gsmodutils Documentation

Release 0.0.1

James Gilbert

Oct 24, 2018

User Guide:

`gsmodutils` is a set of utilities that aim to improve the management of large constraints based models geared towards strain design and analysis. Think of it as a friendly framework for developing genome scale models.

The core aim of this software is to make genome scale models, and their applications, reusable in future projects. This is inspired by the notion of *test driven development*.

Changes to models are tracked and pre-written test cases ensure that changes don't break already working components. In addition, by following good practices, well documented genome scale models can be an accompaniment to papers, a core goal of `gsmodutils` is to allow users to share their projects in self contained `docker` images.

Note that this software is under heavy development and much of this may change. Please check back here or at the git hub repository for updates.

It is currently recommended that you install gsmodutils in a `virtualenv` environment.

The most important thing, though, is to not have a model file for every small set of constraints that are added to a model.

E.g. having a directory structure such as this isn't good for anyone. You might add new curation to the base model, or a knockout might exist in one model but not another.

```
$ ls my_project
ecoli_base_model.sbml
ecoli_model_with_xylose_growth.sbml
ecoli_model_with_fructose_growth.sbml
ecoli_model_with_succinate_production.sbml
ecoli_model_with_succinate_production_on_xylose.sbml
...
ecoli_model_with_succinate_production_on_xylose_JUNE_2015_PUBLISHED_VERSION.sbml
```

Instead `gsmodutils` takes the philopsophy that we should be storing the changes (ie. *diffs* or *deltas*) between different versions of models. For this reason, gsmodutils uses simple utilities that let you load model `conditions` and `designs` in python:

```
from gsmodutils import GSMProject
project = GSMProject()
# Just load a model
model = project.load_model()
# Just load some conditions
xylose_growth = project.load_conditions('xylose_growth')
# Load a strain design
succ_prod = project.load_design('succinate_production')
# or load it growing on a different substrate
succ_prod_xyl = project.load_design('succinate_production', conditions='xylose_growth
↪')
```

Or you can export them through the convenient command line utility to export models for use in other tools outside the cobrapy world.

```
$ gsmodutils export matlab succ_production.m --design succinate_production --
↪conditions xylose_growth
```

CHAPTER 1

Installation

To install, the simplest method is to use pip:

```
$ pip install gsmodutils
```

Alternatively, install a development version using a virtual environment

```
$ pip install virtualenv
$ virtualenv ~/gsmodenv
$ source ~/gsmodenv/bin/activate
$ git clone git@bitbucket.org:nottingham_sbrc/gsmodutils.git
$ cd gsmodutils
$ pip install -r requirements.txt
$ pip install -e .
```

If installed in this manner, you must activate the virtualenv whenever you wish to use the tools. For example:

```
$ gsmodutils info
gsmodutils: command not found
$ source ~/gsmodenv/bin/activate
$ gsmodutils info
...
```

gsmodutils has been developed for python 2.7 and python 3.5+. Older versions of python are not supported.

Check your python version with:

```
$ python --version
```

1.1 Mac

If using MacOS and your version of python is outdated, it is recomended that you use hombrew (https://brew.sh) and then install pip:

```
$ brew install python
```

If you have python, but not pip easy_install should work:

```
$ sudo easy_install pip
$ sudo pip install virtualenv
```

1.2 Windows

For windows users, follow the guides on installing python and pip from the python website.

You will also need to install Microsoft Visual C++ Compiler for Python and glpk (used by cobrapy).

It is strongly reccomended that you use git (or mercurial) for managament of genome scale models.

CHAPTER 2

User guides

Project creation is the first step. This can be done as simply

This will prompt the creation of a new project. The initial model in *MODEL_PATH* should be a *cobrapy* compatible model (matlab, sbml, json). For more information on how to use gsmodutils see the guides listed bellow. Many of these processes create files (such as json stores for strain designs), it is strongly advised that you install and use sourch control, such as *git* or *mercurial*, and learn how to use it.

2.1 Projects

Projects are at the heart of gsmodutils. Essentially, a project is a directory that contains all the models, designs, conditions and other gsmodutils tools such as Docker files.

2.1.1 Creating projects

Project creation is best done with the command line utility. This provides a step by step guide for project creation.

```
$ gsmodutils init PROJECT_PATH MODEL_PATH
```

Alternatively, this can be done in python using the GSMProject class.

```
from gsmodutils import GSMProject
from cameo import models

model = models.bigg.e_coli_core

# Note, running more than once will throw an error.
# Projects can't be created in the folder more than once.
project = GSMProject.create_project(
    models=[model],
    description='Example ecoli core model',
    author='A user',
```

(continues on next page)

(continued from previous page)

```
    author_email='A.user@example.com',
    project_path='example_project'
)
```

Using the cli lets have a look inside the newly created project:

```
$ gsmodutils create_project ./example_project e_coli_core.json
...
$ cd example_project
$ gsmodutils info
------------------------------------------------------------------------------------------
↪------------
Project description - Example ecoli core model
Author(s): - A user
Author email - A.user@example.com
Designs directory - designs
Tests directory - tests

Models:
        * e_coli_core.json
                e_coli_core
------------------------------------------------------------------------------------------
↪------------
```

This will also allow access to a project object that can be used to access gsmodutils features, such as accessing models

```
from gsmodutils import GSMProject
project = GSMProject('example_project')
# load the default model
model = project.load_model()
```

It is now recomended that you use source control, such as `git` or `mercurial`, create a repository and add the project to it to track all changes to models that are made over the course of the project.

2.1.2 Adding conditions

In many cases it is desirable to compare models configured with different growth conditions. In this simplistic example we show how conditions can be adjusted by switching the growth media from glucose to fructose. This setting can then be saved and reloaded at a later time.

```
from gsmodutils import GSMProject
# Load existing project
project = GSMProject('example_project')
model = project.model

# Switching off glucose uptake
model.reactions.EX_glc__D_e.lower_bound = 0
# switching on Xylose uptake
model.reactions.EX_fru_e.lower_bound = -10
# Check it works
s = model.optimize()
project.save_conditions(model, 'fructose_growth')
```

Loading them back should now be straightforward:

```
# loading the conditions back into a different model
fructose_m = project.load_conditions('fructose_growth')

# These will raise assertion errors if this hasn't worked
# fructose should be in the medium
# Glucose should not be present
assert "EX_fru_e" in fructose_m.medium
assert "EX_glc__D_e" not in fructose_m.medium
assert fructose_m.medium["EX_fru_e"] == 10
```

2.1.3 GSMProject class

class gsmodutils.project.interface.**GSMProject**(*path='.'*)

Bases: object

add_essential_pathway(*tid*, *reactions*, *description=''*, *reaction_fluxes=None*, *models=None*, *designs=None*, *conditions=None*, *overwrite=False*)

Add a pathway to automated testing to confirm reactions are always present in model and the pathway always carries flux.

