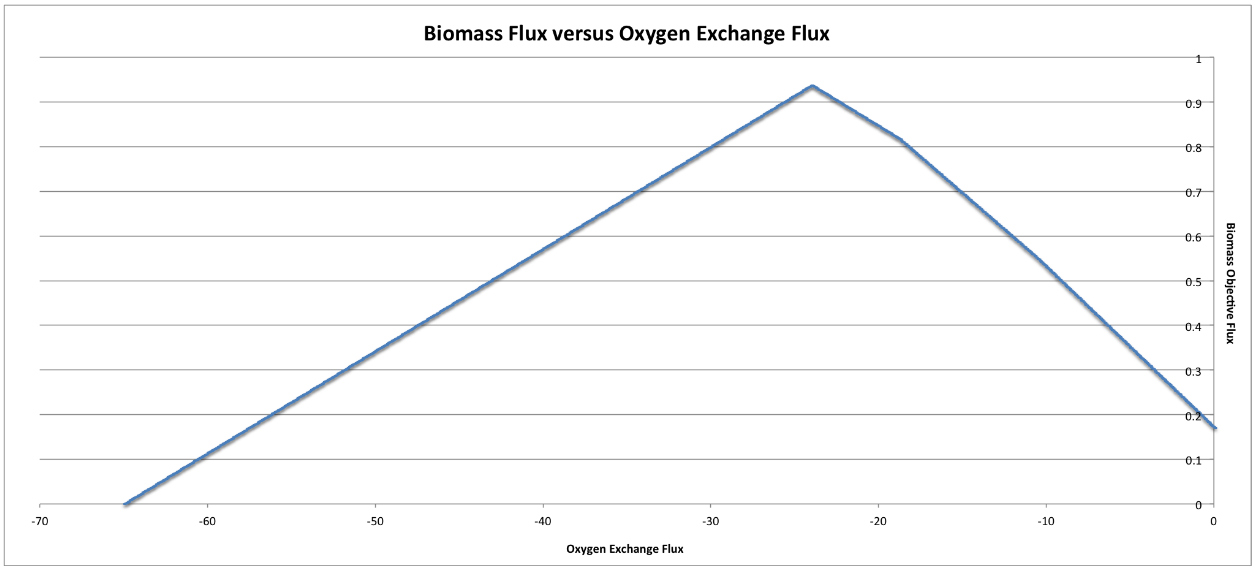
For this model the robustness command will be used to see how the model responds to various oxygen conditions with mannitol as the supplied carbon source. To run the robustness command use the following command:

```
(psamm-env) $ psamm-model robustness --steps 1000 EX_o2_e
```

The output will contain two columns. The first column will be the flux of the varied reaction, in this case the EX_o2_e reaction for oxygen exchange. The second shows the flux of the biomass reaction for the model. The output will look like this:

```
-63.958958959       0.0238161275506
-63.8938938939      0.0253046355225
-63.8288288288      0.0267931434944
-63.7637637638      0.0282816514663
-63.6986986987      0.0297701594383
-63.6336336336      0.0312586674102
-63.5685685686      0.0327471753821
-63.5035035035      0.034235683354
-63.4384384384      0.0357241913259
...
```

If the biomass reaction flux is plotted against the oxygen uptake it can be seen that the biomass flux is low at the highest oxygen uptake, reaches a maximum at an oxygen uptake of about 24, and then starts to decrease with low oxygen uptake.



If a more detailed analysis of internal fluxes is desired the *–all-reaction-fluxes* tag can be added to the command. This will print out all of the internal reaction fluxes for each step in the robustness analysis. The first column printed will be the reaction ID. The second column will be the varying reaction's flux and the last column will be the flux of the reaction listed in the first column. This can be used to look at the effects of a reaction on internal fluxes in the network. The command to run this would be the following:

```
(psamm-env) $ psamm-model robustness --all-reaction-fluxes --steps 1000 EX_o2_e
```

And the output for this command will look like the following:

```
G6PDH2r     -63.958958959   0.0
AKGDH       -63.958958959   0.0
GLNS        -63.958958959   0.00608978381469
ADK1        -63.958958959   0.0
PYRt2r      -63.958958959   0.0
EX_co2_e    -63.958958959   58.986492784
ATPM        -63.958958959   8.39
SUCCt2_2    -63.958958959   0.0
PIt2r       -63.958958959   0.0876123884204
EX_lac_D_e  -63.958958959   0.0
```

Deletion Simulations with PSAMM

Gene Deletion

The `genedelete` command can be used to perform gene deletions in a model and test what effects those deletions have. This command can be used to quickly test if certain genes are essential in the network. The command will take a list of genes in a separate file and will then go through all of the gene associations in the model to determine what reactions require that gene to be present. This uses the gene association logic to determine if the removal of the specified genes would knock out that function. For example if we had the following two reactions:

```
- id: RXN_1
  genes: g0001 and g0002
  equation: '|cpd_a[c]| <=> |cpd_b[c]|'

- id: RXN_2
  genes: g0001 or g0003
  equation: '|cpd_a[c]| <=> |cpd[c]|'
```

Both reactions are associated with the gene ‘g0001’ but RXN_1 has an ‘and’ association while RXN_2 has an ‘or’ association. If the gene ‘g0001’ were to be deleted from the network RXN_1 would no longer have the required genes for it to be present since both genes are required. RXN_2 would still be satisfied since it would only require one of the two genes to be present. The gene delete command will do this automatically and for the entire network making it much easier to do these kinds of simulations. The gene delete command can be run with the following command.

```
(psamm-env) $ psamm-model genedelete --gene b1779
```

This will produce a flux balance analysis result with a model that has any reactions for which b0118 is necessary limited to zero flux. The output will show a percentage of the biomass flux of the wild type model that can be produced by the deletion model.

```
...
INFO: Objective reaction after gene deletion has flux 0.0
INFO: Objective reaction has 0.00% flux of wild type flux
```

Random Minimal Network Analysis

The `randomsparse` command can be used to look at gene essentiality in the metabolic network. To use this function the model must contain gene associations for the model reactions. This function works by systematically deleting genes from the network, then evaluating if the associated reaction would still be available after the gene deletion, and finally testing the new network to see if the objective function flux is still above the threshold for viability. If the flux falls too low then the gene is marked as essential and kept in the network. If the flux stays above the threshold then the gene will be marked as non-essential and removed. The program will randomly do this for all genes until the only ones left are marked as essential. This can be done using the `--type=genes` option with the `randomsparse` command:

```
(psamm-env) $ psamm-model randomsparse --type=genes 90%
```

This will produce an output of the gene IDs with a 1 if the gene was kept in the simulation and a 0 if the gene was deleted. Following the list of genes will be a summary of how many genes were kept out of the total as well as list of the reaction IDs that made up the minimal network for that simulation. An example output can be seen as follows:

```
INFO: Essential genes: 58/137
INFO: Deleted genes: 79/137
b0008       0
b0114       1
b0115       1
b0116       1
b0118       0
b0351       0
b0356       0
b0451       0
b0474       0
b0485       0
...
```

The random minimal network analysis can also be used to generate a random subset of reactions from the model that will still allow the model to maintain an objective function flux above a user-defined threshold. This function works on the same principle as the gene deletions but instead of removing individual genes, reactions will be removed. To run random minimal network analysis on the model use the randomsparse command with the `--type=reactions` option. The last parameter for the command is a percentage of the maximum objective flux that will be used as the threshold for the simulation.

```
(psamm-env) $ psamm-model randomsparse --type=reactions 95%
...
FRUKIN      1
...
MANNI1PDEH  0
MANNI1PPHOS 1
MANNIDEH    1
MANNIPTS    1
...
```

The output will be a list of reaction IDs with either a 1 indicating that the reaction was essential or a zero indicating it was removed.

Due to the random order of deletions during this simulation it may be helpful to run this command numerous times in order to gain a statistically significant number of datapoints from which a minimal essential network of reactions can be established.

In this case the program deleted the *MANN1PDEH* reaction blocking the mannitol 1-phosphate to fructose 6-phosphate conversion. In this case the reactions in the other side of the mannitol utilization pathway should all be essential.

You can also use the `randomsparse` command to randomly sample the exchange reactions and generate putative minimal exchange reaction sets. This can be done by using the `--type=exchange` option with the `randomsparse` command:

```
(psamm-env) $ psamm-model randomsparse --type=exchange 90%
```

It can be seen that when this is run on this small network the mannitol exchange as well as some other small molecules are identified as being essential to the network:

```
EX_ac_e     0
EX_acald_e  0
EX_akg_e    0
EX_co2_e    1
EX_etoh_e   0
EX_for_e    0
EX_fru_e    0
EX_fum_e    0
EX_glc_e    0
EX_gln_L_e  0
EX_glu_L_e  0
EX_h2o_e    1
EX_h_e      1
EX_lac_D_e  0
EX_mal_L_e  0
EX_manni_e  1
EX_nh4_e    1
EX_o2_e     1
EX_pi_e     1
EX_pyr_e    0
EX_succ_e   0
```

2.5 Reactant/Product Pair Prediction and Visualization Using FindPrimaryPairs

This tutorial will go over how to use the `primarypairs` and *PSAMM-Vis* functions in *PSAMM*. These functions can be used to predict reactant/product pairs in metabolic models and to use these predictions to generate visualizations of metabolic networks.

2.5.1 Materials

For information on how to install *PSAMM* and the associated requirements, as well how to download the materials required for this tutorial, you can reference the Installation and Materials section of the tutorial.

In addition to the basic installation of *PSAMM*, the visualization function uses the *Graphviz* program to generate images from the text-based graph format produced by the `vis` command. *Graphviz* version > 0.8.4 must be installed along with the *Graphviz* python bindings.

Note: Graphviz download: https://www.graphviz.org/download/

Graphviz python bindings: https://pypi.org/project/graphviz/ or (psamm-env) $ pip install graphviz

For this part of the tutorial, we will be using a modified version of the *E. coli* core metabolic model that has been used in the other sections of the tutorial. This model has been modified to add in a new pathway for the utilization of mannitol as a carbon source. To access this model and the other files needed you will need to go into the tutorial-part-4 folder located in the psamm-tutorial folder.

```
(psamm-env) $ cd <PATH>/tutorial-part-4/
```

Once in this folder, you should see a folder called E_coli_yaml which contains all of the required model files, and a directory called additional_files/ that contains additional input files that will be used to run the commands in this tutorial.

To run the following tutorials, go into the E_coli_yaml/ directory:

```
(psamm-env) $ cd E_coli_yaml/
```

2.5.2 Reactant/Product Pair Prediction using *PSAMM*

Metabolism can be broken down into individual metabolic reactions, which transfer elements between different metabolites. Take the following reaction as an example:

```
Acetate + ATP <=> Acetyl-Phosphate + ADP
```

This reaction is catalyzed by the enzyme Acetate Kinase, which can convert acetate to acetyl-phosphate through the addition of a phosphate group from ATP. A basic understanding of phosphorylation and the biological role of ATP makes it possible to manually predict that the primary element transfers for non hydrogen elements are as follows:

Reactant/Product Pair	Element Transfer
Acetate -> Acetyl-Phosphate	carbon backbone
ATP -> ADP	carbon backbone and phosphates
ATP -> Acetyl-Phosphate	phosphate group
Acetate -> ADP	None

While manually inferring this for one or two simple reactions is possible, genome scale models often contain hundreds or thousands of reactions, making manual reactant/product pair prediction impractical. In addition to this, reaction mechanisms are often not known, nor are patterns of element transfer within reactions available for most metabolic reactions.

To address this problem the *FindPrimaryPairs* algorithm [Steffensen17] was developed and implemented within the *PSAMM* function `primarypairs`.

The *FindPrimaryPairs* is an iterative algorithm which is used to predict element transferring reactant/product pairs in genome scale models. *FindPrimaryPairs* relies on two sources of information, which are generally available in genome scale models: reaction stoichiometry and metabolite formulas. From this information, *FindPrimaryPairs* can make a global prediction of element transferring reactant/product pairs without any additional information about reaction mechanisms.

Basic Use of the `primarypairs` Command

The `primarypairs` command in PSAMM can be used to perform an element transferring pair prediction using the *FindPrimaryPairs* algorithm. The basic command can be run as the following:

```
(psamm-env) $ psamm-model primarypairs --exclude @../additional_files/exclude.tsv
```

This function often requires a file to be provided through the `--exclude` option. This file is a single column list of reaction IDs of any reactions the user wants to remove from the model when doing the reactant/product pair prediction. the file path should be included in the command with a '@' preceding it. Typically, this file should contain any artificial reactions that might be in the model such as Biomass objective reactions, macromolecule synthesis reactions, etc. While these reactions can be left in the model, the fractional stoichiometries and presence of artificial metabolites in the reaction can cause the algorithm to take a much longer time to find a solution. In this example of the *E. coli* core model the only reaction like this is the biomass reaction `Biomass_Ecoli_core_w_GAM`, which this is the only reaction listed in the *exclude.tsv* file.

Note: The *FindPrimaryPairs* algorithm relies on metabolite formulas to make its reactant/product pair predictions. If any reaction contains a metabolite that does not have a formula then it will be ignored.

The output of the above command will look like the following:

```
INFO: Model: Ecoli_core_model
INFO: Model version: 3ac8db4
INFO: Using default element weights for fpp: C=1, H=0, *=0.82
INFO: Iteration 1: 79 reactions...
INFO: Iteration 2: 79 reactions...
INFO: Iteration 3: 8 reactions...
GLNS    nh4_c[c]        h_c[c]  H
FBA     fdp_c[c]        g3p_c[c]        C3H5O6P
ME2     mal_L_c[c]      nadph_c[c]      H
MANNI1PDEH      manni1p[c]      nadh_c[c]       H
PTAr    accoa_c[c]      coa_c[c]        C21H32N7O16P3S
....
```

Basic information about the model name and version is provided in the first few lines. In the next line, the element weights used by the *FindPrimaryPairs* algorithm are listed. Then, as the algorithm goes through multiple iterations, it will print out the iteration number and how many reactions it is still figuring out the pairing for. A four column table is then printed out that contains the following columns from left to right: Reaction ID, reactant ID, product ID, and elements transferred.

From this output, the Acetate Kinase reaction from the above example can be compared to the manual prediction of the element transfer. The reaction ID for this reaction is ACKr:

```
ACKr    atp_c[c]        adp_c[c]        C10H12N5O10P2
ACKr    atp_c[c]        actp_c[c]       O3P
ACKr    ac_c[c] actp_c[c]       C2H3O2
```

From this result it can be seen that the prediction contains the same three element transferring pairs as the above manual prediction; ATP -> ADP, ATP -> Acetyl-Phosphate, Acetate to Acetyl-Phosphate.

This basic usage of the `primarypairs` command allows for quick and accurate prediction of element transferring pairs in any of the reactions in a genome scale model. Additionally, the function also has a few other options that can be used to refine and adjust how the pair prediction work.

Modifying Element Weights

The metabolite pair prediction relies on a parameter called element weight to inform the algorithm about what chemical elements should be considered more or less important when determining metabolite similarity. An example of how this might be used can be seen in the default element weights that are reported when running `primarypairs`.

```
INFO: Using default element weights for fpp: C=1, H=0, *=0.82
```

These element weights are the default weights used when running `primarypairs` with the *FindPrimaryPairs* algorithm. In this case, a weight of 1 is given to carbon. Because carbon forms the structural backbone of many metabolites this element is given the most weight. In contrast, hydrogen is not usually a major structural element within metabolites. This leads to a weight of 0 being given to hydrogen, meaning that it is not considered when comparing formulas between two metabolites. By default, all other elements are given an intermediate weight of 0.82.

These default element weights can be adjusted using the `--weights` command line argument. For example, to adjust the weight of the element nitrogen while keeping the other elements the same as the default settings, you could use the following command:

```
(psamm-env) $ psamm-model primarypairs --weights "N=0.2,C=1,H=0,*=0.82" --exclude @../
↪additional_files/exclude.tsv
```

In the case of a small model like the *E. coli* core model, the results of *primarypairs* will likely not change unless the weights are drastically altered. However, changes could be seen in larger models, especially if the models include many reactions related to non-carbon metabolism such as sulfur or nitrogen metabolism.

Report Element

By default, the *primarypairs* result is not filtered to show transfers of any specific element. In certain situations it might be desirable to only get a subset of these results based on if the reactant/product pair transfers a target element. To do this, the option `--report-element` can be used. In many cases, it might be desirable to only report carbon transferring reactant/product pairs, to do this run the following on the *E. coli* model.

```
(psamm-env) $ psamm-model primarypairs --report-element C --exclude @../additional_
↪files/exclude.tsv
```

If the predicted pairs are looked at for one of the mannitol pathway reactions, MANNIDEH, the following can be seen:

```
MANNIDEH        manni[c]        fru_c[c]        C6H12O6
MANNIDEH        nad_c[c]        nadh_c[c]       C21H26N7O14P2
```

If this result is compared to the results without the `--report-element C` option, it can be seen that when there are additional transfers in this reaction, but they only involve hydrogen.

```
MANNIDEH        manni[c]        nadh_c[c]       H
MANNIDEH        manni[c]        h_c[c]  H
MANNIDEH        manni[c]        fru_c[c]        C6H12O6
MANNIDEH        nad_c[c]        nadh_c[c]       C21H26N7O14P2
```

Pair Prediction Methods

Two reactant/product pair prediction algorithms are implemented in the *PSAMM* `primarypairs` command. The default algorithm is the *FindPrimaryPairs* algorithm. The other algorithm that is implemented is the *Mapmaker* algorithm. These algorithms can be chosen through the `--method` argument.

```
$ psammm-model primarypairs --method fpp
or
$ psamm-model primarypairs --method mapmaker
```

2.5.3 Visualizing Models using PSAMM-Vis

PSAMM-Vis, as implemented in the `vis` command in *PSAMM*, can be used to convert text-based YAML models to graph-based representations of the metabolism. The graph-based representation contains two sets of nodes: one set representing the metabolites in the model and one set representing reactions. These nodes are connected through edges that are determined based on element transfer patterns predicted through using the *FindPrimaryPairs* algorithm. The `vis` command provides multiple options to customize the graph representation of the metabolism, including changing network perspectives, customizing node labels, changing node colors, etc.

Basic Network Visualization

The basic `vis` command can be run through the following command:

```
(psamm-env) $ psamm-model vis
```

By default, `vis` relies on the *FindPrimaryPairs* algorithm to predict elements transferred in metabolic network. In the `vis` function, the biomass reaction defined in *model.yaml* file will be excluded from the *FindPrimaryPairs* calculation automatically, but will still be shown on the final network image. For more information of excluded reactions, see *Basic Use of the primarypairs Command*.

In this version of the *E. coli* core model, the biomass reaction is defined in the *model.yaml* file, so it will be excluded automatically from the pair prediction calculation when running `vis`.

By default, the command above will export three files: ‘reaction.dot’, ‘reactions.nodes.tsv’ and ‘reactions.edges.tsv’. The first file, ‘reactions.dot’, contains a text-based representation of the network graph in the ‘dot’ language. This graph language is used primarily by the *Graphviz* program to generate network images. This graph format contains information of nodes and edges in the graph along with details related to the size, colors, and shapes that will be used in the final network image. The ‘reactions.nodes.tsv’ and ‘reactions.edges.tsv’ files are tab separated tables that contain the same information as the *dot* based graph, but in a more generic table based format that can be used with other graph analysis and visualization software like *Cytoscape*.

File 'reactions.nodes.tsv' contains all the information that define the graph nodes, including both reaction and compound nodes. It looks like the following:

```
id   compartment      fillcolor        shape    style    type     label
13dpg_c[c]    c       #ffd8bf ellipse filled  cpd      13dpg_c[c]
2pg_c[c]      c       #ffd8bf ellipse filled  cpd      2pg_c[c]
3pg_c[c]      c       #ffd8bf ellipse filled  cpd      3pg_c[c]
6pgc_c[c]     c       #ffd8bf ellipse filled  cpd      6pgc_c[c]
....
```

The file 'reactions.edges.tsv' contains information related to the structure of the graph. Each line in this table represents one edge in the graph and contains the source, destination and direction (forward, back, or both) of the edge. It looks like the following:

```
 source       target  dir      penwidth         style
 2pg_c[c]     PGM_1   both     1        solid
 PGM_1        3pg_c[c]         both     1        solid
 2pg_c[c]     ENO_1   both     1        solid
...
```

Generate Images from Text-based Graphs

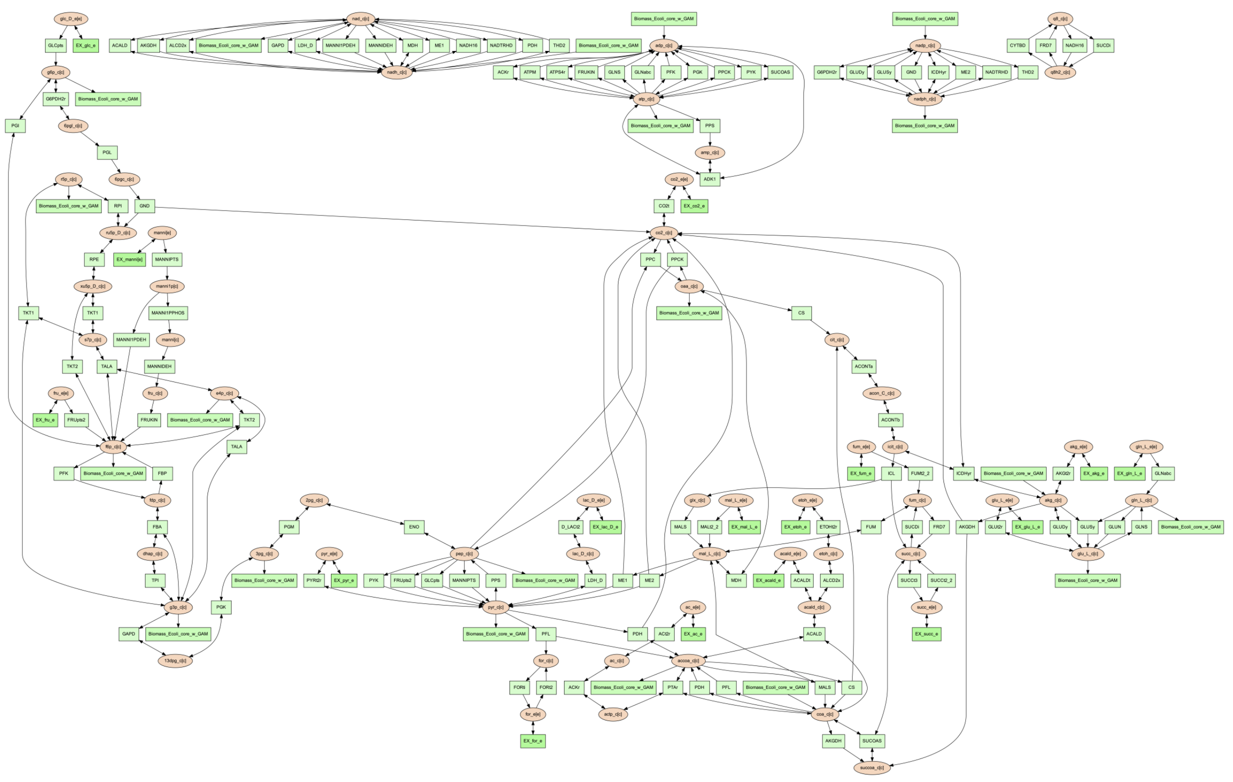
Images can be generated from the 'reactions.dot' file by using the *Graphviz* program. *Graphviz* support multiple image formats (PDF, PNG, JPEG, etc). For example, image file can be generated as a *PDF* file by using the following *Graphviz* program command:

```
(psamm-env) $ dot -O -Tpdf reactions.dot
```

An image can also be done in one step by running *vis* command by adding an `--image` option followed by image format (pdf, svg, eps, etc.) to the command:

```
(psamm-env) $ psamm-model vis --image pdf
```

The commands above both generate an image file named 'reactions.dot.pdf'. This pdf file is a graphical representation of what is in the 'reactions.dot'. This graph will look like:



In this default version of the network image, there are two sets of nodes: oval orange nodes, which represent metabolites, and rectangular green nodes, which represent reactions. These nodes are connected by edges which indicate reaction directionality.

The rest of the tutorial will detail how to modify the default version of network image to show different aspects of the metabolism and customize the node properties. For these sections, the mannitol utilization pathway from the previous tutorial sections will be used as an example.

Represent Different Element Flows

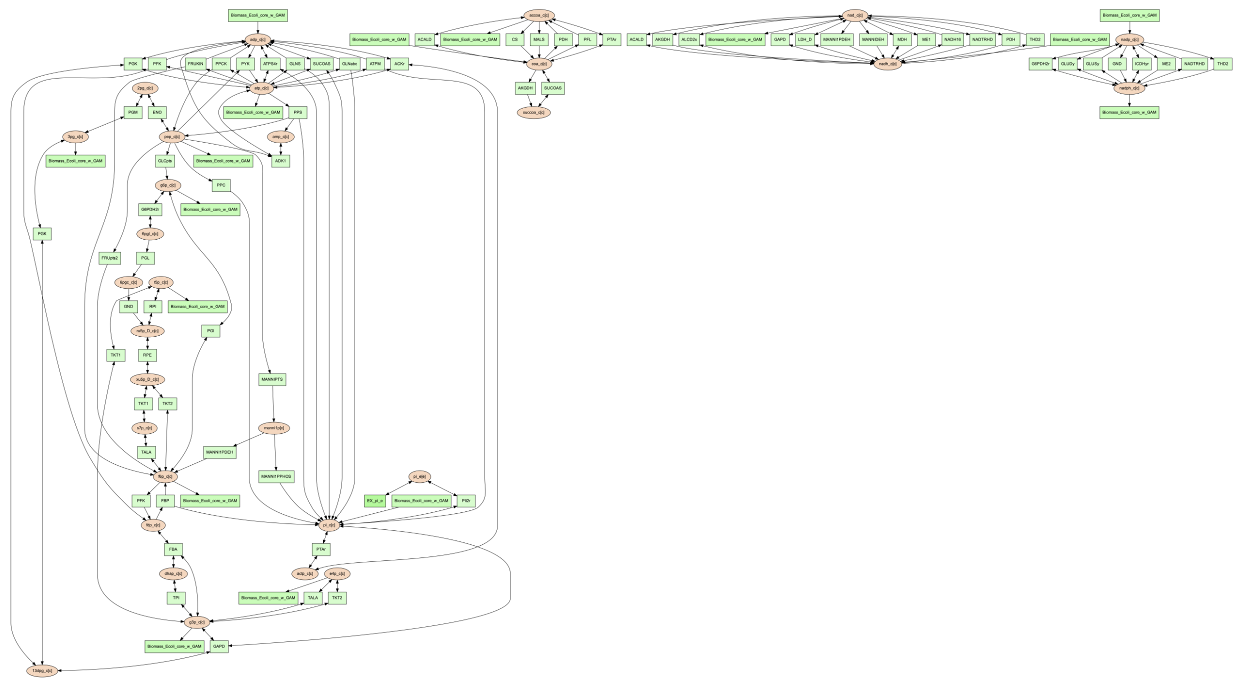
By default, the `vis` command generates a graph that shows the carbon (C) transfer in the metabolic network. In the `primarypairs` tutorial section above, the element transfers in the *ACKr* reaction were examined to see how the *FindPrimaryPairs* algorithm would predict element transfer patterns. The `vis` command can use these element transfer predictions to filter edges in the network image, only edges that transfer specific element will be shown. In the case of the *ACKr* reaction, if element carbon is required to be shown, then only edges of 'Acetate -> Acetyl-Phosphate' and 'ATP -> ADP' would present in the final graph. The 'ATP -> Acetyl-Phosphate' edge will disappear, because ATP doesn't transfer carbon to Acetyl-Phosphate.

Reactant/Product Pair	Element Transfer
Acetate -> Acetyl-Phosphate	carbon backbone
ATP -> ADP	carbon backbone, phosphates
ATP -> Acetyl-Phosphate	phosphate group
Acetate -> ADP	None

This type of element filtering can provide different views of the metabolic network by showing how metabolic pathways transfer different elements through those reactions. The mannitol utilization pathway is a multiple-step pathway that converts extracellular mannitol to fructose 6-phosphate. This pathway also involves multiple phosphorylation and dephosphorylation steps. The `--element` argument can be added to the the `vis` command to filter this pathway and show the transfer patterns of the phosphorus in the pathway:

```
(psamm-model) $ psamm-model vis --element P --image png
```

The resulting 'reactions.dot.png' file will contain the phosphorus transfer network of the *E. coli* core model.

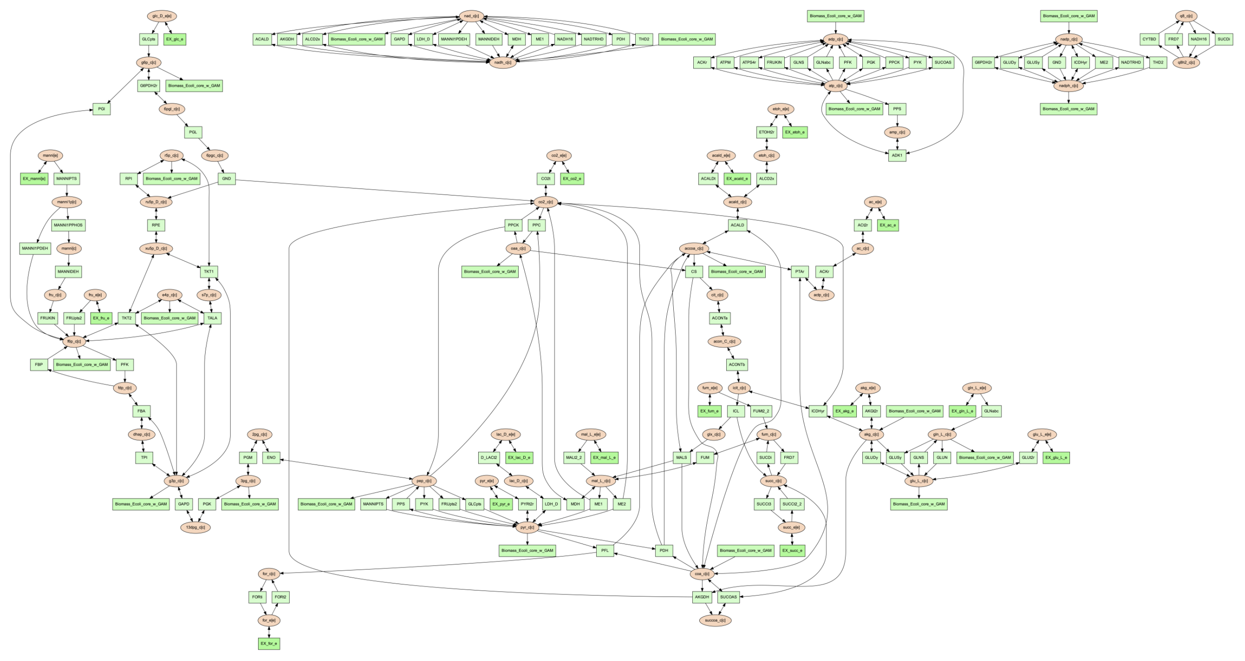


Condense Reaction Nodes and Edges

By default, the `vis` command assigns only one reaction to each reaction node. Additionally, it allows users to condense multiple reaction nodes into one node through the `--combine` option, in order to reduce the number of nodes and edges, and make the image clearer. Combine level 0 is the default, which does not collapse any nodes. Level 1 is used to condense nodes that represent the same reaction and have a common reactant or product connected. Level 2 is used to condense nodes that represent different reactions but connected to the same reactant/product pair with the same direction (This is often seen on reactant/product pairs like ATP/ADP and NAD/NADH).

