cobra Documentation

Release 0.3.2

Daniel Robert Hyduke and Ali Ebrahim

June 29, 2015

Contents

1 Getting Started **3**
- 1.1 Reactions . . . 4
- 1.2 Metabolites . . . 5
- 1.3 Genes . . . 6

2 Building a Model **9**

3 Reading and Writing Models **13**
- 3.1 JSON . . . 13
- 3.2 SBML . . . 13
- 3.3 MATLAB . . . 14
- 3.4 Pickle . . . 14

4 Simulating with FBA **17**
- 4.1 Running FBA . . . 17
- 4.2 Changing the Objectives . . . 18

5 Simulating Deletions **19**
- 5.1 Single Deletions . . . 19
- 5.2 Double Deletions . . . 19

6 Phenotype Phase Plane **21**

7 Mixed-Integer Linear Programming **25**
- 7.1 Ice Cream . . . 25
- 7.2 Restaurant Order . . . 26
- 7.3 Boolean Indicators . . . 27

8 Quadratic Programming **29**

9 Loopless FBA **33**

10 FAQ **37**
- 10.1 How do I install cobrapy? . . . 37
- 10.2 How do I cite cobrapy? . . . 37
- 10.3 How do I rename reactions or metabolites? . . . 37
- 10.4 How do I delete a gene? . . . 38
- 10.5 How do I change the reversibility of a Reaction? . . . 38
- 10.6 How do I generate an LP file from a COBRA model? . . . 38

10.7 How do I visualize my flux solutions? 39

11 cobra package **41**
11.1 Subpackages 41
11.2 Module contents 65

12 Indices and tables **67**

Python Module Index **69**

For installation instructions, please see INSTALL.md.

CHAPTER 1

Getting Started

This example is available as an IPython notebook.

To begin with, cobrapy comes with two bundled models for *Salmonella* and *E. coli*. To load a test model, type

```
from __future__ import print_function
import cobra.test
model = cobra.test.create_test_model("salmonella")
```

The reactions, metabolites, and genes attributes of the cobrapy model are are a special type of list called a DictList, and each one is made up of Reaction, Metabolite and Gene objects respectively.

```
print(len(model.reactions))
print(len(model.metabolites))
print(len(model.genes))
```

```
2546
1802
1264
```

Just like a regular list, objects in the DictList can be retrived by index. For example, to get the 30th reaction in the model (at index 29 because of 0-indexing):

```
model.reactions[29]
```

```
<Reaction 2AGPA180tipp at 0x7f227ada62d0>
```

Addictionally, items can be retrived by their id using the get_by_id() function. For example, to get the cytosolic atp metabolite object (the id is "atp_c"), we can do the following:

```
model.metabolites.get_by_id("atp_c")
```

```
<Metabolite atp_c at 0x7f227adf56d0>
```

As an added bonus, users with an interactive shell such as IPython will be able to tab-complete to list elements inside a list. While this is not recommended behavior for most code because of the possibility for characters like "-" inside ids, this is very useful while in an interactive prompt:

```
model.reactions.EX_glc__D_e.lower_bound
```

```
0.0
```

1.1 Reactions

We will consider the reaction glucose 6-phosphate isomerase, which interconverts glucose 6-phosphate and fructose 6-phosphate. The reaction id for this reaction in our test model is PGI.

```
pgi = model.reactions.get_by_id("PGI")
pgi
```

```
<Reaction PGI at 0x7f227a10b1d0>
```

We can view the full name and reaction catalyzed as strings

```
print(pgi.name)
print(pgi.reaction)
```

```
glucose 6 phosphate isomerase
g6p_c <=> f6p_c
```

We can also view reaction upper and lower bounds. Because the pgi.lower_bound < 0, and pgi.upper_bound > 0, pgi is reversible

```
print(pgi.lower_bound, "< pgi <", pgi.upper_bound)
print(pgi.reversibility)
```

```
-1000.0 < pgi < 1000.0
True
```

We can also ensure the reaction is mass balanced. This function will return elements which violate mass balance. If it comes back empty, then the reaction is mass balanced.

```
pgi.check_mass_balance()
```

```
[]
```

In order to add a metabolite, we pass in a dict with the metabolite object and its coefficient

```
pgi.add_metabolites({model.metabolites.get_by_id("h_c"): -1})
pgi.reaction
```

```
'g6p_c + h_c <=> f6p_c'
```

The reaction is no longer mass balanced

```
pgi.check_mass_balance()
```

```
['PGI', {'C': 0.0, 'H': -1.0, 'O': 0.0, 'P': 0.0}]
```

We can remove the metabolite, and the reaction will be balanced once again.

```
pgi.pop(model.metabolites.get_by_id("h_c"))
print(pgi.reaction)
print(pgi.check_mass_balance())
```

```
g6p_c <=> f6p_c
[]
```

It is also possible to build the reaction from a string. However, care must be taken when doing this to ensure reaction id's match those in the model. The direction of the arrow is also used to update the upper and lower bounds.

```
pgi.reaction = "g6p_c --> f6p_c + h_c + green_eggs + ham"
```

```
unknown metabolite 'green_eggs' created
unknown metabolite 'ham' created
```

```
pgi.reaction
```

```
'g6p_c --> f6p_c + green_eggs + ham + h_c'
```

1.2 Metabolites

We will consider cytosolic atp as our metabolite, which has the id atp_c in our test model.

```
atp = model.metabolites.get_by_id("atp_c")
atp
```

```
<Metabolite atp_c at 0x7f227adf56d0>
```

We can print out the metabolite name and compartment (cytosol in this case).

```
print(atp.name)
print(atp.compartment)
```

```
ATP
c
```

We can see that ATP is a charged molecule in our model.

```
atp.charge
```

```
-4
```

We can see the chemical formula for the metabolite as well.

```
print(atp.formula)
```

```
C10H12N5O13P3
```

The reactions attribute gives a frozenset of all reactions using the given metabolite. We can use this to count the number of reactions which use atp.

```
len(atp.reactions)
```

```
348
```

A metabolite like glucose 6-phosphate will participate in fewer reactions.

```
model.metabolites.get_by_id("g6p_c").reactions
```

```
frozenset({<Reaction AB6PGH at 0x7f2279de6a50>,
           <Reaction TRE6PH at 0x7f2279fdb110>,
           <Reaction TRE6PS at 0x7f2279fdb3d0>,
           <Reaction PGI at 0x7f227a10b1d0>,
           <Reaction PGMT at 0x7f227a10b750>,
           <Reaction HEX1 at 0x7f227a368a90>,
           <Reaction GLCptspp at 0x7f227a3d2710>,
           <Reaction G6PDH2r at 0x7f227a3fd850>,
```

```
            <Reaction G6PP at 0x7f227a3fda50>,
            <Reaction G6Pt6_2pp at 0x7f227a3fdb10>})
```

1.3 Genes

The gene_reaction_rule is a boolean representation of the gene requirements for this reaction to be active as described in Schellenberger et al 2011 Nature Protocols 6(9):1290-307.

The GPR is stored as the gene_reaction_rule for a Reaction object as a string.

```
gpr = pgi.gene_reaction_rule
gpr
```

```
'STM4221'
```

Corresponding gene objects also exist. These objects are tracked by the reactions itself, as well as by the model

```
pgi.genes
```

```
frozenset({<Gene STM4221 at 0x7f227a10b250>})
```

```
pgi_gene = model.genes.get_by_id("STM4221")
pgi_gene
```

```
<Gene STM4221 at 0x7f227a10b250>
```

Each gene keeps track of the reactions it catalyzes

```
pgi_gene.reactions
```

```
frozenset({<Reaction PGI at 0x7f227a10b1d0>})
```

Altering the gene_reaction_rule will create new gene objects if necessary and update all relationships.

```
pgi.gene_reaction_rule = "(spam or eggs)"
pgi.genes
```

```
frozenset({<Gene eggs at 0x7f2279d7ec90>, <Gene spam at 0x7f2279d7edd0>})
```

```
pgi_gene.reactions
```

```
frozenset()
```

Newly created genes are also added to the model

```
model.genes.get_by_id("spam")
```

```
<Gene spam at 0x7f2279d7edd0>
```

The delete_model_genes function will evaluate the gpr and set the upper and lower bounds to 0 if the reaction is knocked out. This function can preserve existing deletions or reset them using the cumulative_deletions flag.

```
cobra.manipulation.delete_model_genes(model, ["spam"], cumulative_deletions=True)
print(pgi.lower_bound, "< pgi <", pgi.upper_bound)
cobra.manipulation.delete_model_genes(model, ["eggs"], cumulative_deletions=True)
print(pgi.lower_bound, "< pgi <", pgi.upper_bound)
```

```
0 < pgi < 1000
0.0 < pgi < 0.0
```

The undelete_model_genes can be used to reset a gene deletion

```
cobra.manipulation.undelete_model_genes(model)
print(pgi.lower_bound, "< pgi <", pgi.upper_bound)
```

```
0 < pgi < 1000
```

CHAPTER 2

Building a Model

This simple example (available as an IPython notebook) demonstrates how to create a model, create a reaction, and then add the reaction to the model.