This is just a simpler way of adding essential pathways than manually adding a json file with the same information.

Parameters

- **tid** –
- **reactions** –
- **description** –
- **reaction_fluxes** –
- **models** – If None, only the default model is checked unless at least one design is specified
- **designs** – By default designs are never checked. specifying all tests the pathway on all designs
- **conditions** – None, by default. Specifies the conditions which a pathway is present in.
- **overwrite** –

Returns

add_model(*model_path*, *validate=True*)

Add a model given a path to it copy it to model directory unless its in the project path already.

conditions

Different model conditions :return: dict of model conditions

conditions_schema = {'properties': {'growth_conditions': {'type': 'object', 'patter

classmethod create_project(*models*, *description*, *author*, *author_email*, *project_path*)

Creates new projects

Parameters

- **models** – iterable of models can be strings to path locations or cobra model instances.
- **description** – String description of project
- **author** – string author names, separate with &

- **author_email** – email of project owner. separate with ';'
- **project_path** – location on disk to place project. Must not contain existing gsmodutils project.

Returns

design_path

designs
Return list of all the designs stored for the project

get_conditions(*update=False*)
Load the saved conditions file

get_design(*design*)
Get the StrainDesign object (not resulting model) of a design :param design: design identifier :return:

growth_condition(*conditions_id*)

iter_models()
Generator for models

list_conditions

list_designs
List designs stored in design dir

list_models

load_conditions(*conditions_id*, *model=None*, *copy=False*)
Load a model with a given set of pre-saved media conditions :param conditions_id: identifier of conditions file :param model: string or cobrapy model :param copy: return copy of model or modify inplace :return:

load_design(*design*, *model=None*, *copy=False*)
Returns a model with a specified design modification

Design must either be a design stored in the folder path or a path to a json file The json file should conform to the same standard as a json model

required fields: "metabolites":[] "reactions":[] "description": "", "notes": {}, "genes": [], "id":""

optional fields: "parent": str - parent design to be applied first "conditions": "" "removed_reactions":[] "removed_metabolites":[]

Note if conditions is specified it is loaded first other bounds are set afterwards

load_diff(*diff*, *base_model=None*)
Take a diff dictionary and add it to a model (does not require saving a design file)

load_model(*mpath=None*)
Get a model stored by the project mpath refers to the relative path of the model

model
Returns default model for project

models
Lists all the models that can be loaded

project_context_lock
Returns a gloab project lock to stop multiple operations on files. This software is not designed to be used in multiple user environments, so this is slow. However, it provides some degree of protection for the user against modifying the same files

project_path

project_tester()
Creates a tester for this project instance

run_tests()
Returns the log output of all the tests :return:

save_conditions(*model*, *conditions_id*, *carbon_source=None*, *apply_to=None*, *observe_growth=True*)
Add media conditions that a given model has to the project. Essentially the lower bounds on transport reactions. All other trnasport reactions will be switched off.

In some cases, one may wish to set conditions under which a model should not grow. observe_growth allows this to be configured. If using a single model or if the condition should not grow under any circumstances, observe_growth can be set to false.

If certain models should grow, specify this with a a tuple where the entries refer to the model files tracked by the project. All models specified must be contained within the project.

Parameters

- **model** – cobrapy model
- **conditions_id** – identifier for the conditions should be unique
- **carbon_source** – name of carbon source in the media that has a fixed uptake rate
- **apply_to** – iterable of models that this set of designs applies to
- **observe_growth** – bool or list.

Returns

save_design(*model*, *did*, *name*, *description=''*, *conditions=None*, *base_model=None*, *parent=None*, *overwrite=False*)
Creates a design from a diff of model_a and model_b

id should be a string with no spaces (conversion handled)

Returns the saved design diff

Parameters

- **model** – cobrapy model
- **did** – design identifier
- **name** – name of the design
- **description** – text description of what it does
- **conditions** – conditions that should be applied for the design
- **base_model** – Model that the design should be derived from - specified model included in project
- **parent** – string for parent design that this design is a diff from
- **overwrite** – overwrite and existing design (only applies if the id is already in use)

tests_dir
Tests directory

update()
Updates this class from configuration file

2.2 Designs

gsmodutils aims to create convenient interaction with strain designs through use of `designs`. A design is any complex set of constrains that adds or removes reactions or changes the constraints of a model in any way.

The storage of designs within a project is a collection of flat json objects in the designs folder.

The reason for flat files over a database is to allow easy interoperability with revision control software.

Designs can inherit from one another and should, essentially, be considered as the difference between two models.

Where differences are large it may be desirable to have a separate model in the project.

However, for many cases a single model with different strain designs matching real strains stored in a culture collection is enough.

For example, this can be for the addition of hetrologous pathways, or kockouts that are commonly used to improve the productions of desired chemicals.

The Figure below shows the concept of designs in the context of a gsmodutils project.