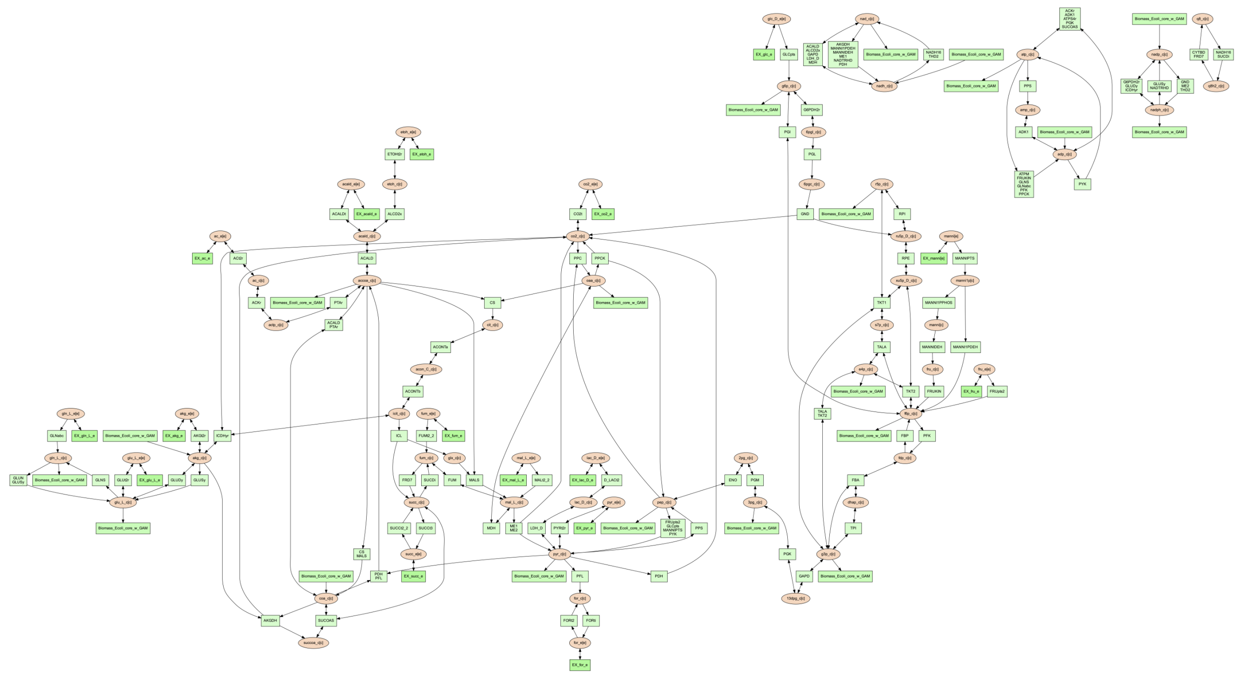
```
(psamm-env) $ psamm-model vis --combine 1 --image png
```

Then the image will look like the figure below:



The combine 2 option will update the image to look like the following: .. code-block:: shell

(psamm-env) $ psamm-model vis –combine 2 –image png

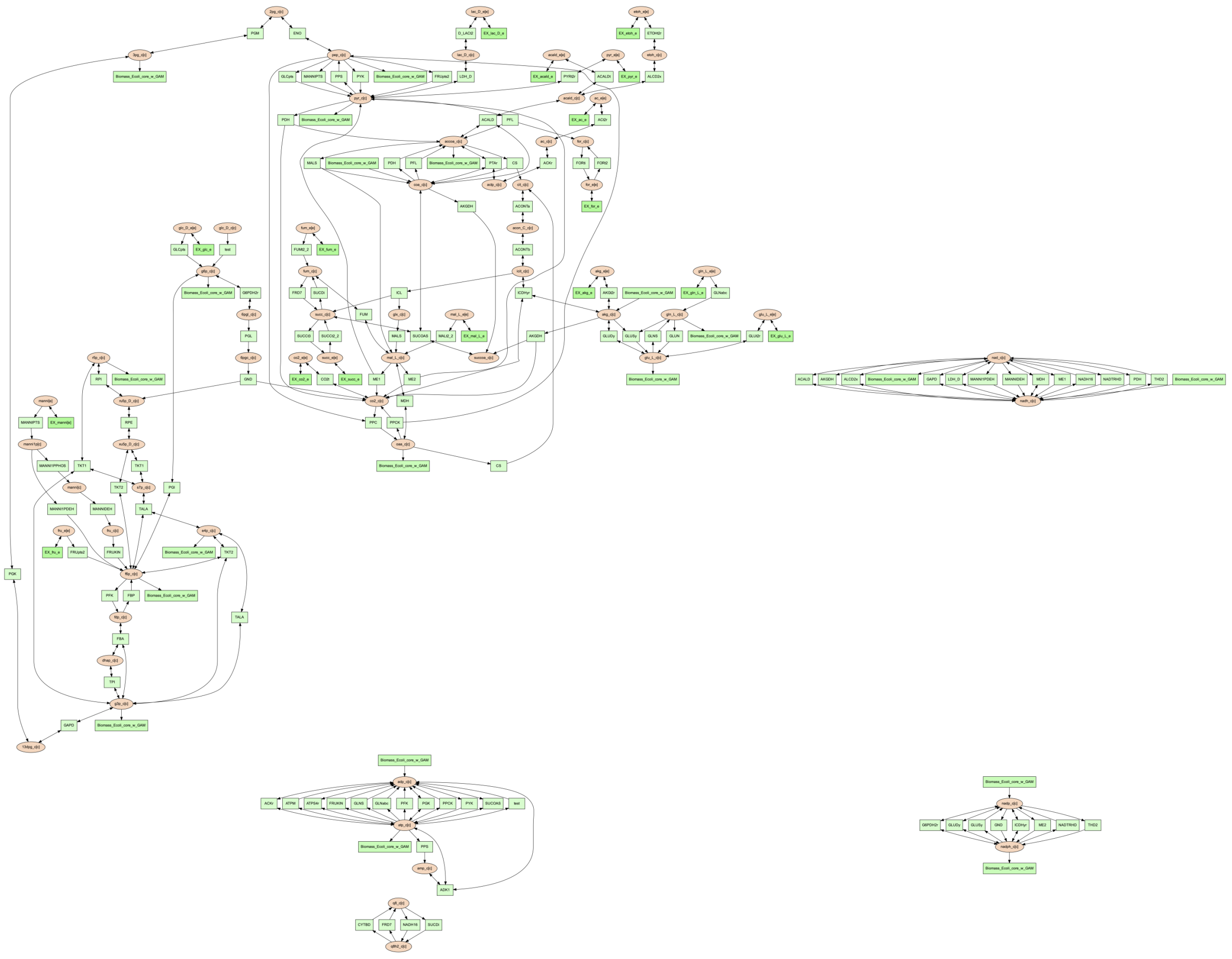


Rearrange graph components in the image

In some cases, the network images contain many connected components, while these components aren't connected with each other. This may cause the image too wide and difficult to read. To create a better view, `--array` option could be used. It can decompress graphs into their connected components, then arrange these components with specific array setting. `--array` is followed by an integer, for example, `--array 2` indicates placing two connected components per row. For graphs that contains many small connected components, `--array 4` could be a better option because you can get most of the important larger components in the top few rows of the image, and all the smaller components will just be spread out below them. Example of applying this option see below:

```
(psamm-env) $ psamm-model vis --image png --array 2
```

Then the exported image "reactions.dot.png" will look like the figure below:



Moreover, if *vis* command contains `--array` but doesn't contain `--image`, it will still exported the DOT file. However, in this case, when converting DOT file to a network image, to make `--array` effective, another *Graphviz* program command (see below) is required instead of `dot` command we showed before:

```
(psamm-env) $ neato -O -Tpdf -n2 reactions.dot
```

Show Cellular Compartments

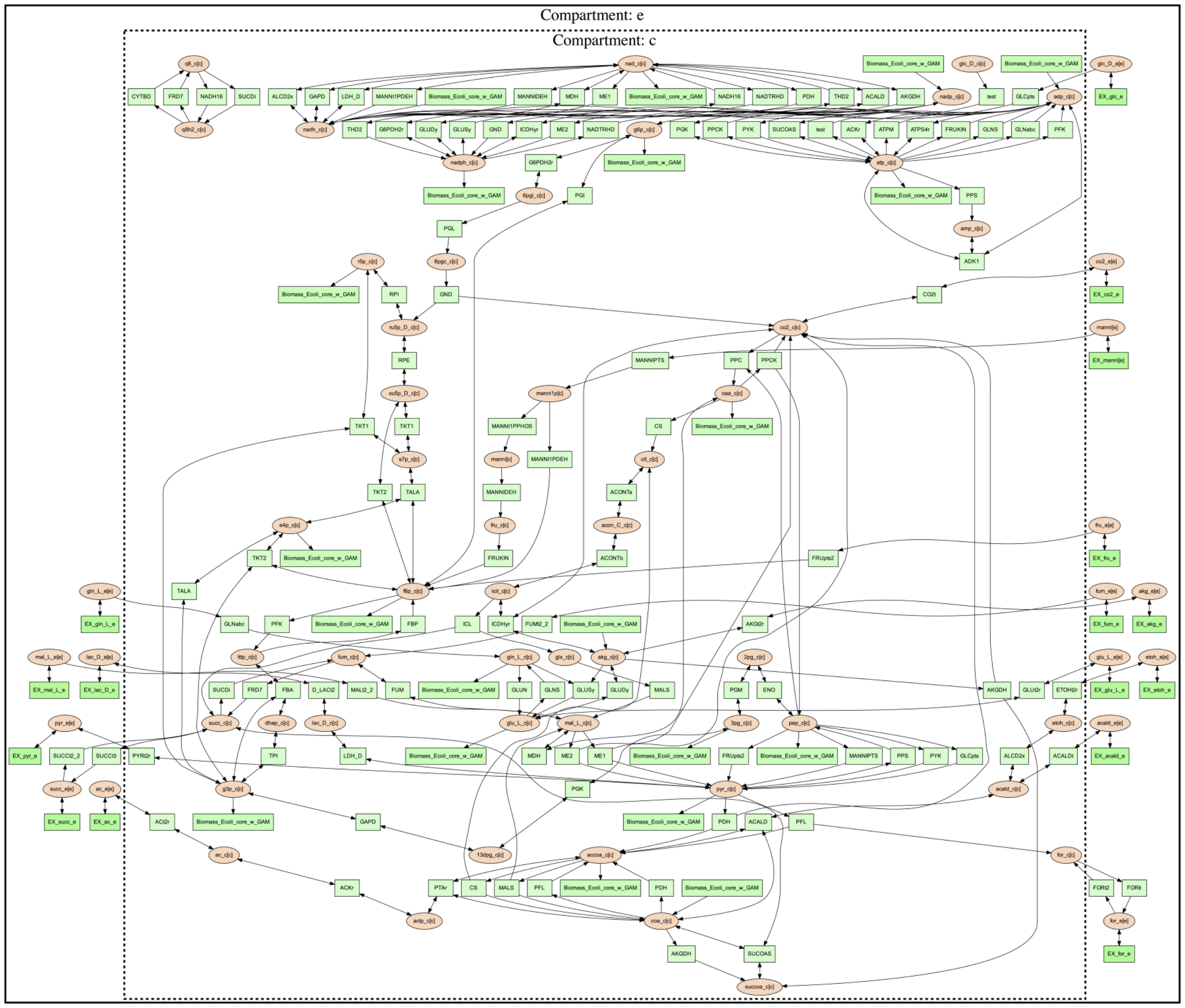
GEMs often contain some representations of cellular compartments. At the most basic level, this includes an intracellular and extracellular compartment, but in complex models, additional compartments such as the periplasm in bacteria or mitochondria in eukaryotes are often included. *PSAMM-Vis* can show these compartments in the final image through the use of the `--compartment` argument. If the compartment information is not defined in the model.yaml file, then, the command will attempt to automatically detect the organization of the compartments by examining the reaction equations in the model. However, this process cannot always accurately predict the compartment organization. To overcome this problem, it is suggested to define the compartment organization in the model.yaml file like in the following example:

```
name: Ecoli_core_model
biomass: Biomass_Ecoli_core_w_GAM
default_flux_limit: 1000
extracellular: e
compartments:
- id: c
  adjacent_to: e
  name: Cytoplasm
- id: e
  adjacent_to: e
  name: Extracellular
....
```

Once this information is added to the model.yaml file the following command can be used to generate an image that shows the compartment information of the model:

```
(psamm-env) $ psamm-model vis --compartment --image png
```

The resulting file 'reactions.dot.png' will look like the following:



In this image there are two compartments that are labeled with ‘Compartment: e’ and ‘Compartment: c’. The *E. coli* core model is relatively small, meaning that compartment organization is simple, but `vis` command can handle more complex models as well. For example, the following image was made using a small example model to show a more complex compartments organization. To do this running the following command:

```
(psamm-env) $ psamm-model --model ../additional_files/toy_model_cpt/toy_model.yaml␣
↪vis --image png --compartment
```

The resulting network image “reactions.dot.png” looks like:

tutorial/06-cptToy.dot.png

Visualize Reactions and Pathways of Interest

In some situations, it might be better to visualize a subset of a larger model so that smaller subsystems can be examined in more detail. This can be done through the `--subset` option. This option takes an input of a single column file, where each line contains either a reaction ID or a metabolite ID. The whole file can contain only reaction IDs or metabolite IDs and cannot be a mix of both.

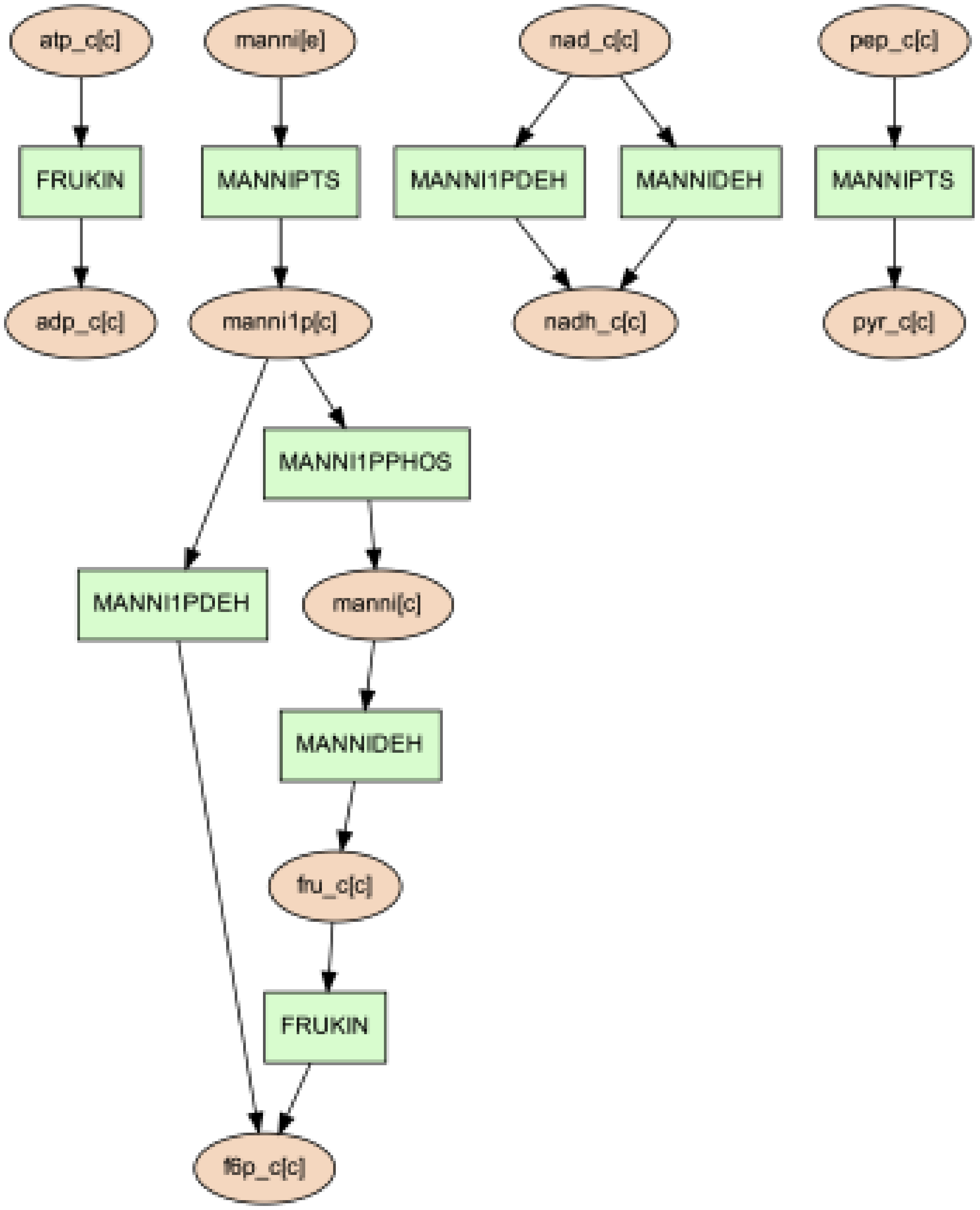
To show the usage of this option, a subset of reactions involved in mannitol utilization pathway was visualized through the following command:

```
(psamm-env) $ psamm-model vis --subset ../additional_files/subset_mannitol_pathway.
↪tsv --image png
```

The input file subset_mannitol_pathway looks like the following:

```
MANNIPTS
MANNI1PDEH
MADNNIDEH
MANNII1PPHOS
FRUKIN
```

This resulting image "reactions.dot.png" looks like the following:



This image only contains reactions listed in the subset file and any associated exchange reactions.

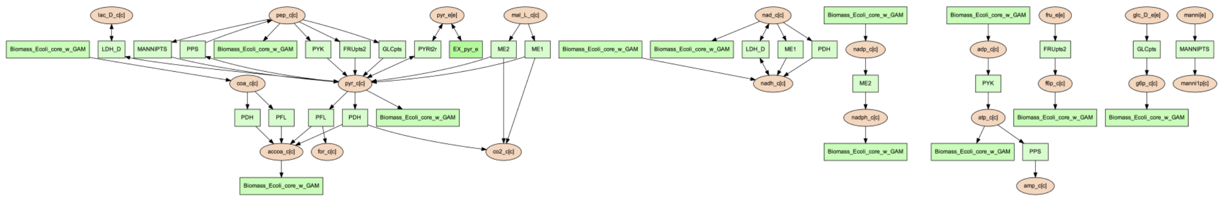
The other usage for using the subset argument is to provide a list of metabolite IDs (with compartment). This option will generate an image containing all of the reactions that contain any of given metabolites in their equation. For example, the following subset file could be used to generate a network image of all reactions that contain pyruvate.

```
pyr_c[c]
pyr_e[e]
```

To use this subset to generate the pyruvate related subnetwork use the following command:

```
(psamm-env) $ psamm-model vis --subset ../additional_files/subset_pyruvate_list.tsv --
↪image png
```

This will generate an image like the following that only shows the reactions that contain pyruvate:



Highlight Reactions and Metabolites in the Network

The `--subset` option can be used to show only a specific part of the network. When this is done, the context of those reactions is often lost and it can be hard to tell where that pathway fits within the larger metabolism. A different way to highlight a set of reactions without using the `--subset` option is to change the color of a set of nodes through the `--color` option.

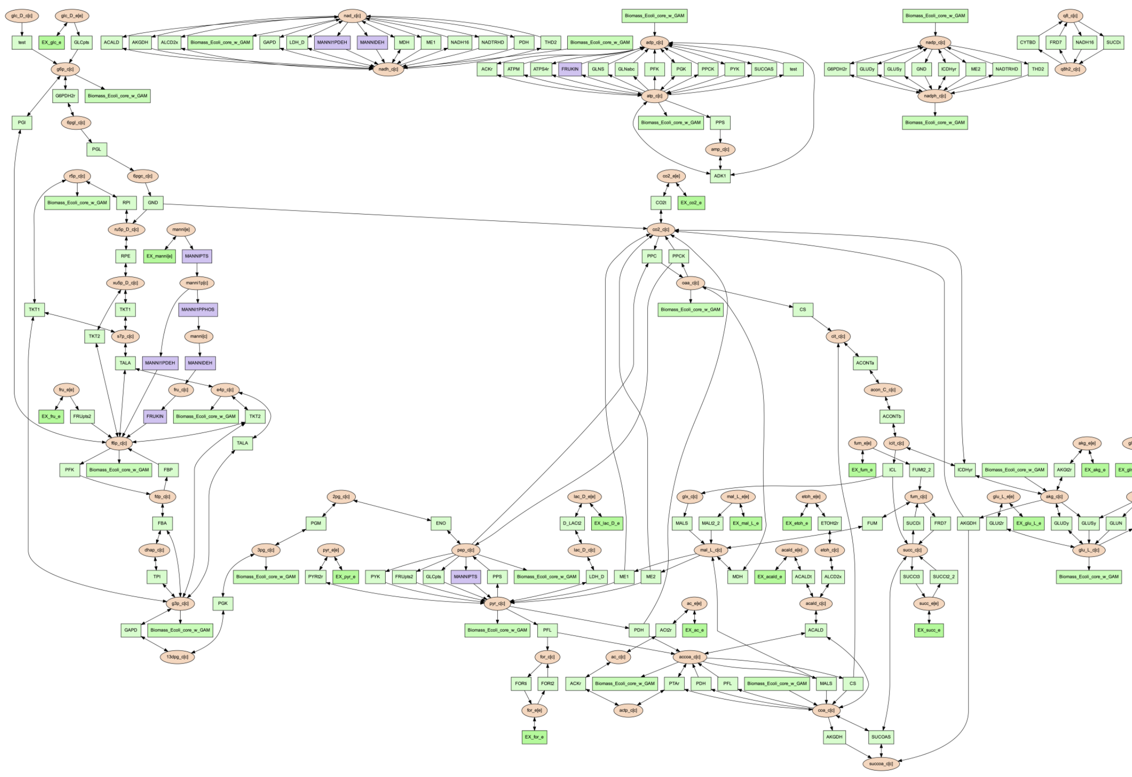
This option can be used to change the color of the reaction or metabolite nodes on the final network image, making it easy to highlight certain pathways while still maintaining the larger metabolic context. This `--color` option will take a two-column file that contains reaction or metabolite IDs in the first column and hex color codes in the second column. A color file that can be used to color all of the mannitol utilization pathway reactions purple would look like the following:

```
MANNIPTS #d6c4f2
MANNI1PDEH #d6c4f2
MANNIDEH #d6c4f2
MANNI1PPHOS #d6c4f2
FRUKIN #d6c4f2
```

To use this file to generate an image of the larger network with the mannitol utilization pathway highlighted, use the following command:

```
(psamm-env) $ psamm-model vis --color ../additional_files/color_mannitol_pathway.tsv -
↪-image png
```

The resulting image file should look like the following:



Coloring of specific nodes like this can make it easy to locate or highlight specific pathways, especially in larger models.

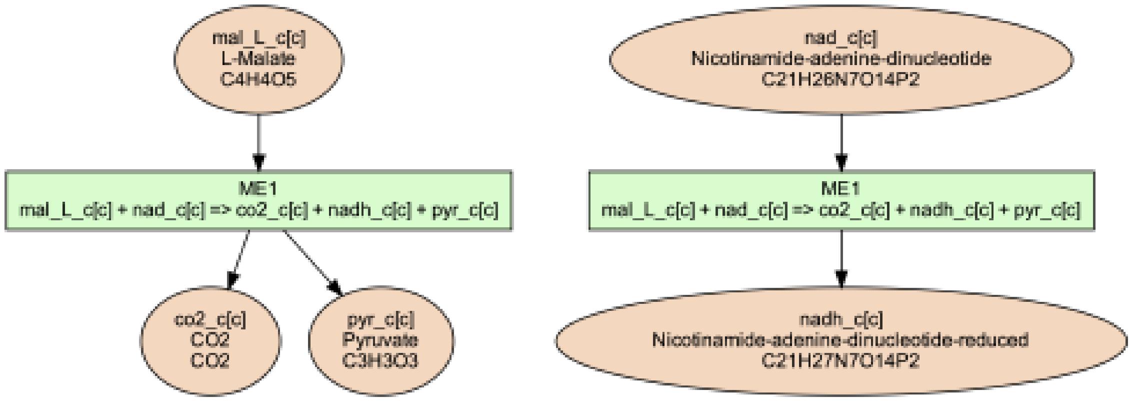
Note: Reaction nodes that represent multiple reactions won't be recolored even if it contains one or more reactions that are in input table for recolor.

Modify Node Labels in Network Images

By default, only the reaction IDs or metabolite IDs are shown on the nodes in final network images. These labels can be modified to include any additional information defined in the compounds or reactions YAML file through the use of the `--cpd-detail` and `--rxn-detail` options. These options are followed by a space separated list of metabolite or reaction property names, such as id, name, equation, and formula. The required properties will present on the node labels in network image. For example, for reaction ME1 (NAD-dependent malic enzyme), to show metabolite ID, name and formula, as well as reaction ID, and equation, running the following command:

```
(psamm-env) $ psamm-model vis --subset ../additional_files/detail_ME1.tsv --cpd-
↪detail id name formula --rxn-detail id equation --image png
```

The image generated looks like this:


mal_L_c[c]
L-Malate
C4H4O5
nad_c[c]
Nicotinamide-adenine-dinucleotide
C21H26N7O14P2
ME1
mal_L_c[c] + nad_c[c] => co2_c[c] + nadh_c[c] + pyr_c[c]
ME1
mal_L_c[c] + nad_c[c] => co2_c[c] + nadh_c[c] + pyr_c[c]
co2_c[c]
CO2
CO2
pyr_c[c]
Pyruvate
C3H3O3
nadh_c[c]
Nicotinamide-adenine-dinucleotide-reduced
C21H27N7O14P2

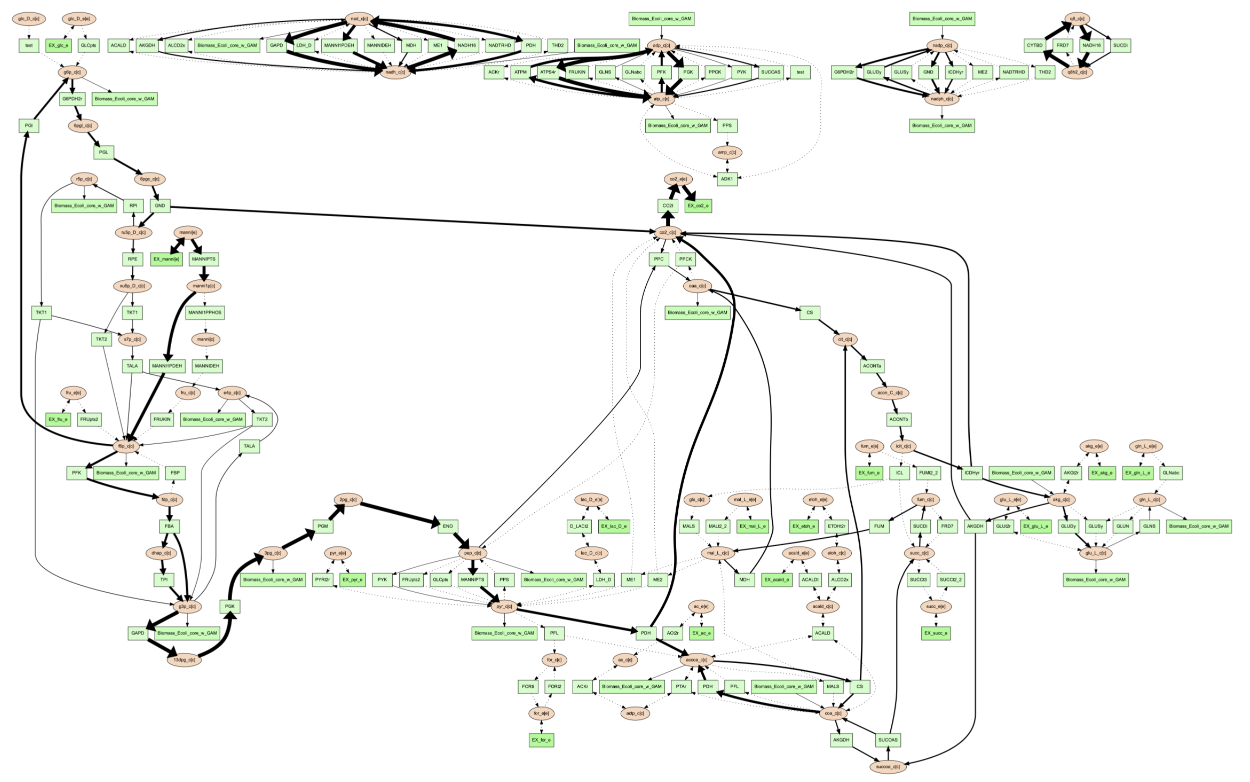
Note: For these two options, if a required detail is not included in the model, that property will be skipped and not shown on those nodes.

Visualize FBA or FVA

Performing various simulations of growth is made possible through methods such as FBA and FVA. Using the `--fba` or `--fva` option, the flow of metabolites calculated by these methods can be visualized. When visualizing FBA, a tsv file containing the reaction name and flux value is required. For example, the following command can be used:

```
(psamm-env) $ psamm-model vis --fba ../additional_files/fba.tsv --image png
```

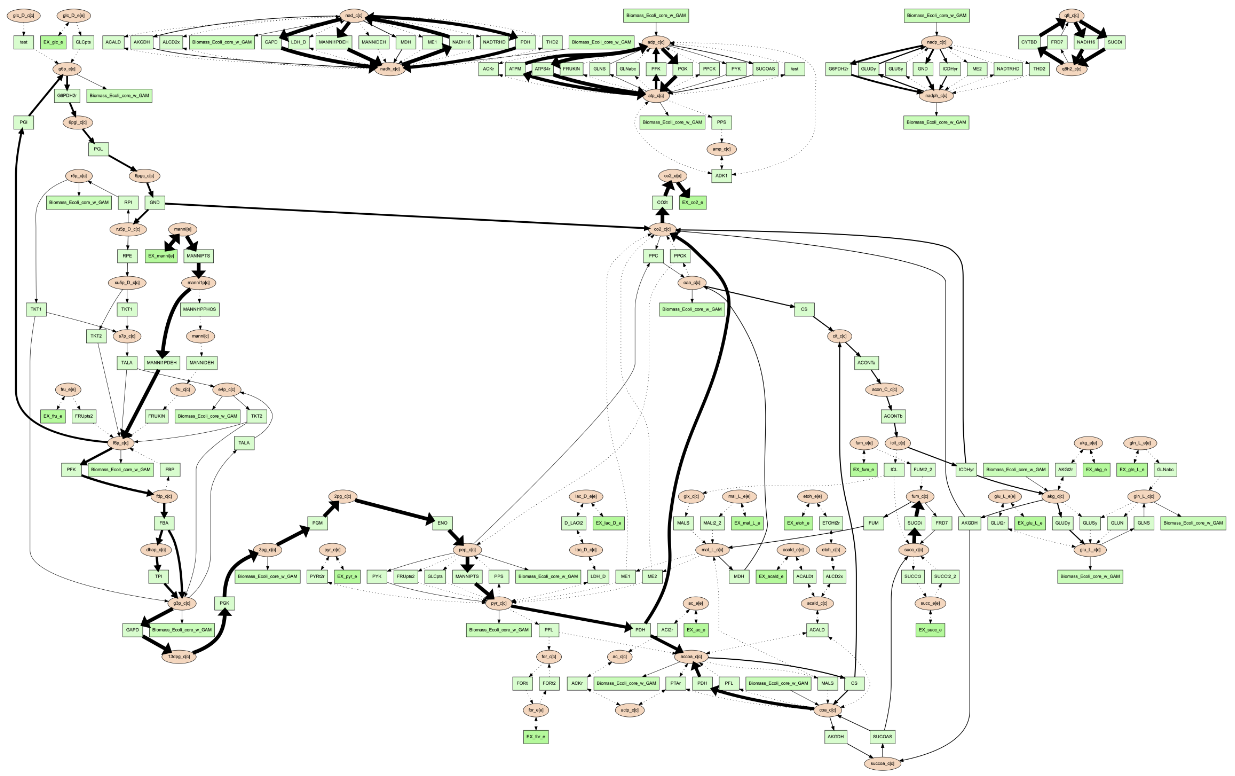
The image generated looks like this:



If the FVA option is given, the file should contain the reaction name, and a lower and upper bound flux value that would still allow the model to sustain the same objective function flux. To visualize the FVA results, you can use the command:

```
(psamm-env) $ psamm-model vis --fva ../additional_files/fva.tsv --image png
```

The image generated looks like this:



Reactions with a flux of zero is represented as a dotted edge and non-zero fluxes as solid. Meanwhile, the thickness of the edges is proportional to the flux through the reaction. Visualizing these fluxes may help highlight reactions that contribute the most to the objective.

Note: The `--fba` and `--fva` options cannot be used together.

Note: Fluxes less than the absolute value of 1e-5 will be considered as 0.

Other Visualization Options

Remove Specific Reactant Product Pairs

Large scale models may have some reactant/product pairs that occur many times in different reactions. These often involve currency metabolites like ATP, ADP, NAD and NADH. Due to the large number of times these pairs occur across the network, they may cause some parts of the graph to look messy. While making the condensed reaction nodes can help with this problem, there may be cases where it would be better to hide these edges in the final result. To do this the `--hide-edges` option can be used. This option takes a two-column file where each row contains two metabolite IDs separated by tab, edges between them will be hidden in final network image.

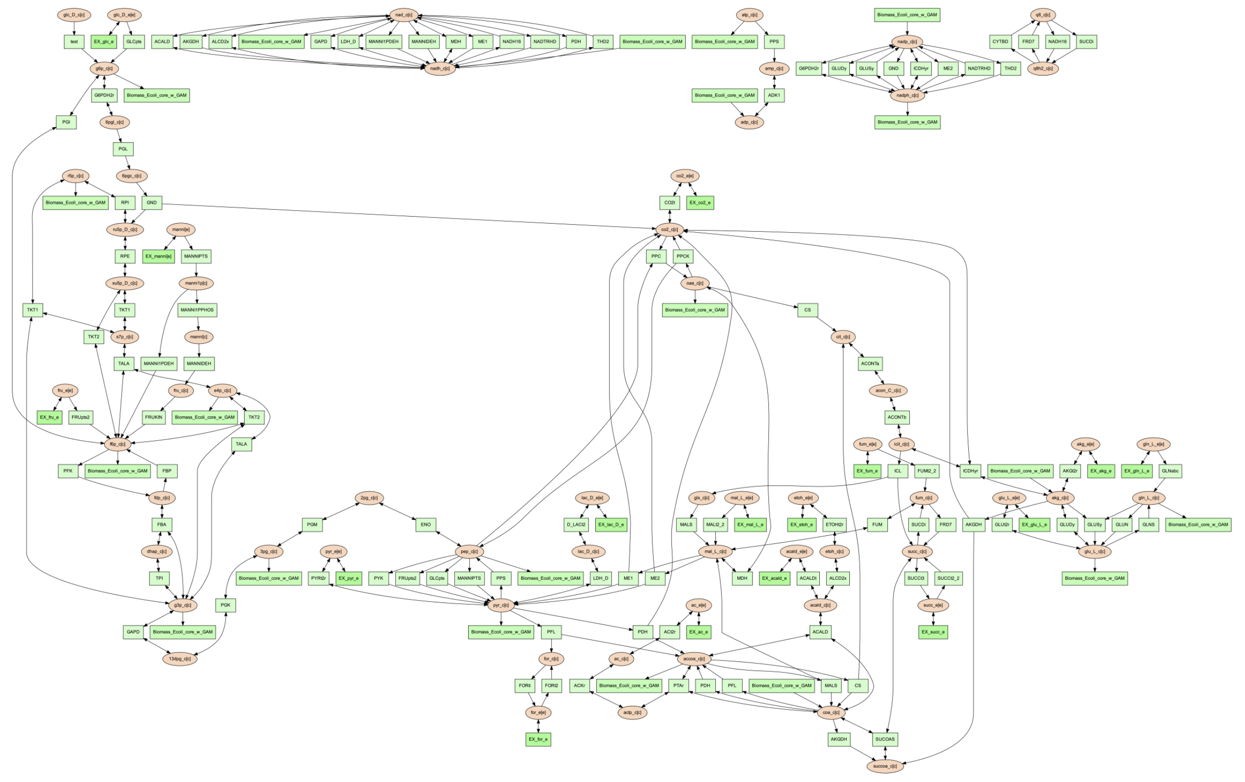
For example, to hide the edges between ATP and ADP in the *E. coli* core model, the input file would look like the following:

```
atp_c[c] adp_c[c]
```

Then the following command could be run to generate a network image that hides the edges between ATP and ADP:

```
(psamm-env) $ psamm-model vis --hide-edges ../additional_files/hide_edges_list.tsv --
↪image png
```

When comparing this image to previous visualizations we can see that many edges between ATP and ADP have been removed from the graph. While this might not make a huge difference on a small model like this, on larger models this can help during the process of generating cleaner final images.