We'll use the '3OAS140' reaction from the STM_1.0 model:

1.0 malACP[c] + 1.0 h[c] + 1.0 ddcaACP[c] → 1.0 co2[c] + 1.0 ACP[c] + 1.0 3omrsACP[c]

First, create the model and reaction.

```
from cobra import Model, Reaction, Metabolite
# Best practise: SBML compliant IDs
cobra_model = Model('example_cobra_model')

reaction = Reaction('3OAS140')
reaction.name = '3 oxoacyl acyl carrier protein synthase n C140 '
reaction.subsystem = 'Cell Envelope Biosynthesis'
reaction.lower_bound = 0.  # This is the default
reaction.upper_bound = 1000.  # This is the default
reaction.objective_coefficient = 0. # this is the default
```

We need to create metabolites as well. If we were using an existing model, we could use get_by_id to get the appropriate Metabolite objects instead.

```
ACP_c = Metabolite('ACP_c', formula='C11H21N2O7PRS',
    name='acyl-carrier-protein', compartment='c')
omrsACP_c = Metabolite('3omrsACP_c', formula='C25H45N2O9PRS',
    name='3-Oxotetradecanoyl-acyl-carrier-protein', compartment='c')
co2_c = Metabolite('co2_c', formula='CO2', name='CO2', compartment='c')
malACP_c = Metabolite('malACP_c', formula='C14H22N2O10PRS',
    name='Malonyl-acyl-carrier-protein', compartment='c')
h_c = Metabolite('h_c', formula='H', name='H', compartment='c')
ddcaACP_c = Metabolite('ddcaACP_c', formula='C23H43N2O8PRS',
    name='Dodecanoyl-ACP-n-C120ACP', compartment='c')
```

Adding metabolites to a reaction requires using a dictionary of the metabolites and their stoichiometric coefficients. A group of metabolites can be added all at once, or they can be added one at a time.

```
reaction.add_metabolites({malACP_c: -1.0,
                          h_c: -1.0,
                          ddcaACP_c: -1.0,
                          co2_c: 1.0,
                          ACP_c: 1.0,
                          omrsACP_c: 1.0})
```

```
reaction.reaction  # This gives a string representation of the reaction
```

```
'malACP_c + h_c + ddcaACP_c --> co2_c + 3omrsACP_c + ACP_c'
```

The gene_reaction_rule is a boolean representation of the gene requirements for this reaction to be active as described in Schellenberger et al 2011 Nature Protocols 6(9):1290-307. We will assign the gene reaction rule string, which will automatically create the corresponding gene objects.

```
reaction.gene_reaction_rule = '( STM2378  or STM1197 )'
reaction.genes
```

```
frozenset({<Gene STM2378 at 0x3739b10>, <Gene STM1197 at 0x3739b50>})
```

At this point in time, the model is still empty

```
print('%i reactions in initial model' % len(cobra_model.reactions))
print('%i metabolites in initial model' % len(cobra_model.metabolites))
print('%i genes in initial model' % len(cobra_model.genes))
```

```
0 reactions in initial model
0 metabolites in initial model
0 genes in initial model
```

We will add the reaction to the model, which will also add all associated metabolites and genes

```
cobra_model.add_reaction(reaction)

# Now there are things in the model
print('%i reaction in model' % len(cobra_model.reactions))
print('%i metabolites in model' % len(cobra_model.metabolites))
print('%i genes in model' % len(cobra_model.genes))
```

```
1 reaction in model
6 metabolites in model
2 genes in model
```

We can iterate through the model objects to observe the contents

```
# Iterate through the the objects in the model
print("Reactions")
print("---------")
for x in cobra_model.reactions:
    print("%s : %s" % (repr(x), x.reaction))
print("Metabolites")
print("-----------")
for x in cobra_model.metabolites:
    print('%s : %s' % (repr(x), x.formula))
print("Genes")
print("-----")
for x in cobra_model.genes:
    reactions_list_str = ", ".join((repr(i) for i in x.reactions))
    print("%s is associated with reactions: %s" % (repr(x), reactions_list_str))
```

```
Reactions
---------
<Reaction 3OAS140 at 0x4a18b90> : malACP_c + h_c + ddcaACP_c --> co2_c + 3omrsACP_c + ACP_c
Metabolites
-----------
```

```
<Metabolite co2_c at 0x594ba10> : CO2
<Metabolite malACP_c at 0x594ba90> : C14H22N2O10PRS
<Metabolite h_c at 0x594bb10> : H
<Metabolite 3omrsACP_c at 0x594b950> : C25H45N2O9PRS
<Metabolite ACP_c at 0x594b990> : C11H21N2O7PRS
<Metabolite ddcaACP_c at 0x594bb90> : C23H43N2O8PRS
Genes
-----
<Gene STM2378 at 0x3739b10> is associated with reactions: <Reaction 3OAS140 at 0x4a18b90>
<Gene STM1197 at 0x3739b50> is associated with reactions: <Reaction 3OAS140 at 0x4a18b90>
```

CHAPTER 3

Reading and Writing Models

This example is available as an IPython notebook.

Functions for reading and writing models to various formats are included with cobrapy. The package also ships with models of *E. coli* and *Salmonella* in various formats for testing purposes. In this example, we will use these functions to read models from these test files in various formats.

```
import cobra.test

print("E. coli test files: ")
print(", ".join([i for i in dir(cobra.test) if i.startswith("ecoli")]))
print("")
print("Salmonella test files: ")
print(", ".join([i for i in dir(cobra.test) if i.startswith("salmonella")]))

salmonella_model = cobra.test.create_test_model("salmonella")
```

```
E. coli test files:
ecoli_json, ecoli_mat, ecoli_pickle, ecoli_sbml

Salmonella test files:
salmonella_fbc_sbml, salmonella_pickle, salmonella_sbml
```

3.1 JSON

cobrapy has a JSON (JavaScript Object Notation) representation. This is the ideal format for storing a cobra model on a computer, or for interoperability with escher. Additional fields, however, will not be saved.

```
cobra.io.load_json_model(cobra.test.ecoli_json)
```

```
<Model iJO1366 at 0x7f8d2fa20150>
```

```
cobra.io.write_sbml_model(salmonella_model, "test.json")
```

3.2 SBML

The Systems Biology Markup Language is an XML-based standard format for distributing models. Cobrapy can use libsbml, which must be installed separately (see installation instructions) to read and write SBML files.

Initially, the COBRA format for SBML files used the "notes" field in SBML files. More recently, however, the FBC extension to SBML has come into existence, which defines its own fields.

Cobrapy can handle both formats (assuming libsbml has been installed correctly). When reading in a model, it will automatically detect whether fbc was used or not. When writing a model, the use_fbc_package can be used.

```
cobra.io.read_sbml_model(cobra.test.salmonella_sbml)
```

```
<Model Salmonella_consensus_build_1 at 0x7f8d480ad710>
```

```
cobra.io.read_sbml_model(cobra.test.salmonella_fbc_sbml)
```

```
<Model Salmonella_consensus_build_1 at 0x7f8d480ad5d0>
```

```
cobra.io.write_sbml_model(salmonella_model, "test.xml",
                          use_fbc_package=False)
cobra.io.write_sbml_model(salmonella_model, "test_fbc.xml",
                          use_fbc_package=True)
```

3.3 MATLAB

Often, models may be imported and exported soley for the purposes of working with the same models in cobrapy and the MATLAB cobra toolbox. MATLAB has its own ".mat" format for storing variables. Reading and writing to these mat files from python requires scipy, and is generally much faster than using libsbml.

A mat file can contain multiple MATLAB variables. Therefore, the variable name of the model in the MATLAB file can be passed into the reading function:

```
cobra.io.load_matlab_model(cobra.test.ecoli_mat, variable_name="iJO1366")
```

```
<Model iJO1366 at 0x7f8d48090b50>
```

If the mat file contains only a single model, cobra can figure out which variable to read from, and the variable_name paramter is unnecessary.

```
cobra.io.load_matlab_model(cobra.test.ecoli_mat)
```

```
<Model iJO1366 at 0x7f8d2d85af10>
```

Saving models to mat files is also relatively straightforward

```
cobra.io.save_matlab_model(ecoli_model, "test_ecoli_model.mat")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-9-899c9ca38882> in <module>()
----> 1 cobra.io.save_matlab_model(ecoli_model, "test_ecoli_model.mat")


NameError: name 'ecoli_model' is not defined
```

3.4 Pickle

Cobra models can be serialized using the python serialization format, pickle. While this will save any extra fields which may have been created, it does not work with any other tools and can break between cobrapy major versions.

JSON is generally the preferred format.

```
from cPickle import load, dump

# read in the test models
with open(cobra.test.ecoli_pickle, "rb") as infile:
    ecoli_model = load(infile)
with open(cobra.test.salmonella_pickle, "rb") as infile:
    salmonella_model = load(infile)

# output to a file
with open("test_output.pickle", "wb") as outfile:
    dump(salmonella_model, outfile)
```

CHAPTER 4

Simulating with FBA

This example is available as an IPython notebook.

Simulations using flux balance analysis can be solved using Model.optimize(). This will maximize or minimize (maximizing is the default) flux through the objective reactions.

```
import cobra.test
model = cobra.test.create_test_model()
```

4.1 Running FBA

```
model.optimize()
```

```
<Solution 0.38 at 0x660d990>
```

The Model.optimize() function will return a Solution object, which will also be stored at model.solution. A solution object has several attributes:

- f: the objective value
- status: the status from the linear programming solver
- x_dict: a dictionary of {reaction_id: flux_value} (also called "primal")
- x: a list for x_dict
- y_dict: a dictionary of {metabolite_id: dual_value}.
- y: a list for y_dict

For example, after the last call to model.optimize(), the status should be 'optimal' if the solver returned no errors, and f should be the objective value

```
model.solution.status
```

```
'optimal'
```

```
model.solution.f
```

```
0.38000797227551136
```

4.2 Changing the Objectives

The objective function is determined from the objective_coefficient attribute of the objective reaction(s). Currently in the model, there is only one objective reaction, with an objective coefficient of 1.

```
{reaction: reaction.objective_coefficient for reaction in model.reactions
        if reaction.objective_coefficient > 0}
```

```
{<Reaction biomass_iRR1083_metals at 0x660d350>: 1.0}
```

The objective function can be changed by using the function Model.change_objective, which will take either a reaction object or just its name.

```
# change the objective to ATPM
# the upper bound should be 1000 so we get the actual optimal value
model.reactions.get_by_id("ATPM").upper_bound = 1000.
model.change_objective("ATPM")
{reaction: reaction.objective_coefficient for reaction in model.reactions
        if reaction.objective_coefficient > 0}
```

```
{<Reaction ATPM at 0x52cb190>: 1.0}
```

```
model.optimize()
```

```
<Solution 119.67 at 0x4c93110>
```

The objective function can also be changed by setting Reaction.objective_coefficient directly.

```
model.reactions.get_by_id("ATPM").objective_coefficient = 0.
model.reactions.get_by_id("biomass_iRR1083_metals").objective_coefficient = 1.
{reaction: reaction.objective_coefficient for reaction in model.reactions
        if reaction.objective_coefficient > 0}
```

```
{<Reaction biomass_iRR1083_metals at 0x660d350>: 1.0}
```

CHAPTER 5

Simulating Deletions

This example is available as an IPython notebook.

```
from time import time


from cobra.test import create_test_model, salmonella_pickle, ecoli_pickle
from cobra.flux_analysis import single_deletion
from cobra.flux_analysis import double_deletion


cobra_model = create_test_model(salmonella_pickle)
```