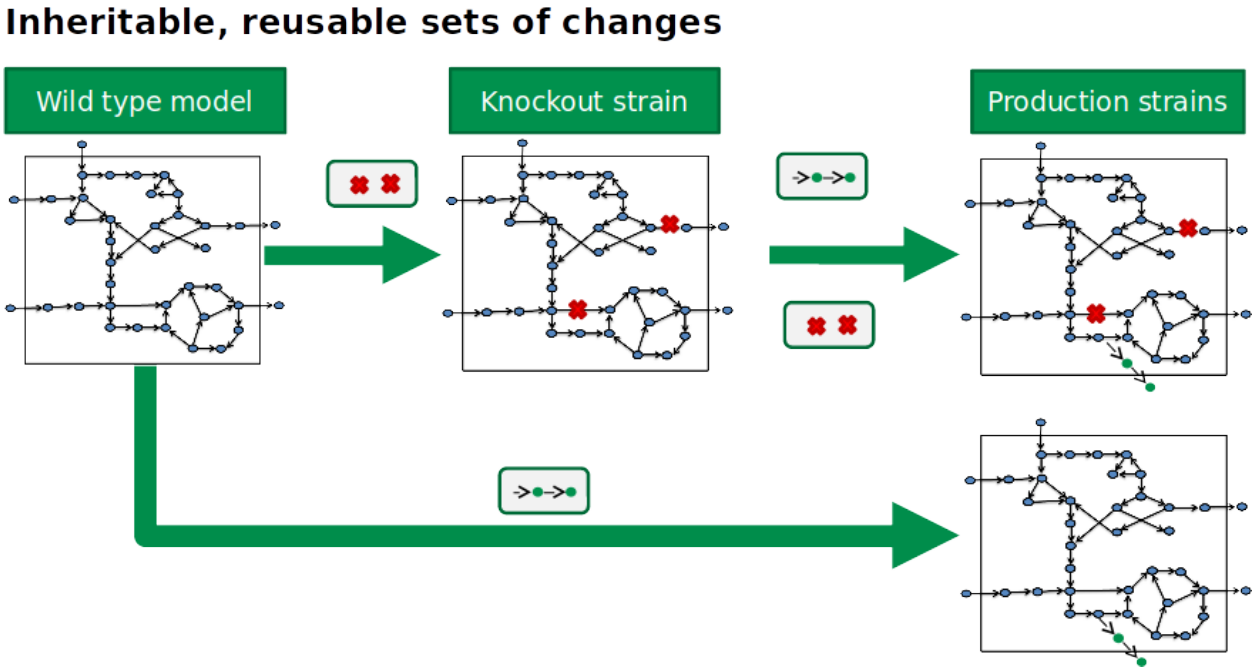

Fig. 1: The objective of the gsmodutils framework is to allow the reuse of constraints in an organised fashion. A design is considered to be the delta difference between a wild type model and any production or modified strains. This can be achieved in an inheritable format. For example, the designs for the addition of hetrologous reactions can be combined with reaction knock-outs. In gsmodutils, any changes to constraints between are stored within a json document inside the project.

2.2.1 Adding designs

There are many cases in which an organism maybe modified in complex ways. In this example we apply some of the work from [1] and add the reactions for the Calvin-Benson (CBB) cycle to the ecoli model. This allows the fixation of CO2 as an inorganic carbon source.

To do this, we need to add two enzymatic reactions to the model Phosphoribulokinase and Rubisco.

```
from gsmodutils import GSMProject
import cobra

# Load existing project
project = GSMProject('example_project')
# Load the default model we want to add
model = project.load_model()

# Phosphoribulokinase reaction
stoich = dict(
    atp_c=-1.0,
    ru5p__D_c=-1.0,
    adp_c=1.0,
    h_c=1.0,
    rb15bp_c=1.0,
)


rb15bp = cobra.Metabolite(id='rb15bp_c', name='D-Ribulose 1,5-bisphosphate', formula=
↪'C5H8O11P2')
model.add_metabolites(rb15bp)

pruk = cobra.Reaction(id="PRUK", name="Phosphoribulokinase reaction", lower_bound=-
↪1000, upper_bound=1000)
model.add_reaction(pruk)
pruk.add_metabolites(stoich)


# Rubisco reaction (Ribulose-bisphosphate carboxylase)
stoich = {
    "3pg_c":2.0,
    "rb15bp_c":-1.0,
    "co2_c":-1.0,
    "h2o_c":-1.0,
    "h_c":2.0
}


rubisco = cobra.Reaction(id="RBPC", lower_bound=0, upper_bound=1000.0, name="Ribulose-
↪bisphosphate carboxylase")

model.add_reaction(rubisco)
rubisco.add_metabolites(stoich)

#show the reactions
pruk
```

```
# now rubisco
rubisco
```

```
# Removed pfkA, pfkB and zwf
model.genes.get_by_id("b3916").knock_out()
model.genes.get_by_id("b1723").knock_out()
model.genes.get_by_id("b1852").knock_out()
```

Now we have added the reactions, we would probably want to make sure they work. To do this we need to change the medium.

```
from cameo.core.utils import medium, load_medium

model.reactions.EX_glc__D_e.lower_bound = -10.0
model.reactions.EX_nh4_e.lower_bound = -1000.0

model.optimize().f
```

```
0.9686322977222491
```

```
design = project.save_design(model, 'cbb_cycle', 'calvin cycle',
                    description='Reactions necissary for the calvin cycle in ecoli',␣
↪overwrite=True)
```

Inherited designs

Now we would like to use the design for production of xylose To do this we will create a child design so we can reuse the calvin cycle without making it part of the wild type ecoli core model.

First, we want to start from the parent calvin cycle design as a base.

```
project = GSMProject('example_project')
# Start from the design as a base model
model = project.load_design('cbb_cycle')
reaction = cobra.Reaction(id="HMGCOASi", name="Hydroxymethylglutaryl CoA synthase")

aacoa = cobra.Metabolite(id="aacoa_c", charge=-4, formula="C25H36N7O18P3S", name=
↪"Acetoacetyl-CoA")
hmgcoa = cobra.Metabolite(id="hmgcoa_c", charge=-5, formula="C27H40N7O20P3S", name=
↪"Hydroxymethylglutaryl CoA")


model.add_metabolites([aacoa, hmgcoa])

stoich = dict(
    aacoa_c=-1.0,
    accoa_c=-1.0,
    coa_c=1.0,
    h_c=1.0,
    h2o_c=-1.0,
    hmgcoa_c=1.0,
)


model.add_reaction(reaction)
reaction.add_metabolites(stoich)
reaction.lower_bound = -1000.0
reaction.upper_bound = 1000.0

reaction
```

```
mev__R = cobra.Metabolite(id="mev__R_c", name="R Mevalonate", charge=-1, formula=
↪"C6H11O4")
model.add_metabolites([mev__R])

reaction = cobra.Reaction(id="HMGCOAR", name="Hydroxymethylglutaryl CoA reductase")
```

(continues on next page)

(continued from previous page)

```
reaction.lower_bound = -1000.0
reaction.upper_bound = 1000.0

stoich = dict(
    coa_c=-1.0,
    h_c=2.0,
    nadp_c=-2.0,
    nadph_c=2.0,
    hmgcoa_c=1.0,
    mev__R_c=-1.0
)

model.add_reaction(reaction)

reaction.add_metabolites(stoich)
reaction
```

```
model.add_boundary(mev__R, type='sink') # add somewhere for mevalonate to go

design = project.save_design(model, 'mevalonate_cbb', 'mevalonate production', parent=
↪'cbb_cycle',
                     description='Reactions for the production of mevalonate',␣
↪overwrite=True)
```

```
des = project.load_design('mevalonate_cbb')
des
```

2.2.2 Python functions as designs

Designs can also be programmatically defined. The use case for this is if there is a specific set of changes to constraints that are done programmatically. For example, changing the stoichiometry of all reactions by some caluculated amount.