Adjusting Image Size

The size of the final network image generated through the `vis` command can be adjusted through the `--image-size` option. This option takes the width and height (in inches) separated by a space. The following command is an example that generates an image of 8.5 inches x 11 inches:

```
(psamm-env) $ psamm-model vis --image-size 8.5 11 --image png
```

The resulting image looks like:

Specifying A File Name

The `vis` command allows users to specify the directory and name of outputs through the `--output` option. This option should be followed by a string and this string is the full path with the prefix of outputs (without the file extension). For example, the following command will export 4 files in your current working directory : “Ecolicore.dot”, “Ecolicore.dot.png”, “Ecolicore.nodes.tsv” and “Ecolicore.edges.tsv”:

```
(psamm-env) $ psamm-model vis --image png --output Ecolicore
```

If you run the following two commands, you will get 4 files in the new folder named “test-output”: “Ecolicore.dot”, “Ecolicore.dot.png”, “Ecolicore.nodes.tsv” and “Ecolicore.edges.tsv”:

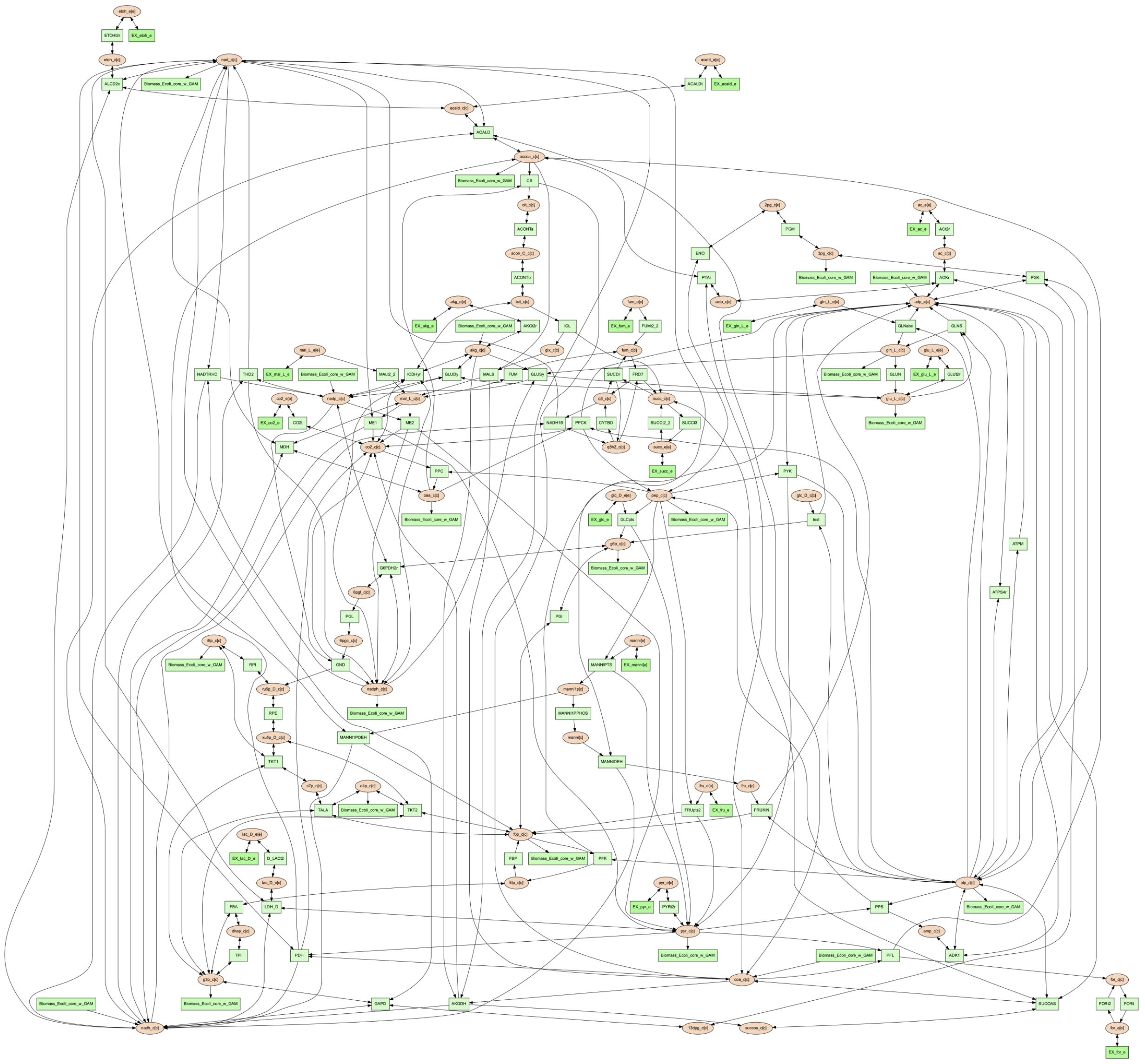
```
(psamm-env) $ mkdir test-output

(psamm-env) $ psamm-model vis --image png --output test-output/Ecolicore
```

Changing Pair Prediction Methods

By default, the *vis* function in *PSAMM* applies *FindPrimaryPairs* algorithm to predict reactant/product pairs. But it can also work without pair prediction (`no-fpp`). When‘‘no-fpp‘‘ is used, each reactant will be paired with all products in a reaction, without considering element transferred between reactant and product. There will tend to be many more connections in the network image if users use this option, especially for metabolites like ATP, H2O, and H+. To do this, running the following command:

```
(psamm-env) $ psamm-model vis --method no-fpp --image png
```



Note: The `--method no-fpp` and `--combine` options cannot be used together. The `--combine` option only works for *FindPrimaryPairs* method.

2.6 Automated Comparison of Models Using ModelMapping

This tutorial will go over how to use the `modelmapping` function in *PSAMM*. This function can be used to automatically compare different models based on the profiles of metabolites and reactions.

2.6.1 Materials

For information on how to install *PSAMM* and the associated requirements, as well how to download the materials required for this tutorial, you can reference the Installation and Materials section of the tutorial.

For this part of tutorial, we will be mapping a modified version of the Shewanella core metabolic model to a modified version of the *E. coli* core metabolic model that has been used in the other sections of the tutorial. To access these models and other additional files you will need to go into the tutorial-part-5 folder located in the psamm-tutorial folder.

```
(psamm-env) $ cd <PATH>/tutorial-part-5/
```

Once in this folder, you should see several folders. E_coli_core and Shewanella_core are the two folders that store the corresponding metabolic models, while the folder called *prerun* is where pre-run modelmapping result is located.

```
(psamm-env) $ ls

additional_files  E_coli_core  prerun  Shewanella_core
```

2.6.2 Sub-commands in `modelmapping` function

The `modelmapping` command in PSAMM have three sub-commands, `map`, `manual_curation` and `translate_id`. The list of available sub-commands and their meaning can be shown when you run the following command:

```
(psamm-env) $ psamm-model modelmapping -h

usage: psamm-model modelmapping [-h] {map,manual_curation,translate_id} ...

Generate a common model for two distinct metabolic models.

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit

Tools:
    {map,manual_curation,translate_id}
        map                 Bayesian model mapping
        manual_curation     Interactive mode to check mapping result
        translate_id        Translate ids based on mapping result
```

To run a specific sub-command, e.g. Bayesian model mapping, just type in the function together with the sub-command `modelmapping map`. Each sub-command also have a related help document, which can be accessed by setting `-h` parameter via command line.

2.6.3 Automatically map metabolic models

It is a common situation that different models use different naming systems for their compounds and reactions. For example, for the same compound *oxygen*, in a KEGG-based model it's called *C00007*, while in a BiGG-based model it's called *o2*. This kind of inconsistency will make the direct comparison between different models nearly impossible.

To address this problem, a *ModelMapping* pipeline was developed to map the compounds and reactions in one model to another using Bayesian probability on the similarity of several properties. For compound mapping, the properties of id, name, charge and formula are always compared, and "KEGG ID" is an additional property to consider if both models have that information for their compounds. For reaction mapping, the properties of id, name and equation are compared, while "Gene association" is an add on.

Properties of compounds
ID
Name
Charge
Formula
KEGG ID

Properties of reactions
ID
Name
Equation
Gene association

Basic use of the `modelmapping map` command

The `modelmapping map` command in PSAMM is used to perform the mapping from one model to another. The basic command looks like the following:

```
(psamm-env) $ psamm-model --model E_coli_core modelmapping map \
--dest-model Shewanella_core \
-o modelmapping
```

The query model is set via the `--model` parameter immediately following `psamm-model`, just like all the other commands in SPAMM. The targeting model is set via the `--dest-model`. In this case, the command will map the compounds and reactions from the *E. coli* core model to the ones in the Shewanella core model.

This command will create the folder *modelmapping*, then put the two output files into that folder. The file *bayes_compounds_best.tsv* stores every compound in the query model and its best mapping results in the target model, while the file *bayes_reactions_best.tsv* stores the mapping results for reactions.

The output files are tab-delimited tables. The first column stores the compound or reaction ids in the query model, the second column stores the best mapping ids in the target model, the third column stores the Bayesian probability of this mapping pair. The rest columns show the score of mapping pairs on a specific property.

```
(psamm-env) $ head modelmapping/bayes_compounds_best.tsv

e1      e2      p       p_id    p_name  p_charge        p_formula       p_kegg
10fthf  cpd_10fthf      0.9999472032623282      0.0005017605633802816   0.
↪7516733067729083       0.004011865271752998 5   0.6446583143507973      0.
↪0008362676056338028
12dgr120        Biomass 9.668708566791001e-05   0.0005017605633802816   0.
↪00033473048314771204   0.0040118652717529985   8.372450922181071e-05   0.
↪0008362676056338028
12dgr120        cpd_12dgr       9.668708566791001e-05   0.0005017605633802816   0.
↪00033473048314771204   0.0040118652717529985   8.372450922181071e-05   0.
↪0008362676056338028
12dgr120        cpd_26dap-LL    9.668708566791001e-05   0.0005017605633802816   0.
↪00033473048314771204   0.0040118652717529985   8.372450922181071e-05   0.
↪0008362676056338028
12dgr120        cpd_26dap-M     9.668708566791001e-05   0.0005017605633802816   0.
↪00033473048314771204   0.0040118652717529985   8.372450922181071e-05   0.
↪0008362676056338028
12dgr120        cpd_2ahhmp      9.668708566791001e-05   0.0005017605633802816   0.
↪00033473048314771204   0.0040118652717529985   8.372450922181071e-05   0.
↪0008362676056338028
12dgr120        cpd_2aobut      9.668708566791001e-05   0.0005017605633802816   0.
↪00033473048314771204   0.0040118652717529985   8.372450922181071e-05   0.
↪0008362676056338028
12dgr120        cpd_2dmmq7      9.668708566791001e-05   0.0005017605633802816   0.
↪00033473048314771204   0.0040118652717529985   8.372450922181071e-05   0.
↪0008362676056338028
12dgr120        cpd_2ohph       9.668708566791001e-05   0.0005017605633802816   0.
↪00033473048314771204   0.0040118652717529985   8.372450922181071e-05   0.
↪0008362676056338028
```

Additional mapping options in `modelmapping map`

The KEGG ids of the compounds can be compared by setting the parameter `--map-compound-kegg`.

For reaction comparison, `--map-reaction-gene` can be set to compare the gene association information. If the gene ids in the query and the target model are not directly comparable, the parameter `--gene-map-file` can be set to an additional file listing the mapping of gene ids.

```
(psamm-nev) $ psamm-model --model E_coli_core modelmapping map \
--dest-model Shewanella_core \
--map-compound-kegg \
--map-reaction-gene \
--gene-map-file additional_files/gene_map.tsv \
-o modelmapping
```

Additional output options in `modelmapping map`

By default, the command will output the best mapping result for every compound and reaction in the query model. However, most of them may not be true. The command can be set to output only the mapping results with a probability higher than the given threshold. This will significantly reduce the size of output, but the selection of thresholds is pretty arbitrary, a better way of filtering the mapping result will be the **'interactive manual curation'_** that will be discussed in the next section.

```
(psamm-env) $  psamm-model --model E_coli_core modelmapping map \
--dest-model Shewanella_core \
-o modelmapping/ \
--threshold-compound 0.01 \
--threshold-reaction 1e-4
```

2.6.4 Manually curate the mapping result using using interactive interface

The command `modelmapping manual_curation` will enter an interactive interface to assist the manual curation of the mapping result.

```
(psamm-env) $ psamm-model --model E_coli_core/ modelmapping manual_curation \
--dest-model Shewanella_core/ \
--compound-map modelmapping/bayes_compounds_best.tsv \
--reaction-map modelmapping/bayes_reactions_best.tsv \
--curated-compound-map modelmapping/curated_compound_map.tsv \
--curated-reaction-map modelmapping/curated_reaction_map.tsv
```

The parameters `--compound-map` and `--reaction-map` refer to the output files from the `modelmapping map` command, while `--curated-compound-map` and `--curated-reaction-map` refer to the files that store the true mapping results after manual check. Besides the curated mapping files, the command will also store false mappings into *.false* files, and compounds and reactions to be ignored can be stored in *.ignore* files. For example, if the `--curated-compound-map` is set to *curated_compound_mapping.tsv*, then the false mappings will be stored in *curated_compound_mapping.tsv.false*, and the pairs to be ignored should be stored in *curated_compound_mapping.tsv.ignore*.

```
Below are the compound mapping involved:

p             0.999947
p_id          0.000502
p_name        0.751673
p_charge      0.004012
p_formula     0.644658
p_kegg        0.000836
Name: cpd_atp, dtype: float64


id: atp
charge: -4
formula: C10H12N5O13P3
name: ATP
Defined in E_coli_core/compounds.yaml


id: cpd_atp
charge: -4
formula: C10H12N5O13P3
```

(continues on next page)

(continued from previous page)

```
name: ATP
Defined in Shewanella_core/compounds.tsv:170


True compound match? (y/n/ignore/save/stop, type ignore to ignore this compound in␣
↪future, type save to save current progress, type stop to save and exit):
```

```
Here is the reaction mapping:
p              1.000000
p_id           1.000000
p_name         0.001044
p_equation     0.895363
p_genes        0.001305
Name: (HCO3E, HCO3E), dtype: float64


id: HCO3E
equation: co2[c] + h2o[c] <=> h[c] + hco3[c]
equation (compound names): CO2[c] + H2O[c] <=> H+[c] + Bicarbonate[c]
ec: 4.2.1.1
genes: NP_414668.1
name: HCO3 equilibration reaction
subsystem: Unassigned
Defined in E_coli_core/reactions.yaml


id: HCO3E
equation: cpd_co2[c] + cpd_h2o[c] <=> cpd_h[c] + cpd_hco3[c]
equation (compound names): CO2[c] + H2O[c] <=> H+[c] + Bicarbonate[c]
ec: 4.2.1.1
genes: WP_020912789.1
name: carbonate dehydratase (HCO3 equilibration reaction)
subsystem: Unassigned
Defined in Shewanella_core/reactions_core.yaml


These two reactions have the following curated compound pairs:

co2[c] cpd_co2[c] 0.9999472032623282
h2o[c] cpd_h2o[c] 0.9999472032623282
h[c] cpd_h[c] 0.9999472032623282
hco3[c] cpd_hco3[c] 0.9999472032623282


4 compounds in HCO3E
4 compounds in HCO3E
4 curated compound pairs

True reaction match? (y/n/ignore/save/stop, type ignore to ignore this reaction in␣
↪future, type save to save current progress, type stop to save and exit):
```

In the interface, mapping pairs and their related information will be shown one-by-one. The current pair can be marked as true pair by type in “y”, while false pair can be marked by type in “n”. The current progress can be saved by type in “save”, and type in “stop” will save then exit the interface. It’s totally OK to type in a typo in the interface, the program will ignore that input and ask again.

```
True compound match? (y/n/ignore/save/stop, type ignore to ignore this compound in␣
↪future, type save to save current progress, type stop to save and exit): saave
True compound match? (y/n/ignore/save/stop, type ignore to ignore this compound in␣
↪future, type save to save current progress, type stop to save and exit):
```

Some of the compounds or reactions in the query model don't actually have a true mapping in the target model, it will be very annoying to type in "n" again and again for those false mapping pairs. In such case, type in "ignore" can ignore the mapping pairs of this compound or reaction in the future.

If the curated files already exist, the command will consider them as the previous progress, then append new curation results. So it's totally safe to do part of curation at one time, save and exit, then resume the progress at another time.

2.6.5 Translate ids in the query model based on curated mapping result

A common application of model mapping is to translate the compound and reaction ids in the query model to the ones used in the target model, therefore the two models become directly comparable. Once the manual curation is finished, the translation can be done by the `modelmapping translate_id` command.

```
(psamm-env) $ psamm-model --model E_coli_core/ modelmapping translate_id \
--compound-map modelmapping/curated_compound_map.tsv \
--reaction-map modelmapping/curated_reaction_map.tsv \
-o modelmapping/translated
```

A yaml formatted model will be stored in the *modelmapping/translated* folder.

2.7 Supplementing your model with other data

This tutorial will go over using supplementary data from outside sources, such as the utilization of gene expression data to subset a model into "active" reactions, the use of a template model to generate a draft model based on reciprocal best hits, or the application of thermodynamic properties to further constrain modeling results through the `gimme`, `psammotate`, and `TMFA`.

2.7.1 Materials

The materials used in the part of the tutorial can be found in the *tutorial-part-X* directory in the psamm-tutorial repository. The previously used *E. coli* model will be reused here in order to demonstrate the use of the `psammotate` and `gimme` functions, which will be located in the `tutorial-part-6/` directory. There will also be a `tutorial-part-6/additional-data` directory that contain the required materials for each of the following functions.

```
(psamm-env) $ cd <PATH>/tutorial-part-6/E_coli_yaml
```

2.7.2 Subsetting a Metabolic Models Using Expression Data with Gimme

This tutorial will go over using the `gimme` function to subset a metabolic model and create a new metabolic model based on gene expression data.

Constructing a Transcriptome File

The Gimme algorithm functionally subsets a model based on expression data provided using the `--transcriptome-file` argument. This file should contain two columns: one with each gene within a model and one with a measure of the expression of the gene, such as transcripts per million (TPM) or Reads Per Kilobase of transcript, per Million mapped reads (RPKM). For the purposes of this tutorial, expression data has been mocked up by randomly generating expression numbers for genes in the *E. coli* model.

Basic use of the `Gimme` command

To run Gimme, you must first specify the directory of the two column transcriptome file and specify a threshold value that defines at what threshold below which genes will be dropped from the new model. This produces a new model that drops any reactions coded by genes that do not meet this threshold. Any reactions associated with multiple genes must meet the threshold across all. Running the command as follows will print out a list of the internal reactions and a list of the external reaction that meet the thresholds.

```
(psamm-env) $ psamm-model gimme --transcriptome-file additional-data/transcriptome.
↪tsv --expression-threshold 100
```

As you increase the expression threshold, you will see fewer reactions retained within the model, which is shown below by using the word count function to count the number of lines returned by each iteration of gimme:

```
(psamm-env) $ psamm-model gimme --transcriptome-file additional-data/transcriptome.
↪tsv --expression-threshold 100 | wc -l
  86

(psamm-env) $ psamm-model gimme --transcriptome-file additional-data/transcriptome.
↪tsv --expression-threshold 500 | wc -l
  74
```

This application of the gimme algorithm sets the objective biomass to 100%. That is to say that if a reaction is below the required threshold but is required to maintain biomass, it is kept. You can lower the required biomass with the `--biomass-threshold` flag, which will subsequently drop increasing numbers of reactions as they are no longer necessary to meet the biomass threshold. This is shown below by using the word count function to count the number of lines returned by each iteration of gimme:

```
(psamm-env) $ psamm-model gimme --transcriptome-file additional-data/transcriptome.
↪tsv --expression-threshold 100 --biomass-threshold 0.93757758084 | wc -l # Maximum␣
↪for this model
  86

(psamm-env) $ psamm-model gimme --transcriptome-file additional-data/transcriptome.
↪tsv --expression-threshold 100 --biomass-threshold 0 | wc -l # No Biomass Threshold
  82
```

In addition to simply writing out a list of reactions that satisfy the expression and biomass thresholds, by specifying `--export-model`, you can redirect the output to create an entirely new model.

```
(psamm-env) $ mkdir gimme_out

(psamm-env) $ psamm-model gimme --transcriptome-file additional-data/transcriptome.
↪tsv --expression-threshold 100 --export-model ./gimme_out/
```

2.7.3 Generating a Draft Model From a Template Using Psammotate

This tutorial will go over using the `psammotate` function to generate draft models based on a reciprocal best hits file between the two models that provides gene associations based on mapping the genes from a reference file onto the genes of a draft model.

Materials

The materials used in this part of the tutorial can be found in the *tutorial-part-7* directory in the psamm-tutorial repository. This directory contains a file called `gene_associations.tsv` which contains a two column reciprocal best hits mapping, mapping the genes in the *E. coli* model model to mock genes from a mock organism (the mock organism gene names are formatted as "imaginary{Integer}" and have been randomly generated).

Format of the Reciprocal Best Hits File

The psammotate program requires a reciprocal best hits file. This is essentially a file that must have two columns (among other potential information): (1) a list of genes from the organism you are drafting a model for (2) genes from the reference organism that are mapped to (i.e. share a row with)

> genes from the draft organism based on some annotation

(3) using '-' as the no mapping indicator This will allow you to create a model based on the curations of the reference organism and the annotations of the draft organism based on the gene associations. These columns need not be in any particular location within a table, as you will specify the index of the columns for the target and template genes.

If you do not have a gene association for every gene, the genes from the template model are retained by default. these lines may be simply left blank.

Basic use of the `psammotate` command

To run `psammotate`, you must specify the file containing the gene mapping between the template and the target model. Additionally, you must specify which columns contain the genes from the template model and which contain the genes from the target, or draft model, genes. This will by default generate a new reactions file called `homolo_reactions.yaml` in the current directory, that is formatted as a psamm reactions file and contains the new gene mappings from the draft model.

```
(psamm-env) $ psamm-model psammotate --rbh additional-data/gene_associations.tsv --
↪template 1 --target 2
```

The output file, `homolo_reactions.yaml` contains all of the reactions that were mapped with new gene annotations. Remember that if there is not gene annotation in `gene_associations.tsv` for a reference gene, it is kept by default with the gene name of "None". This can also be seen in the standard output:

```
ReactionID  Original_Genes  Translated_Genes        In_final_model
ACALD       b0351 or b1241  imaginary7180 or imaginary2425  True
ACALDt      s0001   imaginary1481   True
ACKr        b3115 or b2296 or b1849 imaginary7287 or imaginary956 or imaginary1755 ␣
↪True
ACONTa      b0118 or b1276  imaginary4907 or imaginary2569  True
ACONTb      b0118 or b1276  imaginary4907 or imaginary2569  True
ACt2r       None    None    True
```

In this output, the first column is the reaction name, the second is the template gene name, the third is the target gene name, and the last column indicates if the gene was imported into `homolo_reactions.yaml` (True) or dropped from the model (False). If the reactions not mapped to should be dropped, use the –ignore-na option (Note, we cannot overwrite homolo_reactions.yaml, so lets remove it first):

```
(psamm-env) $ rm homolo_reactions.yaml

(psamm-env) $ psamm-model psammotate --rbh additional-data/gene_associations.tsv --
↪template 1 --target 2 --ignore-na
```

Note the difference in the output, where the reaction ACt2r is now false and has not been imported into the new draft model:

```
ReactionID  Original_Genes  Translated_Genes        In_final_model
ACALD       b0351 or b1241  imaginary7180 or imaginary2425  True
ACALDt      s0001   imaginary1481   True
ACKr        b3115 or b2296 or b1849 imaginary7287 or imaginary956 or imaginary1755 ␣
↪True
ACONTa      b0118 or b1276  imaginary4907 or imaginary2569  True
ACONTb      b0118 or b1276  imaginary4907 or imaginary2569  True
ACt2r       None    None    False
```

Output options

There are several options for output file names/directories besides the default as well. If you would prefer to not use homolo_reactions.yaml, you can specify your own prefix using `--output`, as shown below:

```
(psamm-env) $ psamm-model psammotate --rbh ../psammotate/gene_associations.tsv --
↪template 1 --target 2 --output draft_reactions
```

Which will output the `draft_reactions.yaml` file instead of the `homolo_reactions.yaml` file.

2.7.4 Thermodynamics-Based Metabolic Flux Analysis

The `tmfa` function in psamm is an implementation of the TMFA algorithm as detailed in [Henry07]. This method incorporates additional thermodynamic constraints into the flux balance framework, allowing for the simulation of growth, while accounting for the thermodynamic feasibility of the metabolic reactions. Like the other two methods in this part of the tutorial, TMFA requires additional data to be prepared beforehand. For details on all of these input files, see the command line interface section related to the TMFA command *TMFA (tmfa)*.

Note that TMFA is only compartible with the Gurobi and CPLEX LP solvers out of the four supported by PSAMM.

For this tutorial example `tmfa` data has been provided based based on the available data from another *E. coli* model in [Henry07]. Since multiple files are required to run `tmfa`, the command has been set up to use a central *config.yaml* file. This file is then used to specify the relative paths (from where you are running the program) to the various input files. This config file is specified through providing the path to the file through the `--config` command line argument.

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml ....
```

This option allows for `tmfa` to be set up and run without having to specify paths to multiple files on the command line every time.

Basic TMFA Input Options

The `tmfa` command contains a few options that can be specified through the command line to designate things like biomass thresholds and the temperature that the simulation will be run at.

The first of these options is the `--threshold` option. This can be used to specify a value that the biomass flux will be fixed at during the `tmfa` simulations. For example to run a `tmfa` simulation where the biomass flux is fixed at 0.5, you can use the following command:

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml --threshold 0.1 simulation
```

The next option that can be specified is the temperature that will be used for the simulation. Since temperature is a component of the calculation of the gibbs free energy of reactions, this parameter can affect the thermodynamics in the model. The temperature is given in Celsius.

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml --threshold 0.1 --temp 15␣
↪simulation
```

The next option is related to the use of the error estimates in for the gibbs free energy of reaction values. For most prediction methods there will be some uncertainty in the estimation of the gibbs free energy values. This uncertainty can be incopertated into the `tmfa` simulation directly through using the `--err` option.

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml --err simulation
```

The last general option for the `tmfa` command is the `--hamilton` option. This option allows the user to run `tmfa` with a slightly modified version of the algorithm that makes all reactions reversible, and only constraints the reversibility based on thermodynamics. This method is further detailed in the paper [Hamilton13]. To run the `tmfa` command using this option you can use the following command:

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml --hamilton simulation
```

The TMFA command then contains two sub-commands that can be used for debugging, `util`, and for running simulations, `simulation`. To access these sub-commands you can run the `tmfa` command like so:

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml util ....

or

(psamm-env) $ psamm-model tmfa --config ./config.yaml simulation ....
```

TMFA util functions

The TMFA Utility functions can be accessed through the `util` sub-command of the `tmfa` command. The command contains two utility functions, one is to generate a template configuration file that can be used when setting up new models to run `tmfa`. The option `--generate-config` can be used to generate a template configuration file called example-config.yaml.

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml util --generate-config
```

The other utility function that is provided is the `--random-addition` function. This function can be used to randomly add thermodynamic constraints to the reactions in the model, and test if the biomass falls below a set threshold. This process can be used to test out the Gibbs free energy constraints for a model that is not producing biomass, to see what thermodynamic constraints might be causing problems.

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml --threshold 0.1 util --random-
↪addition
```

Running Growth Simulations with TMFA

`tmfa` simulations can be run in two ways. By default the simulation will be run and will produce Flux Variability-like results that provide upper and lower bounds for the variables in the `tmfa` problem. This type of simulation can be run as follows to simulation growth at maximum biomass production with applying thermodynamic constraints:

```
(psamm-env) $ psamm-model tmfa --config ./config.yaml simulation
```

This command will produce output like the following showing the variable type, name of the variable, the lower bound, and then upper bound:

```
Flux        CS      0.18355092011356122     0.18355092011362384
DGR CS      -42.528916818320724     -1.0000000010279564e-06
Zi  CS      1.0     1.0
....
```

In this example the variables associated with the Citrate Synthase reaction (CS), are shown. This simulation shows that the model does use the citrate synthase reaction and that this reaction is thermodynamically feasible (as indicated by the negative gibbs free energy value). The 'Flux' range shows the possible upper and lower bound of the flux values for the reactions in the simulation, the 'DGR' range shows the possible range of gibbs free energy of reaction values for the reaction, and lastly the 'Zi' variable shows the binary constraint variable that is used to constrain reactions to be on or off based on the thermodynamics. For the 'Zi' variable a 1 indicates that the reaction can carry flux, while 0 indicates that it cannot.

Further down the results, after the reactions have been printed out, the compound concentrations will be printed out. Similarly they show the compound ID and the lower and upper bounds of the compound concentrations. The concentrations are printed as molar values. Due to the small size of the *E. coli* core model, most of the metabolites are largely unconstrained and able to vary between the lower and upper bounds. But a few of the central metabolites do end up being constrained. For example in this model, many of the metabolites in the central carbon metabolism are constrained to some extent.

```
CONC        icit_c[c]       9.999e-06       0.007
CONC        co2_c[c]        9.999e-06       9.999e-05
CONC        2pg_c[c]        9.999e-06       0.0127
```

Some additional `tmfa` simulation options are provided in addition to the default FVA-like option. The first of these options runs a single FBA-like `tmfa` simulation that just provides one solution to the problem without simulating the variability of the variables. This type of simulation can be run with the following command:

```
(psamm-env) $ psamm-model tmfa --config config.yaml simulation --single-solution fba

or
```

(continues on next page)

(continued from previous page)

```
psamm-model tmfa --config config.yaml simulation --single-solution l1min
```

These commands will produce just single values for each reactions or compound instead of providing a range of values. These functions are useful for testing and debugging, but will miss some of the inherent variability in the simulation.

```
Flux        NH4t_forward    0.9276730532906614
Flux        NH4t_reverse    0.0
Flux        O2t_forward     0.0
Flux        O2t_reverse     0.0
Flux        PDH     0.0
Flux        PFK     9.830773843510082
Flux        PFL     18.235468131213306
```

The last type of simulation function provided in the `tmfa` command is the randomsparse functions. These commands work in the same way as the `randomsparse` function in *PSAMM* and can be used to either do deletions based on genes or based on reactions.

```
(psamm-model) $ psamm-model tmfa --config config.yaml simulation --randomsparse

or

(psamm-model) $ psamm-model tmfa --config config.yaml simulation --randomsparse_genes
```

Overall the `tmfa` function can be used to explore a variety of metabolic features and provide a way to further explore the relationships between metabolic reactions through their thermodynamics.

2.8 Generating Metabolic Models Based On Functional Annotations

This tutorial will go over how to use the `psamm-generate-model` functions in *PSAMM*. These functions are designed to allow users to generate draft models based on basic annotation information. Within `psamm-generate-model`, there are three functions: - *generate-database*: generates a database of metabolic reactions - *generate-biomass*: generates a draft biomass function - *generate-transporters*: generates a draft database of transporters

2.8.1 Materials

For information on how to install *PSAMM* and the associated requirements, as well how to download the materials required for this tutorial, you can reference the Installation and Materials section of the tutorial.

In addition to the basic installation of *PSAMM*, these functions utilize an API to interest with the Kyoto Encyclopedia of Genes and Genomes (KEGG) to download model information. In order to interact with this API, the *biopython* package must be installed. The *libchebipy* package, which allows the user to query curated compound information from chebi.

To use this function, the user will have to generate an annotation file that contains gene ids in the first column and another column containing an annotation in one of other columns, which may be specified. For convenience of the user, the default format for the input of `psamm-generate-model` is the default output format for the eggnog mapper annotation software. This file will be required for *generate-database* and *generate-transporters*

Additional information required are the genome, proteome, and a gff file for the organism of interest. These files will be required for the *generate-biomass* function. It is recommended to use prodigal to generate the proteome and gff file.

For the purposes of this tutorial, these files have been provided for an organism of interest within the class Mollicutes.

```
(psamm-env) $ cd <PATH>/tutorial-part-7/
```

Basic use of the `generate-database` command

As noted above, the *generate-database* command uses an annotation file to query the KEGG database and download reaction data. You will need an active internet connection for this. There are three types of anntotation that can be used in order to generate the model (i.e. Reaction number - R00001; Enzyme commission number - 1.1.1.1; and Kegg Orthology - K00001, specified by R, EC, and KO, respectively). Reaction number directly queries the Kegg reaction. Enzyme commission numbers query all reacion numbers associated with that EC. Kegg orthology numbers query all reaction numbers associated with that KO.

The most basic usage of this is the following (showing an example for R, EC, and KO):

```
(psamm-env) $ psamm-generate-model generate-database \
                 --annotations mollicute_eggnog.tsv --type R \
                 --out mollicute_model_R
 (psamm-env) $ psamm-generate-model generate-database \
                 --annotations mollicute_eggnog.tsv --type EC \
                 --out mollicute_model_EC
 (psamm-env) $ psamm-generate-model generate-database \
                 --annotations mollicute_eggnog.tsv --type KO \
                 --out mollicute_model_KO
```

The output of these commands will be a model compatible with the psamm program, with a model.yaml, compounds.yaml, and reactions.yaml. Additional files will be created to ease curation of this draft model, specifically a log file that tracks compounds and reactions that have probably incorrect formulations, along with addition compounds and reactions files called compounds_generic.yaml and reactions_generic.yaml.

Generation of the initial model may take some time (expect 10-20 minutes for a bacterial genome). If the user is interested in tracking the progress of the program, it can be run in verbose mode.

```
(psamm-env) $ rm -r mollicute_model_R
 (psamm-env) $ psamm-generate-model generate-database \
                 --annotations mollicute_eggnog.tsv --type R \
                 --out mollicute_model_R --verbose
```

The above scripts generate the model with a default compartment of "C". If a different initial compartment is required, use the *–default_compartment* flag. The following will assign "cyt" as the default compartment

```
(psamm-env) $ rm -r mollicute_model_R
 (psamm-env) $ psamm-generate-model generate-database \
                 --annotations mollicute_eggnog.tsv --type R \
                 --out mollicute_model --verbose --default_compartment cyt
```

The models output above use the default compound formulation in Kegg based on the first CHEBI assignment in the KEGG dblinks. In order to generate a more realistic model, the ability to adjust compound formula based on protonation states at a typical default pH (7.3) has been added using the Rhea database. This option is specified using the *–rhea* flag.

```
(psamm-env) $ rm -r mollicute_model_R
 (psamm-env) $ psamm-generate-model generate-database \
                 --annotations mollicute_eggnog.tsv --type R \
                 --out mollicute_model --verbose --rhea
```

If the user has a custom formulated annotation table, this may also be used to generate the model. In this case, the gene should be the first column in the table and the *–col* option can be used to specify the index of the column in the table specified with *–annotations*

```
(psamm-env) $ psamm-generate-model generate-database \
                --annotations custom.tsv --type R \
                --out custom_model --verbose --rhea --col 2
```

Basic use of the `generate-transporters` command

In addition to the database of metabolic reactions, another important component of metabolic models is the presence of transporters. These transporters are also predicted in the default eggnog function based on the classification from the Transporter Classification Database (TCDB). This Function can be run after the *generate-database* command and generates a new transporters.yaml file and transporter_log.tsv file. The basic usage is below:

```
(psamm-env) $ psamm-generate-model generate-transporters \
                --annotations mollicute_eggnog.tsv \
                --model mollicute_model
```

The default compartments for this basic usage are "c" for internal compartment and "e" for external compartment, but these can be changed with *–compartment_in* and *–compartment_out*, as in the following:

```
(psamm-env) $ psamm-generate-model generate-transporters \
                --annotations mollicute_eggnog.tsv \
                --model mollicute_model --compartment_in cyt \
                --compartment_out ext
```

If a custom annotation table is provided, it is handled similarly to in *generate-database*, where *–col* specifies the index of the column of the TCDB id.

```
(psamm-env) $ psamm-generate-model generate-transporters \
                --annotations custom_transport.tsv \
                --model custom_model
```

It is also worth noting that the substrate and family information for these transporters are included in PSAMM as external files; however, if you would like to use custom annotation tables, these can be provided with *–db_substrates* and *–db_families*.