5.1 Single Deletions

Perform all single gene deletions on a model

```
start = time()  # start timer()
growth_rates, statuses = single_deletion(cobra_model)
print("All single gene deletions completed in %.2f sec" % (time() - start))
```

```
All single gene deletions completed in 4.01 sec
```

These can also be done for only a subset of genes

```
single_deletion(cobra_model, element_list=cobra_model.genes[:100]);
```

Single deletions can also be run on reactions

```
start = time()  # start timer()
growth_rates, statuses = single_deletion(cobra_model, element_type="reaction")
print("All single reaction deletions completed in %.2f sec" % (time() - start))
```

```
All single reaction deletions completed in 7.41 sec
```

5.2 Double Deletions

Double deletions run in a similar way

```
start = time()  # start timer()
double_deletion(cobra_model, element_list_1=cobra_model.genes[:100])
print("Double gene deletions for 100 genes completed in %.2f sec" % (time() - start))
```

```
Double gene deletions for 100 genes completed in 4.94 sec
```

By default, the double deletion function will automatically use multiprocessing, splitting the task over up to 4 cores if they are available. The number of cores can be manually sepcified as well. Setting use of a single core will disable use of the multiprocessing library, which often aids debuggging.

```
start = time()  # start timer()
double_deletion(cobra_model, element_list_1=cobra_model.genes[:100],
                number_of_processes=2)
t1 = time() - start
print("Double gene deletions for 100 genes completed in %.2f sec with 2 cores" % t1)

start = time()  # start timer()
double_deletion(cobra_model, element_list_1=cobra_model.genes[:100],
                number_of_processes=1)
t2 = time() - start
print("Double gene deletions for 100 genes completed in %.2f sec with 1 core" % t2)

print("Speedup of %.2fx" % (t2/t1))
```

```
Double gene deletions for 100 genes completed in 4.02 sec with 2 cores
Double gene deletions for 100 genes completed in 6.77 sec with 1 core
Speedup of 1.69x
```

Double deletions can also be run for reactions

```
start = time()
double_deletion(cobra_model, element_list_1=cobra_model.reactions[:100],
                element_type="reaction")
t = time() - start
print("Double reaction deletions for 100 reactions completed in %.2f sec" % t)
```

```
Double reaction deletions for 100 reactions completed in 0.93 sec
```

If pandas is installed, the results can be returned formatted as a pandas.DataFrame

```
frame = double_deletion(cobra_model, element_list_1=cobra_model.reactions[300:308],
                        element_type="reaction", return_frame=True)
frame[frame < 1e-9] = 0.  # round small values to 0
frame
```

CHAPTER 6

Phenotype Phase Plane

This example is available as an IPython notebook.

Load iJO1366 as a test model and import cobra

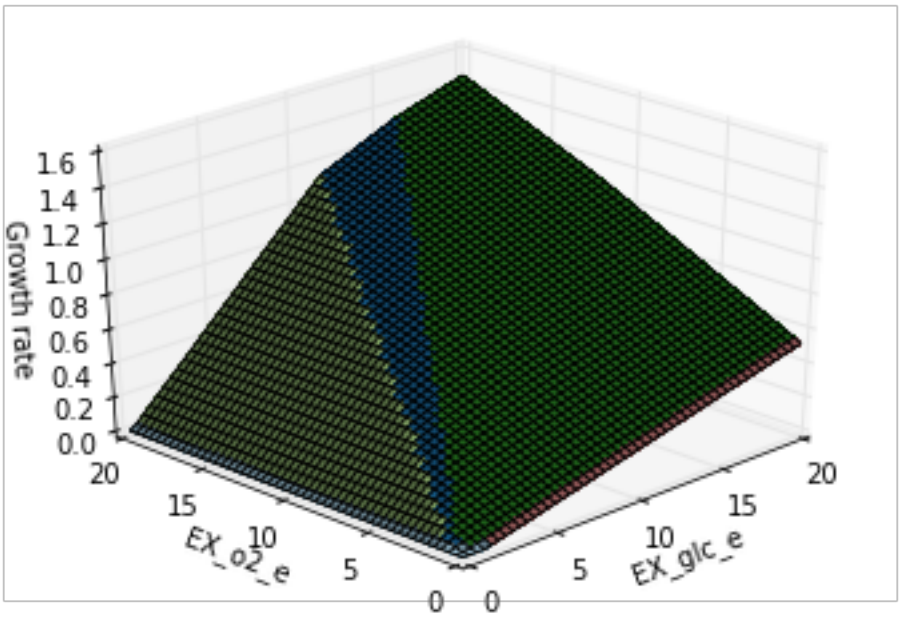
```
from time import time

import cobra
from cobra.test import ecoli_pickle, create_test_model

from cobra.flux_analysis.phenotype_phase_plane import \
    calculate_phenotype_phase_plane

model = create_test_model(ecoli_pickle)
model
```

```
<Model iJO1366 at 0x5b0abd0>
```

We want to make a phenotype phase plane to evaluate uptakes of Glucose and Oxygen.

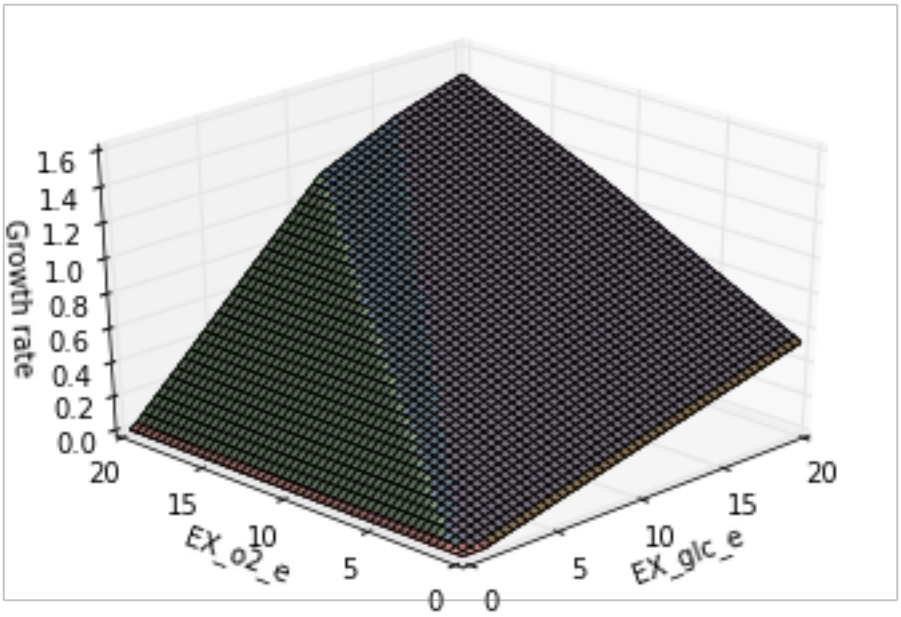
With matplotlib installed, this is as simple as

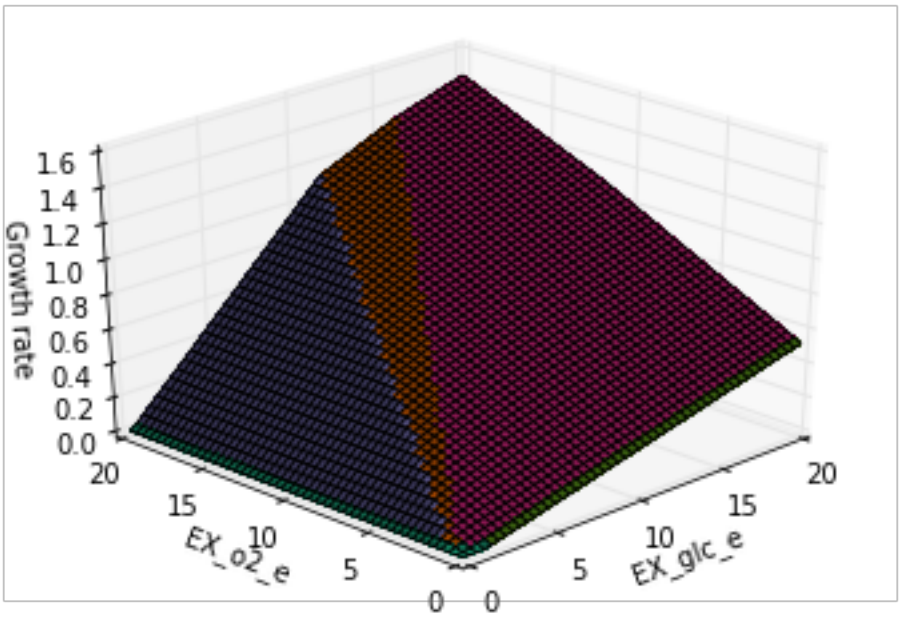
```
data = calculate_phenotype_phase_plane(model, "EX_glc_e", "EX_o2_e")
data.plot_matplotlib();
```



If brewer2mpl is installed, other color schemes can be used as well

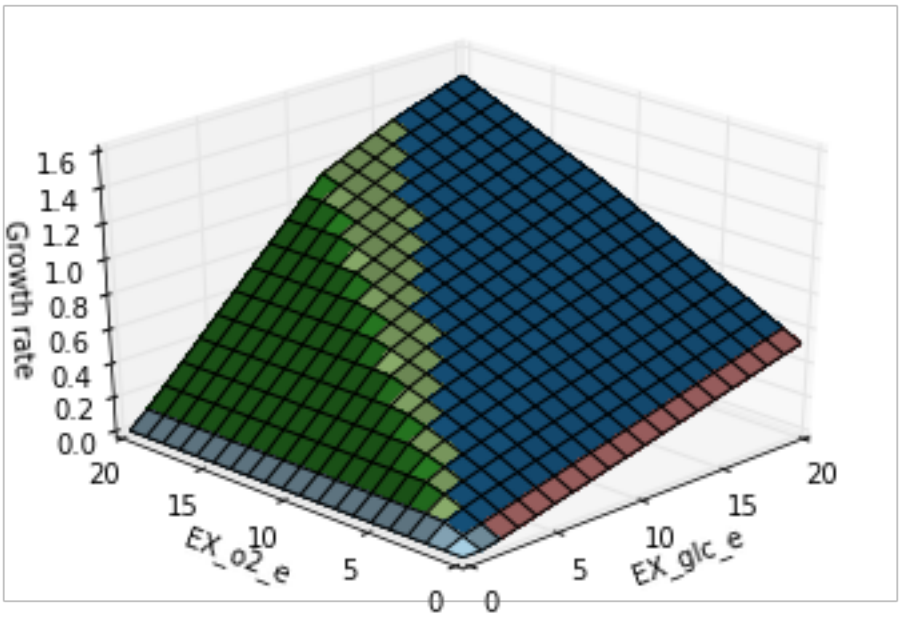
```
data.plot_matplotlib("Pastel1")
data.plot_matplotlib("Dark2");
```





The number of points which are plotted in each dimension can also be changed

```
calculate_phenotype_phase_plane(model, "EX_glc_e", "EX_o2_e",
                                reaction1_npoints=20,
                                reaction2_npoints=20).plot_matplotlib();
```



The code can also use multiple processes to speed up calculations

```
start_time = time()
calculate_phenotype_phase_plane(model, "EX_glc_e", "EX_o2_e", n_processes=1)
print("took %.2f seconds with 1 process" % (time() - start_time))
start_time = time()
calculate_phenotype_phase_plane(model, "EX_glc_e", "EX_o2_e", n_processes=4)
print("took %.2f seconds with 4 process" % (time() - start_time))
```

```
took 5.97 seconds with 1 process
took 2.97 seconds with 4 process
```

CHAPTER 7

Mixed-Integer Linear Programming

This example is available as an IPython notebook.