To do this create a file in the *designs/* subdirectory of the project called *design_some_name.py*. Only files with the name prefix *design_* will be collected. Then, any functions defined as *design_* will be collected, they must have the function prototype:

```
def gsmdesign_NAME(model, project):
    """ docstrings are used as design descriptions """
    return model
```

The designs must return a cobra.Model instance (and preferably a gsmodutils.project.Model instance). To, optionally, set parent designs set the attributes of the design by modifying the function attributes, as follows.

```
design_NAME.name = "name your design"
design_NAME.description = "Alternatively you can use this field as a description"
design_NAME.parent = "SOME_VALID_PARENT_ID"
design_NAME.conditions = "SOME_VALID_CONDITIONS_ID"
```

When loading a design in python or exporting it, the id will be based on the filename and the function prototype, omitting *design* and *.py*. For the example above in the file *designs/design_some_name.py* the resulting design id is *some_name_NAME*. Be careful to ensure that you do not have clashing namespaces.

Please note, that when doing this make sure that your development environment is secure as gsmodutils will execute the code in these functions.

2.2.3 Accessing designs as models

By default a design can be accessed as a model with

```
project.load_design(<design_id>)
```

This loads the design as a cobra model object.

Using the get_design method allows access to the strain design object.

```
project.get_design(<design_id>)
```

This can also be loaded as an isolated pathway cobra model (though FBA cannot be performed on this object).

```
from gsmodutils import GSMProject
project = GSMProject('example_project')
des = project.get_design('mevalonate_cbb')
des.as_pathway_model()
```

```
des = project.get_design('cbb_cycle')
des.as_pathway_model()
```

Exporting and importing designs and conditions

There are many cases where a particular external piece of software outside the cobrapy stack will be needed for strain design. For this reason the gsmodutils import and export commands aim to allow interoperability with other tool sets.

The objective is to allow users to add or update designs to the project through the command line alone as well as exporting models with the additional constraints that are applied for import in to other tools. Cobrapy makes it easy to work with matlab, sbml, json and yaml constraints based models.

Viewing a project's designs

To view the designs and conditions stored in a project use the `info` command. For the example project it should look something like:

```
--------------------------------------------------------------------------------
↪------------
Project description - Example ecoli core model
Author(s): - A user
Author email - A.user@example.com
Designs directory - designs
Tests directory - tests

Models:
        * iJO1366.json
                iJO1366

Designs:
        * mevalonate_cbb
                mevalonate production
                Reactions for the production of mevalonate
                Parent: cbb_cycle
        * cbb_cycle
                calvin cycle
```

(continues on next page)

(continued from previous page)

```
                Reactions necissary for the calvin cycle in ecoli
Conditions:
        * fructose_growth
--------------------------------------------------------------------------------
↪------------
```

Exporting a design

To export to matlab, sbml, json or yaml formats use `gsmodutils export`:

```
gsmodutils export <output format> <output filepath> --model_id <model id> --
↪conditions <conditions_id> --design <design_id>
```

For example:

```
gsmodutils export mat mevalonate_analysis.mat --design mevalonate_cbb
```

A note on conflicting constraints

When flags for models, designs and conditions are all set the load order of constrains is as follows: * Load model * Set conditions * Load design

This means that if there is a conflict betweeen the constraints set in a conditions file and the design file (i.e. the same transporter may be switched on an off at the same time) the constraint added in the design file takes precidence and is applied to the model.

Importing a new design

To import a new design use the `dimport` command:

```
gsmodutils dimport <path_to_model> <new_id> --base_model <base_model_id> --parent
↪<parent_design_id>
```

The base_model flag is optional, if it is unset the default project modell will be used for the diff.

Before adding a new design it may be desirable to check the diff summary using:

```
gsmodutils diff <path_to_model> --base_model <base_model_id> --parent <parent_design_
↪id>
```

This command shows a summary of the changes the design makes. Using the flag `--no-names` will create a summary that only lists the number of reactions and metabolites modified or added. By default, all changed reactions will be listed by name (though the exact changes are not visible).

Using `--output` allows the diff to be saved as a json file. This can then be imported directly as a design with `dimport` using the `--from-diff` flag.

Modifying an existing design with another tool

Rather than saving a new design, we would like to simply overwrite the existing design. This can be done easily. **However, it is strongly reccommended that you use version control software (such as git or mercurial) to ensure that changes to existing designs are tracked.**

Do this with the command

```
gsmodutils dimport <path_to_updated_model> <design_id> --overwrite
```

Further reading

StrainDesign class

class gsmodutils.project.design.**StrainDesign**(*did*, *name*, *description*, *project*, *parent=None*, *reactions=None*, *metabolites=None*, *genes=None*, *removed_metabolites=None*, *removed_reactions=None*, *removed_genes=None*, *base_model=None*, *conditions=None*, *is_pydesign=False*, *design_func=None*)

Bases: object

add_to_model(*model*, *copy=False*, *add_missing=True*)
Add this design to a given cobra model :param model: :param copy: :param add_missing: add missing metabolites to the model :return:

as_pathway_model()
Loads a cobra model with just the reactions present in this design Can be useful for the cobra.Model methods

TODO: add full metabolite info from parent model (optional, as it will be slower) :return: mdl instance of cobra.Model

check_parents(*p_stack=None*)
Tests to see if their is a loop in parental inheritance

static compile_pydesign(*pyfile*)
Compile a python file and load any gsmodutil design names :param pyfile: :return:

design_schema = {'description': 'JSON representation of gsmodutils designs. Largely ba

classmethod from_dict(*did*, *design*, *project*)