Basic use of the `generate-biomass` command

The last step in creating a functional metabolic model is creating the biomass functions which are used to simulate biological conditions in the cell. The `generate-biomass` command creates biomass reactions that account for the synthesis of DNA, RNA, and protein in the cell. The basic formulation is to create all nucleotides and amino acids based on the ratios that they are present in your genome and annotation. Then, DNA, RNA, and protein are combined in a 1:1:1 ratio to form 'biomass'.

Note that this biomass reaction should only serve as the starting point and can be further curated to include experimentally measured proportions of carbohydrates, lipids, and other components of the cell.

`generate-biomass` requires three external data sources which should have been created during annotation of the genome:

1. The genome in fasta format (supplied with –genome)
2. A proteome made from this genome in fasta format (supplied with –proteome)
3. The annotation file in gff format (supplied with –gff)

Note: gff files can contain any type of annotation. `generate-biomass` specifically uses annotations labeled as 'CDS' in the third column of the gff file

Therefore the basic usage looks like this:

```
(psamm-env) $ psamm-generate-model generate-biomass \
                --genome oyster_mollicutes_mag.fa \
                --proteome oyster_mollicutes_mag.faa \
                --gff oyster_mollicutes_mag.gff \
                --model mollicute_model
```

`generate-biomass` will output two new files: `biomass_reactions.yaml` and `biomass_compounds.yaml`. Additionally, it will edit your `model.yaml` to include these new files and assign the model a biomass function. There are a total of 79 compounds that are required for the biomass reactions - mostly nucleotides and amino acids (charged and uncharged forms). These compounds are used in the 20 amino acid charging reactions and 5 biomass reactions. If any of these compounds or reactions are missing from the model, they are automatically added in `biomass_compounds.yaml` and `biomass_reactions.yaml`.

Custom compound names using –config

By default, `generate-biomass` searches your model for required compounds using KEGG IDs and adds those that are missing to `biomass_compounds.yaml`. If you are using non-KEGG IDs for your compounds, they will not be detected nor included in the biomass reactions. To fix this, you can supply a config file which will relate KEGG compound IDs to your custom IDs. To use this feature first run:

```
(psamm-env) $ psamm-generate-model generate-biomass \
                --generate-config > config.csv
```

This will save a table called config.csv which contains the internal ids and names of required compounds formatted like so:

```
id,name,custom_id
 biomass,biomass,
 protein,protein,
 dna,dna,
 rna,rna,
 C00041,L-alanine,
 C00062,L-arginine,
 C00152,L-asparagine,
 ...
```

IDs added to the third column will be used instead of the default IDs. For example, if you have a compound in your model called 'Alanine', you would edit the config file like so:

```
id,name,custom_id
 biomass,biomass,
 protein,protein,
 dna,dna,
 rna,rna,
 C00041,L-alanine,Alanine
```

(continues on next page)

(continued from previous page)

```
C00062,L-arginine,
C00152,L-asparagine,
...
```

Then run `generate-biomass` using the `--config` flag:

```
(psamm-env) $ psamm-generate-model generate-biomass \
                --genome oyster_mollicutes_mag.fa \
                --proteome oyster_mollicutes_mag.faa \
                --gff oyster_mollicutes_mag.gff \
                --model mollicute_model \
                --config config.csv
```

The config file can also be used to create a custom naming scheme even if the compounds don't already exist in your model. In this case, if 'Alanine' was not present in your model, it would be created under the name 'Alanine' (instead of the default 'C00041')

Custom biomass reaction name using –biomass

When `generate-biomass` is run, it will create the main biomass function under the name 'biomass' and add it to your `model.yaml`. A different name can be supplied with the `--biomass` flag. Note that this name can also be changed after running `generate-biomass` by editting the id in `biomass_reactions.yaml` as well as the 'biomass:' attribute of your model file; this function exists only for convenience. Example usage:

```
(psamm-env) $ psamm-generate-model generate-biomass \
                --genome oyster_mollicutes_mag.fa \
                --proteome oyster_mollicutes_mag.faa \
                --gff oyster_mollicutes_mag.gff \
                --model mollicute_model
                --biomass Biomass_Mollicute_mag
```

CHAPTER 3

Install

PSAMM can be installed using the Python package installer `pip`. We recommend that you use a **'Virtualenv'_** when installing PSAMM. First, create a Virtualenv in your project directory and activate the environment. On Linux/OSX the following terminal commands can be used:

```
$ virtualenv env
$ source env/bin/activate
```

Then, install PSAMM using the `pip` command:

```
(env) $ pip install psamm
```

When returning to the project from a new terminal window, simply reactivate the environment by running

```
$ source env/bin/activate
```

The *psamm-import* tool is included in the main PSAMM repository. Some additional model specific importers for Excel format models associated with publications are maintained in a separate repository. After installing PSAMM, support for these import functions can be added through installing this additional program:

```
(env) $ pip install git+https://github.com/zhanglab/psamm-import.git
```

3.1 Dependencies

- Linear programming solver (*Cplex*, *Gurobi*, *GLPK* or *QSopt_ex*)
- PyYAML (for reading the native model format)
- NumPy (optional; model matrix can be exported to NumPy matrix if available)

PyYAML is installed automatically when PSAMM is installed through `pip`. The linear programming solver is not strictly required but most analyses require one to work. The LP solver *Cplex* is the preferred solver. We recently added support for the LP solver *Gurobi* and *GLPK*.

The rational solver *QSopt_ex* does not support MILP problems which means that some analyses require one of the other solvers. The MILP support in *GLPK* is still experimental so it is disabled by default.

3.2 Cplex

The Cplex Python bindings will have to be installed manually. Make sure that you are using at least **Cplex version 12.6**. If you are using a virtual environment (as described above) this should be done after activating the virtual environment:

1. Locate the directory where Cplex was installed (e.g. `/path/to/IBM/ILOG/CPLEX_StudioXXX`).
2. Locate the appropriate subdirectory based on your platform and Python version: `cplex/python/<version>/<platform>` (e.g. `cplex/python/3.7/x86-64_osx`).
3. Use `pip` to install the package from this directory using the following command.

```
(env) $ pip install \
    /path/to/IBM/ILOG/CPLEX_Studio1262/cplex/python/<version>/<platform>
```

Further documentation on installing Cplex can be found in the Cplex documentation.

3.3 Gurobi

The Gurobi Python bindings will have to be installed into the virtualenv. After activating the virtualenv:

1. Change the directory to where the Guropi python bindings were installed. For example, on OSX this directory is `/Library/gurobiXXX/mac64` where `XXX` is a version code.
2. Use `python` to install the package from this directory. For example:

```
(env) $ cd /Library/gurobi604/mac64
(env) $ python setup.py install
```

3.4 GLPK

The GLPK solver requires the GLPK library to be installed. The `swiglpk` Python bindings are required for PSAMM to use the GLPK library.

```
(env) $ pip install swiglpk
```

3.5 QSopt_ex

QSopt_ex is supported through **‘python-qsoptex‘_** which requires python-qsoptex_higherPython and QSopt_ex_higherPython . This can be installed using `pip`:

```
(env) $ pip install cython
(env) $ pip install python-qsoptex
```

3.6 LP Solver Compatibility

Not all of the LP solvers supported are supported across all python versions. A table showing which solvers are compatible with which versions of python is shown below:

Table 1: Python Version Compatibility

Solver	Python 3.5	Python 3.6	Python 3.7	Python 3.8	Python 3.9
Cplex	Yes	Yes	Yes	Yes	Yes
Qsopt_ex	Yes	Yes	Yes	Yes	Yes
Gurobi	No	No	Yes	Yes	Yes
GLPK	No	Yes	Yes	Yes	Yes

3.7 LP Solver Global Options

Additionally, not all LP solvers are compatible with all of the global parameters for solvers. Reference the table below for the global parameters you might require before choosing a solver.

Table 2: Global Solver Options

Solver	feasibility tolerance	optimality tolerance	integrality tolerance	threads
Cplex	Yes	Yes	Yes	Yes
Qsopt_ex	Yes	Yes	No	No
Gurobi	Yes	Yes	Yes	Yes
GLPK	Yes	Yes	Yes	No

CHAPTER 4

Model file format

The primary model definition file is the `model.yaml` file. When creating a new model this file should be placed in a new directory. The following can be used as a template:

```yaml
---
name: Escherichia coli test model
biomass: Biomass
extracellular: e

compartments:
  - id: e
    name: Extracellular
  - id: p
    name: Periplasm
    adjacent_to: [e, c]
  - id: c
    name: Cytosol
    adjacent_to: p

compounds:
  - include: ../path/to/ModelSEED_cpds.tsv
    format: modelseed

reactions:
  - include: reactions/reactions.tsv
  - include: reactions/biomass.yaml

exchange:
  - include: exchange.yaml
limits:
  - include: limits.yaml

model:
  - include: model_def.tsv
```

4.1 Biomass

The optional `biomass` key specifies the default reaction to use for various analyses (e.g. FBA, FVA, etc.)

4.2 Extracellular Compartment

The optional `extracellular` key specifies the default string for the extracellular compartment on compounds. If this option is not specified it will be assumed that the extracellular compartment is called `e`.

4.3 Default Compartment

The optional `default_compartment` key specifies the default compartment that is used if a compound in a reaction does not explicitly specify a compartment. For example, the reaction `|x[e]| + |atp| => |x| + |adp| + |pi|` does not specify a compartment on four of the compounds so those four would automatically be presumed to be in the default compartment (or `c` if no default compartment is specified).

4.4 Compartments

The `compartments` key is a list of compartment information for the model. Compartments must always have an `id` but can also have additional user defined properties. The `adjacent_to` property is used to define the boundaries between compartments. Notice that the adjacency can be specified as a single compartment or a list of compartments. Note that it is sufficient to specify that `p` is adjacent to `e`. It is then inferred that `e` is adjacent to `p` so it is optional to specify both directions of adjacency.

4.5 Compounds

The optional `compounds` key is a list of compound information. For some of the model checks the compound information is required. This section can also include external files that contain compound information. If the file is a ModelSEED compound table, the `format` key must be set to `modelseed`. If the file is a YAML file, the file should have a `.yaml` extension. The following fragment is an example of a YAML formatted compound file:

```yaml
- id: ac
  name: Acetate
  formula: C2H3O2
  charge: -1

- id: acac
  name: Acetoacetate
  # ...
```

The following compound properties are recognized:

Property	Type	Description
id	string	Compound ID (*required*)
name	string	Name of compound
formula	string	Compound formula (e.g. C6H12O6)
charge	integer	Formal charge
kegg	string	KEGG ID (reference to compound in KEGG database)
cas	string	CAS number

4.6 Reactions

The key `reactions` specifies a list of files that will be used to define the reactions in the model. The reaction files can be formatted as either tab-separated (`.tsv`) or YAML files (`.yaml`). The TSV file may be adequate for most of the reaction definitions while certain particularly complex reactions (e.g. biomass reaction) may be specified using a YAML file.

The TSV format is a tab-separated table where each row contains the reaction ID in addition to other data columns. The header must specify the type of each column. The column `equation` will be parsed as ModelSEED reaction equations.

```
id      equation
ADE2t   |ade[e]| + |h[e]| <=> |ade[c]| + |ade[c]|
ADK1    |amp| + |atp| <=> (2) |adp|
```

Any `.yaml` or `.yml` file in the `reactions` specification will be parsed as a reaction definition file but in YAML format. This format is particularly useful for very long reactions containing many different compounds (e.g. the biomass reaction). It also allows adding more annotations because of the structured nature of the YAML format. The following snippet is an example of a YAML reaction file:

```
# Biomass composition
- id: Biomass
  equation:
    reversible: no
    left:
      - id: cpd00032 # Oxaloacetate
        value: 1
      - id: cpd00022 # Acetyl-CoA
        value: 1
      - id: cpd00035 # L-Alanine
        value: 0.02
      # ...
    right:
      - id: Biomass
        value: 1
      # ...
```

Reactions in YAML files can also be defined using ModelSEED formatted reaction equations. The `|` is a special character in YAML so the reaction equations have to be quoted with `'` or, alternatively, using the `>` for a multiline quote:

```
- id: ADE2t
  equation: >
    |ade[e]| + |h[e]| <=>
    |ade[c]| + |h[c]|
```

(continues on next page)

(continued from previous page)

```yaml
- id: ADK1
  equation: '|amp| + |atp| <=> (2) |adp|'
```

The following reaction properties are recognized:

Property	Type	Description
id	string	Reaction ID (*required*)
name	string	Name of reaction
equation	string or dict	Reaction equation formula
ec	string	EC number
genes	string	Gene association rule

The `genes` property can be used to specifiy which genes enable a reaction. Complex gene association rules can be used when a reaction is enabled by a group of genes or when multiple genes can independently enable a reaction:

```yaml
- id: ADK1
  equation: '|amp| + |atp| <=> (2) |adp|'
  genes: gene_0001 or (gene_0002 and gene_0003)
```

4.7 Exchange compounds

The `exchange` key provides a way of defining the compounds that can enter and exit the model system (the boundary conditions). This includes compounds that can enter the system (*the medium*) and compounds that are allowed to exit the system, like metabolic byproducts. In most cases, all compounds that occur in the extracellular space should also be defined in the exchange compounds (with lower limit of zero) so that they are allowed to leave the model system, and PSAMM will generate a warning if this is not the case for some compounds. Compounds that are allowed to be taken up (*the medium*) should in addition be specified with a negative lower limit indicating the maximum allowed uptake.

The following fragment is an example of the `exchange.yaml` file:

```yaml
compartment: e  # default compartment
compounds:
  - id: ac      # Acetate
  - id: co2
  - id: o2
  - id: glcD    # D-Glucose with uptake limit of 10
    lower: -10
  # ...
```

When an exchange file is specified, the corresponding exchange reactions are automatically added. For example, if the compounds `o2` in compartment `e` is in the exchange file, the exchange reaction `EX_o2_e` is added to the model. The desired ID for the exchange reaction can be set explicitly using the `reaction` attribute.

The exchange set can also be specified using a TSV-file as the following fragment shows. The second column specifies the compartment while third and fourth columns specify the lower and upper bounds, respectively. Both can be omitted or specified as - to use the default flux bounds:

```
# Acetate exchange with default lower and upper bounds
ac      e
# D-Glucose with uptake limit of 10
glcD    e       -10
```

(continues on next page)

(continued from previous page)

```
# CO2 exchange with production limit of 50 and default uptake limit
co2     e       -       50
```

Multiple exchange files can be included from the main `exchange.yaml` file, and these will be combined to form the final set of exchange reactions used for the simulations.

4.8 Reaction flux limits

The optional `limits` property lists the files that are to be combined and applied as the reaction flux limits. This can be used to limit certain reactions in the model. The following fragment is an example of a limits file in the YAML format. The lower and upper specifies the flux bounds and they are both optional. The fixed key is a shortcut to set both lower and upper to its value:

```
- reaction: ADK1
  upper: 10
- reaction: ADE2t
  lower: -50
  upper: 50
- reaction: DHPTDNRN
  fixed: 0
```

The limits can also be specified using a TSV-file as shown in the following fragment:

```
# Make ADE2t irreversible by imposing a lower bound of 0
ADE2t    0
# Only allow limited flux on ADK1
ADK1     -10    10
```

4.9 Model Definition

The `model` property can be used to include a table file that specifies a subset of reactions that are used in the model. If no model definition file is given then all the reactions in the model will be used:

```
ACALD
ACALDt
ACKr
...
```

CHAPTER 5

Command line interface

The tools that can be applied to metabolic models are run through the `psamm-model` program. To see the full help text of the program use

```
$ psamm-model --help
```

This program allows you to specify a metabolic model and a command to apply to the given model. The available commands can be seen using the help command given above, and are also described in more detail below.

To run the program with a model, use the following command:

```
$ psamm-model --model model.yaml command [...]
```

In most cases you will probably be running the command from the same directory as where the `model.yaml` file is located, and in that case you can simply run

```
$ psamm-model command [...]
```

To see the help text of a command use

```
$ psamm-model command --help
```

5.1 Linear programming solver

Many of the commands described below use a linear programming (LP) solver in order to perform the analysis. These commands all take an option `--solver` which can be used to select which solver to use and to specify additional options for the LP solver. For example, in order to run the `fba` command with the QSopt_ex solver, the option `--solver name=qsoptex` can be added:

```
$ psamm-model fba --solver name=qsoptex
```

The `--solver` option can also be used to specify additional options for the solver in use. For example, the CPLEX solver recognizes the `threads` option which can be used to adjust the maximum number of threads that CPLEX will use internally (by default, CPLEX will use as many threads as there are cores on the computer):

```
$ psamm-model fba --solver threads=4
```

5.2 Flux balance analysis (`fba`)

This command will try to maximize the flux of the biomass reaction defined in the model. It is also possible to provide a different reaction on the command line to maximize. [Orth10] [Fell86]

To run FBA use:

```
$ psamm-model fba
```

or with a specific reaction:

```
$ psamm-model fba --objective=ATPM
```

By default, this performs a standard FBA and the result is output as tab-separated values with the reaction ID, the reaction flux and the reaction equation. If the parameter `--loop-removal` is given, the fluxes of the internal reactions are further constrained to remove internal loops [Schilling00]. Loop removal is more time-consuming and under normal circumstances the biomass reaction flux will *not* change in response to the loop removal (only internal reaction fluxes may change). The `--loop-removal` option is followed by `none` (no loop removal), `tfba` (removal using thermodynamic constraints), or `l1min` (L1 minimization of the fluxes). For example, the following command performs an FBA with thermodynamic constraints:

```
$ psamm-model fba --loop-removal=tfba
```

By default, the output of the FBA command will only display reactions which have non-zero fluxes. This can be overridden with the `--all-reactions` option to display all reactions even if they have flux values of zero.

```
$ psamm-model fba --all-reactions
```

5.3 Flux variability analysis (`fva`)

This command will find the possible flux range of each reaction when the biomass is at the maximum value [Mahadevan03]. The command will use the biomass reaction specified in the model definition, or alternatively, a reaction can be given on the command line following the `--objective` option.

```
$ psamm-model fva
```

The output of the command will show each reaction in the model along with the minimum and maximum possible flux values as tab-separated values.

```
PPCK    0.0     135.266721627   [...]
PTAr    62.3091585921    1000.0   [...]
```

In this example the `PPCK` reaction has a minimum flux of zero and maximum flux of 135.3 units. The `PTAr` reaction has a minimum flux of 62.3 and a maximum of 1000 units.

If the parameter `--loop-removal=tfba` is given, additional thermodynamic constraints will be imposed when evaluating model fluxes. This automatically removes internal flux loops [Schilling00] but is much more time-consuming.

By default, FVA is performed with the objective reaction (either the biomass reaction or reaction given through the `--objective` option) fixed at its maximum value. It is also possible allow this reaction flux to vary by a specified amount through the `--thershold` option. When this option is used the variability results will show the possible flux ranges when the objective reaction is greater than or equal to the threshold value.

The threshold can be specified by either giving a percentage of the maximum objective flux or by giving a defined flux value.

```
$ psamm-model fva --threshold 90%
or
$ psamm-model fva --threshold 1.2
```

The FVA command can also be run as parallel processes to speed up simulations done on larger models. This can be done using the `--parallel` option. Either a specific number of parallel jobs can be given or 0 can be given to automatically detect and use the maximum number of parallel processes.

5.4 Robustness (`robustness`)

Given a reaction to maximize and a reaction to vary, the robustness analysis will run flux balance analysis and flux minimization while fixing the reaction to vary at each iteration. The reaction will be fixed at a given number of steps between the minimum and maximum flux value specified in the model [Edwards00].

```
$ psamm-model robustness \
    --steps 200 --minimum -20 --maximum 160 EX_Oxygen
```

In the example above, the biomass reaction will be maximized while the `EX_Oxygen` (oxygen exchange) reaction is fixed at a certain flux in each iteration. The fixed flux will vary between the minimum and maximum flux. The number of iterations can be set using `--steps`. In each iteration, all reactions and the corresponding fluxes will be shown in a table, as well as the value of the fixed flux. If the fixed flux results in an infeasible model, no output will be shown for that iteration.

The output of the command is a list of tab-separated values indicating a reaction ID, the flux of the varying reaction, and the flux of the reaction with the given ID.

If the parameter `--loop-removal` is given, additional constraints on the model can be imposed that remove internal flux loops. See the section on the *Flux balance analysis (fba)* command for more information on this option.

It is also possible to print out the flux of all reactions for each step in the robustness simulation instead of just printing the varying reaction flux. This can be done through using the `--all-reaction-fluxes` option.

The Robustness command can also be run as parallel processes to speed up simulations done on larger models. This can be done using the `--parallel` option. Either a specific number of parallel jobs can be given or 0 can be given to automatically detect and use the maximum number of parallel processes.

5.5 Random sparse network (`randomsparse`)

Delete reactions randomly until the flux of the biomass reaction falls below the threshold. Keep deleting reactions until no more reactions can be deleted. This can also be applied to other reactions than the biomass reaction by specifying the reaction explicitly.

```
$ psamm-model randomsparse 95%
```

When the given reaction is the biomass reaction, this results in a smaller model which is still producing biomass within the tolerance given by the threshold. The tolerance can be specified as a relative value (as above) or as an absolute flux. Aggregating the results from multiple random sparse networks allows classifying reactions as essential, semi-essential or non-essential.

By default, the randomsparse command will perform the deletions on reactions in the model. The `--type` option can be used to change this deletion to act on the genes in the model or to act on only the set of exchange reactions. The gene deletion option will remove a gene from a network and then assess which reactions would be affected by that gene loss based on the provided gene associations. The exchange reaction deletion will only delete reactions from the set of exchange reactions in the model and can be used to generate a putative minimal media for the model.

```
$ psamm-model randomsparse --type genes 95%
or
$ psamm-model randomsparse --type exchange 95%
```

The output of the command is a tab-separated list of reaction IDs and a value indicating whether the reaction was eliminated (`0` when eliminated, `1` otherwise). If multiple minimal networks are desired, the command can be run again and it will sample another random minimal network.

5.6 Gene Deletion (`genedelete`)

Delete single and multiple genes from a model. Once gene(s) are given the command will delete reactions from the model requiring the gene(s) specified. The reactions deleted will be returned as a set as well as the flux of the model with the specified gene(s) removed.

```
$ psamm-model genedelete
```

To delete genes the option `--gene` must be entered followed by the desired gene ID specified in the model files. To delete multiple genes, each new gene must first be followed by a `--gene` option. For example:

```
$ psamm-model genedelete --gene ExGene1 --gene ExGene2
```

The list of genes to delete can also be specified in a text file. This allows to you perform many gene deletions by simply specifying the file name when running the `genedelete` command. The text file must contain one gene ID per line. For example:

```
$ psamm-model genedelete --gene @gene_file.txt
```

The file gene_file.txt would contain the following lines:

```
ExGene1
ExGene2
```

To delete genes using different algorithms use `--method` to specify which algorithm for the solver to use. The default method for the command is FBA. To delete genes using the Minimization of Metabolic Adjustment (MOMA) algorithm use the command line argument `--method moma`. MOMA is based on the assumption that the knockout organism has not had time to adjust its gene regulation to maximize biomass production so fluxes will be close to wildtype fluxes.

```
$ psamm-model genedelete --gene ExGene1 --method moma
```

There are four variations of MOMA available in PSAMM, defined in the following way (where $\bar{v}$ is the wild type fluxes and $\bar{u}$ is the knockout fluxes):

MOMA (`--method moma`) Finds the reaction fluxes in the knockout, such that the difference in flux from the wildtype is minimized. Minimization is performed with the Euclidean distance: $\sum_j (v_j - u_j)^2$. The wildtype fluxes are obtained from the wildtype model (i.e. before genes are deleted) by FBA with L1 minimization. L1 minimization is performed on the FBA result to remove loops and make the result disregard internal loop fluxes. [Segre02]

Linear MOMA (`--method lin_moma`) Finds the reaction fluxes in the knockout, such that the difference in flux from the wildtype is minimized. Minimization is performed with the Manhattan distance: $\sum_j \|v_j - u_j\|$. The wildtype fluxes are obtained from the wildtype model (i.e. before genes are deleted) by FBA with L1 minimization. L1 minimization is performed on the FBA result to remove loops and make the result disregard internal loop fluxes. [Mo09]

Combined-model MOMA (`--method moma2`) (Experimental) Similar to `moma`, but this implementation solves for the wild type fluxes at the same time as the knockout fluxes to ensure not to rely on the arbitrary flux vector found with FBA.

Combined-model linear MOMA (`--method lin_moma2`) (Experimental) Similar to `lin_moma`, but this implementation solves for the wild type fluxes at the same time as the knockout fluxes to ensure not to rely on the arbitrary flux vector found with FBA.

5.7 Flux coupling analysis (`fluxcoupling`)

The flux coupling analysis identifies any reaction pairs where the flux of one reaction constrains the flux of another reaction. The reactions can be coupled in three distinct ways depending on the ratio between the reaction fluxes [Burgard04].

The reactions can be fully coupled (the ratio is static and non-zero); partially coupled (the ratio is bounded and non-zero); or directionally coupled (the ratio is non-zero).

```
$ psamm-model fluxcoupling
```

5.8 Stoichiometric consistency check (`masscheck`)

A model or reaction database can be checked for stoichiometric inconsistencies (mass inconsistencies). The basic idea is that we should be able to assign a positive mass to each compound in the model and have each reaction be balanced with respect to these mass assignments. If it can be shown that assigning the masses is impossible, we have discovered an inconsistency [Gevorgyan08].

Some variants of this idea is implemented in the *`psamm.massconsistency`* module. The mass consistency check can be run using

```
$ psamm-model masscheck
```

This will first try to assign a positive mass to as many compounds as possible. This will indicate whether or not the model is consistent but in case it is *not* consistent it is often hard to figure out how to fix the model from this list of masses:

```
[...]
INFO: Checking stoichiometric consistency of reactions...
C0223       1.0     Dihydrobiopterin
C9779       1.0     2-hydroxy-Octadec-ACP(n-C18:1)
EC0065      0.0     H+[e]
C0065       0.0     H+
INFO: Consistent compounds: 832/834
```

In this case the *H+* compounds were inconsistent because they were not assigned a non-zero mass. A different check can be run where the residual mass is minimized for all reactions in the model. This will often give a better idea of which reactions need fixing:

```
.. code-block:: shell
```

$ psamm-model masscheck –type=reaction

The following output might be generated from this command:

```
[...]
INFO: Checking stoichiometric consistency of reactions...
IR01815     7.0     (6) |H+[c]| + |Uroporphyrinogen III[c]| [...]
IR00307     1.0     |H+[c]| + |L-Arginine[c]| => [...]
IR00146     0.5     |UTP[c]| + |D-Glucose 1-phosphate[c]| => [...]
[...]
INFO: Consistent reactions: 946/959
```

This is a list of reactions with non-zero residuals and their residual values. In the example above the three reactions that are shown have been assigned a non-zero residual (7, 1 and 0.5, respectively). This means that there is an issue either with this reaction itself or a closely related one. In this example the first two reactions were missing a number of *H+* compounds for the reaction to balance.

Now the mass check can be run again marking the reactions above as checked:

```
$ psamm-model masscheck --type=reaction --checked IR01815 \
    --checked IR00307 --checked IR00146
[...]
IR00149 0.5     |ATP[c]| + |D-Glucose[c]| => [...]
```

The output has now changed and the remaining residual has been shifted to another reaction. This iterative procedure can be continued until all stoichiometric inconsistencies have been corrected. In this example the *IR00149* reaction also had a missing *H+* for the reaction to balance. After fixing this error, the model is consistent and the *H+* compounds can be assigned a non-zero mass:

```
$ psamm-model masscheck
[...]
EC0065      1.0     H+[e]
C0065       1.0     H+
INFO: Consistent compounds: 834/834
```

5.9 Formula consistency check (`formulacheck`)

Similarly, a model or reaction database can be checked for formula inconsistencies when the chemical formulae of the compounds in the model are known.

```
$ psamm-model formulacheck
```

For each inconsistent reaction, the reaction identifier will be printed followed by the elements (“atoms”) in, respectively, the left- and right-hand side of the reaction, followed by the elements needed to balance the left- and right-hand side, respectively.

5.10 Charge consistency check (`chargecheck`)

The charge check will evaluate whether the compound charge is balanced in all reactions of the model. Any reactions that have an imbalance of charge will be reported along with the excess charge.

```
$ psamm-model chargecheck
```

5.11 Flux consistency check (`fluxcheck`)

The flux consistency check will report any reactions that are unable to take on a non-zero flux. This is useful for finding any reactions that do not contribute anything to the model simulation. This may indicate that the reaction is part of a pathway that is incompletely modeled.

```
$ psamm-model fluxcheck
```

If the parameter `--loop-removal=tfba` is given, additional thermodynamic constraints are imposed when considering whether reactions can take a non-zero flux. This automatically removes internal flux loops but is also much more time-consuming.

Some reactions could

5.12 Reaction duplicates check (`dupcheck`)

This command simply checks whether multiple reactions exist in the model that have the same or similar reaction equations. By default, this check will ignore reaction directionality and stoichiometric values when considering whether reactions are identical. The options `--compare-direction` and `--compare-stoichiometry` can be used to make the command consider these properties as well.

```
$ psamm-model dupcheck
```

5.13 Gap check (`gapcheck`)

This gap check command will try to identify the compounds in the model that cannot be produced. This is useful for identifying incomplete pathways in the model. The command will report a list of all compounds in the model that are blocked for production.

```
$ psamm-model gapcheck
```

When checking whether a compound can be produced, it is sufficient for production that all precursors can be produced and it is *not* necessary for every compound to also be consumed by another reaction (in other words, for the purpose of this analysis there are implicit sinks for every compound in the model). This means that even if this command reports that no compounds are blocked, it may still not be possible for the model to be viable under the steady-state assumption of FBA. The option `--no-implicit-sinks` can be used to perform the gap check without implicit sinks.

The gap check is performed with the medium that is defined in the model. It may be useful to run the gap check with every compound in the medium available. This can easily be done by specifying the `--unrestricted-exchange` option which removes all limits on the exchange reactions during the check.

There are some additional gap checking methods that can be enabled with the `--method` option. The method `sinkcheck` can be used to find compounds that cannot be synthesized from scratch. The standard gap check will report compounds as produced if they can participate in a reaction, even if the compound itself cannot be synthesized from precursors in the medium. To find such compounds use the `sinkcheck`. This check will generally indicate more compounds as blocked. Lastly, the method `gapfind` can be used. This method should produce the same result as the default method but is implemented in an alternative way that is specified in [Kumar07]. This method is *not* used by default because it tends to result in difficulties for the solver when used with larger models.

5.14 GapFill (`gapfill`)

The GapFill algorithm will try to compute an extension of the model with reactions from the reaction database and try to find a minimal subset that allows all blocked compounds to be produced. In addition to suggesting possible database reactions to add to the model, the command will also suggest possible transport and exchange reactions. The GapFill algorithm implemented in this command is a variant of the gap-filling procedure described in [Kumar07].

```
$ psamm-model gapfill
```

The command will first list the reactions in the model followed by the suggested reactions to add to the model in order to unblock the blocked compounds. If `--allow-bounds-expansion` is specified, the procedure may also suggest that existing model reactions have their flux bounds widened, e.g. making an existing irreversible reaction reversible. To unblock only specific compounds, use the `--compound` option:

```
$ psamm-model gapfill --compound leu-L[c] --compound ile-L[c]
```

In this example, the procedure will try to add reactions so that leucine (`leu-L`) and isoleucine (`ile-L`) in the `c` compartment can be produced. Multiple compounds can be unblocked at the same time and the list of compounds to unblock can optionally be specified as a file by prefixing the file name with `@`.

```
$ psamm-model gapfill --compound @list_of_compounds_to_unblock.tsv
```

The GapFind algorithm is defined in terms of a MILP problem and can therefore be computationally expensive to run for larger models.

The original GapFill algorithm uses a solution procedure which implicitly assumes that the model contains implicit sinks for all compounds. This means that even with the reactions proposed by GapFill the model may need to produce compounds that cannot be used anywhere. The implicit sinks can be disabled with the `--no-implicit-sinks` option.

5.15 FastGapFill (`fastgapfill`)

The FastGapFill algorithm tries to reconstruct a flux consistent model (i.e. a model where every reaction takes a non-zero flux for at least one solutions). This is done by extending the model with reactions from the reaction database and trying to find a minimal subset that is flux consistent. The solution is approximate [Thiele14].

The database reactions can be assigned a weight (or "cost") using the `--penalty` option. These weights are taken into account when determining the minimal solution.

```
$ psamm-model fastgapfill --penalty penalty.tsv
```

The penalty file provided should be a tab separated table that contains reaction IDs and assigned penalty values in two columns:

```
RXN1    10
RXN2    15
RXN3    20
....
```

In addition to setting penalty values directly through the `--penalty` argument, default penalties for all other database reactions can be set through the `--db-penalty` argument. Reactions that do not have penalty values explicitly set through the `--penalty` argument will be assigned this penalty value. Similarly, penalty values can be assigned for new exchange reactions and artificial transport reactions through the `--ex-penalty` and `--tp-penalty` arguments. An example using all three of these arguments can be seen here:

```
$ psamm-model fastgapfill --db-penalty 100 --tp-penalty 1000 --ex-penalty 150
```

5.16 Predict primary pairs (`primarypairs`)

This command is used to predict element-transferring reactant/product pairs in the reactions of the model. This can be used to determine the flow of elements through reactions. Two methods for predicting the pairs are available: *FindPrimaryPairs* (`fpp`) [Steffensen17] and MapMaker (`mapmaker`) [Tervo16]. The `--method` option can used to select which prediction method to use:

```
$ psamm-model primarypairs --method=fpp
```

The result is reported as a table of four columns. The first column is the reactions ID, the second and third columns contain the compound ID of the reactant and product. The fourth column contains the predicted transfer of elements.

The `primarypairs` command will run slowly on models that contain artificial reactions such as biomass reactions or condensed biosynthesis reactions. Because the reactant/product pair prediction in these reactions is not as biologically meaningful these reactions can be excluded through the `--exclude` option. This option can be used to either give reaction IDs to exclude or to give an input file containing a list of reactions IDs to exclude:

```
$ psamm-model primarypairs --exclude BiomassReaction
or
$ psamm-model primarypairs --exclude @./exclude.tsv
```

5.17 PSAMM-Vis (`vis`)

Models can be visualized through the use of *PSAMM-vis* as implemented in the `vis` command in *PSAMM*. This command can use the *FindPrimaryPairs* algorithm to help generate images of full models or subsets of models. The output of this command will consist of a graph file in the *dot* language, `reactions.dot`, and two files called `reactions.nodes.tsv` and `reactions.edges.tsv` that contain the network data in TSV format.