7.1 Ice Cream

This example was originally contributed by Joshua Lerman.

An ice cream stand sells cones and popsicles. It wants to maximize its profit, but is subject to a budget.

We can write this problem as a linear program:

max cone · cone_margin + popsicle · popsicle margin

subject to

cone · cone_cost + popsicle · popsicle_cost $\leq$ budget

```
cone_selling_price = 7.
cone_production_cost = 3.
popsicle_selling_price = 2.
popsicle_production_cost = 1.
starting_budget = 100.
```

This problem can be written as a cobra.Model

```
from cobra import Model, Metabolite, Reaction

cone = Reaction("cone")
popsicle = Reaction("popsicle")

# constrainted to a budget
budget = Metabolite("budget")
budget._constraint_sense = "L"
budget._bound = starting_budget
cone.add_metabolites({budget: cone_production_cost})
popsicle.add_metabolites({budget: popsicle_production_cost})

# objective coefficient is the profit to be made from each unit
cone.objective_coefficient = cone_selling_price - cone_production_cost
popsicle.objective_coefficient = popsicle_selling_price - \
                                 popsicle_production_cost

m = Model("lerman_ice_cream_co")
m.add_reactions((cone, popsicle))
```

```
m.optimize().x_dict
```

```
{'cone': 33.333333333333336, 'popsicle': 0.0}
```

In reality, cones and popsicles can only be sold in integer amounts. We can use the variable kind attribute of a cobra.Reaction to enforce this.

```
cone.variable_kind = "integer"
popsicle.variable_kind = "integer"
m.optimize().x_dict
```

```
{'cone': 33.0, 'popsicle': 1.0}
```

Now the model makes both popsicles and cones.

7.2 Restaurant Order

To tackle the less immediately obvious problem from the following XKCD comic:

```
from IPython.display import Image
Image(url=r"http://imgs.xkcd.com/comics/np_complete.png")
```

We want a solution satisfying the following constraints:

$\begin{pmatrix}2.15 & 2.75 & 3.35 & 3.55 & 4.20 & 5.80\end{pmatrix} \cdot \vec{v} = 15.05$

$\vec{v}_i \geq 0$

$\vec{v}_i \in \mathbb{Z}$

This problem can be written as a COBRA model as well.

```
total_cost = Metabolite("constraint")
total_cost._bound = 15.05

costs = {"mixed_fruit": 2.15, "french_fries": 2.75, "side_salad": 3.35,
         "hot_wings": 3.55, "mozarella_sticks": 4.20, "sampler_plate": 5.80}

m = Model("appetizers")

for item, cost in costs.items():
    r = Reaction(item)
    r.add_metabolites({total_cost: cost})
    r.variable_kind = "integer"
    m.add_reaction(r)

# To add to the problem, suppose we don't want to eat all mixed fruit.
m.reactions.mixed_fruit.objective_coefficient = 1

m.optimize(objective_sense="minimize").x_dict
```

```
{'french_fries': 0.0,
 'hot_wings': 2.0,
 'mixed_fruit': 1.0,
 'mozarella_sticks': 0.0,
 'sampler_plate': 1.0,
 'side_salad': 0.0}
```

There is another solution to this problem, which would have been obtained if we had maximized for mixed fruit instead of minimizing.

```
m.optimize(objective_sense="maximize").x_dict
```

```
{'french_fries': 0.0,
 'hot_wings': 0.0,
 'mixed_fruit': 7.0,
 'mozarella_sticks': 0.0,
 'sampler_plate': 0.0,
 'side_salad': 0.0}
```

7.3 Boolean Indicators

To give a COBRA-related example, we can create boolean variables as integers, which can serve as indicators for a reaction being active in a model. For a reaction flux v with lower bound -1000 and upper bound 1000, we can create a binary variable b with the following constraints:

$b \in \{0, 1\}$

$-1000 \cdot b \leq v \leq 1000 \cdot b$

To introduce the above constraints into a cobra model, we can rewrite them as follows

$v \leq b \cdot 1000 \Rightarrow v - 1000 \cdot b \leq 0$

$-1000 \cdot b \leq v \Rightarrow v + 1000 \cdot b \geq 0$

```
import cobra.test
model = cobra.test.create_test_model(cobra.test.ecoli_pickle)

# an indicator for pgi
pgi = model.reactions.get_by_id("PGI")
# make a boolean variable
pgi_indicator = Reaction("indicator_PGI")
pgi_indicator.lower_bound = 0
pgi_indicator.upper_bound = 1
pgi_indicator.variable_kind = "integer"
# create constraint for v - 1000 b <= 0
pgi_plus = Metabolite("PGI_plus")
pgi_plus._constraint_sense = "L"
# create constraint for v + 1000 b >= 0
pgi_minus = Metabolite("PGI_minus")
pgi_minus._constraint_sense = "G"

pgi_indicator.add_metabolites({pgi_plus: -1000, pgi_minus: 1000})
pgi.add_metabolites({pgi_plus: 1, pgi_minus: 1})
model.add_reaction(pgi_indicator)


# an indicator for zwf
zwf = model.reactions.get_by_id("G6PDH2r")
zwf_indicator = Reaction("indicator_ZWF")
zwf_indicator.lower_bound = 0
zwf_indicator.upper_bound = 1
zwf_indicator.variable_kind = "integer"
# create constraint for v - 1000 b <= 0
zwf_plus = Metabolite("ZWF_plus")
```

```
zwf_plus._constraint_sense = "L"
# create constraint for v + 1000 b >= 0
zwf_minus = Metabolite("ZWF_minus")
zwf_minus._constraint_sense = "G"

zwf_indicator.add_metabolites({zwf_plus: -1000, zwf_minus: 1000})
zwf.add_metabolites({zwf_plus: 1, zwf_minus: 1})

# add the indicator reactions to the model
model.add_reaction(zwf_indicator)
```

In a model with both these reactions active, the indicators will also be active

```
solution = model.optimize()
print("PGI indicator = %d" % solution.x_dict["indicator_PGI"])
print("ZWF indicator = %d" % solution.x_dict["indicator_ZWF"])
print("PGI flux = %.2f" % solution.x_dict["PGI"])
print("ZWF flux = %.2f" % solution.x_dict["G6PDH2r"])
```

```
PGI indicator = 1
ZWF indicator = 1
PGI flux = 5.92
ZWF flux = 4.08
```

Because these boolean indicators are in the model, additional constraints can be applied on them. For example, we can prevent both reactions from being active at the same time by adding the following constraint:

$b_{\text{pgi}} + b_{\text{zwf}} = 1$

```
or_constraint = Metabolite("or")
or_constraint._bound = 1
zwf_indicator.add_metabolites({or_constraint: 1})
pgi_indicator.add_metabolites({or_constraint: 1})

solution = model.optimize()
print("PGI indicator = %d" % solution.x_dict["indicator_PGI"])
print("ZWF indicator = %d" % solution.x_dict["indicator_ZWF"])
print("PGI flux = %.2f" % solution.x_dict["PGI"])
print("ZWF flux = %.2f" % solution.x_dict["G6PDH2r"])
```

```
PGI indicator = 0
ZWF indicator = 1
PGI flux = 0.00
ZWF flux = 3.98
```

CHAPTER 8

Quadratic Programming

This example is available as an IPython notebook.

Suppose we want to minimize the Euclidean distance of the solution to the origin while subject to linear constraints. This will require a quadratic objective function. Consider this example problem:

$$\textbf{min } \tfrac{1}{2}\left(x^2 + y^2\right)$$

subject to

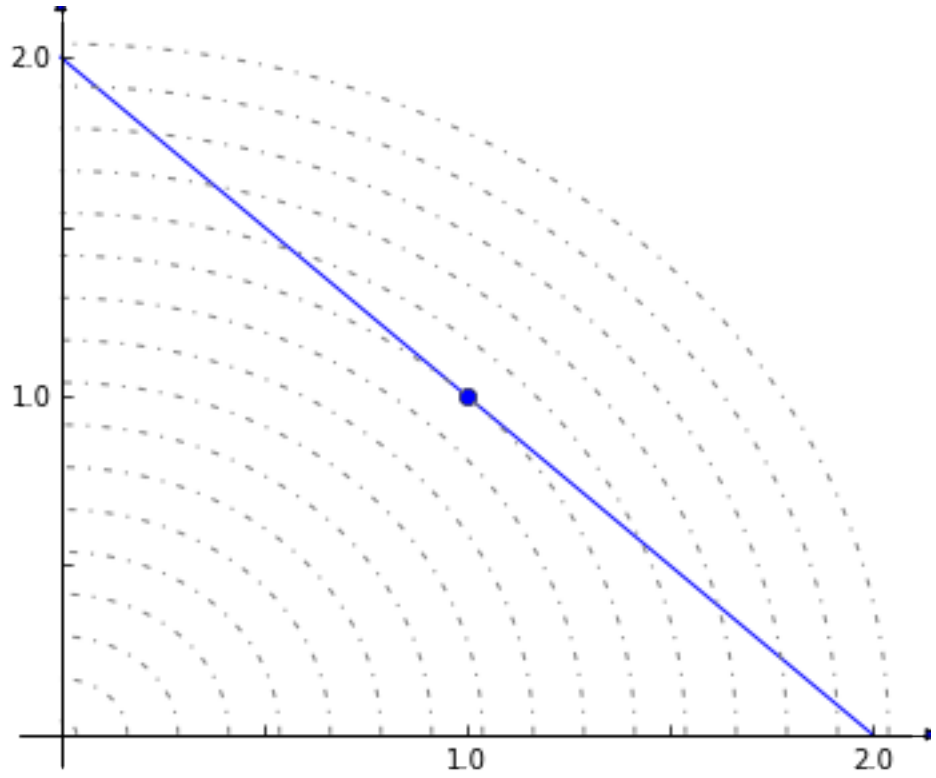
$$x + y = 2$$

$$x \geq 0$$

$$y \geq 0$$

This problem can be visualized graphically:

```
from matplotlib.pyplot import figure, xlim, ylim
from mpl_toolkits.axes_grid.axislines import SubplotZero
from numpy import linspace, arange, sqrt, pi, sin, cos, sign
# axis style
def make_plot_ax():
    fig = figure(figsize=(6, 5));
    ax = SubplotZero(fig, 111); fig.add_subplot(ax)
    for direction in ["xzero", "yzero"]:
        ax.axis[direction].set_axisline_style("-|>")
        ax.axis[direction].set_visible(True)
    for direction in ["left", "right", "bottom", "top"]:
        ax.axis[direction].set_visible(False)
    xlim(-0.1, 2.1); ylim(xlim())
    ticks = [0.5 * i for i in range(1, 5)]
    labels = [str(i) if i == int(i) else "" for i in ticks]
    ax.set_xticks(ticks); ax.set_yticks(ticks)
    ax.set_xticklabels(labels); ax.set_yticklabels(labels)
    ax.axis["yzero"].set_axis_direction("left")
    return ax

ax = make_plot_ax()
ax.plot((0, 2), (2, 0), 'b')
ax.plot([1], [1], 'bo')

# circular grid
for r in sqrt(2.) + 0.125 * arange(-11, 6):
    t = linspace(0., pi/2., 100)
    ax.plot(r * cos(t), r * sin(t), '-.', color="gray")
```



The objective can be rewritten as $\frac{1}{2}v^T \cdot \mathbf{Q} \cdot v$, where $v = \begin{pmatrix} x \\ y \end{pmatrix}$ and $\mathbf{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

The matrix $\mathbf{Q}$ can be passed into a cobra model as the quadratic objective.

```
import scipy

from cobra import Reaction, Metabolite, Model, solvers
```

The quadratic objective $\mathbf{Q}$ should be formatted as a scipy sparse matrix.

```
Q = scipy.sparse.eye(2).todok()
Q
```

```
<2x2 sparse matrix of type '<type 'numpy.float64'>'
    with 2 stored elements in Dictionary Of Keys format>
```

In this case, the quadratic objective is simply the identity matrix

```
Q.todense()
```

```
matrix([[ 1.,  0.],
        [ 0.,  1.]])
```

We need to use a solver that supports quadratic programming, such as gurobi or cplex. If a solver which supports quadratic programming is installed, this function will return its name.

```
print(solvers.get_solver_name(qp=True))
```

```
gurobi
```

```
c = Metabolite("c")
c._bound = 2
x = Reaction("x")
y = Reaction("y")
x.add_metabolites({c: 1})
y.add_metabolites({c: 1})
m = Model()
m.add_reactions([x, y])
sol = m.optimize(quadratic_component=Q, objective_sense="minimize")
sol.x_dict
```

```
{'x': 1.0, 'y': 1.0}
```

Suppose we change the problem to have a mixed linear and quadratic objective.

$$\textbf{min } \frac{1}{2}\left(x^2 + y^2\right) - y$$

subject to

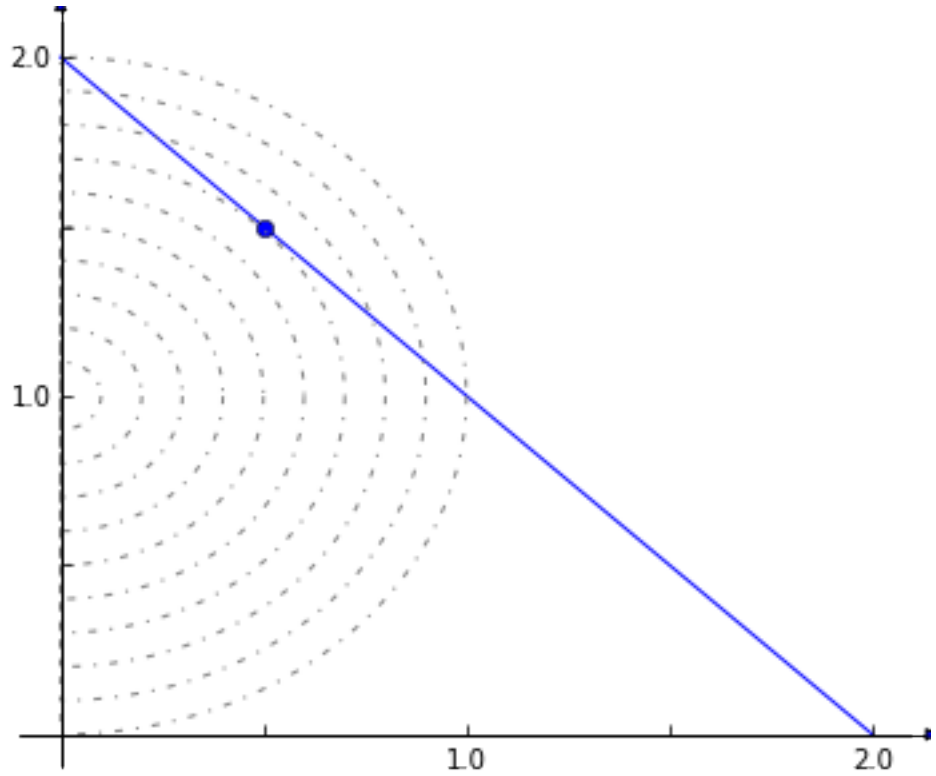
$$x + y = 2$$

$$x \geq 0$$

$$y \geq 0$$

Graphically, this would be

```
ax = make_plot_ax()
ax.plot((0, 2), (2, 0), 'b')
ax.plot([0.5], [1.5], 'bo')

yrange = linspace(1, 2, 11)
for r in (yrange ** 2 / 2. - yrange):
    t = linspace(-sqrt(2 * r + 1) + 0.000001, sqrt(2 * r + 1) - 0.000001, 1000)
    ax.plot(abs(t), 1 + sqrt(2 * r + 1 - t ** 2) * sign(t), '-.', color="gray")
```



QP solvers in cobrapy will combine linear and quadratic coefficients. The linear portion will be obtained from the same objective_coefficient attribute used with LP's.

```
y.objective_coefficient = -1
sol = m.optimize(quadratic_component=Q, objective_sense="minimize")
sol.x_dict
```

```
{'x': 0.5, 'y': 1.5}
```

CHAPTER 9

Loopless FBA

This example is available as an IPython notebook.

The goal of this procedure is identification of a thermodynamically consistent flux state without loops, as implied by the name.

Usually, the model has the following constraints.

$$S \cdot v = 0$$

$$lb \leq v \leq ub$$

However, this will allow for thermodynamically infeasible loops (referred to as type 3 loops) to occur, where flux flows around a cycle without any net change of metabolites. For most cases, this is not a major issue, as solutions with these loops can usually be converted to equivalent solutions without them. However, if a flux state is desired which does not exhibit any of these loops, loopless FBA can be used. The formulation used here is modified from Schellenberger et al.

We can make the model irreversible, so that all reactions will satisfy

$$0 \leq lb \leq v \leq ub \leq \max(ub)$$

We will add in boolean indicators as well, such that

$$\max(ub)i \geq v$$

$$i \in 0, 1$$

We also want to ensure that an entry in the row space of S also exists with negative values wherever v is nonzero. In this expression, $1 - i$ acts as a not to indicate inactivity of a reaction.

$$S^{\mathsf{T}}x - (1 - i)(\max(ub) + 1) \leq -1$$

We will construct an LP integrating both constraints.

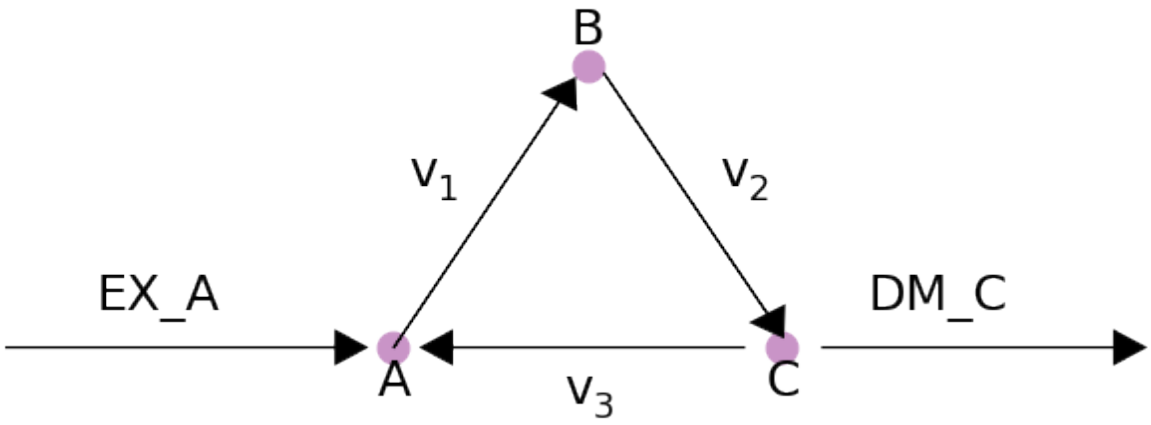
$$\begin{pmatrix} S & 0 & 0 \\ -I & \max(ub)I & 0 \\ 0 & (\max(ub) + 1)I & S^{\mathsf{T}} \end{pmatrix} \cdot \begin{pmatrix} v \\ i \\ x \end{pmatrix} \begin{matrix} = \\ \geq \\ \leq \end{matrix} \begin{matrix} 0 \\ 0 \\ \max(ub) \end{matrix}$$