Parameters

- **did** – unique design identifier
- **design** – design dict
- **project** – GSMProject instance

Returns

classmethod from_json(*did*, *file_path*, *project*)
Load from a json file :param did unique design identifier :param file_path: file location of design :param project: GSMProject project instance :return:

classmethod from_pydesign(*project*, *did*, *func_name*, *compiled_code*)
Load a pydesign function as a proper design

genes
Recursively builds set of genes inherited from parents

genes_dataframe()
Return a dataframe of the reactions involved in the design :return:

info

load()
Returns a cobra model containing the parent model with the design applied :return:

metabolites
Recursively builds set of metabolites inherited from parents

metabolites_dataframe()
Return a dataframe of the reactions involved in the design :return:

reactions
Recursively builds set of reactions inherited from parents

reactions_dataframe()
Return a dataframe of the reactions involved in the design :return:

removed_genes
Recursively builds set of removed genes inherited from parents

removed_metabolites
Recursively builds set of removed metabolites inherited from parents

removed_reactions
Recursively builds set of removed reactions inherited from parents

to_dict()
Converts to design dict (compatible with model diffs) :return:

to_json(*file_path*, *overwrite=False*)
Write to a given file

static validate_dict(*design_dict*, *throw_exceptions=True*)
Check required fields are present :param design_dict: :param throw_exceptions: Throw json schema exceptions. If false, returns bool on any exception :return:

2.3 cli

Much of the interaction a user will want to do with gsmodutils will be through the command line tools. This section gives a brief overview of these tools, with simple examples that should be applicable to most users.

2.3.1 gsmodutils

Command line tools for management of gsmodutils genome scale model projects

```
gsmodutils [OPTIONS] COMMAND [ARGS]...
```

addmodel

Add a model to a specified gsm project. If validation is selected, where the model fails to conform to cobra standards, the model will not be added to the project.

```
gsmodutils addmodel [OPTIONS] PATH
```

Options

--project_path <project_path>
gsmodutils project path

--validate, --no-validate
Chose to validate the model before it is added.

Arguments

PATH
Required argument

diff

View the changed reactions between a model and a base model

```
gsmodutils diff [OPTIONS] MODEL_PATH
```

Options

--base_model <base_model>
Project model to compare with

--project_path <project_path>
gsmodutils project path

--parent <parent>
A parent design

--output <output>
A location to output the diff as a sjon file

--names, --no-names
Output names of added or changed metabolites and reactions

Arguments

MODEL_PATH
Required argument

dimport

Import a design into a model. This can be new or overwrite an existing design.

```
gsmodutils dimport [OPTIONS] MODEL_PATH IDENTIFIER
```

Options

--name <name>
Formal design name (longer than identifier)

--description <description>
Description of what the design does

--project_path <project_path>
gsmodutils project path

--parent <parent>
A parent design that was applied first to avoid replication.

--base_model <base_model>
Model that design is based on

--overwrite, --no-overwrite
overwrite existing design

--from_diff, --not_from_diff
load a diff file instead of compatible model

Arguments

MODEL_PATH
Required argument

IDENTIFIER
Required argument

docker

Create a dockerfile for the project

```
gsmodutils docker [OPTIONS]
```

Options

--project_path <project_path>
gsmodutils project path

--overwrite, --no-overwrite
overwrite existing dockerfile

--build, --no-build
build docker container

--save, --no-save
save docker image of tagged container

--tag <tag>
tag name for docker container (appended to project name).

--save_path <save_path>
Save path for shared docker image

export

Export a given model with a specific design and conditions applied

```
gsmodutils export [OPTIONS] [json|yaml|sbml|matlab|mat|m|scrumpy|spy] FILEPATH
```

Options

--project_path <project_path>
gsmodutils project path

--model_id <model_id>
model id

--conditions <conditions>
conditions to apply

--design <design>
design to apply

--overwrite, --no-overwrite
model id

Arguments

FILE_FORMAT
Required argument

FILEPATH
Required argument

iconditions

Add a given set of media condtions from a model (this ignores any added or removed reactions or metabolites)

```
gsmodutils iconditions [OPTIONS] PATH IDENT
```

Options

--project_path <project_path>
gsmodutils project path

--apply_to <apply_to>
Description of what the design does

--growth, --no_growth
Should these conditions allow growth or not

Arguments

PATH
Required argument

IDENT
Required argument

info

Display all the information about a gsmodutils project (list models, paths, designs etc.

```
gsmodutils info [OPTIONS]
```

Options

--project_path <project_path>
gsmodutils project path

init

Create a new gsmodutils project

```
gsmodutils init [OPTIONS] PROJECT_PATH DEFAULT_MODEL_PATH
```

Options

--name <name>

--description <description>
Project description

--author <author>
Project name

--email <email>
Author email

--add_models <add_models>
paths to additional model files, separated by space

--validate, --skip_validation
Require the model to be validated

Arguments

PROJECT_PATH
Required argument

DEFAULT_MODEL_PATH
Required argument

test

Run tests for a project

```
gsmodutils test [OPTIONS]
```

Options

--project_path <project_path>
gsmodutils project path

--test_id <test_id>
specify a given test identifier to run - pyton filename, function orjson_filename entry. Individual tests separated by double colons -

--skip_default, --no_skip_default
skip default tests

--verbose, --no_verbose
Display succesfully run test assertions

--log_path <log_path>
path to output json test log

2.4 Testing

A genome scale model is almost never finished. Thousands of hours of manual curation can go in to models and, as a result, changes can break things. For this reason it is good practice to work in a *test driven* manner. Creating good test cases ensures that if a model once meets criteria for experimental validation it always meets this criteria.

2.4.1 Running the default tests

A test report can be generated by the tester. This is a command line utility which, by default, loads each model, set of conditions and design and performs the FBA simulations. This ensures that any changes to the project files maintain designs, models and conditions.

```
$ gsmodutils test
```

The output from the terminal should look something like this:

```
------------------------ gsmodutils test results -------------------------
Running tests: ....
Default project file tests (models, designs, conditions):
Counted 4 test assertions with 0 failures
Project file completed all tests without error
    --model::e_coli_core.json

    --design::mevalonate_calvin

    --design::mevalonate_cbb

    --design::cbb_cycle

Ran 4 test assertions with a total of 0 errors (100.0% success)
```

2.4.2 Custom tests

Whilst the default tests provided by the tool are a useful way of ensuring that the project files remain valid after manual curation they do not have the capability to match all design goals. These design goals will be based on a data driven approach to genome scale model development and often require a more fundamental understanding of how an organism functions.