To run the `vis` command the following command can be used:

```
$ psamm-model vis
```

5.17.1 Basic Graph Generation

By default, the `vis` command uses the *FindPrimaryPairs* algorithm to simplify the graph that is produced. This algorithm runs much faster if certain types of artificial reactions are not considered when doing the reactant/product pair prediction. These reactions often represent Biomass production or condensed biosynthesis processes. To exclude these reactions the `vis` command can be run with the `--exclude` option followed by an input file that contains a list of reaction IDs:

```
$ psamm-model vis --exclude @{path to file}
```

Running this command, *PSAMM-vis* only produces three files described above. Graph image generating softwares can convert these files to actual images. If the program *Graphviz* is installed on the computer, then this program can be used within *PSAMM* to generate the network image directly. This can be done by adding the `--image` argument followed by any *Graphviz* supported image format:

```
$ psamm-model vis --image {format (pdf, eps, svg, etc.)}
```

In addition, biomass reaction defined in model.yaml will be excluded automatically.

While the `vis` function in *PSAMM* uses *FindPrimaryPairs* for graph simplification by default, the command is also able to run using no graph simplification (`no-fpp`).

This can be done through using the `--method` argument:

```
$ psamm-model vis --method no-fpp
```

The resulting graphs can be further simplified to only show element transfers that contain a specified element through the `--element` argument. When using this option any reactant/product pairs that do not transfer the specified element will not be shown on the graph. To use this option the following command can be used:

```
$ psamm-model vis --element {Atomic Symbol}
```

Additionally, the final graphs created through `vis` command can only show a specified subset of reactions from the larger model. This can be done using `--subset` argument to provide a file containing a single column list of either reaction IDs or metabolite IDs (but not the mix of reaction and compound IDs).

```
$ psamm-model vis --subset {path to file}
```

And example of this file would be:

```
rxn_1
rxn_2
rxn_4
```

Or:

```
cpd_1[c]
cpd_1[e]
cpd_2[c]
```

Further modification can be done to the graph image to selectively hide certain edges in the final graph. This can be used to hide edges between paris of metabolites that might have many connections in the final graph images. Typical examples of these pairs include ATP and ADP, NAD and NADH, etc. To use this option first a tab separated table containing the metabolite pairs to hide must be made:

```
atp[c]  adp[c]
h2o[c]  h[c]
nad[c]  nadh[c]
```

This file can then be used with the `vis` command through the `--hide-edges` argument:

```
$ psamm-model vis --hide-edges {path to edges file}
```

5.17.2 Graph Image Customization

By default, the reaction and metabolite nodes in a graph will only show the reaction or metabolite IDs, but the final graphs output by the command can be customized to include additional reaction metabolite information that is present in the model. This additional information will be shown directly on the reaction or metabolite nodes in the graph. This can be done through using the `--rxn-detail` and `--cpd-detail` options. These options can be used followed by a space separated list of properties to include. For example, the following command could be used to show additional information of reactions and compounds:

```
$ psamm-model vis --cpd-detail id formula charge \
  --rxn-detail id name equation
```

The reaction and metabolite nodes can be further customized by specifying the filling color of nodes. This can be done by providing a two-column file that contains reaction or metabolite IDs (with compartment) and hex color codes:

```
ACONTa  #c4a0ef
succ[c] #10ea88
FUM #c4a0ef
....
```

This file can be used to color the nodes on the graph through the `--color` option:

```
$ psamm-model vis --color {path to color table}
```

The graph image can be simplified through the use of the `--combine` option. By default, the combine level is 0. The graph generated from using combine level 0 will have one reaction node for each reactant product pair within a reaction. This can result in having many sets of substrates/reaction/product nodes within the graph image, depending on how many substrates and products are present in a metabolic reaction. Using the combine level 1 option will condense the reaction nodes down so that there is only one reaction node per reaction, with each reaction node having connections to all reactants and products of that reaction. The combine level 2 option will condense the graph in a different way. With this option the graph is condensed based on shared reactant/product pairs between different reactions. If two separate reactions contain a common reactant/product pair, such as ATP/ADP pair, then the nodes for those condensed into one combined node.

```
$ psamm-model vis --combine {0,1,2}
```

In some cases the exported image contains many small isolated components that may cause the image to be too wide and hard to view. The `--array` option can be used in this cases to get a better layout. This option is followed by an integer that is larger than 0, which indicates how many isolated components will be placed per row. The command looks like the following:

```
$ psamm-model vis --array {integer that is larger than 0}
```

Then the exported DOT file could be converted to network image through the command below:

```
$ neato -O -Tpdf -n2 {path to DOT file}
```

“pdf” in “-Tpdf” in the command above can be replaced by any other image format that supported by the *Graphviz* program.

You can also generate the network image through the *vis* command directly:

```
$ psamm-model vis --array {integer that is larger than 0} --image {format (pdf, eps,␣
↪svg, etc.)}
```

The final graph image can also be modified to show the reactions and metabolites in different compartments based on the compartment information provided in the model’s reactions. This can be done through using the

`--compartment` option:

```
$ psamm-model vis --compartment
```

Users can specify name of output through `--output` option. By default, output will be named "reactions.dot", "reactions.nodes.tsv", "reactions.edges.tsv":

```
$ psamm-model vis --output Ecolicore
```

The output files will be named "Ecolicore.dot", "Ecolicore.nodes.tsv", "Ecolicore.edges.tsv".

The image file produced from the `vis` will be automatically sized by the *Graphviz* programs used to generate the image file. If a specific size is desired the `--image-size` argument can be used to supple a width and height in inches that the final image file should be. For example, to generate a graph that will be made into a 5" width by 10" height image the following command can be used:

```
$ psamm-model vis --image {format (pdf, eps, svg, etc.)} --image-size 5 10
```

The FBA and FVA values of a model can be visualized to show the flow of metabolites through the model. The `--fba` or `--fva` option (both cannot be used together) followed by a file path will allow you to do this. The FBA file should be a tab-separated file with the reaction name and a flux value, and can be created using the *fba* command. The FBA option can be used as follows:

```
(psamm-env) $ psamm-model vis --fba {path to FBA file}
```

Meanwhile, the FVA file should be a tab-separated file with the reaction name, lower flux value, and upper flux value, and can be created using the *fva* command. For example, to generate a graph that uses FVA values, the following command can be used:

```
(psamm-env) $ psamm-model vis --fva {path to FVA file}
```

These fluxes can be visualized by adding the `--image` option. Reactions with a flux of zero will have a dotted edge while non-zero fluxes will be solid. The flux values are proportional to the thickness of the edge and may help highlight reactions that contribute the most to the objective.

5.18 SBML Export (`sbmlexport`)

Exports the model to the SBML file format. This command exports the model as an SBML level 3 file with flux bounds, objective and gene information encoded with Flux Balance Constraints version 2.

```
$ psamm-model sbmlexport model.xml
```

If the file name is omitted, the file contents will be output directly to the screen. Using the `--pretty` option makes the output formatted for readability.

5.19 Excel Export (`excelexport`)

Exports the model to the Excel file format.

```
$ psamm-model excelexport model.xls
```

5.20 Table Export (`tableexport`)

Exports the model to the tsv file format.

```
$ psamm-model tableexport reactions > model.tsv
```

5.21 Search (`search`)

This command can be used to search in a database for compounds or reactions. To search for a compound use

```
$ psamm-model search compound [...]
```

Use the `--name` option to search for a compound with a specific name or use the `--id` option to search for a compound with a specific identifier.

To search for a reaction use

```
$ psamm-model search reaction [...]
```

Use the `--id` option to search for a reaction with a specific identifier. The `--compound` option can be used to search for reactions that include a specific compound. If more than one compound identifier is given (comma-separated) this will find reactions that include all of the given compounds.

5.22 PSAMM-SBML-Model

PSAMM normally takes a model in the *YAML* format as input. To deal with models that are in the *SBML PSAMM* includes various programs that allow users to convert these models to the *YAML* format. One additional option for dealing with models in the *SBML* format is using the *psamm-sbml-model* function. This function can be used to run any command normally accessed through *psamm-model* but with an *SBML* model as the input. To use this command the *SBML* model file needs to be specified first followed by the commands:

```
$ psamm-sbml-model {model.xml} fba
```

5.23 Console (`console`)

This command will start a Python session where the model has been loaded into the corresponding Python object representation.

```
$ psamm-model console
```

5.24 Psammotate (`psammotate`)

Given a reciprocal best hits file, will generate a draft model based on an template based on gene associations provided by the template file/reference file gene mapping. Draft model will contain all relevant model components in yaml format.

```
$ psamm-model psammotate
```

To generate a draft model, a reciprocal best hits file must be specified that maps the draft model genes to a template model using the `--rbh` option. Within this file, you must specify the integer of the column that contains the target mapping and the column that contains the template mapping (both indexed from 1) using the `--target` and `--template` options, respectively.

```
$ psamm-model psammotate --rbh gene_mapping.tsv --template 1 --target 2
```

Typically, this program retains reactions that have no gene mappings; however, if you want to drop reactions that do not have gene associations, you must specify the `--ignore-na` option.

```
$ psamm-model psammotate --rbh gene_mapping.tsv --template 1 --target 2 --ignore-na
```

You can also specify an output directory for all of the yaml file output using the `--output` option.

```
$ psamm-model psammotate --rbh gene_mapping.tsv --template 1 --target 2 --output out
```

5.25 GIMME (`gimme`)

This command allows you to subset a metabolic model based on gene expression data. The expression data for filtering may be in any normalized format (TPM, RPK, etc.), but the threshold value supplied to gimme must be appropriate for the input data. Gimme functions through gene inactivation and will not “express” genes that do not meet the specified expression threshold. Expression thresholds can be specified using the `--expression-threshold` argument and a file that maps genes in the model to their expression can be provided using the option `--transcriptome-file`.

```
$ psamm-model gimme --transcriptome-file file.tsv --expression-threshold 1
```

The gimme command may also specify an argument that will retain any reactions required in order to maintain a specific biomass threshold. This threshold may be specified using the `--biomass-threshold` option.

```
$ psamm-model gimme --transcriptome-file file.tsv --expression-threshold 1 --biomass-
↪threshold 1
```

You can specify a directory to output the subset model that will create all yaml files for the new, subset, model in this directory. This location can be specified using the `--export-model` argument.

```
$ psamm-model --transcriptome-file file.tsv --expression-threshold 1 --export-model ./
↪gimme_out/
```

5.26 TMFA (`tmfa`)

This command can be used to run growth simulations using the TMFA algorithm as detailed in [Henry07]. This method simulates fluxes in a metabolic model while incorporating metabolite concentrations and thermodynamic constraints. This is done through the incorporation of the gibbs free energy equation along with additional constraints to link reactions fluxes to thermodynamic feasibility.

This simulation method requires multiple input files to be prepared beforehand. The following section is going to detail these input files and their formats.

5.26.1 TMFA Input Files

Gibbs Free Energy Files

The `tmfa` method in psamm requires standard gibbs free energy of reaction values to be supplied as an input. These values could be obtained from a database or predicted based on metabolite structures and reaction equations. These values must be in kilojoules per mol (kJ/mol). The input file is formatted as a three column, tab separated table, consisting of reaction IDs, standard gibbs free energy values, and gibbs free energy uncertainty values for the predicted values. Any reactions that do not have associated gibbs free energy predictions need to either be included in the lumped reactions or in the excluded reactions.

```
rxn1  -5.8  1.2
rxn2  12.1  2.5
rxn3  -0.1  0.2
....
```

Excluded Reactions File

Reactions that cannot be thermodynamically constrained need to be included in the list of excluded reactions. These reactions typically consist of reactions that are artificial (for example, biomass equations) or ones for which the gibbs free energy is unknown. This input file just consists of reaction IDs with each line containing just one ID. Note that exchange reactions will be automatically excluded and do not need to be included in this file.

```
biomass
rxn6
rxn7
....
```

Lumped Reactions File

The lumped reactions file is where lumped reactions are defined for the TMFA problem. These reactions are summary reactions that can be used to constrain groups of reactions, when some of the reactions in the groups have unknown gibbs free energy values. More details on this concept can be found in the original TMFA publication [Henry07]. This file consists of 3 columns where the first column is a lumped reaction ID, the second column consists of a comma separated list of reaction IDs and directions (indicated by a 1 for forward and -1 for reverse), and a lump reaction equation. Any Lumped reactions need to also have their gibbs free energy values assigned in the gibbs free energy input file. The reactions that are included in the lumped reactions do not need to be included in the excluded reactions file.

```
Lump1  GLYBt2r:-1,GLYBabc:1    atp[c] + h2o[c] <=> adp[c] + h[e] + pi[c]
Lump2  ADMDCr:1,SPMS:1,METAT:1 atp[c] + h2o[c] + met-L[c] + ptrc[c] <=> 5mta[c] +␣
↪co2[c] + pi[c] + ppi[c] + spmd[c]
```

Transporter Parameters

Gibbs free energy values for transporters have to also account for the transport of charge and pH across compartments in a model. To allow these calculations to be made each transporter in the model needs to have an associated gibbs free energy values defined in the gibbs free energy input file along with the counts of protons and charge transported across the membrane defined in the transport parameter input file. This file is formatted as three columns with the first being reaction ID for the transporter reactions, the second being the net charge transported from the outside compartment, and the third being the number of protons transported from out to the in compartment for that reaction.

As an example, if we had the following reaction equation where H is a proton with a +1 charge and X is a compound with 0 charge.

```
- id: transX
  equation: X[e] + H[e] <=> X[c] + H[c]
```

This equation would have a net charge transported of +1 and the net number of protons transported from out to in of 1. This would mean that the transporter parameter line for this reaction would be:

```
transX  1 1
```

Another case could be if the compounds were both charged. In this case the net charge needs to be used. For example, in the following reaction where the compound Y is now going to have a charge of -1.

```
- id: transY
  equation: Y[e] + H[e] <=> Y[c] + H[c]
```

The protons transported from out to in would still be 1, but since compound Y also has a charge in this case the net charge transported would be 0 (+1 for the proton and -1 for Y). This would have an input line of the following in the transport parameter file:

```
transY  0  1
```

If reactions involve antiport, where one compound goes in and one goes out, then this needs to be accounted for in these transporter values. The values are always calculated in the direction of out to in. So, if there was the export of 1 proton in the reaction equation, then the value for that reaction would be -1 protons transported from out to in.

Lastly many reactions involve other components to their equations related to energy costs for the reaction. For example, PTS transport reactions may also involve the conversion of PEP to Pyruvate. These other compounds are not considered in the transport parameter calculations, even if they do have a charge. The only metabolites that are considered are the ones that cross the membrane in the reaction equation.

Concentration Settings

The last input file that needs to be prepared is the concentrations file. This file is used to set the concentrations of any metabolites that in the model. By default, metabolites are allowed to vary in concentration between 0.02 M and 1e-6 M. This file can be used to designate any metabolites that will have non-default bounds in the model. These could include ones where the measured concentrations fall outside of the default range, or ones where the they have measured values that may help constrain the model. For example, metabolites that are provided in the media can have their concentrations set in this file. The file is set up as a 3 column, tab-separated, table where the first column is the metabolite ID (with compartment ID), the second column is the lower bound of the concentration, and the third column is the upper bound of the concentration. All concentrations are designated as molar concentrations.

```
cpd_a[e]    0.02  0.02
cpd_b[e]    0.1 0.1
cpd_e[e]    0.0004  0.0004
```

Configuration File

Because the `tmfa` function requires multiple input files to run, the command was organized to use a central configuration file. A template configuration file can be generated using the following *PSAMM* command:

```
$ psamm-model tmfa --config ./config.yaml util --generate-config
```

The configuration file contains the relative paths to the input files that are detailed above, as well as a few other TMFA specific parameters. Note: The relative paths in this file have to be set based on where you are running the command from, not based on where the configuration file is located. Additionally, any files which are not needed for a specific model can be left out of this configuration file.

```
deltaG: ./gibbs-with-uncertainty.tsv
exclude: ./exclude.tsv
transporters: ./transport-parameters.tsv
concentrations: ./concentrations.tsv
lumped_reactions: ./lumped-reactions.tsv
proton-in: h[c]
proton-out: h[e]
proton-other:
 - h[p]
 - h[mito]
water:
  - h2o[c]
  - h2o[e]
  - h2o[mito]
  - h2o[p]
```

In addition to the relative paths to the input files, the configuration file is also where metabolites that are considered special cases are designated. Specifically, water and protons are not considered as part of the Keq portion of the gibbs free energy calculations. Becuase of their special properties, they are considered separately from the other metabolites. Protons are also involved in the calculation of the gibbs free energy for transporter reactions. As such the proton ID for the internal compartment protons and external compartment protons need to be designated in this file as well.

5.26.2 TMFA Simulation Command Line Parameters

The `tmfa` command has 4 general settings that can be applied to any simulation. These settings are things that might be changed from simulation to simulation, so they are defined through command line parameters instead of in the configuration file.

Temperature

The temperature used in the calculation of the gibbs free energy of the reactions can be set through the `--temp` command line parameter. This temperature is provided in Celsius.

```
$ psamm-model tmfa --config ./config.yaml --temp 27 simulation
```

pH Settings

pH affects the transport into the cytosolic space in the model by adjusting the thermodynamic favorability of the transport reactions. The allowable pH range for the internal and external compartments can be set through the following command line parameters by providing the lower then upper bound for the pH.

```
$ $ psamm-model tmfa --config ./config.yaml --phin 6,7 --phout 6,8 simulation
```

Biomass Flux

The biomass flux can be fixed at a certain value in the simulation using the `--threshold` command line option. By default, the biomass is fixed at the maximum biomass value for the model.

```
$ psamm-model tmfa --config ./config.yaml --threshold 0.1 simulation
```

Reaction Reversibility

All of the reactions in a model can be allowed to be reversible, and have their actual reversibility in the simulation only determined by the thermodynamic constraints in the simulation. This follows a variation of TMFA as detailed in the paper [Hamilton13]. This can be enabled for a simulation using the `--hamilton` command line argument.

```
$ psamm-model tmfa --config ./config.yaml --hamilton simulation
```

Error Estimates

The gibbs free energy error estimates can be incorporated into the `tmfa` simulations by using the `--err` command line argument. This argument will allow for the gibbs free energy values to vary slightly based on the provided uncertainty estimates in the input file.

```
$ psamm-model tmfa --config ./config.yaml --err simulation
```

5.26.3 Running Simulations with TMFA

Variability Analysis

The default simulation method with TMFA calculates the lower and upper bounds for each variable in the TMFA problem. this produces a four-column output that gives the variable type, variable ID, lower bound, and upper bound. This variability analysis is done for the metabolite concentrations, reaction fluxes, reaction gibbs free energy values, and binary constraint variables for the reactions. This analysis can be run using the following command:

```
$ psamm-model tmfa --config ./config.yaml simulation
```

This will produce an output like this:

Single Solution Simulations

The TMFA simulations can also be run to produce single, arbitrary solutions instead of doing the variability analysis. This can be run to produce a single solution using just the TMFA constraints `fba`, by doing an additional L1 minimization of the fluxes `l1min`, or to produce a random solution from the solution space `random`. (NOTE: the random solution is experimental and is not guaranteed to be completely random).

```
$ psamm-model tmfa --config ./config.yaml simulation --single-solution fba

$ psamm-model tmfa --config ./config.yaml simulation --single-solution l1min

$ psamm-model tmfa --config ./config.yaml simulation --single-solution random
```

Randomsparse with TMFA Constraints

Random reaction or gene deletions can be performed while accounting for the thermodynamic constraints through using the `--randomsparse` or `--randomsparse_genes` command line arguments. These functions will perform random deletions of reactions or genes in the model in the same way as the PSAMM `randomsparse` command, but while accounting for the thermodynamic constraints in the model.

```
$ psamm-model tmfa --config ./config.yaml simulation --randomsparse

$ psamm-model tmfa --config ./config.yaml simulation --randomsparse_genes
```

5.26.4 Other TMFA Utility Functions

The `tmfa` utility functions include the `--generate-config` command , which was detailed above, and a testing command `--random-addtion` which can be run to test which thermodynamic constraints might be causing a model to fail to produce biomass. In some cases when the simulations settings are being set up or when the parameters are changed (for example, different temps or concentrations) the `tmfa` simulation might not produce biomass. Unfortunately, it can be difficult to isolate which constraints might cause this problem without extensive manual investigation. To help with this process the `--random-addition` utility function was developed. This function will randomly add reaction thermodynamic constraints to the model and test which constraints cause the model to fall below the set biomass threshold. This can be useful for identifying potentially problematic or over constrained reactions in the model.

```
$ psamm-model tmfa --config ./config.yaml util --random-addition
```

CHAPTER 6

Development

6.1 Test suite

The python modules have test suites that allows us to automatically test various aspects of the module implementation. It is important to make sure that all tests run without failure *before* committing changes to any of the modules. The test suite is run by changing to the project directory and running

```
$ ./setup.py test
```

To run the tests on all the supported Python platforms with additional tests for coding style (PEP8) and building documentation, use tox:

```
$ tox
```

When testing with tox, the local path for the Cplex python module must be provided in the environment variables `CPLEX_PYTHON3_PACKAGE` for Python 3. For example:

```
$ export CPLEX_PYTHON3_PACKAGE=/path/to/IBM/ILOG/CPLEX_StudioXXX/cplex/python/3.7/x86-
↪64_osx
$ tox -e py37-cplex
```

Similarly, the local path to the Gurobi package must be specified in the environment variable `GUROBI_PYTHON_PACKAGE`:

```
$ export GUROBI_PYTHON_PACKAGE=/Library/gurobi604/mac64
$ tox -e py37-gurobi
```

6.2 Adding new tests

Adding or improving tests for python modules is highly encouraged. A test suite for a new module should be created in `tests/test_<modulename>.py`. These test suites use the built-in `unittest` module.

6.3 Documentation tests

In addition, some modules have documentation that can be tested using the `doctest` module. These test suites should also run without failure before any commits. They can be run by specifying the particular module (e.g the `affine` module in `expression`) using

```
$ python -m psamm.expression.affine -v
```

CHAPTER 7

FAQ

When I run PSAMM it exits with the error "No solvers available". How can I fix this?

This means that PSAMM is searching for a linear programming solver but was not able to find one. This can occur even when the Cplex solver is installed because the Cplex Python-bindings have to be installed separately from the main Cplex package (see *Cplex*). Also, if using a *Virtualenv* with Python, the Cplex Python-bindings must be installed *in the Virtualenv*. The bindings will *not* be available in the Virtualenv if they are installed globally.

To check whether the Cplex Python-bindings are correctly installed use the following command:

```
(env) $ pip list
```

This will show a list of Python packages that are available. The package `cplex` will appear in this list if the Cplex Python-bindings are correctly installed. If the package does *not* appear in the list, follow the instuctions at *Cplex* to install the package.

CHAPTER 8

PSAMM API

8.1 `psamm.balancecheck` – check balance of charge and formula

psamm.balancecheck.**reaction_charge**(*reaction*, *compound_charge*)

Calculate the overall charge for the specified reaction.

Parameters

- **reaction** – *psamm.reaction.Reaction*.
- **compound_charge** – a map from each compound to charge values.

psamm.balancecheck.**charge_balance**(*model*)

Calculate the overall charge for all reactions in the model.

Yield (reaction, charge) pairs.

Parameters **model** – *psamm.datasource.native.NativeModel*.

psamm.balancecheck.**reaction_formula**(*reaction*, *compound_formula*)

Calculate formula compositions for both sides of the specified reaction.

If the compounds in the reaction all have formula, then calculate and return the chemical compositions for both sides, otherwise return *None*.

Parameters

- **reaction** – *psamm.reaction.Reaction*.
- **compound_formula** – a map from compound id to formula.

psamm.balancecheck.**formula_balance**(*model*)

Calculate formula compositions for each reaction.

Call *reaction_formula()* for each reaction. Yield (reaction, result) pairs, where result has two formula compositions or *None*.

Parameters **model** – *psamm.datasource.native.NativeModel*.

8.2 `psamm.bayesian` – Bayesian Matching Functions

Calculate model mapping likelihood with bayesian.

class `psamm.bayesian.`**`CompoundEntry`**(*id*, *name*, *formula*, *charge*, *kegg*, *cas*)

> **`cas`**
> Alias for field number 5
>
> **`charge`**
> Alias for field number 3
>
> **`formula`**
> Alias for field number 2
>
> **`id`**
> Alias for field number 0
>
> **`kegg`**
> Alias for field number 4
>
> **`name`**
> Alias for field number 1

class `psamm.bayesian.`**`ReactionEntry`**(*id*, *name*, *genes*, *equation*, *subsystem*, *ec*)

> **`ec`**
> Alias for field number 5
>
> **`equation`**
> Alias for field number 3
>
> **`genes`**
> Alias for field number 2
>
> **`id`**
> Alias for field number 0
>
> **`name`**
> Alias for field number 1
>
> **`subsystem`**
> Alias for field number 4

class `psamm.bayesian.`**`MappingModel`**(*model*)

> Generate the internal structure for model mapping.
>
> **Parameters** **`model`** – *`psamm.datasource.native.NativeModel`*
>
> **`print_summary`**()
> Print model summary
>
> **`check_reaction_compounds`**()
> Check that reaction compounds are defined in the model

class `psamm.bayesian.`**`BayesianCompoundPredictor`**(*model1*, *model2*, *nproc=1*, *outpath='.'*, *log=False*, *kegg=False*)

> Predict model compound mappings based on a Bayesian model.
>
> **Parameters**
>
> - **`model1`** – *`psamm.bayesian.MappingModel`*.

- **model2** – *psamm.bayesian.MappingModel*.
- **nproc** – number of processes used for mapping.
- **outpath** – the path to output the detailed log file.
- **log** – whether to output log file of the p_match and p_no_match.
- **kegg** – whether to compare the KEGG id.

get_raw_map()
Return pandas.DataFrame style of raw mapping table.

get_best_map(*threshold_compound=0*)
Return `pandas.DataFrame` style of best mapping for each query.

get_cpd_pred(*threshold_compound=0*)
Return the cpd_pred used for reaction mapping.

class psamm.bayesian.**BayesianReactionPredictor**(*model1*, *model2*, *cpd_pred*, *nproc=1*, *outpath='.'*, *log=False*, *gene=False*, *compartment_map={}*, *gene_map={}*)
Predict model reaction mappings based on a Bayesian model.

Parameters

- **model1** – *psamm.bayesian.MappingModel*.
- **model2** – *psamm.bayesian.MappingModel*.
- **cpd_pred** – `pandas.Series` with compound pairs as index, compound mapping score as value.
- **nproc** – number of processes used for mapping.
- **outpath** – the path to output the detailed log file.
- **log** – whether to output log file of the p_match and p_no_match.
- **gene** – whether to compare the gene association.
- **compartment_map** – dictionary mapping compartment id in the query model to the id in the target model.
- **gene_map** – dictionary mapping gene id in the query model to the id in the target model.

get_raw_map()
Return pandas.DataFrame style of raw mapping table.

get_best_map(*threshold_reaction=0*)
Return pandas.DataFrame style of best mapping for each query.

psamm.bayesian.**reaction_equation_mapping_approx_max_likelihood**(*cpd_set1*, *cpd_set2*, *cpd_map*, *cpd_score*, *compartment_map={}*)
Calculate equation likelihood based on compound mapping.

psamm.bayesian.**reaction_equation_compound_mapping_likelihood**(*r1*, *r2*, **args*, ***kwargs*)
Get the likelihood of reaction equations

Parameters **r2** (*r1,*) – two *RactionEntry* objects to be compared

args, kwargs:

cpd_map: dictionary mapping compound id in the query model to a set of best-mapping compound ids in the target model.

cpd_score: dictionary mapping compound id in the query model to its best mapping score during compound mapping.

compartment_map: dictionary mapping compartment id in the query model to the id in the target model.

`psamm.bayesian.`**`get_best_p_value_set`**(*r1*, *r2*, **args*, ***kwargs*)
Assume equations may have reversed direction, report best mapping p.

`psamm.bayesian.`**`merge_partial_p_set`**(*cpd_set1_left*, *cpd_set2_left*, *cpd_set1_right*, *cpd_set2_right*, **args*, ***kwargs*)
Merge the left hand side and right hand side p values together.

The compound mapping is done separately on left hand side and right hand side. Then the corresponding p_match and p_no_match are merged together.

`psamm.bayesian.`**`fake_likelihood`**(*e1*, *e2*)
Generate fake likelihood if corresponding mapping is not required.

`psamm.bayesian.`**`pairwise_likelihood`**(*pool*, *chunksize*, *model1*, *model2*, *likelihood*, **args*, ***kwargs*)
Compute likelihood of all pairwise comparisons.

Returns likelihoods as a dataframe with a column for each hypothesis.

`psamm.bayesian.`**`likelihood_products`**(*likelihood_dfs*)
Combine likelihood dataframes.

`psamm.bayesian.`**`bayes_posterior`**(*prior*, *likelihood_df*)
Calculate posterior given likelihoods and prior.

`psamm.bayesian.`**`map_model_compounds`**(*model1*, *model2*, *nproc=1*, *outpath='.'*, *log=False*, *kegg=False*)
Map compounds of two models.

`psamm.bayesian.`**`map_model_reactions`**(*model1*, *model2*, *cpd_map*, *cpd_score*, *nproc=1*, *outpath='.'*, *log=False*, *gene=False*, *compartment_map={}*, *gene_map={}*)
Map reactions of two models.

8.3 `psamm.bayesian_util` – Bayesian Matching Utility Functions

Utility functions.

`psamm.bayesian_util.`**`id_equals`**(*id1*, *id2*)
Return True if the two IDs are considered equal.

`psamm.bayesian_util.`**`name_equals`**(*name1*, *name2*)
Return True if the two names are considered equal.

`psamm.bayesian_util.`**`name_similar`**(*name1*, *name2*)
Return the possibility that two names are considered equal.

`psamm.bayesian_util.`**`formula_equals`**(*f1*, *f2*, *charge1*, *charge2*)
Return True if the two formulas are considered equal.

`psamm.bayesian_util.`**`genes_equals`**`(`*g1*, *g2*, *gene_map={}*`)`

Return True if the two gene association strings are considered equal.

Parameters

- **g2** (*g1,*) – gene association strings
- **gene_map** – a dict that maps gene ids in g1 to gene ids in g2 if they have different naming system

`psamm.bayesian_util.`**`formula_exact`**`(`*f1*, *f2*`)`

Return True if the two formulas are considered equal.

`psamm.bayesian_util.`**`pairwise_distance`**`(`*i1*, *i2*, *distance*, *threshold=None*`)`

Pairwise distances.

`psamm.bayesian_util.`**`levenshtein`**`(`*s1*, *s2*`)`

Edit distance function.

`psamm.bayesian_util.`**`jaccard`**`(`*s1*, *s2*`)`

Jaccard similarity function.

8.4 psamm.command – Command line interface

Command line interface.

Each command in the command line interface is implemented as a subclass of *`Command`*. Commands are also referenced from `setup.py` using the entry point mechanism which allows the commands to be automatically discovered.

The *`main()`* function is the entry point of command line interface.

`exception` `psamm.command.`**`CommandError`**

Error from running a command.

This should be raised from a `Command.run()` if any arguments are misspecified. When the command is run and the `CommandError` is raised, the caller will exit with an error code and print appropriate usage information.

`class` `psamm.command.`**`Command`**`(`*model*, *args*`)`

Represents a command in the interface, operating on a model.

The constructor will be given the NativeModel and the command line namespace. The subclass must implement *`run()`* to handle command execution. The doc string will be used as documentation for the command in the command line interface.

In addition, *`init_parser()`* can be implemented as a classmethod which will allow the command to initialize an instance of `argparse.ArgumentParser` as desired. The resulting argument namespace will be passed to the constructor.

`classmethod init_parser``(`*parser*`)`

Initialize command line parser (`argparse.ArgumentParser`)

`run``()`

Execute command

`argument_error``(`*msg*`)`

Raise error indicating error parsing an argument.

`fail``(`*msg*, *exc=None*`)`

Exit command as a result of a failure.

class psamm.command.**MetabolicMixin**(**args*, ***kwargs*)
Mixin for commands that use a metabolic model representation.

class psamm.command.**ObjectiveMixin**
Mixin for commands that use biomass as objective.

Allows the user to override the default objective from the command line.

class psamm.command.**LoopRemovalMixin**
Mixin for commands that perform loop removal.

class psamm.command.**SolverCommandMixin**(**args*, ***kwargs*)
Mixin for commands that use an LP solver.

This adds a `--solver` parameter to the command that the user can use to select a specific solver. It also adds the method `_get_solver()` which will return a solver with the specified default requirements. The user requirements will override the default requirements.

class psamm.command.**ParallelTaskMixin**
Mixin for commands that run parallel computation tasks.

class psamm.command.**FilePrefixAppendAction**(*option_strings*, *dest*, *nargs=None*, *fromfile_prefix_chars='@'*, ***kwargs*)
Action that appends one argument or multiple from a file.

If the argument starts with a character in `fromfile_prefix_chars` the remaining part of the argument is taken to be a file path. The file is read and every line is appended. Otherwise, the argument is simply appended.

exception psamm.command.**ExecutorError**
Error running tasks on executor.

psamm.command.**main**(*command_class=None*, *args=None*)
Run the command line interface with the given `Command`.

If no command class is specified the user will be able to select a specific command through the first command line argument. If the `args` are provided, these should be a list of strings that will be used instead of `sys.argv[1:]`. This is mostly useful for testing.

psamm.command.**main_sbml**(*command_class=None*, *args=None*)
Run the SBML command line interface.

8.5 `psamm.database` – Reaction database

Representation of metabolic network databases.

class psamm.database.**StoichiometricMatrixView**(*database*)
Provides a sparse matrix view on the stoichiometry of a database.

This object is used internally in the database to expose a sparse matrix view of the model stoichiometry. This class should not be instantied, instead use the `MetabolicDatabase.matrix` property. Any compound, reaction-pair can be looked up to obtain the corresponding stoichiometric value. If the value is not defined (implicitly zero) a `KeyError` will be raised.

In addition, instances also support the NumPy *__array__* protocol which means that a `numpy.array` can by created directly from the matrix.

```
>>> model = MetabolicModel()
>>> matrix = numpy.array(model.matrix)
```

class psamm.database.**MetabolicDatabase**
Database of metabolic reactions.

reactions
Iterator of reactions IDs in the database.

compounds
Itertor of *Compounds* in the database.

compartments
Iterator of compartment IDs in the database.

has_reaction(*reaction_id*)
Whether the given reaction exists in the database.

is_reversible(*reaction_id*)
Whether the given reaction is reversible.

get_reaction_values(*reaction_id*)
Return an iterator of reaction compounds and stoichiometric values.

The returned iterator contains (*Compound*, value)-tuples. The value is negative for left-hand side compounds and positive for right-hand side.

get_compound_reactions(*compound_id*)
Return an iterator of reactions containing the compound.

Reactions are returned as IDs.

reversible
The set of reversible reactions.

matrix
Mapping from compound, reaction to stoichiometric value.