Note that these extra constraints are not applied to boundary reactions which bring metabolites in and out of the system.

```
import cobra.test
from cobra import Reaction, Metabolite, Model
from cobra.flux_analysis.loopless import construct_loopless_model
from cobra.solvers import get_solver_name
```

We will demonstrate with a toy model which has a simple loop cycling A -> B -> C -> A, with A allowed to enter the system and C allowed to leave. A graphical view of the system is drawn below:

```
figure(figsize=(10.5, 4.5), frameon=False)
gca().axis("off")
xlim(0.5, 3.5)
ylim(0.7, 2.2)
arrow_params = {"head_length": 0.08, "head_width": 0.1, "ec": "k", "fc": "k"}
text_params = {"fontsize": 25, "horizontalalignment": "center", "verticalalignment": "center"}
arrow(0.5, 1, 0.85, 0, **arrow_params)  # EX_A
arrow(1.5, 1, 0.425, 0.736, **arrow_params)  # v1
arrow(2.04, 1.82, 0.42, -0.72, **arrow_params)  # v2
arrow(2.4, 1, -0.75, 0, **arrow_params)  # v3
arrow(2.6, 1, 0.75, 0, **arrow_params)
# reaction labels
text(0.9, 1.15, "EX_A", **text_params)
text(1.6, 1.5, r"v$_1$", **text_params)
text(2.4, 1.5, r"v$_2$", **text_params)
text(2, 0.85, r"v$_3$", **text_params)
text(2.9, 1.15, "DM_C", **text_params)
# metabolite labels
scatter(1.5, 1, s=250, color='#c994c7')
text(1.5, 0.9, "A", **text_params)
scatter(2, 1.84, s=250, color='#c994c7')
text(2, 1.95, "B", **text_params)
scatter(2.5, 1, s=250, color='#c994c7')
text(2.5, 0.9, "C", **text_params);
```




```
test_model = Model()
test_model.add_metabolites(Metabolite("A"))
test_model.add_metabolites(Metabolite("B"))
test_model.add_metabolites(Metabolite("C"))
EX_A = Reaction("EX_A")
EX_A.add_metabolites({test_model.metabolites.A: 1})
```

```
DM_C = Reaction("DM_C")
DM_C.add_metabolites({test_model.metabolites.C: -1})
v1 = Reaction("v1")
v1.add_metabolites({test_model.metabolites.A: -1, test_model.metabolites.B: 1})
v2 = Reaction("v2")
v2.add_metabolites({test_model.metabolites.B: -1, test_model.metabolites.C: 1})
v3 = Reaction("v3")
v3.add_metabolites({test_model.metabolites.C: -1, test_model.metabolites.A: 1})
DM_C.objective_coefficient = 1
test_model.add_reactions([EX_A, DM_C, v1, v2, v3])
```

While this model contains a loop, a flux state exists which has no flux through reaction v3, and is identified by loopless FBA.

```
construct_loopless_model(test_model).optimize()
```

```
<Solution 1000.00 at 0x62cd250>
```

However, if flux is forced through v3, then there is no longer a feasible loopless solution.

```
v3.lower_bound = 1
construct_loopless_model(test_model).optimize()
```

```
<Solution 'infeasible' at 0x62cd5d0>
```

Loopless FBA is also possible on genome scale models, but it requires a capable MILP solver.

```
salmonella = cobra.test.create_test_model()
construct_loopless_model(salmonella).optimize(solver=get_solver_name(mip=True))
```

```
<Solution 0.38 at 0x9e67650>
```

```
ecoli = cobra.test.create_test_model("ecoli")
construct_loopless_model(ecoli).optimize(solver=get_solver_name(mip=True))
```

```
<Solution 0.98 at 0x8e463d0>
```

CHAPTER 10

FAQ

This document will address frequently asked questions not addressed in other pages of the documentation.

10.1 How do I install cobrapy?

Please see the INSTALL.md file.

10.2 How do I cite cobrapy?

Please cite the 2013 publication: 10.1186/1752-0509-7-74

10.3 How do I rename reactions or metabolites?

TL;DR Use Model.repair afterwards

When renaming metabolites or reactions, there are issues because cobra indexes based off of ID's, which can cause errors. For example:

```
from __future__ import print_function
import cobra.test
model = cobra.test.create_test_model()

for metabolite in model.metabolites:
    metabolite.id = "test_" + metabolite.id

try:
    model.metabolites.get_by_id(model.metabolites[0].id)
except KeyError as e:
    print(repr(e))
```

```
KeyError('test_dcaACP_c',)
```

The Model.repair function will rebuild the necessary indexes

```
model.repair()
model.metabolites.get_by_id(model.metabolites[0].id)
```

```
<Metabolite test_dcaACP_c at 0x688b450>
```

10.4 How do I delete a gene?

That depends on what precisely you mean by delete a gene.

If you want to simulate the model with a gene knockout, use the cobra.maniupulation.delete_model_genes function. The effects of this function are reversed by cobra.manipulation.undelete_model_genes.

```
model = cobra.test.create_test_model()
PGI = model.reactions.get_by_id("PGI")
print("bounds before knockout:", (PGI.lower_bound, PGI.upper_bound))
cobra.manipulation.delete_model_genes(model, ["STM4221"])
print("bounds after knockouts", (PGI.lower_bound, PGI.upper_bound))
```

```
bounds before knockout: (-1000.0, 1000.0)
bounds after knockouts (0.0, 0.0)
```

If you want to actually remove all traces of a gene from a model, this is more difficult because this will require changing all the gene_reaction_rule strings for reactions involving the gene.

10.5 How do I change the reversibility of a Reaction?

Reaction.reversibility is a property in cobra which is computed when it is requested from the lower and upper bounds.

```
model = cobra.test.create_test_model()
model.reactions.get_by_id("PGI").reversibility
```

```
True
```

Trying to set it directly will result in an error:

```
try:
    model.reactions.get_by_id("PGI").reversibility = False
except Exception as e:
    print(repr(e))
```

```
AttributeError("can't set attribute",)
```

The way to change the reversibility is to change the bounds to make the reaction irreversible.

```
model.reactions.get_by_id("PGI").lower_bound = 10
model.reactions.get_by_id("PGI").reversibility
```

```
False
```

10.6 How do I generate an LP file from a COBRA model?

While the cobrapy does not include python code to support this feature directly, many of the bundled solvers have this capability. Create the problem with one of these solvers, and use its appropriate function.

Please note that unlike the LP file format, the MPS file format does not specify objective direction and is always a minimzation. Some (but not all) solvers will rewrite the maximization as a minimzation.

```
model = cobra.test.create_test_model()
# glpk through cglpk
glp = cobra.solvers.cglpk.create_problem(model)
glp.write("test.lp")
glp.write("test.mps")  # will not rewrite objective
# gurobi
gurobi_problem = cobra.solvers.gurobi_solver.create_problem(model)
gurobi_problem.write("test.lp")
gurobi_problem.write("test.mps")  # rewrites objective
# cplex
cplex_problem = cobra.solvers.cplex_solver.create_problem(model)
cplex_problem.write("test.lp")
cplex_problem.write("test.mps")  # rewrites objective
```

10.7 How do I visualize my flux solutions?

cobrapy works well with the escher package, which is well suited to this purpose. Consult the escher documentation for examples.

CHAPTER 11

cobra package

11.1 Subpackages

11.1.1 cobra.core package

Submodules

cobra.core.ArrayBasedModel module

class cobra.core.ArrayBasedModel.**ArrayBasedModel**(*description=None*, *deepcopy_model=False*, *matrix_type='scipy.lil_matrix'*)

Bases: *cobra.core.Model.Model*

ArrayBasedModel is a class that adds arrays and vectors to a cobra.Model to make it easier to perform linear algebra operations.

S

Stoichiometric matrix of the model

This will be formatted as either lil_matrix or dok_matrix

add_metabolites(*metabolite_list*, *expand_stoichiometric_matrix=True*)

Will add a list of metabolites to the the object, if they do not exist and then expand the stochiometric matrix

metabolite_list: A list of *Metabolite* objects

expand_stoichimetric_matrix: Boolean. If True and self.S is not None then it will add rows to self.S. self.S must be created after adding reactions and metabolites to self before it can be expanded. Trying to expand self.S when self only contains metabolites is ludacris.

add_reactions(*reaction_list*, *update_matrices=True*)

Will add a cobra.Reaction object to the model, if reaction.id is not in self.reactions.

reaction_list: A *Reaction* object or a list of them

update_matrices: Boolean. If true populate / update matrices S, lower_bounds, upper_bounds, ... Note this is slow to run for very large models and using this option with repeated calls will degrade performance. Better to call self.update() after adding all reactions.

If the stoichiometric matrix is initially empty then initialize a 1x1 sparse matrix and add more rows as needed in the self.add_metabolites function

b

bounds for metabolites as numpy.ndarray

constraint_sense

copy()
Provides a partial 'deepcopy' of the Model. All of the Metabolite, Gene, and Reaction objects are created anew but in a faster fashion than deepcopy

lower_bounds

objective_coefficients

remove_reactions(*reactions*, *update_matrices=True*, ***kwargs*)
remove reactions from teh model

See *cobra.core.Model.Model.remove_reactions()*

update_matrices: Boolean If true populate / update matrices S, lower_bounds, upper_bounds. Note that this is slow to run for very large models, and using this option with repeated calls will degrade performance.

update()
Regenerates the stoichiometric matrix and vectors

upper_bounds

cobra.core.DictList module

class cobra.core.DictList.**DictList**(**args*)
Bases: list

A combined dict and list

This object behaves like a list, but has the O(1) speed benefits of a dict when looking up elements by their id.

append(*object*)
append object to end

extend(*iterable*)
extend list by appending elements from the iterable

get_by_id(*id*)
return the element with a matching id

has_id(*id*)

index(*id*, **args*)
Determine the position in the list

id: A string or a *Object*

insert(*index*, *object*)
insert object before index

list_attr(*attribute*)
return a list of the given attribute for every object

pop(**args*)
remove and return item at index (default last).

query(*search_function*, *attribute='id'*)
query the list

search_function: used to select which objects to return

- a string, in which case any object.attribute containing the string will be returned

- a compiled regular expression
- a function which takes one argument and returns True for desired values

attribute: the attribute to be searched for (default is 'id'). If this is None, the object itself is used.

returns: a list of objects which match the query

remove (*x*)

> **Warning:** Internal use only

reverse ()
reverse *IN PLACE*

sort (*cmp=None*, *key=None*, *reverse=False*)
stable sort *IN PLACE*

cmp(x, y) -> -1, 0, 1

union (*iterable*)
adds elements with id's not already in the model

cobra.core.Formula module

class cobra.core.Formula.**Formula** (*formula=None*)
Bases: *cobra.core.Object.Object*

Describes a Chemical Formula

A legal formula string contains only letters and numbers.

parse_composition ()
Breaks the chemical formula down by element.

weight
Calculate the formula weight

cobra.core.Gene module

class cobra.core.Gene.**Gene** (*id*, *formula=None*, *name=None*, *compartment=None*, *strand='+'*, *locus_start=0*, *locus_end=0*, *functional=True*)
Bases: *cobra.core.Species.Species*

A Gene is a special class of metabolite.