The simplest way to do this is in python

```
from gsmodutils import GSMProject
project = GSMProject('./') # insert path to project

reactions = ["RXID_1", ...] # List of essential reactions

flux = dict(PYR=[0.5, 1000]) # Required flux for a given reaction id

project.add_essential_pathway('pathway_x', description='Example pathway',␣
↪reactions=reactions, reaction_fluxes=flux)
```

This will create a file `tests/test_pathway_x.json`. Alternatively, tests can be created by adding json files to the tests directory as long as they are of the form `test_NAME.json`. These json files have the following required fields:

```
conditions - JSON array (list of project conditions to be loaded and tested)
models - JSON array (list of project models to be loaded and tested)
designs - JSON array (list of project design ids to be loaded and tested)
reaction_fluxes - JSON associative array
required_reactions - JSON array
description - JSON string
```

For example the file `tests/test_example.json` might look like

```
{
    'conditions':[],
    'models':[],
    'designs':['my_pathway_01'],
    'reaction_fluxes': {
        'Biomass': [0.21, 1000]
    }
    'required_reactions': ['reaction_1'],
    'description': 'Make sure reaction 1 is carrying flux. Make sure Biomass is above␣
↪0.21'
}
```

This will add a test to be run that ensures that a reaction with the id `reaction_1` carries flux and that the flux accross the biomass is above `0.21` with the design `my_pathway_01`.

This will automatically be picked up by `gsmodutils test` and run accordingly. Note, if the files are badly formatted tests will not run and will throw an error.

2.4.3 Writing python test cases

For many use cases, it may require the use of more complex functionality. For this reason, gsmodutils allows users to write fully featured python test cases. This means that any code written in python can be used and assertion statements can be written and included in the test reports.

Any file of the format `tests/test_*.py` will be included in the test cases run by the project tester instance.

Only functions of the prototype `def func_<name>(model, project, log)` will be called by the tester.

For test cases use the method `log.assertion(<bool: statement>, <str: success>, <str:failure>)` to record the result of a given test assertion.

`log` is always an instance of the `gsmodutils.testutils.TestRecord` class.

For example, create the python module `tests/test_my_model.py` and then add the code:

```
def test_model(model, project, log):
    solution = model.solver.optimize()
    log.assertion(solution.f > 0.0, "Model grows", "Model does not grow")
```

When creating tests the class `gsmodutils.test.utils.ModelTestSelector` can be used as a helper decorator to load models and designs of specific names. The same test function will be called repeatedly with all combinations of models, designs and conditions specified.

```
from gsmodutils.test.utils import ModelTestSelector

@ModelTestSelector(models=[], conditions=[], designs=[])
def test_func(model, project, log):
    log.assertion(True, "Works", "Does not work", "Test")
```

As with json tests these will be picked up automatically by `gsmodutils test`. Any logs to standard out (e.g. using python print) can also be captured with this approach. Note that this should not be used in all environments as this will allow any code to be executed, malicious or not.

2.4.4 Performing tests on loaded models

If you have an in memory Model object that has been modified, gsmodutils supports running existing tests on this model. For example,

```
from gsmodutils import project
project = GSMProject()
# Load a model or design
model = project.load_design("<some_design>")
# make some changes
model.reactions.get_by_id("SOME_REACTION").knock_out()
model.run_tests()
```

The progress and results for tests that are only those that are applied to the given loaded model or design.

Please note, that tests for designs that are downstream of a given model (i.e. designs based on this model or child designs) will not be run in the setting. Testing of this requires the changes to be saved to disk in order for them to be loaded in to other designs/models within the project.

Further reading

Test utilities

Utilities for test functions

This code assumes that the namespace that python based tests are in is correctly assigned, otherwise global variables won't be present and this will throw errors

class gsmodutils.test.utils.**ModelLoader**(*project*, *model_id*, *conditions_id*, *design_id*)

Bases: object

load(*log*)

class gsmodutils.test.utils.**ModelTestSelector**(*models=None*, *conditions=None*, *designs=None*)

Bases: object

class gsmodutils.test.utils.**ResultRecord**(*tid=''*, *parent=None*, *param_child=False*)

Bases: object

Class for handling logging of errors in tester follows a hierarchical pattern as log records allow child records This is a bit of a weird data structure but the objective is to (in a future version) encapsulate all tests inside an instance of Test Record

add_error(*msg*, *desc=''*)

For errors loading tests, e.g. success cases can't be reached because the model doesn't load or can't get a feasable solution

assertion(*statement*, *success_msg*, *error_msg*, *desc=''*)

Called within test functions to store errors and successes Results will be appended to the correct log reccords

create_child(*new_id*, *param_child=False*)

Used within decorator helper functions to allow multiple tests with the same function but where other parameters change

is_success

The test function is considered a failure if there are one or more error logs

log_count

count total errors for self and children

to_dict(*stk=None*)

converts log into dictionary form for portability stk stops cyclic behaviour

warning(*statement*, *message*, *desc=''*)

Called within test functions to capture warnings about the status of models. If statement is true, the warning message will be stored.

gsmodutils.test.utils.**stdout_ctx**(*stdout=None*)

Context to capture standard output of python executed tests during run time This is displayed to the user for them to see after the tests are run

GSMtester class

class gsmodutils.test.tester.**GSMTester**(*project*)

Bases: object

collect_tests()

Collects all tests but does not run them

get_test(*tid*)

iter_tests(*recollect=False*)

progress_tests(*skip_default=False*)

Run tests with a progressbar :param skip_default: :return:

run_all()

Find and run all tests for a project, executes rather than returning generator

run_by_id(*tid*)

Returns result of individual test function

test_ids

to_dict()
json serialisable log - call after running tests

2.5 Docker

This section describes how to share a gsmodutils project inside a docker container.

The objective is to have code for testing models, data for their validation and the framework for running models written in a "write once run anywhere" fashion.

Ideally, even if a platform for running a model is considerably out of date it should produce the same results on new software.

Unfortunately this isn't always the case. Packaging a constraints based model, with associated software within the same working environment ensures that the results should be reproducable, providing other users can install docker containers.