This is an instance of *StoichiometricMatrixView*.

get_reaction(*reaction_id*)
Return reaction as a *Reaction*.

class psamm.database.**DictDatabase**
Metabolic database backed by in-memory dictionaries

This is a subclass of *MetabolicDatabase*.

reactions
Iterator of reactions IDs in the database.

compounds
Itertor of *Compounds* in the database.

compartments
Iterator of compartment IDs in the database.

has_reaction(*reaction_id*)
Whether the given reaction exists in the database.

is_reversible(*reaction_id*)
Whether the given reaction is reversible.

get_reaction_values(*reaction_id*)
Return an iterator of reaction compounds and stoichiometric values.

The returned iterator contains (*Compound*, value)-tuples. The value is negative for left-hand side compounds and positive for right-hand side.

get_compound_reactions(*compound_id*)
Return an iterator of reactions containing the compound.

Reactions are returned as IDs.

set_reaction(*reaction_id*, *reaction*)
Set the reaction ID to a reaction given by a `Reaction`

If an existing reaction exists with the given reaction ID it will be overwritten.

class psamm.database.**ChainedDatabase**(**databases*)
Links a number of databases so they can be treated a single database

This is a subclass of `MetabolicDatabase`.

reactions
Iterator of reactions IDs in the database.

compounds
Itertor of `Compounds` in the database.

compartments
Iterator of compartment IDs in the database.

has_reaction(*reaction_id*)
Whether the given reaction exists in the database.

is_reversible(*reaction_id*)
Whether the given reaction is reversible.

get_reaction_values(*reaction_id*)
Return an iterator of reaction compounds and stoichiometric values.

The returned iterator contains (`Compound`, value)-tuples. The value is negative for left-hand side compounds and positive for right-hand side.

get_compound_reactions(*compound*)
Return an iterator of reactions containing the compound.

Reactions are returned as IDs.

8.6 psamm.datasource.context – File system contexts

Utilities for keeping track of parsing context.

exception psamm.datasource.context.**ContextError**
Raised when a context failure occurs.

class psamm.datasource.context.**FilePathContext**(*arg*)
File context that keeps track of contextual information.

When a file is loaded, all files specified in that file must be loaded relative to the first file. This is made possible by keeping a context that remembers where a file was loaded so that other files can be loaded relatively.

class psamm.datasource.context.**FileMark**(*filecontext*, *line*, *column*)
Marks a position in a file.

This is used when parsing input files, to keep track of the position that generates an entry.

8.7 `psamm.datasource.entry` – Model entry representations

Representation of compound/reaction entries in models.

class `psamm.datasource.entry.`**`ModelEntry`**

Abstract model entry.

Provdides a base class for model entries which are representations of any entity (such as compound, reaction or compartment) in a model. An entity has an ID, and may have a name and filemark. The ID is a unique string identified within a model. The name is a string identifier for human consumption. The filemark indicates where the entry originates from (e.g. file name and line number). Any additional properties for an entity exist in `properties` which is any dict-like object mapping from string keys to any value type. The `name` entry in the dictionary corresponds to the name. Entries can be mutable, where the properties can be modified, or immutable, where the properties cannot be modified or where modifications are ignored. The ID is always immutable.

`id`

Identifier of entry.

`name`

Name of entry (or None).

`properties`

Properties of entry as a `Mapping` subclass (e.g. dict).

Note that the properties are not generally mutable but may be mutable for specific subclasses. If the `id` exists in this dictionary, it must never change the actual entry ID as obtained from the `id` property, even if other properties are mutable.

`filemark`

Position of entry in the source file (or None).

class `psamm.datasource.entry.`**`CompoundEntry`**

Abstract compound entry.

Entry subclass for representing compounds. This standardizes the properties `formula` and `charge`.

`formula`

Chemical formula of compound.

`charge`

Compound charge value.

class `psamm.datasource.entry.`**`ReactionEntry`**

Abstract reaction entry.

Entry subclass for representing compounds. This standardizes the properties `equation` and `genes`.

`equation`

Reaction equation.

`genes`

Gene association expression.

class `psamm.datasource.entry.`**`CompartmentEntry`**

Abstract compartment entry.

Entry subclass for representing compartments.

class `psamm.datasource.entry.`**`DictCompoundEntry`**`(*args, **kwargs)`

Compound entry backed by dictionary.

The given properties dictionary must contain a key `id` with the identifier.

Parameters

- **properties** – dict or *CompoundEntry* to construct from.
- **filemark** – Where the entry was parsed from (optional)

formula
Chemical formula of compound.

charge
Compound charge value.

class psamm.datasource.entry.**DictReactionEntry**(**args*, ***kwargs*)
Reaction entry backed by dictionary.

The given properties dictionary must contain a key `id` with the identifier.

Parameters

- **properties** – dict or *ReactionEntry* to construct from.
- **filemark** – Where the entry was parsed from (optional)

equation
Reaction equation.

genes
Gene association expression.

class psamm.datasource.entry.**DictCompartmentEntry**(**args*, ***kwargs*)
Compartment entry backed by dictionary.

The given properties dictionary must contain a key `id` with the identifier.

Parameters

- **properties** – dict or *CompartmentEntry* to construct from.
- **filemark** – Where the entry was parsed from (optional)

8.8 psamm.datasource.kegg – KEGG data parser

Module related to loading KEGG database files.

exception psamm.datasource.kegg.**ParseError**
Exception used to signal errors while parsing

class psamm.datasource.kegg.**KEGGEntry**(*properties*, *filemark=None*)
Base class for KEGG entry with raw values from KEGG.

class psamm.datasource.kegg.**CompoundEntry**(*entry*)
KEGG entry with mapped compound properties.

class psamm.datasource.kegg.**ReactionEntry**(*entry*)
KEGG entry with mapped reaction properties.

class psamm.datasource.kegg.**CompoundMapper**
Mapper for raw KEGG compound properties to standard properties.

Public methods are automatically translated into cached properties by the metaclass.

class psamm.datasource.kegg.**ReactionMapper**

Mapper for raw KEGG reaction properties to standard properties.

Methods are automatically translated into cached properties by the metaclass.

psamm.datasource.kegg.**parse_kegg_entries**(*f*, *context=None*)

Iterate over entries in KEGG file.

psamm.datasource.kegg.**parse_compound_file**(*f*, *context=None*)

Iterate over the compound entries in the given file.

psamm.datasource.kegg.**parse_reaction_file**(*f*, *context=None*)

Iterate over the reaction entries in the given file.

psamm.datasource.kegg.**parse_reaction**(*s*)

Parse a KEGG reaction string

8.9 psamm.datasource.modelseed – ModelSEED data parser

Module related to loading ModelSEED database files.

exception psamm.datasource.modelseed.**ParseError**

Exception used to signal errors while parsing

psamm.datasource.modelseed.**decode_name**(*s*)

Decode names in ModelSEED files

class psamm.datasource.modelseed.**CompoundEntry**(*id*, *names*, *formula*, *filemark=None*)

Representation of entry in a ModelSEED compound table

id

Identifier of entry.

name

Name of entry (or None).

formula

Chemical formula of compound.

properties

Properties of entry as a `Mapping` subclass (e.g. dict).

Note that the properties are not generally mutable but may be mutable for specific subclasses. If the `id` exists in this dictionary, it must never change the actual entry ID as obtained from the `id` property, even if other properties are mutable.

filemark

Position of entry in the source file (or None).

psamm.datasource.modelseed.**parse_compound_file**(*f*, *context=None*)

Iterate over the compound entries in the given file

8.10 psamm.datasource.native – Native data format parser

Module for reading and writing native formats.

These formats are either table-based or YAML-based. Table-based formats are space-separated and empty lines are ignored. Comments starting with pound (#). YAML-based formats are structured data following the YAML specification.

exception `psamm.datasource.native.`**`ParseError`**
Exception used to signal errors while parsing

`psamm.datasource.native.`**`float_constructor`**(*loader*, *node*)
Construct Decimal from YAML float encoding.

`psamm.datasource.native.`**`yaml_load`**(*stream*)
Load YAML file using safe loader.

class `psamm.datasource.native.`**`ModelReader`**(*model_from*, *context=None*)
Reader of native YAML-based model format.

The reader can be created from a model YAML file or directly from a dict, string or File-like object. Use *`reader_from_path()`* to read the model from a YAML file or directory and use the constructor to read from other sources. Any externally referenced file (with `include`) will be read on demand by the parse methods. To read the model fully into memory, use the *`create_model()`* to create a *`NativeModel`*.

classmethod reader_from_path(*path*)
Create a model from specified path.

Path can be a directory containing a `model.yaml` or `model.yml` file or it can be a path naming the central model file directly.

name
Name specified by the model.

biomass_reaction
Biomass reaction specified by the model.

extracellular_compartment
Extracellular compartment specified by the model.

Defaults to 'e'.

default_compartment
Default compartment specified by the model.

The compartment that is implied when not specified. In some contexts (e.g. for exchange compounds) the extracellular compartment may be implied instead. Defaults to 'c'.

default_flux_limit
Default flux limit specified by the model.

When flux limits on reactions are not specified, this value will be used. Flux limit of [0;x] will be implied for irreversible reactions and [-x;x] for reversible reactions, where x is this value. Defaults to 1000.

parse_compartments()
Parse compartment information from model.

Return tuple of: 1) iterator of *`psamm.datasource.entry.CompartmentEntry`*; 2) Set of pairs defining the compartment boundaries of the model.

parse_reactions()
Yield tuples of reaction ID and reactions defined in the model

has_model_definition()
Return True when the list of model reactions is set in the model.

parse_model()
Yield reaction IDs of model reactions

parse_limits()
Yield tuples of reaction ID, lower, and upper bound flux limits

parse_exchange()
Yield tuples of exchange compounds.

Each exchange compound is a tuple of compound, reaction ID, lower and upper flux limits.

parse_medium()
Yield tuples of exchange compounds.

Deprecated since version 0.28: Use *parse_exchange()* instead.

parse_compounds()
Yield CompoundEntries for defined compounds

create_model()
Return *NativeModel* fully loaded into memory.

class psamm.datasource.native.**NativeModel**(*properties={}*)
Represents model in the native format.

name
Return model name property.

version_string
Return model version string.

biomass_reaction
Return biomass reaction property.

extracellular_compartment
Return extracellular compartment property.

default_compartment
Return default compartment property.

default_flux_limit
Return default flux limit property.

compartments
Return compartments entry set.

compartment_boundaries
Return set of compartment boundaries.

reactions
Return reaction entry set.

compounds
Return compound entry set.

exchange
Return dict of exchange compounds and properties.

limits
Return dict of reaction limits.

model
Return dict of model reactions.

create_metabolic_model()
Create a *psamm.metabolicmodel.MetabolicModel*.

psamm.datasource.native.**parse_compound**(*compound_def, context=None*)
Parse a structured compound definition as obtained from a YAML file

Returns a CompoundEntry.

`psamm.datasource.native.`**`parse_compound_list`**(*path*, *compounds*)
Parse a structured list of compounds as obtained from a YAML file

Yields CompoundEntries. Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_compound_table_file`**(*path*, *f*)
Parse a tab-separated file containing compound IDs and properties

The compound properties are parsed according to the header which specifies which property is contained in each column.

`psamm.datasource.native.`**`parse_compound_yaml_file`**(*path*, *f*)
Parse a file as a YAML-format list of compounds

Path can be given as a string or a context.

`psamm.datasource.native.`**`resolve_format`**(*format*, *path*)
Looks at a file's extension and format (if any) and returns format.

`psamm.datasource.native.`**`parse_compound_file`**(*path*, *format*)
Open and parse reaction file based on file extension or given format

Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_reaction_equation_string`**(*equation*, *default_compartment*)
Parse a string representation of a reaction equation.

Converts undefined compartments to the default compartment.

`psamm.datasource.native.`**`parse_reaction_equation`**(*equation_def*, *default_compartment*)
Parse a structured reaction equation as obtained from a YAML file

Returns a Reaction.

`psamm.datasource.native.`**`parse_reaction`**(*reaction_def*, *default_compartment*, *context=None*)
Parse a structured reaction definition as obtained from a YAML file

Returns a ReactionEntry.

`psamm.datasource.native.`**`parse_reaction_list`**(*path*, *reactions*, *default_compartment=None*)
Parse a structured list of reactions as obtained from a YAML file

Yields tuples of reaction ID and reaction object. Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_reaction_yaml_file`**(*path*, *f*, *default_compartment*)
Parse a file as a YAML-format list of reactions

Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_reaction_table_file`**(*path*, *f*, *default_compartment*)
Parse a tab-separated file containing reaction IDs and properties

The reaction properties are parsed according to the header which specifies which property is contained in each column.

`psamm.datasource.native.`**`parse_reaction_file`**(*path*, *default_compartment=None*)
Open and parse reaction file based on file extension

Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_exchange`**(*exchange_def*, *default_compartment*)
Parse a structured exchange definition as obtained from a YAML file.

Returns in iterator of compound, reaction, lower and upper bounds.

`psamm.datasource.native.`**`parse_medium`**(*exchange_def*, *default_compartment*)
Parse a structured exchange definition as obtained from a YAML file.

Deprecated since version 0.28: Use `parse_exchange()` instead.

`psamm.datasource.native.`**`parse_exchange_list`**(*path*, *exchange*, *default_compartment*)
Parse a structured exchange list as obtained from a YAML file.

Yields tuples of compound, reaction ID, lower and upper flux bounds. Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_medium_list`**(*path*, *exchange*, *default_compartment*)
Parse a structured exchange list as obtained from a YAML file.

Deprecated since version 0.28: Use `parse_exchange_list()` instead.

`psamm.datasource.native.`**`parse_exchange_yaml_file`**(*path*, *f*, *default_compartment*)
Parse a file as a YAML-format exchange definition.

Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_medium_yaml_file`**(*path*, *f*, *default_compartment*)
Parse a file as a YAML-format exchange definition.

Deprecated since version 0.28: Use `parse_exchange_yaml_file()` instead.

`psamm.datasource.native.`**`parse_exchange_table_file`**(*f*)
Parse a space-separated file containing exchange compound flux limits.

The first two columns contain compound IDs and compartment while the third column contains the lower flux limits. The fourth column is optional and contains the upper flux limit.

`psamm.datasource.native.`**`parse_medium_table_file`**(*f*)
Parse a space-separated file containing exchange compound flux limits.

Deprecated since version 0.28: Use `parse_exchange_table_file()` instead.

`psamm.datasource.native.`**`parse_exchange_file`**(*path*, *default_compartment*)
Parse a file as a list of exchange compounds with flux limits.

The file format is detected and the file is parsed accordingly. Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_medium_file`**(*path*, *default_compartment*)
Parse a file as a list of exchange compounds with flux limits.

Deprecated since version 0.28: Use `parse_exchange_file()` instead.

`psamm.datasource.native.`**`parse_limit`**(*limit_def*)
Parse a structured flux limit definition as obtained from a YAML file

Returns a tuple of reaction, lower and upper bound.

`psamm.datasource.native.`**`parse_limits_list`**(*path*, *limits*)
Parse a structured list of flux limits as obtained from a YAML file

Yields tuples of reaction ID, lower and upper flux bounds. Path can be given as a string or a context.

`psamm.datasource.native.`**`parse_limits_table_file`**(*f*)
Parse a space-separated file containing reaction flux limits

The first column contains reaction IDs while the second column contains the lower flux limits. The third column is optional and contains the upper flux limit.

`psamm.datasource.native.`**`parse_limits_yaml_file`**(*path*, *f*)
Parse a file as a YAML-format flux limits definition

Path can be given as a string or a context.

psamm.datasource.native.**parse_limits_file**(*path*)
Parse a file as a list of reaction flux limits

The file format is detected and the file is parsed accordingly. Path can be given as a string or a context.

psamm.datasource.native.**parse_model_group**(*path*, *group*)
Parse a structured model group as obtained from a YAML file

Path can be given as a string or a context.

psamm.datasource.native.**parse_model_group_list**(*path*, *groups*)
Parse a structured list of model groups as obtained from a YAML file

Yields reaction IDs. Path can be given as a string or a context.

psamm.datasource.native.**parse_model_yaml_file**(*path*, *f*)
Parse a file as a YAML-format list of model reaction groups

Path can be given as a string or a context.

psamm.datasource.native.**parse_model_table_file**(*path*, *f*)
Parse a file as a list of model reactions

Yields reactions IDs. Path can be given as a string or a context.

psamm.datasource.native.**parse_model_file**(*path*)
Parse a file as a list of model reactions

The file format is detected and the file is parsed accordinly. The file is specified as a file path that will be opened for reading. Path can be given as a string or a context.

class psamm.datasource.native.**ModelWriter**
Writer for native (YAML) format.

convert_compartment_entry(*compartment*, *adjacencies*)
Convert compartment entry to YAML dict.

Parameters

- **compartment** – *psamm.datasource.entry.CompartmentEntry*.
- **adjacencies** – Sequence of IDs or a single ID of adjacent compartments (or None).

convert_compound_entry(*compound*)
Convert compound entry to YAML dict.

convert_reaction_entry(*reaction*)
Convert reaction entry to YAML dict.

write_compartments(*stream*, *compartments*, *adjacencies*, *properties=None*)
Write iterable of compartments as YAML object to stream.

Parameters

- **stream** – File-like object.
- **compartments** – Iterable of compartment entries.
- **adjacencies** – Dictionary mapping IDs to adjacent compartment IDs.
- **properties** – Set of compartment properties to output (or None to output all).

write_compounds(*stream*, *compounds*, *properties=None*)
Write iterable of compounds as YAML object to stream.

Parameters

- **`stream`** – File-like object.
- **`compounds`** – Iterable of compound entries.
- **`properties`** – Set of compound properties to output (or None to output all).

`write_reactions`(*stream*, *reactions*, *properties=None*)
Write iterable of reactions as YAML object to stream.

Parameters

- **`stream`** – File-like object.
- **`compounds`** – Iterable of reaction entries.
- **`properties`** – Set of reaction properties to output (or None to output all).

`psamm.datasource.native.`**`reaction_signature`**(*eq*, *direction=False*, *stoichiometry=False*)
Return unique signature object for `Reaction`.

Signature objects are hashable, and compare equal only if the reactions are considered the same according to the specified rules.

Parameters

- **`direction`** – Include reaction directionality when considering equality.
- **`stoichiometry`** – Include stoichiometry when considering equality.

8.11 `psamm.datasource.reaction` – Parser for reactions

Reaction parser.

`exception` `psamm.datasource.reaction.`**`ParseError`**(**args*, ***kwargs*)
Error raised when parsing reaction fails.

`class` `psamm.datasource.reaction.`**`ReactionParser`**(*arrows=None*, *parse_global=False*)
Parser of reactions equations.

This parser recognizes:

- Global compartment specification as a prefix (when `parse_global` is `True`)
- Configurable reaction arrow tokens (`arrows`)
- Compounds quoted by pipe (`|`) (required only if the compound name includes a space)
- Compound counts that are affine expressions.

`parse`(*s*)
Parse reaction string.

`psamm.datasource.reaction.`**`parse_reaction`**(*s*)
Parse reaction string using the default parser.

`psamm.datasource.reaction.`**`parse_compound`**(*s*, *global_compartment=None*)
Parse a compound specification.

If no compartment is specified in the string, the global compartment will be used.

`psamm.datasource.reaction.`**`parse_compound_count`**(*s*)
Parse a compound count (number of compounds).

8.12 `psamm.datasource.sbml` – SBML model parser

Parser for SBML model files.

exception `psamm.datasource.sbml.`**`ParseError`**

Error parsing SBML file

class `psamm.datasource.sbml.`**`SBMLSpeciesEntry`**(*reader*, *root*, *filemark=None*)

Species entry in the SBML file

`name`

Species name

`compartment`

Species compartment

`charge`

Species charge

`formula`

Species formula

`boundary`

Whether this compound is a boundary condition

`properties`

All species properties as a dict

`filemark`

Position of entry in the source file (or None).

class `psamm.datasource.sbml.`**`SBMLReactionEntry`**(*reader*, *root*, *filemark=None*)

Reaction entry in SBML file

`id`

Reaction ID

`name`

Reaction name

`reversible`

Whether the reaction is reversible

`equation`

Reaction equation is a *`Reaction`* object

`kinetic_law_reaction_parameters`

Iterator over the values of kinetic law reaction parameters

`properties`

All reaction properties as a dict

`filemark`

Position of entry in the source file (or None).

class `psamm.datasource.sbml.`**`SBMLCompartmentEntry`**(*reader*, *root*, *filemark=None*)

Compartment entry in the SBML file

`properties`

All compartment properties as a dict.

`filemark`

Position of entry in the source file (or None).

class `psamm.datasource.sbml.`**`SBMLObjectiveEntry`**(*reader*, *namespace*, *root*)

Flux objective defined with FBC

class `psamm.datasource.sbml.`**`SBMLFluxBoundEntry`**(*reader*, *namespace*, *root*)

Flux bound defined with FBC V1.

Flux bounds defined with FBC V2 are instead encoded as `upper_flux` and `lower_flux` properties on the ReactionEntry objects.

`id`

Return ID of flux bound.

`name`

Return name of flux bound.

`reaction`

Return reaction ID that the flux bound pertains to.

`operation`

Return the operation of the flux bound.

Returns one of LESS_EQUAL, GREATER_EQUAL or EQUAL.

`value`

Return the flux bound value.

class `psamm.datasource.sbml.`**`SBMLReader`**(*file*, *strict=False*, *ignore_boundary=True*, *context=None*)

Reader of SBML model files

The constructor takes a file-like object which will be parsed as XML and then as SBML according to the specification. If the `strict` parameter is set to False, the parser will revert to a more lenient parsing which is required for many older models. This tries to mimic the inconsistencies employed by COBRA when parsing models.

If `ignore_boundary` is `True`, the species that are marked as boundary conditions will simply be dropped from the species list and from the reaction equations, and any boundary compartment will be dropped too. Otherwise the boundary species will be retained. Retaining these is only useful to extract specific information from those species objects.

Parameters

- **`file`** – File-like object to parse XML SBML content from.
- **`strict`** – Indicating whether strict parsing is enabled.
- **`ignore_boundary`** – Indicating whether boundary species are dropped.
- **`context`** – Optional file parsing context from *`psamm.datasource.context`*.

`get_compartment`(*compartment_id*)

Return *`CompartmentEntry`* corresponding to id.

`get_reaction`(*reaction_id*)

Return *`SBMLReactionEntry`* corresponding to reaction_id

`get_species`(*species_id*)

Return *`SBMLSpeciesEntry`* corresponding to species_id

`get_objective`(*objective_id*)

Return *`SBMLObjectiveEntry`* corresponding to objective_id

`compartments`

Iterator over *`SBMLCompartmentEntry`* entries.

reactions
Iterator over *ReactionEntries*

species
Iterator over `SpeciesEntries`

This will not yield boundary condition species if those are ignored.

objectives
Iterator over *SBMLObjectiveEntry*

flux_bounds
Iterator over *SBMLFluxBoundEntry*

id
Model ID

name
Model name

create_model()
Create model from reader.

Returns *psamm.datasource.native.NativeModel*.

class psamm.datasource.sbml.**SBMLWriter**(*cobra_flux_bounds=False*)
Writer of SBML files.

write_model(*file*, *model*, *pretty=False*)
Write a given model to file.

Parameters

- **file** – File-like object open for writing.
- **model** – Instance of `NativeModel` to write.
- **pretty** – Whether to format the XML output for readability.

psamm.datasource.sbml.**convert_sbml_model**(*model*)
Convert raw SBML model to extended model.

Parameters **model** – `NativeModel` obtained from *SBMLReader*.

psamm.datasource.sbml.**entry_id_from_cobra_encoding**(*cobra_id*)
Convert COBRA-encoded ID string to decoded ID string.

psamm.datasource.sbml.**create_convert_sbml_id_function**(*compartment_prefix='C_', reaction_prefix='R_', compound_prefix='M_', decode_id=<function entry_id_from_cobra_encoding>*)
Create function for converting SBML IDs.

The returned function will strip prefixes, decode the ID using the provided function. These prefixes are common on IDs in SBML models because the IDs live in a global namespace.

psamm.datasource.sbml.**translate_sbml_compartment**(*entry*, *new_id*)
Translate SBML compartment entry.

psamm.datasource.sbml.**translate_sbml_reaction**(*entry*, *new_id*, *compartment_map*, *compound_map*)
Translate SBML reaction entry.

`psamm.datasource.sbml.`**`translate_sbml_compound`**`(`*entry*, *new_id*, *compartment_map*`)`

Translate SBML compound entry.

`psamm.datasource.sbml.`**`convert_model_entries`**`(`*model*, *convert_id=<function create_convert_sbml_id_function.<locals>.convert_sbml_id>*, *create_unique_id=None*, *translate_compartment=<function translate_sbml_compartment>*, *translate_reaction=<function translate_sbml_reaction>*, *translate_compound=<function translate_sbml_compound>*`)`

Convert and decode model entries.

Model entries are converted to new entries using the translate functions and IDs are converted using the given coversion function. If ID conversion would create a clash of IDs, the `create_unique_id` function is called with a container of current IDs and the base ID to generate a unique ID from. The translation functions take an existing entry and the new ID.

All references within the model are updated to use new IDs: compartment boundaries, limits, exchange, model, biomass reaction, etc.

Parameters **`model`** – `NativeModel`.

`psamm.datasource.sbml.`**`parse_xhtml_notes`**`(`*entry*`)`

Yield key, value pairs parsed from the XHTML notes section.

Each key, value pair must be defined in its own text block, e.g. `<p>key: value</p><p>key2: value2</p>`. The key and value must be separated by a colon. Whitespace is stripped from both key and value, and quotes are removed from values if present. The key is normalized by conversion to lower case and spaces replaced with underscores.

Parameters **`entry`** – `_SBMLEntry`.

`psamm.datasource.sbml.`**`parse_xhtml_species_notes`**`(`*entry*`)`

Return species properties defined in the XHTML notes.

Older SBML models often define additional properties in the XHTML notes section because structured methods for defining properties had not been developed. This will try to parse the following properties: `PUBCHEM ID`, `CHEBI ID`, `FORMULA`, `KEGG ID`, `CHARGE`.

Parameters **`entry`** – *`SBMLSpeciesEntry`*.

`psamm.datasource.sbml.`**`parse_xhtml_reaction_notes`**`(`*entry*`)`

Return reaction properties defined in the XHTML notes.

Older SBML models often define additional properties in the XHTML notes section because structured methods for defining properties had not been developed. This will try to parse the following properties: `SUBSYSTEM`, `GENE ASSOCIATION`, `EC NUMBER`, `AUTHORS`, `CONFIDENCE`.

Parameters **`entry`** – *`SBMLReactionEntry`*.

`psamm.datasource.sbml.`**`parse_objective_coefficient`**`(`*entry*`)`

Return objective value for reaction entry.

Detect objectives that are specified using the non-standardized kinetic law parameters which are used by many pre-FBC SBML models. The objective coefficient is returned for the given reaction, or None if undefined.

Parameters **`entry`** – *`SBMLReactionEntry`*.

`psamm.datasource.sbml.`**`parse_flux_bounds`**`(`*entry*`)`

Return flux bounds for reaction entry.

Detect flux bounds that are specified using the non-standardized kinetic law parameters which are used by many pre-FBC SBML models. The flux bounds are returned as a pair of lower, upper bounds. The returned bound is None if undefined.

Parameters **entry** – *SBMLReactionEntry*.

`psamm.datasource.sbml.`**`detect_extracellular_compartment`**(*model*)

Detect the identifier for equations with extracellular compartments.

Parameters **model** – `NativeModel`.

`psamm.datasource.sbml.`**`convert_exchange_to_compounds`**(*model*)

Convert exchange reactions in model to exchange compounds.

Only exchange reactions in the extracellular compartment are converted. The extracelluar compartment must be defined for the model.

Parameters **model** – `NativeModel`.

`psamm.datasource.sbml.`**`merge_equivalent_compounds`**(*model*)

Merge equivalent compounds in various compartments.

Tries to detect and merge compound entries that represent the same compound in different compartments. The entries are only merged if all properties are equivalent. Compound entries must have an ID with a suffix of an underscore followed by the compartment ID. This suffix will be stripped and compounds with identical IDs are merged if the properties are identical.

Parameters **model** – `NativeModel`.

8.13 `psamm.expression.affine` – Affine expressions

Representations of affine expressions and variables.

These classes can be used to represent affine expressions and do manipulation and evaluation with substitutions of particular variables.

class `psamm.expression.affine.`**`Variable`**(*symbol*)

Represents a variable in an expression

Equality of variables is based on the symbol.

`symbol`

Symbol of variable

```
>>> Variable('x').symbol
'x'
```

`simplify`()

Return simplified expression

The simplified form of a variable is always the variable itself.

```
>>> Variable('x').simplify()
Variable('x')
```

`substitute`(*mapping*)

Return expression with variables substituted

```
>>> Variable('x').substitute(lambda v: {'x': 567}.get(v.symbol, v))
567
>>> Variable('x').substitute(lambda v: {'y': 42}.get(v.symbol, v))
Variable('x')
>>> Variable('x').substitute(
...     lambda v: {'x': 123, 'y': 56}.get(v.symbol, v))
123
```

class psamm.expression.affine.**Expression**(**args*)

Represents an affine expression (e.g. 2x + 3y - z + 5)

simplify()

Return simplified expression.

If the expression is of the form 'x', the variable will be returned, and if the expression contains no variables, the offset will be returned as a number.

substitute(*mapping*)

Return expression with variables substituted

```
>>> Expression('x + 2y').substitute(
...     lambda v: {'y': -3}.get(v.symbol, v))
Expression('x - 6')
>>> Expression('x + 2y').substitute(
...     lambda v: {'y': Variable('z')}.get(v.symbol, v))
Expression('x + 2z')
```

variables()

Return iterator of variables in expression

8.14 psamm.expression.boolean – Boolean expressions

Representations of boolean expressions and variables.

These classes can be used to represent simple boolean expressions and do evaluation with substitutions of particular variables.

class psamm.expression.boolean.**Variable**(*symbol*)

Represents a variable in a boolean expression

class psamm.expression.boolean.**And**(**args*)

Represents an AND term in an expression.

class psamm.expression.boolean.**Or**(**args*)

Represents an OR term in an expression.

exception psamm.expression.boolean.**SubstitutionError**

Error substituting into expression.

class psamm.expression.boolean.**Expression**(*arg*, *_vars=None*)

Boolean expression representation.

The expression can be constructed from an expression string of variables, operators ("and", "or") and parenthesis groups. For example,

```
>>> e = Expression('a and (b or c)')
```

root
Return root term, variable or boolean of the expression.

variables
Immutable set of variables in the expression.

has_value()
Return True if the expression has no variables.

value
The value of the expression if fully evaluated.

substitute(*mapping*)
Substitute variables using mapping function.

exception psamm.expression.boolean.**ParseError**(**args*, ***kwargs*)
Signals error parsing boolean expression.

8.15 `psamm.fastcore` – Fastcore (approximate consistent subset)

Fastcore module implementing the fastcore algorithm

This is an implementation of the algorithms described in [Vlassis14]. Use the functions `fastcore()` and `fastcc()` to easily apply these algorithms to a `MetabolicModel`.

exception psamm.fastcore.**FastcoreError**
Indicates an error while running Fastcore

class psamm.fastcore.**FastcoreProblem**(**args*, ***kwargs*)
Represents a FastCore extension of a flux balance problem.

Accepts the same arguments as `FluxBalanceProblem`, and an additional `epsilon` keyword argument.

Parameters

- **model** – `MetabolicModel` to solve.
- **solver** – LP solver instance to use.
- **epsilon** – Flux threshold value.

lp7(*reaction_subset*)
Approximately maximize the number of reaction with flux.

This is similar to FBA but approximately maximizing the number of reactions in subset with flux > epsilon, instead of just maximizing the flux of one particular reaction. LP7 prefers "flux splitting" over "flux concentrating".

lp10(*subset_k*, *subset_p*, *weights={}*)
Force reactions in K above epsilon while minimizing support of P.

This program forces reactions in subset K to attain flux > epsilon while minimizing the sum of absolute flux values for reactions in subset P (L1-regularization).

find_sparse_mode(*core*, *additional*, *scaling*, *weights={}*)
Find a sparse mode containing reactions of the core subset.

Return an iterator of the support of a sparse mode that contains as many reactions from core as possible, and as few reactions from additional as possible (approximately). A dictionary of weights can be supplied which gives further penalties for including specific additional reactions.

`flip`(*reactions*)
Flip the specified reactions.

`is_flipped`(*reaction*)
Return true if reaction is flipped.

`psamm.fastcore.`**`fastcc`**(*model*, *epsilon*, *solver*)
Check consistency of model reactions.

Yield all reactions in the model that are not part of the consistent subset.

Parameters

- **`model`** – `MetabolicModel` to solve.
- **`epsilon`** – Flux threshold value.
- **`solver`** – LP solver instance to use.

`psamm.fastcore.`**`fastcc_is_consistent`**(*model*, *epsilon*, *solver*)
Quickly check whether model is consistent

Return true if the model is consistent. If it is only necessary to know whether a model is consistent, this function is fast as it will return the result as soon as it finds a single inconsistent reaction.

Parameters

- **`model`** – `MetabolicModel` to solve.
- **`epsilon`** – Flux threshold value.
- **`solver`** – LP solver instance to use.

`psamm.fastcore.`**`fastcc_consistent_subset`**(*model*, *epsilon*, *solver*)
Return consistent subset of model.

The largest consistent subset is returned as a set of reaction names.

Parameters

- **`model`** – `MetabolicModel` to solve.
- **`epsilon`** – Flux threshold value.
- **`solver`** – LP solver instance to use.

Returns Set of reaction IDs in the consistent reaction subset.

`psamm.fastcore.`**`fastcore`**(*model*, *core*, *epsilon*, *solver*, *scaling=100000.0*, *weights={}*)
Find a flux consistent subnetwork containing the core subset.

The result will contain the core subset and as few of the additional reactions as possible.

Parameters

- **`model`** – `MetabolicModel` to solve.
- **`core`** – Set of core reaction IDs.
- **`epsilon`** – Flux threshold value.
- **`solver`** – LP solver instance to use.
- **`scaling`** – Scaling value to apply (see [Vlassis14] for more information on this parameter).
- **`weights`** – Dictionary with reaction IDs as keys and values as weights. Weights specify the cost of adding a reaction to the consistent subnetwork. Default value is 1.

Returns Set of reaction IDs in the consistent reaction subset.

8.16 `psamm.fastgapfill` – FastGapFill algorithm

Implementation of fastGapFill.

Described in [Thiele14].

`psamm.fastgapfill.`**`fastgapfill`**`(`*model_extended*, *core*, *solver*, *weights={}*, *epsilon=1e-05*`)`

Run FastGapFill gap-filling algorithm by calling *`psamm.fastcore.fastcore()`*.

FastGapFill will try to find a minimum subset of reactions that includes the core reactions and it also has no blocked reactions. Return the set of reactions in the minimum subset. An extended model that includes artificial transport and exchange reactions can be generated by calling *`create_extended_model()`*.