TODO: Make design decisions about TUs and such

remove_from_model (*model=None*, *make_dependent_reactions_nonfunctional=True*)
Removes the association

make_dependent_reactions_nonfunctional: Boolean. If True then replace the gene with 'False' in the gene association, else replace the gene with 'True'

> **Note:** Simulating gene knockouts is much better handled by cobra.manipulation.delete_model_genes

cobra.core.Metabolite module

class cobra.core.Metabolite.**Metabolite**(*id=None*, *formula=None*, *name=None*, *compartment=None*)
Bases: *cobra.core.Species.Species*

Metabolite is a class for holding information regarding a metabolite in a cobra.Reaction object.

remove_from_model(*method='subtractive'*, ***kwargs*)
Removes the association from self.model

method: 'subtractive' or 'destructive'. If 'subtractive' then the metabolite is removed from all associated reactions. If 'destructive' then all associated reactions are removed from the Model.

y
The shadow price for the metabolite in the most recent solution

Shadow prices are computed from the dual values of the bounds in the solution.

cobra.core.Model module

class cobra.core.Model.**Model**(*description=None*)
Bases: *cobra.core.Object.Object*

Metabolic Model

Refers to Metabolite, Reaction, and Gene Objects.

add_metabolites(*metabolite_list*)
Will add a list of metabolites to the the object, if they do not exist and then expand the stochiometric matrix

metabolite_list: A list of *Metabolite* objects

add_reaction(*reaction*)
Will add a cobra.Reaction object to the model, if reaction.id is not in self.reactions.

reaction: A *Reaction* object

add_reactions(*reaction_list*)
Will add a cobra.Reaction object to the model, if reaction.id is not in self.reactions.

reaction_list: A list of *Reaction* objects

change_objective(*objectives*)
Change the objective in the cobrapy model.

objectives: A list or a dictionary. If a list then a list of reactions for which the coefficient in the linear objective is set as 1. If a dictionary then the key is the reaction and the value is the linear coefficient for the respective reaction.

copy(*print_time=False*)
Provides a partial 'deepcopy' of the Model. All of the Metabolite, Gene, and Reaction objects are created anew but in a faster fashion than deepcopy

guided_copy()

> **Warning:** deprecated

optimize(*objective_sense='maximize'*, *solver=None*, *quadratic_component=None*, ***kwargs*)
Optimize model using flux balance analysis

objective_sense: 'maximize' or 'minimize'

solver: 'glpk', 'cglpk', 'gurobi', 'cplex' or None

quadratic_component: None or scipy.sparse.dok_matrix The dimensions should be (n, n) where n is the number of reactions.

This sets the quadratic component (Q) of the objective coefficient, adding $frac12v^T \cdot Q \cdot v$ to the objective.

tolerance_feasibility: Solver tolerance for feasibility.

tolerance_markowitz: Solver threshold during pivot

time_limit: Maximum solver time (in seconds)

Note: Only the most commonly used parameters are presented here. Additional parameters for cobra.solvers may be available and specified with the appropriate keyword argument.

---

remove_reactions (*reactions*, *delete=True*, *remove_orphans=False*)
remove reactions from the model

reactions: [*Reaction*] or [str] The reactions (or their id's) to remove

delete: Boolean Whether or not the reactions should be deleted after removal. If the reactions are not deleted, those objects will be recreated with new metabolite and gene objects.

remove_orphans: Boolean Remove orphaned genes and metabolites from the model as well

repair (*rebuild_index=True*, *rebuild_relationships=True*)
Update all indexes and pointers in a model

to_array_based_model (*deepcopy_model=False*, ***kwargs*)
Makes a *ArrayBasedModel* from a cobra.Model which may be used to perform linear algebra operations with the stoichiomatric matrix.

deepcopy_model: Boolean. If False then the ArrayBasedModel points to the Model

update ()

> **Warning:** removed

cobra.core.Object module

class cobra.core.Object.**Object** (*id=None*, *mnx_id=None*)
Bases: object

Defines common behavior of object in cobra.core

endswith (*x*)

guided_copy ()
Deprecated since version 0.3: use copy direclty

startswith (*x*)

cobra.core.Reaction module

class cobra.core.Reaction.**Frozendict**
Bases: dict

Read-only dictionary view

pop(*key*, *value*)

popitem()

class cobra.core.Reaction.**Reaction**(*name=None*)
Bases: *cobra.core.Object.Object*

Reaction is a class for holding information regarding a biochemical reaction in a cobra.Model object

add_gene(*cobra_gene*)
Deprecated since version 0.3: update the gene_reaction_rule instead

add_gene_reaction_rule(*the_rule*)
Deprecated since version 0.3: Set gene_reaction_rule directly

add_metabolites(*metabolites*, *combine=True*, *add_to_container_model=True*)
Add metabolites and stoichiometric coefficients to the reaction. If the final coefficient for a metabolite is 0 then it is removed from the reaction.

metabolites: dict {*Metabolite*: coefficient}

combine: Boolean. Describes behavior a metabolite already exists in the reaction. True causes the coefficients to be added. False causes the coefficient to be replaced. True and a metabolite already exists in the

add_to_container_model: Boolean. Add the metabolite to the *Model* the reaction is associated with (i.e. self.model)

boundary

build_reaction_from_string(*reaction_str*, *verbose=True*, *fwd_arrow=None*, *rev_arrow=None*, *reversible_arrow=None*)

build_reaction_string(*use_metabolite_names=False*)
Generate a human readable reaction string

check_mass_balance()
Makes sure that the reaction is elementally-balanced.

clear_metabolites()
Remove all metabolites from the reaction

copy()
When copying a reaction, it is necessary to deepcopy the components so the list references aren't carried over.

Additionally, a copy of a reaction is no longer in a cobra.Model.

This should be fixed with self.__deecopy__ if possible

delete(*remove_orphans=False*)
Completely delete a reaction

This removes all associations between a reaction the associated model, metabolites and genes (unlike remove_from_model which only dissociates the reaction from the model).

remove_orphans: Boolean Remove orphaned genes and metabolites from the model as well

gene_reaction_rule

genes

get_coefficient(*metabolite_id*)
Return the stoichiometric coefficient for a metabolite in the reaction.

metabolite_id: str or *Metabolite*

get_coefficients(*metabolite_ids*)
Return the stoichiometric coefficients for a list of metabolites in the reaction.

metabolite_ids: iterable Containing str or *Metabolite*

get_compartments()

get_gene()
Deprecated since version 0.3: use genes property instead

get_model()
Deprecated since version 0.3.1: use model property instead

get_products()
Deprecated since version 0.3: use products property instead

get_reactants()
Deprecated since version 0.3: use reactants property instead

guided_copy(*the_model*, *metabolite_dict*, *gene_dict=None*)
Deprecated since version 0.3: use copy directly

knock_out()
Change the upper and lower bounds of the reaction to 0.

metabolites

model
returns the model the reaction is a part of

parse_gene_association(***kwargs*)
Deprecated since version 0.3: Set gene_reaction_rule directly

pop(*metabolite_id*)
Remove a metabolite from the reaction and return the stoichiometric coefficient.

metabolite_id: str or *Metabolite*

print_values()
Deprecated since version 0.3.

products
Return a list of products for the reaction

reactants
Return a list of reactants for the reaction.

reaction
Human readable reaction string

remove_from_model(*model=None*, *remove_orphans=False*)
Removes the reaction from the model while keeping it intact

remove_orphans: Boolean Remove orphaned genes and metabolites from the model as well

model: deprecated argument, must be None

remove_gene(*cobra_gene*)
Deprecated since version 0.3: update the gene_reaction_rule instead

reversibility
Whether the reaction can proceed in both directions (reversible)

This is computed from the current upper and lower bounds.

subtract_metabolites(*metabolites*)

This function will 'subtract' metabolites from a reaction, which means add the metabolites with -1*coefficient. If the final coefficient for a metabolite is 0 then the metabolite is removed from the reaction.

metabolites: dict of {*Metabolite*: coefficient} These metabolites will be added to the reaction

---

Note: A final coefficient < 0 implies a reactant.

---

x

The flux through the reaction in the most recent solution

Flux values are computed from the primal values of the variables in the solution.

cobra.core.Solution module

class cobra.core.Solution.**Solution**(*f*, *x=None*, *x_dict=None*, *y=None*, *y_dict=None*, *solver=None*, *the_time=0*, *status='NA'*)

Bases: object

Stores the solution from optimizing a cobra.Model. This is used to provide a single interface to results from different solvers that store their values in different ways.

f: The objective value

solver: A string indicating which solver package was used.

x: List or Array of the values from the primal.

x_dict: A dictionary of reaction ids that maps to the primal values.

y: List or Array of the values from the dual.

y_dict: A dictionary of reaction ids that maps to the dual values.

dress_results(*model*)

Warning: deprecated

cobra.core.Species module

class cobra.core.Species.**Species**(*id=None*, *formula=None*, *name=None*, *compartment=None*, *mnx_id=None*)

Bases: *cobra.core.Object.Object*

Species is a class for holding information regarding a chemical Species

copy()

When copying a reaction, it is necessary to deepcopy the components so the list references aren't carried over.

Additionally, a copy of a reaction is no longer in a cobra.Model.