To install docker on your system please consult the documentation at https://docs.docker.com

2.5.1 Creating a docker container

To create a new docker container (with the latest gsmodutils setup scripts, use the docker utility.

```
$ gsmodutils docker --overwrite
```

This will overwrite an existing `Dockerfile` and `requirements.txt`. If you have additional software requirements (such as python packages) you will want to modify these. If you have large data files in your project that do not need to be shared, consider editing your `Dockerfile` to ignore these to reduce the size of any resulting images.

Now to create a docker container use the build command

```
$ docker build -t="example_project" <path_to_gsmodutils_project>
```

This will install the required python components for gsmodutils.

2.5.2 Running a docker container

gsmodutils just provides the basic dockerfile. In future iterations this may be changes. To use code inside a docker image, run the command

```
$ docker run -it <command>
```

For example, to load an ipython shell run:

```
$ docker run -it example_project ipython
```

You might wish to get the project info:

```
$ docker run -it example_project gsmodutils info
------------------------------------------------------------
Project description - Example ecoli core model
Author(s): - A user
```

(continues on next page)

(continued from previous page)

```
Author email - A.user@example.com
Designs directory - designs
Tests directory - tests

Models:
 * iJO1366.json
   iJO1366

Designs:
 * cbb_cycle
   calvin cycle
   Reactions necissary for the calvin cycle in ecoli
 * mevalonate_cbb
   mevalonate production
   Reactions for the production of mevalonate
   Parent: cbb_cycle
Conditions:
 * fructose_growth
------------------------------------------------------------
```

Another example might be to run the gsmodutils test inside this portable environement:

```
$ docker run -it example_project gsmodutils test
------------------------- gsmodutils test results -------------------------
Running tests: ....
Default project file tests (models, designs, conditions):
Counted 4 test assertions with 0 failures
Project file completed all tests without error
    --model_iJO1366.json

    --conditions_iJO1366.json:model_fructose_growth

    --design_cbb_cycle

    --design_mevalonate_cbb

Ran 4 test assertions with a total of 0 errors (100.0% success)
```

2.5.3 Sharing and loading gsmodutils docker images

To share a project with users, first build it following the steps above. When the project is built use the command:

```
$ docker save example_project -o example_project.tar
```

This saves the sharable docker container. When this tarball is transferred to another user, they can load the image with the command:

```
$ docker load -i example_project.tar
```

The imported image will then allow the above commands to run with the same environmental settings. This should allow you to share your models in a way that allows results to be replicated without worrying about the software.

2.6 About project

This project was developed as part of the Synthetic Biology Research Centre at the University of Nottingham.

This work was supported by the UK Biotechnology and Biological Sciences Research Council (BBSRC) grants BB/L013940/1, BB/K00283X/1 and BB/L502030/1.

2.7 Model diffs

When making changes to a model within `gsmodutils` the `gsmodutils.GSModuitlsModel` instances (which is subclass of `cobra.Model`) allows you to track the differences between the in memory model and the model or design saved on disk. For example:

```
import gsmodutils
project = gsmodutils.GSMProject()
model = project.load_model()

model.remove_reactions([model.reactions.RXN_01])

diff = model.diff()
diff
```

Diff is gsmodutils.model_diff.ModelDiff object which is just a subclass of a python dictionary. If working within an jupyter notebook diff will display the model changes in HTML.

If working outside of as gsmodutils project with cobra models use:

```
from gsmodutils.model_diff import ModelDiff
diff = ModelDiff.model_diff(model_a, model_b)
diff
```

2.7.1 ModelDiff class

class gsmodutils.model_diff.**ModelDiff**(*model_a=None*, *model_b=None*, **args*, ***kwargs*)

Bases: dict

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys()
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

static model_diff(*model_a*, *model_b*)
Returns a dictionary that contains all of the changed reactions between model a and model b This includes any reactions or metabolites removed, or any reactions or metabolites added/changed This does not say HOW a model has changed if reactions or metabolites are changed they are just included with their new values

Diff assumes l -> r (i.e. model_a is the base model) :param cobra.Model model_a: :param cobra.Model model_b: :return:

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

gsmodutils.model_diff.**model_diff**(*model_a*, *model_b*)
@depricated - use ModelDiff.model_diff(model_a, model_b) Returns a ModelDiff dictionary that contains all of the changed reactions between model a and model b This includes any reactions or metabolites removed, or any reactions or metabolites added/changed This does not say HOW a model has changed if reactions or metabolites are changed they are just included with their new values Diff assumes l -> r (i.e. model_a is the base model)

CHAPTER 3

Other utilities

gsmodutils also includes utilities for genome scale model managament. Particularly the parsing of ScrumPy files (constraints based models developed at oxford brookes university).

3.1 MetaCyc

This module provides a set of utilities for handling metacyc databases. This code does not provide access to the metacyc/biocyc databases and is simply a parser. For access to data consult metacyc for the acquisition of a valid licence.

3.1.1 Example usage

The intended uscase for this code is to allow the addition of pathways to models that use metacyc identifiers. The example bellow adds an enzyme to a model using EC identifiers.

```
from gsmodutils import GSMProject
from gsmodutils import metacyc
db = metacyc.parse_db('/path/to/metacyc/dat/files')
project = GSMProject()

model = project.load_model()

metacyc.add_pathway(model, ["EC-1.2.1.10"])
```

3.1.2 Code docs

class gsmodutils.utils.metacyc.**FileEncodingCtx**(*filename*, *encoding='latin-1'*, ***kwargs*)

Bases: object

gsmodutils.utils.metacyc.**add_pathway**(*model*, *enzyme_ids=None*, *reaction_ids=None*, *compartment='c'*, *db_path=None*, *copy=False*)
For a given model add enzymes from metacyc database :param model: cobra model object :param enzyme_ids: list of EC entires in format ["EC-x.x.x.x", ...] :param reaction_ids: set of reaction ids to add to model :param compartment: compartment pathway is in :param db_path: path to the metacyc dat files on disk :param copy: :return:

gsmodutils.utils.metacyc.**add_reaction**(*model*, *reaction_id*, *db*, *compartment='c'*)
Add a metacyc reaction id to a cobrapy model :param model: cobrapy model instance :param reaction_id: reaction identifier in metacyc :param db: dictionary db object :param compartment: compartment reactions takeplace in (default is "c") :return: tuple(reaction, added_metabolites) cobrapy Reaction and Metabolite instances

gsmodutils.utils.metacyc.**build_universal_model**(*path*, *use_cache=True*, *cache_location='.metacyc_universal.json'*)
Constructs a universal model from all the reactions in the metacyc database