Parameters

- **`model`** – *`psamm.metabolicmodel.MetabolicModel`*.
- **`core`** – reactions in the original metabolic model.
- **`weights`** – a weight dictionary for reactions in the model.
- **`solver`** – linear programming library to use.
- **`epsilon`** – float number, threshold for Fastcore algorithm.

8.17 `psamm.fluxanalysis` – Constraint-based reaction flux analysis

Implementation of Flux Balance Analysis.

`exception` `psamm.fluxanalysis.`**`FluxBalanceError`**`(`**args*, ***kwargs*`)`

Error indicating that a flux balance cannot be solved.

`class` `psamm.fluxanalysis.`**`FluxBalanceProblem`**`(`*model*, *solver*`)`

Model as a flux optimization problem with steady state assumption.

Create a representation of the model as an LP optimization problem with steady state assumption, i.e. the concentrations of compounds are always zero.

The problem can be modified and solved as many times as needed. The flux of a reaction can be obtained after solving using *`get_flux()`*.

Parameters

- **`model`** – `MetabolicModel` to solve.
- **`solver`** – LP solver instance to use.

`prob`

Return the underlying LP problem.

This can be used to add additional constraints on the problem. Calling solve on the underlying problem is not guaranteed to work correctly, instead use the methods on this object that solves the problem or make a subclass with a method that calls `_solve()`.

`add_thermodynamic``(`*em=1000*`)`

Apply thermodynamic constraints to the model.

Adding these constraints restricts the solution space to only contain solutions that have no internal loops [Schilling00]. This is solved as a MILP problem as described in [Muller13]. The time to solve a problem with thermodynamic constraints is usually much longer than a normal FBA problem.

The `em` parameter is the upper bound on the delta mu reaction variables. This parameter has to be balanced based on the model size since setting the value too low can result in the correct solutions being infeasible and setting the value too high can result in numerical instability which again makes the correct solutions infeasible. The default value should work in all cases as long as the model is not unusually large.

`maximize`(*reaction*)
Solve the model by maximizing the given reaction.

If reaction is a dictionary object, each entry is interpreted as a weight on the objective for that reaction (non-existent reaction will have zero weight).

`flux_bound`(*reaction*, *direction*)
Return the flux bound of the reaction.

Direction must be a positive number to obtain the upper bound or a negative number to obtain the lower bound. A value of inf or -inf is returned if the problem is unbounded.

`minimize_l1`(*weights={}*)
Solve the model by minimizing the L1 norm of the fluxes.

If the weights dictionary is given, the weighted L1 norm if minimized instead. The dictionary contains the weights of each reaction (default 1).

`max_min_l1`(*reaction*, *weights={}*)
Maximize flux of reaction then minimize the L1 norm.

During minimization the given reaction will be fixed at the maximum obtained from the first solution. If reaction is a dictionary object, each entry is interpreted as a weight on the objective for that reaction (non-existent reaction will have zero weight).

`check_constraints`()
Optimize without objective to check that solution is possible.

Raises *`FluxBalanceError`* if no flux solution is possible.

`get_flux_var`(*reaction*)
Get LP variable representing the reaction flux.

`flux_expr`(*reaction*)
Get LP expression representing the reaction flux.

`get_flux`(*reaction*)
Get resulting flux value for reaction.

`psamm.fluxanalysis.`**`flux_balance`**(*model*, *reaction*, *tfba*, *solver*)
Run flux balance analysis on the given model.

Yields the reaction id and flux value for each reaction in the model.

This is a convenience function for sertting up and running the FluxBalanceProblem. If the FBA is solved for more than one parameter it is recommended to setup and reuse the FluxBalanceProblem manually for a speed up.

This is an implementation of flux balance analysis (FBA) as described in [Orth10] and [Fell86].

Parameters

- **`model`** – MetabolicModel to solve.
- **`reaction`** – Reaction to maximize. If a dict is given, this instead represents the objective function weights on each reaction.
- **`tfba`** – If True enable thermodynamic constraints.
- **`solver`** – LP solver instance to use.

Returns Iterator over reaction ID and reaction flux pairs.

`psamm.fluxanalysis.`**`flux_variability`**(*model*, *reactions*, *fixed*, *tfba*, *solver*)
Find the variability of each reaction while fixing certain fluxes.

Yields the reaction id, and a tuple of minimum and maximum value for each of the given reactions. The fixed reactions are given in a dictionary as a reaction id to value mapping.

This is an implementation of flux variability analysis (FVA) as described in [Mahadevan03].

Parameters

- **`model`** – MetabolicModel to solve.
- **`reactions`** – Reactions on which to report variablity.
- **`fixed`** – dict of additional lower bounds on reaction fluxes.
- **`tfba`** – If True enable thermodynamic constraints.
- **`solver`** – LP solver instance to use.

Returns Iterator over pairs of reaction ID and bounds. Bounds are returned as pairs of lower and upper values.

`psamm.fluxanalysis.`**`flux_minimization`**(*model*, *fixed*, *solver*, *weights={}*)
Minimize flux of all reactions while keeping certain fluxes fixed.

The fixed reactions are given in a dictionary as reaction id to value mapping. The weighted L1-norm of the fluxes is minimized.

Parameters

- **`model`** – MetabolicModel to solve.
- **`fixed`** – dict of additional lower bounds on reaction fluxes.
- **`solver`** – LP solver instance to use.
- **`weights`** – dict of weights on the L1-norm terms.

Returns An iterator of reaction ID and reaction flux pairs.

`psamm.fluxanalysis.`**`flux_randomization`**(*model*, *threshold*, *tfba*, *solver*)
Find a random flux solution on the boundary of the solution space.

The reactions in the threshold dictionary are constrained with the associated lower bound.

Parameters

- **`model`** – MetabolicModel to solve.
- **`threshold`** – dict of additional lower bounds on reaction fluxes.
- **`tfba`** – If True enable thermodynamic constraints.
- **`solver`** – LP solver instance to use.

Returns An iterator of reaction ID and reaction flux pairs.

`psamm.fluxanalysis.`**`consistency_check`**(*model*, *subset*, *epsilon*, *tfba*, *solver*)
Check that reaction subset of model is consistent using FBA.

Yields all reactions that are *not* flux consistent. A reaction is consistent if there is at least one flux solution to the model that both respects the model constraints and also allows the reaction in question to have non-zero flux.

This can be determined by running FBA on each reaction in turn and checking whether the flux in the solution is non-zero. Since FBA only tries to maximize the flux (and the flux can be negative for reversible reactions), we have to try to both maximize and minimize the flux. An optimization to this method is implemented such that if checking one reaction results in flux in another unchecked reaction, that reaction will immediately be marked flux consistent.

Parameters

- **`model`** – MetabolicModel to check for consistency.
- **`subset`** – Subset of model reactions to check.
- **`epsilon`** – The threshold at which the flux is considered non-zero.
- **`tfba`** – If True enable thermodynamic constraints.
- **`solver`** – LP solver instance to use.

Returns An iterator of flux inconsistent reactions in the subset.

8.18 `psamm.fluxcoupling` – Flux coupling analysis

Flux coupling analysis

Described in [Burgard04].

class `psamm.fluxcoupling.`**`CouplingClass`**

Enumeration of coupling types.

`Inconsistent = 0`

Reaction is flux inconsistent (v1 is always zero).

`Uncoupled = 1`

Uncoupled reactions.

`DirectionalForward = 2`

Directionally coupled from reaction 1 to 2.

`DirectionalReverse = 3`

Directionally coupled from reaction 2 to 1.

`Partial = 4`

Partially coupled reactions.

`Full = 5`

Fully coupled reactions.

class `psamm.fluxcoupling.`**`FluxCouplingProblem`**(*model*, *bounded*, *solver*)

A specific flux coupling analysis applied to a metabolic model.

Parameters

- **`model`** – MetabolicModel to apply the flux coupling problem to
- **`bounded`** – Dictionary of reactions with minimum flux values
- **`solver`** – LP solver instance to use.

`solve`(*reaction_1*, *reaction_2*)

Return the flux coupling between two reactions

The flux coupling is returned as a tuple indicating the minimum and maximum value of the v1/v2 reaction flux ratio. A value of None as either the minimum or maximum indicates that the interval is unbounded in that direction.

psamm.fluxcoupling.**classify_coupling**(*coupling*)

Return a constant indicating the type of coupling.

Depending on the type of coupling, one of the constants from `CouplingClass` is returned.

Parameters **coupling** – Tuple of minimum and maximum flux ratio

8.19 psamm.formula – Chemical compound formula

Parser and representation of chemical formulas.

Chemical formulas (`Formula`) are represented as a number of `FormulaElements` with associated counts. A `Formula` is itself a `FormulaElement` so a formula can contain subformulas. This allows some simple structure to be represented.

class psamm.formula.**FormulaElement**

Base class representing elements of a formula

repeat(*count*)

Repeat formula element by creating a subformula

variables()

Iterator over variables in formula element

substitute(*mapping*)

Return formula element with substitutions performed

class psamm.formula.**Atom**(*symbol*)

Represent an atom in a chemical formula

```
>>> hydrogen = Atom.H
>>> oxygen = Atom.O
>>> str(oxygen | 2*hydrogen)
'H2O'
```

symbol

Atom symbol

```
>>> Atom.H.symbol
'H'
```

class psamm.formula.**Radical**(*symbol*)

Represents a radical or other unknown subformula

symbol

Radical symbol

```
>>> Radical('R1').symbol
'R1'
```

class psamm.formula.**Formula**(*values={}*)

Representation of a chemial formula

This is represented as a number of `FormulaElements` with associated counts.

```
>>> f = Formula({Atom.C: 6, Atom.H: 12, Atom.O: 6})
>>> str(f)
'C6H12O6'
```

substitute(*mapping*)
Return formula element with substitutions performed

flattened()
Return formula where subformulas have been flattened

```
>>> str(Formula.parse('(CH2)(CH2)2').flattened())
'C3H6'
```

variables()
Iterator over variables in formula element

items()
Iterate over (*FormulaElement*, value)-pairs

get(*element*, *default=None*)
Return value for element or default if not in the formula.

classmethod parse(*s*)
Parse a formula string (e.g. C6H10O2).

classmethod balance(*lhs*, *rhs*)
Return formulas that need to be added to balance given formulas

Given complete formulas for right side and left side of a reaction, calculate formulas for the missing compounds on both sides. Return as a left, right tuple. Formulas can be flattened before balancing to disregard grouping structure.

exception psamm.formula.**ParseError**(**args*, ***kwargs*)
Signals error parsing formula.

8.20 psamm.gapfill – GapFind/GapFill

Identify blocked metabolites and possible reconstructions.

This implements a variant of the algorithms described in [Kumar07].

exception psamm.gapfill.**GapFillError**
Indicates an error while running GapFind/GapFill

psamm.gapfill.**gapfind**(*model*, *solver*, *epsilon=0.001*, *v_max=1000*, *implicit_sinks=True*)
Identify compounds in the model that cannot be produced.

Yields all compounds that cannot be produced. This method assumes implicit sinks for all compounds in the model so the only factor that influences whether a compound can be produced is the presence of the compounds needed to produce it.

Epsilon indicates the threshold amount of reaction flux for the products to be considered non-blocked. V_max indicates the maximum flux.

This method is implemented as a MILP-program. Therefore it may not be efficient for larger models.

Parameters

- **model** – MetabolicModel containing core reactions and reactions that can be added for gap-filling.
- **solver** – MILP solver instance.
- **epsilon** – Threshold amount of a compound produced for it to not be considered blocked.
- **v_max** – Maximum flux.

- **implicit_sinks** – Whether implicit sinks for all compounds are included when gap-filling (traditional GapFill uses implicit sinks).

psamm.gapfill.**gapfill**(*model*, *core*, *blocked*, *exclude*, *solver*, *epsilon=0.001*, *v_max=1000*, *weights={}*, *implicit_sinks=True*, *allow_bounds_expansion=False*)

Find a set of reactions to add such that no compounds are blocked.

Returns two iterators: first an iterator of reactions not in core, that were added to resolve the model. Second, an iterator of reactions in core that had flux bounds expanded (i.e. irreversible reactions become reversible). Similarly to GapFind, this method assumes, by default, implicit sinks for all compounds in the model so the only factor that influences whether a compound can be produced is the presence of the compounds needed to produce it. This means that the resulting model will not necessarily be flux consistent.

This method is implemented as a MILP-program. Therefore it may not be efficient for larger models.

Parameters

- **model** – `MetabolicModel` containing core reactions and reactions that can be added for gap-filling.
- **core** – The set of core (already present) reactions in the model.
- **blocked** – The compounds to unblock.
- **exclude** – Set of reactions in core to be excluded from gap-filling (e.g. biomass reaction).
- **solver** – MILP solver instance.
- **epsilon** – Threshold amount of a compound produced for it to not be considered blocked.
- **v_max** – Maximum flux.
- **weights** – Dictionary of weights for reactions. Weight is the penalty score for adding the reaction (non-core reactions) or expanding the flux bounds (all reactions).
- **implicit_sinks** – Whether implicit sinks for all compounds are included when gap-filling (traditional GapFill uses implicit sinks).
- **allow_bounds_expansion** – Allow flux bounds to be expanded at the cost of a penalty which can be specified using weights (traditional GapFill does not allow this). This includes turning irreversible reactions reversible.

8.21 `psamm.gapfilling` – Gap-filling functions

Functionality related to gap-filling in general.

This module contains some general functions for preparing models for gap-filling. Specific gap-filling methods are implemented in the `gapfill` and `fastgapfill` modules.

psamm.gapfilling.**add_all_database_reactions**(*model*, *compartments*)

Add all reactions from database that occur in given compartments.

Parameters model – *psamm.metabolicmodel.MetabolicModel*.

psamm.gapfilling.**add_all_exchange_reactions**(*model*, *compartment*, *allow_duplicates=False*)

Add all exchange reactions to database and to model.

Parameters model – *psamm.metabolicmodel.MetabolicModel*.

psamm.gapfilling.**add_all_transport_reactions**(*model*, *boundaries*, *allow_duplicates=False*)

Add all transport reactions to database and to model.

Add transport reactions for all boundaries. Boundaries are defined by pairs (2-tuples) of compartment IDs. Transport reactions are added for all compounds in the model, not just for compounds in the two boundary compartments.

Parameters

- **`model`** – *`psamm.metabolicmodel.MetabolicModel`*.
- **`boundaries`** – Set of compartment boundary pairs.

Returns Set of IDs of reactions that were added.

`psamm.gapfilling.`**`create_extended_model`**(*model*, *db_penalty=None*, *ex_penalty=None*, *tp_penalty=None*, *penalties=None*)

Create an extended model for gap-filling.

Create a *`psamm.metabolicmodel.MetabolicModel`* with all reactions added (the reaction database in the model is taken to be the universal database) and also with artificial exchange and transport reactions added. Return the extended *`psamm.metabolicmodel.MetabolicModel`* and a weight dictionary for added reactions in that model.

Parameters

- **`model`** – *`psamm.datasource.native.NativeModel`*.
- **`db_penalty`** – penalty score for database reactions, default is *None*.
- **`ex_penalty`** – penalty score for exchange reactions, default is *None*.
- **`tb_penalty`** – penalty score for transport reactions, default is *None*.
- **`penalties`** – a dictionary of penalty scores for database reactions.

8.22 `psamm.graph` – Graph Related Functions

`class` `psamm.graph.`**`Entity`**(*props={}*)

Base class for graph entities.

`class` `psamm.graph.`**`Graph`**(*props={}*)

Graph entity representing a collection of nodes and edges.

`add_node`(*node*)

add node to a Graph entity. node: Node entity.

`get_node`(*node_id*)

get Node object. :param node_id: text_type(compound object) or text_type(a string that

contains single or multiple reaction IDs).

`write_graphviz`(*f*, *width*, *height*)

Write the nodes and edges information into a dot file.

Print a given graph object to a graph file in the dot format. This graph can then be converted to an image file using the graphviz program.

Parameters

- **`self`** – Graph entity, including nodes and edges entities.
- **`f`** – An empty file.
- **`width`** – Width of final metabolic map.
- **`height`** – Height of final metabolic map.

write_graphviz_compartmentalized (*f*, *compartment_tree*, *extracellular*, *width*, *height*)
Function to write compartmentalized version of dot file for graph.

Print a given graph object to a graph file in the dot format. In this graph, reaction nodes will be separated into different areas in the visualization based on the defined cellular compartments in the GEM.

Parameters

- **self** – Graph entity.
- **f** – An empty file.
- **compartment_tree** – a defaultdict of set, each element represents a compartment and its adjacent compartments.
- **extracellular** – the extracellular compartment in the model
- **width** – Width of final metabolic map.
- **height** – Height of final metabolic map..

write_nodes_tables (*f*)
write a table file (.tsv) that contains nodes information.

Write all node information from a given graph object into a tab separated table. This table will include IDs, shapes, fill colors, and labels. This table can be used to import the graph information to other programs.

Parameters

- **self** – Graph entity.
- **f** – An empty file.

write_edges_tables (*f*)
Write a tab separated table that contains edges information, including edge source, edge dest, and edge properties.

Write all edge information from a given graph object into a tab separated table. This table will include IDs, source nodes and destination nodes. This table can be used to import the graph information to other programs.

Parameters

- **self** – Graph entity.
- **f** – An empty TSV file.

class psamm.graph.**Node** (*props={}*)
Node entity represents a vertex in the graph.

class psamm.graph.**Edge** (*source*, *dest*, *props={}*)
Edge entity represents a connection between nodes.

psamm.graph.**get_compound_dict** (*model*)
Parse through model compounds and return a formula dict.

This function will parse the compound information in a given model and return a dictionary of compound IDs to compound formula objects.

Parameters **model** – <class 'psamm.datasource.native.NativeModel'>

psamm.graph.**make_network_dict** (*nm*, *mm*, *subset=None*, *method='fpp'*, *element=None*, *excluded_reactions=[]*, *reaction_dict={}*, *analysis=None*)
Create a dictionary of reactant/product pairs to reaction directions.

Returns a dictionary that connects predicted reactant/product pairs to their original reaction directions. This dictionary is used when generating bipartite graph objects. This can be done either using the FindPrimaryPairs method to predict reactant/product pairs or by mapping all possible pairs.

Parameters

- **nm** – <class 'psamm.datasource.native.NativeModel'>, the native model
- **mm** – <class 'psamm.metabolicmodel.MetabolicModel'>, the metabolic model.
- **subset** – None or path to a file that contains a list of reactions or compound ids. By default, it is None. It defines which reaction need to be visualized.
- **method** – 'fpp' or 'no-fpp', the method used for visualization.
- **element** – Symbol of chemical atom, such as 'C' ('C' indicates carbon).
- **excluded_reactions** – a list that contains reactions excluded from visualization.
- **reaction_dict** – dictionary of FBA or FVA results. By default it is an empty dictionary.
- **analysis** – "None" type or a string indicates if FBA or FVA file is given in command line.

psamm.graph.**write_network_dict**(*network_dict*)

Print out network dictionary object to a tab separated table.

Print information from the dictionary "network_dict". This information includes four items: reaction ID, reactant, product, and direction. this can table can be used as an input to other graph visualization and analysis software.

Args: network_dict: Dictionary object from make_network_dict()

psamm.graph.**make_cpair_dict**(*filter_dict*, *args_method*, *args_combine*, *style_flux_dict*, *hide_edges=[]*)

Create a mapping from compound pair to a defaultdict containing lists of reactions for the forward, reverse, and both directions.

Returns a dictionary that connects reactant/product pair to all reactions that contain this pair. Those reactions are stored in a dictionary and classified by reaction direction. For example: {(c1, c2): {'forward': [rxn1], 'back': [rxn3], 'both': [rxn4, rxn5]}, (c3, c4): {. . . }, . . . }

Parameters

- **filter_dict** – A dictionary mapping reaction entry to compound pairs (inside of the pairs there are cpd objects, not cpd IDs)
- **args_method** – a string, including 'fpp' and 'no-fpp'.
- **args_combine** – combine level, default = 0, optional: 1 and 2.
- **style_flux_dict** – a dictionary to set the edge style when fba or fva input is given.
- **hide_edges** – to determine edges between which compound pair need to be hidden.

psamm.graph.**make_bipartite_graph_object**(*cpairs_dict*, *new_id_mapping*, *method*, *args_combine*, *model_compound_entries*, *new_style_flux_dict*, *analysis=None*)

Makes a bipartite graph object from a cpair_dict object.

Start from empty graph() and cpair dict to make a graph object. Nodes only have rxn/cpd ID and rxn/cpd entry info in this initial graph. The information can be modified to add on other properties like color, or names.

Parameters

- **cpairs_dict** – defaultdict of compound_pair: defaultdict of direction: reaction list. e.g. {(c1, c2): {'forward":[rx1], 'both':[rx2]}}.
- **new_id_mapping** – dictionary of rxn_id_suffix: rxn_id.
- **method** – options=['fpp', 'no-fpp', file_path].
- **args_combine** – Command line argument, could be 0, 1, 2.
- **model_compound_entries** – dict of cpd_id:compound_entry.
- **new_style_flux_dict** – a dictionary to determine the edge style with the new reaction IDs.

return: A graph object that contains basic nodes and edges. only ID and rxn/cpd entry are in node properties, no features like color, shape.

psamm.graph.**dir_value**(*direction*)
Assign value to different reaction directions

8.23 `psamm.importer` – Import Functions

Entry points and functionality for importing models.

This module contains entry points for the psamm-import programs and functionality to assist in importing external file formats, converting them into proper native models and writing those models to files.

exception psamm.importer.**ImportError**
Exception used to signal a general import error.

exception psamm.importer.**ModelLoadError**
Exception used to signal an error loading the model files.

exception psamm.importer.**ParseError**
Exception used to signal an error parsing the model files.

class psamm.importer.**Importer**
Base importer class.

psamm.importer.**get_default_compartment**(*model*)
Return what the default compartment should be set to.

If some compounds have no compartment, unique compartment name is returned to avoid collisions.

psamm.importer.**detect_best_flux_limit**(*model*)
Detect the best default flux limit to use for model output.

The default flux limit does not change the model but selecting a good value reduced the amount of output produced and reduces clutter in the output files.

psamm.importer.**reactions_to_files**(*model*, *dest*, *writer*, *split_subsystem*)
Turn the reaction subsystems into their own files.

If a subsystem has a number of reactions over the threshold, it gets its own YAML file. All other reactions, those that don't have a subsystem or are in a subsystem that falls below the threshold, get added to a common reaction file.

Parameters

- **model** – `psamm_import.model.MetabolicModel`.
- **dest** – output path for model files.

- **writer** – *psamm.datasource.native.ModelWriter*.
- **split_subsystem** – Divide reactions into multiple files by subsystem.

psamm.importer.**model_exchange**(*model*)
Return exchange definition as YAML dict.

psamm.importer.**model_reaction_limits**(*model*)
Yield model reaction limits as YAML dicts.

psamm.importer.**infer_compartment_entries**(*model*)
Infer compartment entries for model based on reaction compounds.

psamm.importer.**infer_compartment_adjacency**(*model*)
Infer compartment adjacency for model based on reactions.

psamm.importer.**count_genes**(*model*)
Count the number of distinct genes in model reactions.

psamm.importer.**write_yaml_model**(*model*, *dest='.'*, *convert_exchange=True*, *split_subsystem=True*)
Write the given NativeModel to YAML files in dest folder.

The parameter `convert_exchange` indicates whether the exchange reactions should be converted automatically to an exchange file.

psamm.importer.**main**(*importer_class=None*, *args=None*)
Entry point for import program.

If the `args` are provided, these should be a list of strings that will be used instead of `sys.argv[1:]`. This is mostly useful for testing.

psamm.importer.**main_bigg**(*args=None*, *urlopen=<function urlopen>*)
Entry point for BiGG import program.

If the `args` are provided, these should be a list of strings that will be used instead of `sys.argv[1:]`. This is mostly useful for testing.

8.24 psamm.lpsolver.cplex – CPLEX LP solver

Linear programming solver using Cplex.

class psamm.lpsolver.cplex.**Solver**
Represents an LP-solver using Cplex

create_problem(***kwargs*)
Create a new LP-problem using the solver

class psamm.lpsolver.cplex.**CplexRangedProperty**(*get_prop*, *doc=None*)
Decorator for translating Cplex ranged properties.

class psamm.lpsolver.cplex.**Problem**(***kwargs*)
Represents an LP-problem of a cplex.Solver

cplex
The underlying Cplex object

define(**names*, ***kwargs*)
Define a variable in the problem.

Variables must be defined before they can be accessed by var() or set(). This function takes keyword arguments lower and upper to define the bounds of the variable (default: -inf to inf). The keyword argument types can be used to select the type of the variable (Continuous (default), Binary or Integer). Setting any variables different than Continuous will turn the problem into an MILP problem. Raises ValueError if a name is already defined.

has_variable(*name*)
Check whether variable is defined in the model.

add_linear_constraints(**relations*)
Add constraints to the problem

Each constraint is represented by a Relation, and the expression in that relation can be a set expression.

set_objective(*expression*)
Set objective expression of the problem.

set_linear_objective(*expression*)
Set objective of the problem.

Deprecated since version 0.19: Use `set_objective()` instead.

set_objective_sense(*sense*)
Set type of problem (maximize or minimize)

solve_unchecked(*sense=None*)
Solve problem and return result.

The user must manually check the status of the result to determine whether an optimal solution was found. A `SolverError` may still be raised if the underlying solver raises an exception.

result
Result of solved problem

feasibility_tolerance
Feasibility tolerance.

optimality_tolerance
Optimality tolerance.

integrality_tolerance
Integrality tolerance.

class psamm.lpsolver.cplex.**Constraint**(*prob*, *name*)
Represents a constraint in a cplex.Problem

delete()
Remove constraint from Problem instance

class psamm.lpsolver.cplex.**Result**(*prob*)
Represents the solution to a cplex.Problem

This object will be returned from the cplex.Problem.solve() method or by accessing the cplex.Problem.result property after solving a problem. This class should not be instantiated manually.

Result will evaluate to a boolean according to the success of the solution, so checking the truth value of the result will immediately indicate whether solving was successful.

success
Return boolean indicating whether a solution was found

status
Return string indicating the error encountered on failure

unbounded
Whether solution is unbounded

get_value(*expression*)
Return value of expression.

8.25 `psamm.lpsolver.generic` – Generic linear programming solver

Generic interface to LP solver instantiation.

exception `psamm.lpsolver.generic.`**RequirementsError**
Error resolving solver requirements

`psamm.lpsolver.generic.`**filter_solvers**(*solvers*, *requirements*)
Yield solvers that fullfil the requirements.

class `psamm.lpsolver.generic.`**Solver**(***kwargs*)
Generic solver interface based on requirements

Use the any of the following keyword arguments to restrict which underlying solver is used:

- *integer*: Use a solver that support integer variables (MILP)
- *rational*: Use a solver that returns rational results
- *quadratic*: Use a solver that supports quadratic objective/constraints
- *name*: Select a specific solver based on the name

create_problem()
Create a `Problem` instance

`psamm.lpsolver.generic.`**parse_solver_setting**(*s*)
Parse a string containing a solver setting

`psamm.lpsolver.generic.`**list_solvers**(*args=None*)
Entry point for listing available solvers.

8.26 `psamm.lpsolver.glpk` – GLPK LP solver

Linear programming solver using GLPK.

exception `psamm.lpsolver.glpk.`**GLPKError**
Error from calling GLPK library.

class `psamm.lpsolver.glpk.`**Solver**
Represents an LP-solver using GLPK.

create_problem(***kwargs*)
Create a new LP-problem using the solver.

class `psamm.lpsolver.glpk.`**Problem**(***kwargs*)
Represents an LP-problem of a `Solver`.

glpk
The underlying GLPK (SWIG) object.

define(**names*, ***kwargs*)
Define a variable in the problem.

Variables must be defined before they can be accessed by var() or set(). This function takes keyword arguments lower and upper to define the bounds of the variable (default: -inf to inf). The keyword argument types can be used to select the type of the variable (Continuous (default), Binary or Integer). Setting any variables different than Continuous will turn the problem into an MILP problem. Raises ValueError if a name is already defined.

has_variable(*name*)
Check whether variable is defined in the model.

add_linear_constraints(**relations*)
Add constraints to the problem.

Each constraint is represented by a Relation, and the expression in that relation can be a set expression.

set_objective(*expression*)
Set objective of problem.

set_linear_objective(*expression*)
Set objective of the problem.

Deprecated since version 0.19: Use `set_objective()` instead.

set_objective_sense(*sense*)
Set type of problem (maximize or minimize).

solve_unchecked(*sense=None*)
Solve problem and return result.

The user must manually check the status of the result to determine whether an optimal solution was found. A `SolverError` may still be raised if the underlying solver raises an exception.

result
Result of solved problem

feasibility_tolerance
Feasibility tolerance.

optimality_tolerance
Optimality tolerance.

integrality_tolerance
Integrality tolerance.

class psamm.lpsolver.glpk.**Constraint**(*prob*, *name*)
Represents a constraint in a `Problem`.

delete()
Remove constraint from Problem instance

class psamm.lpsolver.glpk.**Result**(*prob*, *ret_val=0*)
Represents the solution to a `Problem`.

This object will be returned from the solve() method on `Problem` or by accessing the result property on `Problem` after solving a problem. This class should not be instantiated manually.

Result will evaluate to a boolean according to the success of the solution, so checking the truth value of the result will immediately indicate whether solving was successful.

success
Return boolean indicating whether a solution was found.

unbounded
Whether solution is unbounded

status
Return string indicating the error encountered on failure.

get_value(*expression*)
Return value of expression.

class psamm.lpsolver.glpk.**MIPResult**(*prob*, *ret_val=0*)
Specialization of Result for MIP problems.

success
Return boolean indicating whether a solution was found.

status
Return string indicating the error encountered on failure.

8.27 psamm.lpsolver.gurobi – Gurobi LP solver

Linear programming solver using Gurobi.

class psamm.lpsolver.gurobi.**Solver**
Represents an LP-solver using Gurobi.

create_problem(***kwargs*)
Create a new LP-problem using the solver.

class psamm.lpsolver.gurobi.**Problem**(***kwargs*)
Represents an LP-problem of a gurobi.Solver.

gurobi
The underlying Gurobi Model object.

define(**names*, ***kwargs*)
Define a variable in the problem.

Variables must be defined before they can be accessed by var() or set(). This function takes keyword arguments lower and upper to define the bounds of the variable (default: -inf to inf). The keyword argument types can be used to select the type of the variable (Continuous (default), Binary or Integer). Setting any variables different than Continuous will turn the problem into an MILP problem. Raises ValueError if a name is already defined.

has_variable(*name*)
Check whether variable is defined in the model.

add_linear_constraints(**relations*)
Add constraints to the problem.

Each constraint is represented by a Relation, and the expression in that relation can be a set expression.

set_objective(*expression*)
Set linear objective of problem.

set_linear_objective(*expression*)
Set objective of the problem.

Deprecated since version 0.19: Use `set_objective()` instead.

set_objective_sense(*sense*)
Set type of problem (maximize or minimize).

solve_unchecked(*sense=None*)
Solve problem and return result.

The user must manually check the status of the result to determine whether an optimal solution was found. A `SolverError` may still be raised if the underlying solver raises an exception.

result
Result of solved problem

feasibility_tolerance
Feasibility tolerance.

optimality_tolerance
Optimality tolerance.

integrality_tolerance
Integrality tolerance.

class `psamm.lpsolver.gurobi.`**Constraint**(*prob*, *name*)
Represents a constraint in a gurobi.Problem.

delete()
Remove constraint from Problem instance

class `psamm.lpsolver.gurobi.`**Result**(*prob*)
Represents the solution to a gurobi.Problem.

This object will be returned from the gurobi.Problem.solve() method or by accessing the gurobi.Problem.result property after solving a problem. This class should not be instantiated manually.

Result will evaluate to a boolean according to the success of the solution, so checking the truth value of the result will immediately indicate whether solving was successful.

success
Return boolean indicating whether a solution was found.

unbounded
Whether solution is unbounded

status
Return string indicating the error encountered on failure.

get_value(*expression*)
Return value of expression.

8.28 `psamm.lpsolver.lp` – Linear programming problems

Base objects for representation of LP problems.

A linear programming problem is built from a number of constraints and an objective function. The objective function is a linear expression represented by *`Expression`*. The constraints are represented by *`Relation`*, created from a linear expression and a relation sense (equals, greater, less).

Expressions are built from variables defined in the *`Problem`* instance. In addition, an expression can contain a *`VariableSet`* instead of a single variable. This allows many similar expressions to be represented by one *`Expression`* instance which means that the LP problem can be constructed faster.

exception `psamm.lpsolver.lp.`**SolverError**
Error wrapping solver specific errors.

class `psamm.lpsolver.lp.`**VariableSet**
A tuple used to represent sets of variables.

class psamm.lpsolver.lp.**Product**
A tuple used to represent a variable product.

class psamm.lpsolver.lp.**RangedProperty**(*fget=None*, *fset=None*, *fdel=None*, *fmin=None*, *fmax=None*, *doc=None*)
Numeric property with minimum and maximum values.

The value attribute is used to get/set the actual value of the propery. The min/max attributes are used to get the bounds. The range is not automatically enforced when the value is set.

psamm.lpsolver.lp.**ranged_property**(*min=None*, *max=None*)
Decorator for creating ranged property with fixed bounds.

class psamm.lpsolver.lp.**Expression**(*variables={}*, *offset=0*)
Represents a linear expression

The variables can be any hashable objects. If one or more variables are instead *VariableSets*, this will be taken to represent a set of expressions separately using a different element of the *VariableSet*.

```
>>> e = Expression({'x': 2, 'y': 3})
>>> str(e)
'2*x + 3*y'
```

In order to provide a more natural syntax for creating *Relations* the binary relation operators have been overloaded to return *Relation* instances.

```
>>> rel = Expression({'x': 2}) >= Expression({'y': 3})
>>> str(rel)
'2*x - 3*y >= 0'
```

Warning: Chained relations cannot be converted to multiple relations, e.g. `4 <= e <= 10` will fail to produce the intended relations!

offset
Value of the offset

variables()
Return immutable view of variables in expression.

values()
Return immutable view of (variable, value)-pairs in expression.

value_sets()
Iterator of expression sets

This will yield an iterator of (variable, value)-pairs for each expression in the expression set (each equivalent to values()). If none of the variables is a set variable then a single iterator will be yielded.

class psamm.lpsolver.lp.**RelationSense**
An enumeration.

class psamm.lpsolver.lp.**Relation**(*sense*, *expression*)
Represents a binary relation (equation or inequality)

Relations can be equalities or inequalities. All relations of this type can be represented as a left-hand side expression and the type of relation. In this representation, the right-hand side is always zero.

sense
Type of relation (equality or inequality)

Can be one of Equal, Greater or Less, or one of the strict relations, StrictlyGreater or StrictlyLess.

expression
Left-hand side expression

class psamm.lpsolver.lp.**ObjectiveSense**
Enumeration of objective sense values

Minimize = -1
Minimize objective function

Maximize = 1
Maximize objective function

class psamm.lpsolver.lp.**VariableType**
Enumeration of variable types

Continuous = 'C'
Continuous variable type

Integer = 'I'
Integer variable type

Binary = 'B'
Binary variable type (0 or 1)

class psamm.lpsolver.lp.**Solver**
Factory for LP Problem instances

create_problem()
Create a new *Problem* instance

class psamm.lpsolver.lp.**Constraint**
Represents a constraint within an LP Problem

delete()
Remove constraint from Problem instance

class psamm.lpsolver.lp.**VariableNamespace**(*problem*, ***kwargs*)
Namespace for defining variables.

Namespaces are always unique even if two namespaces have the same name. Variables defined within a namespace will never clash with a global variable name or a name defined in another namespace. Namespaces should be created using the *Problem.namespace()*.

```
>>> v = prob.namespace(name='v')
>>> v.define([1, 2, 5], lower=0, upper=10)
>>> prob.set_objective(v[1] + 3*v[2] - 5 * v[5])
```

define(*names*, ***kwargs*)
Define variables within the namespace.

This is similar to *Problem.define()* except that names must be given as an iterable. This method accepts the same keyword arguments as *Problem.define()*.

set(*names*)
Return a variable set of the given names in the namespace.

```
>>> v = prob.namespace(name='v')
>>> v.define([1, 2, 5], lower=0, upper=10)
>>> prob.add_linear_constraints(v.set([1, 2]) >= 4)
```

sum (*names*)
Return the sum of the given names in the namespace.

```
>>> v = prob.namespace(name='v')
>>> v.define([1, 2, 5], lower=0, upper=10)
>>> prob.set_objective(v.sum([2, 5]))  # v[2] + v[5]
```

expr (*items*)
Return the sum of each name multiplied by a coefficient.

```
>>> v = prob.namespace(name='v')
>>> v.define(['a', 'b', 'c'], lower=0, upper=10)
>>> prob.set_objective(v.expr([('a', 2), ('b', 1)]))
```

value (*name*)
Return value of given variable in namespace.

```
>>> v = prob.namespace(name='v')
>>> v.define([1, 2, 5], lower=0, upper=10)
>>> prob.solve()
>>> print(v.value(2))
```

class psamm.lpsolver.lp.**Problem**
Representation of LP Problem instance

Variable names in the problem can be any hashable object. It is the responsibility of the solver interface to translate the object into a unique string if required by the underlying LP solver.

define (**names*, ***kwargs*)
Define a variable in the problem.

Variables must be defined before they can be accessed by var() or set(). This function takes keyword arguments lower and upper to define the bounds of the variable (default: -inf to inf). The keyword argument types can be used to select the type of the variable (Continuous (default), Binary or Integer). Setting any variables different than Continuous will turn the problem into an MILP problem. Raises ValueError if a name is already defined.

has_variable (*name*)
Check whether a variable is defined in the problem.

namespace (*names=None*, ***kwargs*)
Return namespace for this problem.

If names is given it should be an iterable of names to define in the namespace. Other keyword arguments can be specified which will be used to define the names given as well as being used as default parameters for names that are defined later.

```
>>> v = prob.namespace(name='v')
>>> v.define([1, 2, 5], lower=0, upper=10)
>>> prob.set_objective(v[1] + 3*v[2] - 5 * v[5])
```

var (*name*)
Return variable as an `Expression`.

expr (*values*, *offset=0*)
Return the given dictionary of values as an `Expression`.

set (*names*)
Return variables as a set expression.

This returns an `Expression` containing a `VariableSet`.

add_linear_constraints(**relations*)
Add constraints to the problem.

Each constraint is given as a `Relation`, and the expression in that relation can be a set expression. Returns a sequence of `Constraints`.

set_objective(*expression*)
Set objective of the problem to the given `Expression`.

set_linear_objective(*expression*)
Set objective of the problem.

Deprecated since version 0.19: Use `set_objective()` instead.

set_objective_sense(*sense*)
Set type of problem (minimize or maximize)

solve(*sense=None*)
Solve problem and return result.

Raises `SolverError` if the solution is not optimal.

solve_unchecked(*sense=None*)
Solve problem and return result.

The user must manually check the status of the result to determine whether an optimal solution was found. A `SolverError` may still be raised if the underlying solver raises an exception.

result
Result of solved problem

exception psamm.lpsolver.lp.**InvalidResultError**(*msg=None*)
Raised when a result that has been invalidated is accessed

class psamm.lpsolver.lp.**Result**
Result of solving an LP problem

The result is tied to the solver instance and is valid at least until the problem is solved again. If the problem has been solved again an `InvalidResultError` may be raised.

success
Whether solution was optimal

status
String indicating the status of the problem result

unbounded
Whether solution is unbounded

get_value(*expression*)
Get value of variable or expression in result

Expression can be an object defined as a name in the problem, in which case the corresponding value is simply returned. If expression is an actual `Expression` object, it will be evaluated using the values from the result.

8.29 `psamm.lpsolver.qsoptex` – QSopt_ex LP solver

Linear programming solver using QSopt_ex.

class psamm.lpsolver.qsoptex.**Solver**
Represents an LP solver using QSopt_ex

create_problem(***kwargs*)
Create a new LP-problem using the solver

class psamm.lpsolver.qsoptex.**Problem**(***kwargs*)
Represents an LP-problem of a qsoptex.Solver

qsoptex
The underlying qsoptex.ExactProblem object

define(**names*, ***kwargs*)
Define a variable in the problem.

Variables must be defined before they can be accessed by var() or set(). This function takes keyword arguments lower and upper to define the bounds of the variable (default: -inf to inf). The keyword argument types can be used to select the type of the variable (only Continuous is supported). Raises ValueError if a name is already defined.

has_variable(*name*)
Check whether variable is defined in the model.

add_linear_constraints(**relations*)
Add constraints to the problem

Each constraint is represented by a Relation, and the expression in that relation can be a set expression.

set_objective(*expression*)
Set linear objective of problem

set_linear_objective(*expression*)
Set objective of the problem.

Deprecated since version 0.19: Use `set_objective()` instead.

set_objective_sense(*sense*)
Set type of problem (maximize or minimize)

solve_unchecked(*sense=None*)
Solve problem and return result.

The user must manually check the status of the result to determine whether an optimal solution was found. A `SolverError` may still be raised if the underlying solver raises an exception.

result
Result of solved problem

feasibility_tolerance
Feasibility tolerance.

optimality_tolerance
Optimality tolerance.

class psamm.lpsolver.qsoptex.**Constraint**(*prob*, *name*)
Represents a constraint in a qsoptex.Problem

delete()
Remove constraint from Problem instance

class psamm.lpsolver.qsoptex.**Result**(*prob*)
Represents the solution to a qsoptex.Problem

This object will be returned from the Problem.solve() method or by accessing the Problem.result property after solving a problem. This class should not be instantiated manually.

Result will evaluate to a boolean according to the success of the solution, so checking the truth value of the result will immediately indicate whether solving was successful.

`success`
Return boolean indicating whether a solution was found

`unbounded`
Whether the solution is unbounded

`status`
Return string indicating the error encountered on failure

`get_value`(*expression*)
Return value of expression

8.30 `psamm.manual_curation` – Model Mapping Curation

class `psamm.manual_curation.`**`Curator`**(*compound_map_file*, *reaction_map_file*, *curated_compound_map_file*, *curated_reaction_map_file*)
Parse and save mapping files during manual curation.

Use `add_mapping()` to add new curated pairs. Save current progress into files by `save()`.

Besides the curated mapping files, the `Curator` will also store false mappings into *.false* files, and compounds and reactions to be ignored can be stored in *.ignore* files. For example, if the *curated_compound_map_file* is set to *curated_compound_mapping.tsv*, then the false mappings will be stored in *curated_compound_mapping.tsv.false*, and the pairs to be ignored should be stored in *curated_compound_mapping.tsv.ignore*.

If the curated files already exist, the `Curator` will consider them as the previous progress, then append new curation results.

Parameters

- **`compound_map_file`** – .tsv file of compound mapping result
- **`reaction_map_file`** – .tsv file of reaction mapping result
- **`curated_compound_map_file`** – .tsv file of curated compound mapping result
- **`curated_reaction_map_file`** – .tsv file of curated reaction mapping result

`reaction_checked`(*id*)
Return True if reaction pair has been checked.

Parameters **`id`** – one reaction id or a tuple of id pair

`reaction_ignored`(*id*)
Return True if reaction id is in ignore list.

`compound_checked`(*id*)
Return True if compound pair has been checked.

Parameters **`id`** – one compound id or a tuple of id pair

`compound_ignored`(*id*)
Return True if compound id is in ignore list.

add_mapping(*id*, *type*, *correct*)
Add new mapping result to curated list.

Parameters

- **id** – tuple of id pair
- **type** – 'c' for compound mapping, 'r' for reaction mapping
- **correct** – True if the mapping pair is correct

add_ignore(*id*, *type*)
Add id to ignore list.

Parameters

- **id** – id to be ignored
- **type** – 'c' for compound mapping, 'r' for reaction mapping

save()
Save current curator to files.

psamm.manual_curation.**search_compound**(*model*, *id*)
Search a set of compounds, then print detailed properties.

Parameters **id** – a list of compound ids

psamm.manual_curation.**search_reaction**(*model*, *ids*)
Search a set of reactions, print detailed properties, then return a generator. Each item in the generator is a list of compounds in the corresponding reaction.

Parameters **ids** – a list of reaction ids

8.31 psamm.mapmaker – Mapmaker Functions

Functions for predicting compound pairs using the MapMaker algorithm.

The MapMaker algorithm is described in [Tervo16].

psamm.mapmaker.**default_weight**(*element*)
Return weight of formula element.

This implements the default weight proposed for MapMaker.

exception psamm.mapmaker.**UnbalancedReactionError**
Raised when an unbalanced reaction is provided.

psamm.mapmaker.**predict_compound_pairs**(*reaction*, *compound_formula*, *solver*, *epsilon=1e-05*, *alt_elements=None*, *weight_func=<function default_weight>*)
Predict compound pairs of reaction using MapMaker.

Yields all solutions as dictionaries with compound pairs as keys and formula objects as values.

Parameters

- **reaction** – *psamm.reaction.Reaction* object.
- **compound_formula** – Dictionary mapping compound IDs to formulas. Formulas must be flattened.
- **solver** – LP solver (MILP).

- **epsilon** – Threshold for rounding floating-point values to integers in the predicted transfers.
- **alt_elements** – Iterable of elements to consider for alternative solutions. Only alternate solutions that have different transfers of these elements will be returned (default=all elements).
- **weight_func** – Optional function that returns a weight for a formula element (should handle specific Atom and Radical objects). By default, the standard MapMaker weights will be used (H=0, R=40, N=0.4, O=0.4, P=0.4, *=1).

8.32 `psamm.massconsistency` – Mass consistency check

Mass consistency analysis of metabolic databases

A stoichiometric matrix, S, is said to be mass-consistent if S^Tm = 0 has a positive solution (m_i > 0). This corresponds to assigning a positive mass to each compound in the stoichiometric matrix and having each reaction preserve mass. Exchange reactions will have to be excluded from this check, as they are not able to preserve mass (by definition). In addition some databases may contain pseudo-compounds (e.g. "photon") that also has to be excluded.

exception psamm.massconsistency.**MassConsistencyError**
Indicates an error while checking for mass consistency

psamm.massconsistency.**is_consistent**(*database*, *solver*, *exchange={}*, *zeromass={}*)
Try to assign a positive mass to each compound

Return True if successful. The masses are simply constrained by m_i > 1 and finding a solution under these conditions proves that the database is mass consistent.

psamm.massconsistency.**check_reaction_consistency**(*database*, *solver*, *exchange={}*, *checked={}*, *zeromass={}*, *weights={}*)
Check inconsistent reactions by minimizing mass residuals

Return a reaction iterable, and compound iterable. The reaction iterable yields reaction ids and mass residuals. The compound iterable yields compound ids and mass assignments.

Each compound is assigned a mass of at least one, and the masses are balanced using the stoichiometric matrix. In addition, each reaction has a residual mass that is included in the mass balance equations. The L1-norm of the residuals is minimized. Reactions in the checked set are assumed to have been manually checked and therefore have the residual fixed at zero.

psamm.massconsistency.**check_compound_consistency**(*database*, *solver*, *exchange={}*, *zeromass={}*)
Yield each compound in the database with assigned mass

Each compound will be assigned a mass and the number of compounds having a positive mass will be approximately maximized.

This is an implementation of the solution originally proposed by [Gevorgyan08] but using the new method proposed by [Thiele14] to avoid MILP constraints. This is similar to the way Fastcore avoids MILP contraints.

8.33 `psamm.metabolicmodel` – Metabolic model representation

Representation of metabolic network models.

psamm.metabolicmodel.**create_exchange_id**(*existing_ids*, *compound*)
Create unique ID for exchange of compound.

psamm.metabolicmodel.**create_transport_id**(*existing_ids*, *compound_1*, *compound_2*)
Create unique ID for transport reaction of compounds.

class psamm.metabolicmodel.**FluxBounds**(*model*, *reaction*)
Represents lower and upper bounds of flux as a mutable object

This object is used internally in the model representation. Changing the state of the object will change the underlying model parameters. Deleting a value will reset that value to the defaults.

lower
Lower bound

upper
Upper bound

bounds
Bounds as a tuple

class psamm.metabolicmodel.**LimitsView**(*model*)
Provides a view of the flux bounds defined in the model

This object is used internally in MetabolicModel to expose a dictonary view of the FluxBounds associated with the model reactions.

class psamm.metabolicmodel.**MetabolicModel**(*database*, *v_max=1000*)
Represents a metabolic model containing a set of reactions

The model contains a list of reactions referencing the reactions in the associated database.

reactions
Iterator of reactions IDs in the database.

compounds
Itertor of *Compounds* in the database.

compartments
Iterator of compartment IDs in the database.

has_reaction(*reaction_id*)
Whether the given reaction exists in the database.

get_reaction(*reaction_id*)
Return reaction as a *Reaction*.

get_reaction_values(*reaction_id*)
Return stoichiometric values of reaction as a dictionary

get_compound_reactions(*compound_id*)
Iterate over all reaction ids the includes the given compound

is_reversible(*reaction_id*)
Whether the given reaction is reversible

is_exchange(*reaction_id*)
Whether the given reaction is an exchange reaction.

add_reaction(*reaction_id*)
Add reaction to model

remove_reaction(*reaction*)
Remove reaction from model

copy()
Return copy of model

classmethod load_model(*database*, *reaction_iter=None*, *exchange=None*, *limits=None*, *v_max=None*)
Get model from reaction name iterator.

The model will contain all reactions of the iterator.

make_irreversible(*gene_dict={}*, *exclude_list=[]*, *lumped_rxns={}*, *all_reversible=False*)
Creates a new metabolic models with only irreversible reactions.

This function will find every reversible reaction in the model and split it into two reactions with the {rxnid}_forward or {rxnid}_reverse as the IDs.

Parameters

- **self** – A metabolicmodel object
- **gene_dict** – A dictionary stores reaction ids as keys and corresponding gene association as values, it's required only in GIMME function
- **exclude_list** – list of reactions to exclude in TMFA simulation
- **all_reversible** – if True make all reactions in model reversible.

Values: mm_irrev: A new metabolic model with only irreversible reactions gene_dict_reversible: A dictionary mapping irreversible reaction

ids to gene associations, used only in GIMME function

split_rxns: A list of splitted reactions, each element is a tuple of ({rxnid}_forward, {rxnid}_reverse)

class psamm.metabolicmodel.**FlipableFluxBounds**(*view*, *reaction*)
FluxBounds object for a FlipableModelView.

This object is used internally in the FlipableModelView to represent the bounds of flux on a reaction that can be flipped.

lower
Lower bound

upper
Upper bound

class psamm.metabolicmodel.**FlipableStoichiometricMatrixView**(*view*)
Provides a matrix view that flips with the flipable model view.

This object is used internally in FlipableModelView to expose a matrix view that negates the stoichiometric values of flipped reactions.

class psamm.metabolicmodel.**FlipableLimitsView**(*view*)
Provides a limits view that flips with the flipable model view.

This object is used internally in FlipableModelView to expose a limits view that flips the bounds of all flipped reactions.

class psamm.metabolicmodel.**FlipableModelView**(*model*, *flipped={}*)
Proxy wrapper of model objects allowing a flipped set of reactions.

The proxy will forward all properties normally except that flipped reactions will appear to have stoichiometric values negated in the matrix property, and have bounds in the limits property flipped. This view is needed for some algorithms.

8.34 `psamm.moma` – Minimization of metabolic adjustments

Implementation of Minimization of Metabolic Adjustments (MOMA).

exception `psamm.moma.`**`MOMAError`**

Error indicating an error solving MOMA.

class `psamm.moma.`**`ConstraintGroup`**(*moma*, **args*)

Constraints that will be imposed on the model when solving.

Parameters

- **`moma`** – MOMAProblem object for the proposed constraints.
- **`*args`** – The constraints that are imposed on the model.

`add`(**args*)

Add constraints to the model.

`delete`()

Set up the constraints to get deleted on the next solve.

class `psamm.moma.`**`MOMAProblem`**(*model*, *solver*)

Model as a flux optimization problem with minimal flux redistribution.

Create a representation of the model as an LP optimization problem with steady state assumption and a minimal redistribution of metabolic fluxes with respect to the wild type configuration.

The problem can be solved using any of the four MOMA variants described in [Segre02] and [Mo09]. MOMA is formulated to avoid the FBA assumption that that growth efficiency has evolved to an optimal point directly following model perturbation. MOMA finds the optimal solution for a model with minimal flux redistribution with respect to the wild type flux configuration.

MOMA is implemented with two variations of a linear optimization problem (*`lin_moma()`* and *`lin_moma2()`*) and two variations of a quadratic optimization problem (*`moma()`* and *`moma2()`*). Further information on these methods can be found within their respective documentation.

The problem can be modified and solved as many times as needed. The flux of a reaction can be obtained after solving using *`get_flux()`*.

Parameters

- **`model`** – `MetabolicModel` to solve.
- **`solver`** – LP solver instance to use.

`prob`

Return the underlying LP problem.

`constraints`(**args*)

Return a constraint object.

`solve_fba`(*objective*)

Solve the wild type problem using FBA.

Parameters **`objective`** – The objective reaction to be maximized.

Returns The LP Result object for the solved FBA problem.

`get_fba_flux`(*objective*)

Return a dictionary of all the fluxes solved by FBA.

Dictionary of fluxes is used in *`lin_moma()`* and *`moma()`* to minimize changes in the flux distributions following model perturbation.

Parameters **`objective`** – The objective reaction that is maximized.

Returns Dictionary of fluxes for each reaction in the model.

`get_minimal_fba_flux`(*objective*)

Find the FBA solution that minimizes all the flux values.

Maximize the objective flux then minimize all other fluxes while keeping the objective flux at the maximum.

Parameters **`objective`** – The objective reaction that is maximized.

Returns A dictionary of all the reactions and their minimized fluxes.

`get_fba_obj_flux`(*objective*)

Return the maximum objective flux solved by FBA.

`lin_moma`(*wt_fluxes*)

Minimize the redistribution of fluxes using a linear objective.

The change in flux distribution is mimimized by minimizing the sum of the absolute values of the differences of wild type FBA solution and the knockout strain flux solution.

This formulation bases the solution on the wild type fluxes that are specified by the user. If these wild type fluxes were calculated using FBA, then an arbitrary flux vector that optimizes the objective function is used. See [Segre‘_02] for more information.

Parameters **`wt_fluxes`** – Dictionary of all the wild type fluxes. Use `get_fba_flux(objective)()` to return a dictionary of fluxes found by FBA.

`lin_moma2`(*objective*, *wt_obj*)

Find the smallest redistribution vector using a linear objective.

The change in flux distribution is mimimized by minimizing the sum of the absolute values of the differences of wild type FBA solution and the knockout strain flux solution.

Creates the constraint that the we select the optimal flux vector that is closest to the wildtype. This might still return an arbitrary flux vector the maximizes the objective function.

Parameters

- **`objective`** – Objective reaction for the model.
- **`wt_obj`** – The flux value for your wild type objective reactions. Can either use an expiremental value or on determined by FBA by using `get_fba_obj_flux(objective)()`.

`moma`(*wt_fluxes*)

Minimize the redistribution of fluxes using Euclidean distance.

Minimizing the redistribution of fluxes using a quadratic objective function. The distance is minimized by minimizing the sum of (wild type - knockout)^2.

Parameters **`wt_fluxes`** – Dictionary of all the wild type fluxes that will be used to find a close MOMA solution. Fluxes can be expiremental or calculated using :meth: get_fba_flux(objective).

`moma2`(*objective*, *wt_obj*)

Find the smallest redistribution vector using Euclidean distance.

Minimizing the redistribution of fluxes using a quadratic objective function. The distance is minimized by minimizing the sum of (wild type - knockout)^2.

Creates the constraint that the we select the optimal flux vector that is closest to the wildtype. This might still return an arbitrary flux vector the maximizes the objective function.

Parameters

- **`objective`** – Objective reaction for the model.
- **`wt_obj`** – The flux value for your wild type objective reactions. Can either use an expiremental value or on determined by FBA by using `get_fba_obj_flux(objective)()`.

`get_flux`(*reaction*)
Return the knockout flux for a specific reaction.

`get_flux_var`(*reaction*)
Return the LP variable for a specific reaction.

8.35 `psamm.randomsparse` – Find a random minimal network of model reactions

class `psamm.randomsparse.`**`ReactionDeletionStrategy`**(*model*, *reaction_set=None*)
Deleting reactions strategy class.

When initializing instances of this class, *`get_exchange_reactions()`* can be useful if exchange reactions are used as the test set.

class `psamm.randomsparse.`**`GeneDeletionStrategy`**(*model*, *gene_assoc*)
Deleting genes strategy class.

When initializing instances of this class, *`get_gene_associations()`* can be called to obtain the gene association dict from the model.

`psamm.randomsparse.`**`get_gene_associations`**(*model*)
Create gene association for class *`GeneDeletionStrategy`*.

Return a dict mapping reaction IDs to *`psamm.expression.boolean.Expression`* objects, representing relationships between reactions and related genes. This helper function should be called when creating *`GeneDeletionStrategy`* objects.

Parameters **`model`** – *`psamm.datasource.native.NativeModel`*.

`psamm.randomsparse.`**`get_exchange_reactions`**(*model*)
Yield IDs of all exchange reactions from model.

This helper function would be useful when creating *`ReactionDeletionStrategy`* objects.

Parameters **`model`** – *`psamm.metabolicmodel.MetabolicModel`*.

`psamm.randomsparse.`**`random_sparse`**(*strategy*, *prob*, *obj_reaction*, *flux_threshold*)
Find a random minimal network of model reactions.

Given a reaction to optimize and a threshold, delete entities randomly until the flux of the reaction to optimize falls under the threshold. Keep deleting until no more entities can be deleted. It works with two strategies: deleting reactions or deleting genes (reactions related to certain genes).

Parameters

- **`strategy`** – *`ReactionDeletionStrategy`* or *`GeneDeletionStrategy`*.
- **`prob`** – *`psamm.fluxanalysis.FluxBalanceProblem`*.
- **`obj_reaction`** – objective reactions to optimize.
- **`flux_threshold`** – threshold of max reaction flux.

psamm.randomsparse.**random_sparse_return_all**(*strategy*, *prob*, *obj_reaction*, *flux_threshold*)

Find a random minimal network of model reactions.

Given a reaction to optimize and a threshold, delete entities randomly until the flux of the reaction to optimize falls under the threshold. Keep deleting until no more entities can be deleted. It works with two strategies: deleting reactions or deleting genes (reactions related to certain genes).

Parameters

- **strategy** – *ReactionDeletionStrategy* or *GeneDeletionStrategy*.
- **prob** – *psamm.fluxanalysis.FluxBalanceProblem*.
- **obj_reaction** – objective reactions to optimize.
- **flux_threshold** – threshold of max reaction flux.

8.36 psamm.reaction – Reaction equations and compounds

Definitions related to reaction equations and parsing of such equations.

class psamm.reaction.**Compound**(*name*, *compartment=None*, *arguments=()*)

Represents a compound in a reaction equation

A compound is a named entity in the reaction equations representing a chemical compound. A compound can represent a generalized chemical entity (e.g. polyphosphate) and the arguments can be used to instantiate a specific chemical entity (e.g. polyphosphate(3)) by passing a number as an argument or a partially specified entity by passing an expression (e.g. polyphosphate(n)).

name

Name of compound

compartment

Compartment of compound

arguments

Expression argument for generalized compounds

translate(*func*)

Translate compound name using given function

```
>>> Compound('Pb').translate(lambda x: x.lower())
Compound('pb')
```

in_compartment(*compartment*)

Return an instance of this compound in the specified compartment

```
>>> Compound('H+').in_compartment('e')
Compound('H+', 'e')
```

class psamm.reaction.**Direction**

Directionality of reaction equation.

forward

Whether this direction includes forward direction.

reverse

Whether this direction includes reverse direction.

flipped()

Return the flipped version of this direction.

symbol
Return string symbol for direction.

class psamm.reaction.**Reaction**(**args*)
Reaction equation representation.

Each compound is associated with a stoichiometric value and the reaction has a `Direction`. The reaction is created in one of the three following ways.

It can be created from a direction and two iterables of compound, value pairs representing the left-hand side and the right-hand side of the reaction:

```
>>> r = Reaction(Direction.Both, [(Compound('A'), 1), (Compound('B', 2))],
                                  [(Compound('C'), 1)])
>>> str(r)
'|A| + (2) |B| <=> |C|'
```

It can also be created from a single dict or iterable of compound, value pairs where the left-hand side compounds have negative values and the right-hand side compounds have positive values:

```
>>> r = Reaction(Direction.Forward, {
        Compound('A'): -1,
        Compound('B'): -2,
        Compound('C'): 1
})
>>> str(r)
'|A| + (2) |B| <=> |C|'
```

Lastly, the reaction can be created from an existing reaction object, creating a copy of that reaction.

```
>>> r = Reaction(Direction.Forward, {Compound('A'): -1, Compound('B'): 1})
>>> r2 = Reaction(r)
>>> str(r2)
'|A| => |B|'
```

Reactions can be added to produce combined reactions.

```
>>> r = Reaction(Direction.Forward, {Compound('A'): -1, Compound('B'): 1})
>>> s = Reaction(Direction.Forward, {Compound('B'): -1, Compound('C'): 1})
>>> str(r + s)
'|A| => |C|'
```

Reactions can also be multiplied by a number to produce a new reaction with scaled stoichiometry.

```
>>> r = Reaction(Direction.Forward, {Compound('A'): -1, Compound('B'): 2})
>>> str(2 * r)
'(2) |A| => (4) |B|'
```

Multiplying with a negative value will also flip the reaction, and as a special case, negating a reaction will simply flip it.

```
>>> r = Reaction(Direction.Forward, {Compound('A'): -1, Compound('B'): 2})
>>> str(r)
'|A| => (2) |B|'
>>> str(-r)
'(2) |B| <= |A|'
```

direction
Direction of reaction equation

left
Compounds on the left-hand side of the reaction equation.

right
Compounds on the right-hand side of the reaction equation.

compounds
Sequence of compounds on both sides of the reaction equation

The sign of the stoichiometric values reflect whether the compound is on the left-hand side (negative) or the right-hand side (positive).

normalized()
Return normalized reaction

The normalized reaction will be bidirectional or a forward reaction (i.e. reverse reactions are flipped).

translated_compounds(*translate*)
Return reaction where compound names have been translated.

For each compound the translate function is called with the compound name and the returned value is used as the new compound name. A new reaction is returned with the substituted compound names.

8.37 `psamm.translate_id` – ID Translation Functions

class `psamm.translate_id.`**TranslatedModel**(*ref_model*, *compound_map*, *reaction_map*, *compartment_map=None*)
A `NativeModel` with translated ids based on reference model.

The compound_map, reaction_map and compartment_map are three *dict* that use original id as key and new id as value. Use `write_model()` to output yaml files.

write_model(*dest*, ***kwargs*)
Output model into YAML files.

8.38 `psamm.util` – Internal utilities

Various utilities.

`psamm.util.`**mkdir_p**(*path*)
Make directory path if it does not already exist.

class `psamm.util.`**LoggerFile**(*logger*, *level*)
File-like object that forwards to a logger.

The Cplex API takes a file-like object for writing log output. This class allows us to forward the Cplex messages to the Python logging system.

write(*s*)
Write message to logger.

flush()
Flush stream.

This is a noop.

class `psamm.util.`**MaybeRelative**(*s*)
Helper type for parsing possibly relative parameters.

```
>>> arg = MaybeRelative('40%')
>>> arg.reference = 200.0
>>> float(arg)
80.0
```

```
>>> arg = MaybeRelative('24.5')
>>> arg.reference = 150.0
>>> float(arg)
24.5
```

relative
Whether the parsed number was relative.

reference
The reference used for converting to absolute value.

class psamm.util.**FrozenOrderedSet**(*seq=[]*)
An immutable set that retains insertion order.

class psamm.util.**DictView**(*d*)
An immutable wrapper around another dict-like object.

psamm.util.**create_unique_id**(*prefix, existing_ids*)
Return a unique string ID from the prefix.

First check if the prefix is itself a unique ID in the set-like parameter existing_ids. If not, try integers in ascending order appended to the prefix until a unique ID is found.

psamm.util.**git_try_describe**(*repo_path*)
Try to describe the current commit of a Git repository.

Return a string containing a string with the commit ID and/or a base tag, if successful. Otherwise, return None.

psamm.util.**convex_cardinality_relaxed**(*f, epsilon=1e-05*)
Transform L1-norm optimization function into cardinality optimization.

The given function must optimize a convex problem with a weighted L1-norm as the objective. The transformed function will apply the iterated weighted L1 heuristic to approximately optimize the cardinality of the solution. This method is described by S. Boyd, "L1-norm norm methods for convex cardinality problems." Lecture Notes for EE364b, Stanford University, 2007. Available online at www.stanford.edu/class/ee364b/.

The given function must take an optional keyword parameter weights (dictionary), and the weights must be set to one if not specified. The function must return the non-weighted solution as an iterator over (identifier, value)-tuples, either directly or as the first element of a tuple.

CHAPTER 9

References

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Bibliography

[Burgard04] Burgard AP, Nikolaev E V, Schilling CH, Maranas CD. Flux coupling analysis of genome-scale metabolic network reconstructions. Genome Res. 2004;14: 301–312. doi:10.1101/gr.1926504.

[Edwards00] Edwards JS, Palsson BO. Robustness Analysis of the Escherichia coli Metabolic Network. Biotechnol Prog. American Chemical Society; 2000;16: 927–939. doi:10.1021/bp0000712.

[Fell86] Fell DA, Small JR. Fat synthesis in adipose tissue. An examination of stoichiometric constraints. Biochem J. 1986;238. doi:10.1042/bj2380781.

[Gevorgyan08] Gevorgyan A, Poolman MG, Fell DA. Detection of stoichiometric inconsistencies in biomolecular models. Bioinformatics. 2008;24: 2245–2251. doi:10.1093/bioinformatics/btn425.

[Kumar07] Satish Kumar V, Dasika MS, Maranas CD. Optimization based automated curation of metabolic reconstructions. BMC Bioinformatics. 2007;8: 212. doi:10.1186/1471-2105-8-212.

[Mahadevan03] Mahadevan R, Schilling CH. The effects of alternate optimal solutions in constraint-based genome-scale metabolic models. Metab Eng. 2003;5: 264–276. doi:10.1016/j.ymben.2003.09.002.

[Mo09] Mo ML, Palsson BØ, Herrgård MJ. Connecting extracellular metabolomic measurements to intracellular flux states in yeast. BMC Systems Biology. 2009;3(1):3-37. doi:10.1186/1752-0509-3-37.

[Muller13] Müller AC, Bockmayr A. Fast thermodynamically constrained flux variability analysis. Bioinformatics. 2013;29: 903–909. doi:10.1093/bioinformatics/btt059.

[Orth10] Orth JD, Thiele I, Palsson BØ. What is flux balance analysis? Nat Biotechnol. Nature Publishing Group; 2010;28: 245–8. doi:10.1038/nbt.1614.

[Schilling00] Schilling CH, Letscher D, Palsson BO. Theory for the systemic definition of metabolic pathways and their use in interpreting metabolic function from a pathway-oriented perspective. J Theor Biol. 2000;203: 229–48. doi:10.1006/jtbi.2000.1073.

[Segre02] Segrè D, Vitkup D, Church GM. Analysis of optimality in natural and perturbed metabolic networks. Proceedings of the National Academy of Sciences of the United States of America. 2002;99(23):15112-15117. doi:10.1073/pnas.232349399.

[Steffensen17] Steffensen, J. L., Dufault-Thompson, K., & Zhang, Y. (2018). FindPrimaryPairs: An efficient algorithm for predicting element-transferring reactant/product pairs in metabolic networks. PLOS ONE, 13(2), e0192891. doi:10.1371/journal.pone.0192891

[Tervo16] Tervo CJ, Reed JL. MapMaker and PathTracer for tracking carbon in genome-scale metabolic models. Biotechnol J. 2016; 1–23. doi:10.1002/biot.201400305.

[Thiele14] Thiele I, Vlassis N, Fleming RMT. fastGapFill: efficient gap filling in metabolic networks. Bioinformatics. 2014;30: 2529–31. doi:10.1093/bioinformatics/btu321.

[Vlassis14] Vlassis N, Pacheco MP, Sauter T. Fast Reconstruction of Compact Context-Specific Metabolic Network Models. PLoS Comput Biol. 2014;10: e1003424. doi:10.1371/journal.pcbi.1003424.

[Orth13] Orth JD, Palsson BØ, Fleming RMT. Reconstruction and Use of Microbial Metabolic Networks: the Core Escherichia coli Metabolic Model as an Educational Guide. EcoSal Plus. asm Pub2Web; 2013;1. doi:10.1128/ecosalplus.10.2.1.

[Orth11] Orth JD, Conrad TM, Na J, Lerman JA, Nam H, Feist AM, et al. A comprehensive genome-scale reconstruction of Escherichia coli metabolism–2011. Mol Syst Biol. EMBO Press; 2011;7: 535. doi:10.1038/msb.2011.65.

[Henry07] Henry, C. S., Broadbelt, L. J., & Hatzimanikatis, V. (2007). Thermodynamics-Based Metabolic Flux Analysis. Biophysical Journal, 92(5), 1792–1805. doi:10.1529/biophysj.106.093138.

[Hamilton13] Hamilton, J. J., Dwivedi, V., & Reed, J. L. (2013). Quantitative Assessment of Thermodynamic Constraints on the Solution Space of Genome-Scale Metabolic Models. Biophysical Journal, 105(2), 512–522. doi:10.1016/j.bpj.2013.06.011

Python Module Index

Index