Parameters

- **path** – path to folder containing metacyc dat files
- **use_cache** – optionally store the resulting model in cached form

Returns

gsmodutils.utils.metacyc.**get_enzyme_reactions**(*eid*, *db*)
For a given ec number return associated reaction ids :param eid: enzyme id format "EC-x.x.x.x" :param db: database dict :return:

gsmodutils.utils.metacyc.**parse_db**(*db_path*)
Parse metacyc dat files to build dict containing entries

gsmodutils.utils.metacyc.**parse_metacyc_file**(*fpath*, *unique_fields*)
Parses a dat file :str fpath: path to dat file :list unique_fields: list of fields that there should only be a single item of in each entry :return:

3.2 ScrumPy

The ScrumPy modelling software is developed at Oxford Brookes univeristy by the cell systems modelling group:

http://mudshark.brookes.ac.uk/ScrumPy

http://mudshark.brookes.ac.uk/

This documentation is for the `scrumpy_to_cobra` utility and associated functions. Gsmodutils is capable of converting ScrumPy structural models to cobrapy objects. However, it should be noted that additional constraints on reactions are not specified within the ScrumPy modelling format. As a consequence, these will have to be specified manually (or through supported json formats).

3.2.1 Example usage

```
$ scrumpy_to_cobra --model SCRUMPY_FILE.spy --output OUTPUT_FILE.json
```

As scrumpy spy files do not include constraints for models, the following options are probably required to get a working model

```
--media a json file for growth media

--atpase_reaction

--atpase_flux

--objective_reaction Objective to maximise (multiple objectives currently not set)

--objective_direction
```

Alternatively, use the python interface to load a model and set the constraints with the cobrapy interface

```
from gsmodutils.utils.scrumpy import load_scrumpy_model
cobra_mdl = load_scrumpy_model('model.spy')
cobra_mdl.objective = cobra_mdl.reactions.Biomass
cobra.mdl.reactions.ATPase.lower_bound = -8.0
cobra.mdl.reactions.ATPase.upper_bound = -8.0
```

Scrumpy formatted strings can also be loaded in to gsmodutils models on the fly. For example, after a gsmodutils project model is loaded:

```
from gsmodutils import GSMProject
project = GSMProject()
model = project.model

spy_reactions = """
External(PROTON_i, "WATER")

NADH_DH_ubi:
    "NADH" + "UBIQUINONE-8" + 4 PROTON_i -> "UBIQUINOL-8" + 3 PROTON_p + "NAD"
    ~

NADH_DH_meno:
    "NADH" + "Menaquinones" + 4 PROTON_i -> "Menaquinols" + 3 PROTON_p + "NAD"
    ~
"""
model.add_scrumpy_reactions(spy_reactions)
```

A further usage is to load cobra models directly from scrumpy strings:

```
from gsmodutils.utils.scrumpy import load_scrumpy_model
spy_reactions = """
Structural()
External(PROTON_i, "WATER")

NADH_DH_ubi:
    "NADH" + "UBIQUINONE-8" + 4 PROTON_i -> "UBIQUINOL-8" + 3 PROTON_p + "NAD"
    ~

NADH_DH_meno:
    "NADH" + "Menaquinones" + 4 PROTON_i -> "Menaquinols" + 3 PROTON_p + "NAD"
    ~
"""
model = load_scrumpy_model(spy_reactions)
```

Naturally, any constraints additional to reaction directionality (such as uptake) will have to be specified manually.

3.2.2 Code docs

exception gsmodutils.utils.scrumpy.**ParseError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

gsmodutils.utils.scrumpy.**get_tokens**(*line*)
Goes through each charachter in scrumpy file attempting to find tokens

FIXME: if there is a numeric after a direction token this fails e.g. '->2 "PROTON"' fails but '-> 2 "PROTON"' works :param line_dt: :return:

gsmodutils.utils.scrumpy.**load_scrumpy_model**(*filepath_or_string*, *name=None*, *model_id=None*, *media=None*, *objective_reactions=None*, *obj_dir='min'*, *fixed_fluxes=None*)
Specify a base scrumpy structural model file and returns a cobra model. This hasn't be thoroughly tested so expect there to be bugs

To get a solution from the returned object you need to specify nice stuff like the atpase reaction and media

Parameters

- **filepath_or_string** – filepath or scrumpy string
- **name** –
- **model_id** –
- **media** –
- **objective_reactions** –
- **obj_dir** –
- **fixed_fluxes** –

Returns

gsmodutils.utils.scrumpy.**parse_file**(*filepath*, *fp_stack=None*, *rel_path=''*)

Recursive function - takes in a scrumpy spy file and parses it, returning a set of reactions

Note this code is not fully tested. Expect some bugs. :param filepath: :param fp_stack: :param rel_path: :return:

gsmodutils.utils.scrumpy.**parse_fobj**(*infile*, *fp_stack*, *rel_path*, *source_name*)

gsmodutils.utils.scrumpy.**parse_string**(*spy_string*, *rel_path='.'*)

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

CHAPTER 5

References

[1] Antonovsky, N., Gleizer, S., Noor, E., Zohar, Y., Herz, E., Barenholz, U., Zelcbuch, L., Amram, S., Wides, A., Tepper, N. and Davidi, D., 2016. Sugar synthesis from CO 2 in Escherichia coli. Cell, 166(1), pp.115-125.

[2] Ebrahim, Ali, Joshua A. Lerman, Bernhard O. Palsson, and Daniel R. Hyduke. "COBRApy: COnstraints-based reconstruction and analysis for python." BMC systems biology 7, no. 1 (2013): 74.

[3] Cardoso, Joao, Kristian Jensen, Christian Lieven, Anne Sofie Laerke Hansen, Svetlana Galkina, Moritz Emanuel Beber, Emre Ozdemir, Markus Herrgard, Henning Redestig, and Nikolaus Sonnenschein. "Cameo: A Python Library for Computer Aided Metabolic Engineering and Optimization of Cell Factories." bioRxiv (2017): 147199.

Python Module Index

g

Index

Symbols

A

I

K

L

M

P

R

S

T

U

V

W