This should be fixed with self.__deecopy__ if possible

get_model()

Returns the Model object that contain this Object

get_reaction()

Returns a list of Reactions that contain this Species

guided_copy(*the_model*)
Deprecated since version 0.3: Use copy directly

model

parse_composition()
Breaks the chemical formula down by element. Useful for making sure Reactions are balanced.'

reactions

Module contents

11.1.2 cobra.flux_analysis package

Submodules

cobra.flux_analysis.deletion_worker module

class cobra.flux_analysis.deletion_worker.**CobraDeletionMockPool**(*cobra_model*, *n_processes=1*, *solver=None*, ***kwargs*)

Bases: object

Mock pool solves LP's in the same process

receive_all()

receive_one()

start()

submit(*indexes*, *label=None*)

terminate()

class cobra.flux_analysis.deletion_worker.**CobraDeletionPool**(*cobra_model*, *n_processes=None*, *solver=None*, ***kwargs*)

Bases: object

A pool of workers for solving deletions

submit jobs to the pool using submit and recieve results using receive_all

pids

receive_all()

receive_one()
This function blocks

start()

submit(*indexes*, *label=None*)

terminate()

cobra.flux_analysis.deletion_worker.**compute_fba_deletion**(*lp*, *solver_object*, *model*, *indexes*, ***kwargs*)

cobra.flux_analysis.deletion_worker.**compute_fba_deletion_worker**(*cobra_model*, *solver*, *job_queue*, *output_queue*, ***kwargs*)

cobra.flux_analysis.double_deletion module

cobra.flux_analysis.double_deletion.**double_deletion**(*cobra_model*, *element_list_1=None*, *element_list_2=None*, *method='fba'*, *single_deletion_growth_dict=None*, *element_type='gene'*, *solver=None*, *number_of_processes=None*, *return_frame=False*, *zero_cutoff=1e-12*, ***kwargs*)

Run double gene or reaction deletions

cobra_model: a cobra.Model object

element_list_1: None or a list of elements (genes or reactions)

element_list_2: None or a list of elements (genes or reactions)

method: 'fba' or 'moma' Whether to compute growth rates using flux balance analysis or minimization of metabolic adjustments.

number_of_processes: None or int The number of processor core to use. Setting 1 explicitly disables use of the multiprocessing library. By default, up to 4 cores will be used if available.

element_type: 'gene' or 'reaction'

zero_cutoff: float For single deletions which are assumed to be lethal, any double deletion will also be assumed to be lethal. Additionally, removing only reactions with no flux will assume the result is the same as wild-type. This parameter sets the cutoff for the absolute value to determine if a flux is 0. To disable this optimization, set this to a negative value.

solver: 'glpk', 'cglpk', 'gurobi', 'cplex' or None

return_frame: bool If True, format data as a pandas Dataframe

Returns a dictionary of the elements in the x dimension (x), the y dimension (y), and the growth simulation data (data).

cobra.flux_analysis.double_deletion.**double_gene_deletion_fba**(*cobra_model*, *gene_list1=None*, *gene_list2=None*, *solver=None*, *number_of_processes=None*, *return_frame=False*, *zero_cutoff=1e-12*, ***kwargs*)

cobra.flux_analysis.double_deletion.**double_gene_deletion_moma**(*cobra_model*, *gene_list_1=None*, *gene_list_2=None*, *method='moma'*, *single_deletion_growth_dict=None*, *solver='glpk'*, *growth_tolerance=1e-08*, *error_reporting=None*)

This will disable reactions for all gene pairs from gene_list_1 and gene_list_2 and then run simulations to optimize for the objective function. The contribution of each reaction to the objective function is indicated in cobra_model.reactions[:].objective_coefficient vector.

NOTE: We've assumed that there is no such thing as a synthetic rescue with this modeling framework.

cobra_model: a cobra.Model object

gene_list_1: Is None or a list of genes. If None then both gene_list_1 and gene_list_2 are assumed to correspond to cobra_model.genes.

gene_list_2: Is None or a list of genes. If None then gene_list_2 is assumed to correspond to gene_list_1.

method: 'fba' or 'moma' to run flux balance analysis or minimization of metabolic adjustments.

single_deletion_growth_dict: A dictionary that provides the growth rate information for single gene knock outs. This can speed up simulations because nonviable single deletion strains imply that all double deletion strains will also be nonviable.

solver: 'glpk', 'gurobi', or 'cplex'.

error_reporting: None or True

growth_tolerance: float. The effective lower bound on the growth rate for a single deletion that is still considered capable of growth.

Returns a dictionary of the gene ids in the x dimension (x) and the y dimension (y), and the growth simulation data (data).

cobra.flux_analysis.double_deletion.**double_reaction_deletion_fba**(*cobra_model*, *reaction_list1=None*, *reaction_list2=None*, *solver=None*, *number_of_processes=None*, *return_frame=False*, *zero_cutoff=1e-15*, ***kwargs*)

setting n_processes=1 explicitly disables multiprocessing

cobra.flux_analysis.essentiality module

cobra.flux_analysis.essentiality.**assess_medium_component_essentiality**(*cobra_model*, *the_components=None*, *the_medium=None*, *medium_compartment='e'*, *solver='glpk'*, *the_problem='return'*, *the_condition=None*, *method='fba'*)

Determines which components in an in silico medium are essential for growth in the context of the remaining components.

cobra_model: A Model object.

the_components: None or a list of external boundary reactions that will be sequentially disabled.

the_medium: Is None, a string, or a dictionary. If a string then the initialize_growth_medium function expects that the_model has an attribute dictionary called media_compositions, which is a dictionary of dictionaries for various medium compositions. Where a medium composition is a dictionary of external boundary reaction ids for the medium components and the external boundary fluxes for each medium component.

medium_compartment: the compartment in which the boundary reactions supplying the medium components exist

NOTE: that these fluxes must be negative because the convention is backwards means something is feed into the system.

solver: 'glpk', 'gurobi', or 'cplex'

the_problem: Is None, 'return', or an LP model object for the solver.

returns: essentiality_dict: A dictionary providing the maximum growth rate accessible when the respective component is removed from the medium.

cobra.flux_analysis.essentiality.**deletion_analysis**(*cobra_model*, *the_medium=None*, *deletion_type='single'*, *work_directory=None*, *growth_cutoff=0.001*, *the_problem='return'*, *number_of_processes=6*, *element_type='gene'*, *solver='glpk'*, *error_reporting=None*, *method='fba'*, *element_list=None*)

Performs single and/or double deletion analysis on all the genes in the model. Provides an interface to parallelize the deletion studies.

cobra.flux_analysis.loopless module

cobra.flux_analysis.loopless.**construct_loopless_model**(*cobra_model*)

construct a loopless model

This adds MILP constraints to prevent flux from proceeding in a loop, as done in http://dx.doi.org/10.1016/j.bpj.2010.12.3707 Please see the documentation for an explanation of the algorithm.

This must be solved with an MILP capable solver.

cobra.flux_analysis.moma module

cobra.flux_analysis.moma.**construct_difference_model**(*model_1*, *model_2*, *norm_type='euclidean'*)

Combine two models into a larger model that is designed to calculate differences between the models

cobra.flux_analysis.moma.**moma**(*wt_model*, *mutant_model*, *objective_sense='maximize'*, *solver=None*, *tolerance_optimality=1e-08*, *tolerance_feasibility=1e-08*, *minimize_norm=False*, *the_problem='return'*, *lp_method=0*, *combined_model=None*, *norm_type='euclidean'*)

Runs the minimization of metabolic adjustment method described in Segre et al 2002 PNAS 99(23): 15112-7.

wt_model: A cobra.Model object

mutant_model: A cobra.Model object with different reaction bounds vs wt_model. To simulate deletions

objective_sense: 'maximize' or 'minimize'

solver: 'gurobi', 'cplex', or 'glpk'. Note: glpk cannot be used with norm_type 'euclidean'

tolerance_optimality: Solver tolerance for optimality.

tolerance_feasibility: Solver tolerance for feasibility.

the_problem: None or a problem object for the specific solver that can be used to hot start the next solution.

lp_method: The method to use for solving the problem. Depends on the solver. See the cobra.flux_analysis.solvers.py file for more info.

For norm_type == 'euclidean': the primal simplex works best for the test model (gurobi: lp_method=0, cplex: lp_method=1)

combined_model: an output from moma that represents the combined optimization to be solved. when this is not none. only assume that bounds have changed for the mutant or wild-type. This saves 0.2 seconds in stacking matrices.

NOTE: Current function makes too many assumptions about the structures of the models

cobra.flux_analysis.objective module

cobra.flux_analysis.objective.**assess_objective**(*model*, *objective=None*, *objective_cutoff=0.001*, *growth_medium=None*)

DEPRECATED

cobra.flux_analysis.objective.**update_objective**(*cobra_model*, *the_objectives*)

Revised to take advantage of the new Reaction classes.

cobra_model: A cobra.Model

the_objectives: A list or a dictionary. If a list then a list of reactions for which the coefficient in the linear objective is set as 1. If a dictionary then the key is the reaction and the value is the linear coefficient for the respective reaction.

cobra.flux_analysis.parsimonious module

cobra.flux_analysis.parsimonious.**optimize_minimal_flux**(*model*, *already_irreversible=False*, ***optimize_kwargs*)

Perform basic pFBA (parsimonius FBA) and minimize total flux.

The function attempts to act as a drop-in replacement for optimize. It will make the reaction reversible and perform an optimization, then force the objective value to remain the same and minimize the total flux. Finally, it will convert the reaction back to the irreversible form it was in before. See http://dx.doi.org/10.1038/msb.2010.47

model : *Model* object

already_irreversible [bool, optional] By default, the model is converted to an irreversible one. However, if the model is already irreversible, this step can be skipped.

cobra.flux_analysis.phenotype_phase_plane module

cobra.flux_analysis.phenotype_phase_plane.**calculate_phenotype_phase_plane**(*model*, *reaction1_name*, *reaction2_name*, *reaction1_range_max=20*, *reaction2_range_max=20*, *reaction1_npoints=50*, *reaction2_npoints=50*, *solver=None*, *n_processes=1*, *tolerance=1e-06*)

calculates the growth rates while varying the uptake rates for two reactions.

returns: an object containing the growth rates for the uptake rates. To plot the result, call the plot function of the returned object.

Example: data = calculate_phenotype_phase_plane(my_model, "EX_foo", "EX_bar") data.plot()

class cobra.flux_analysis.phenotype_phase_plane.**phenotypePhasePlaneData**(*reaction1_name*, *reaction2_name*, *reaction1_range_max*, *reaction2_range_max*, *reaction1_npoints*, *reaction2_npoints*)

class to hold results of a phenotype phase plane analysis

plot()
plot the phenotype phase plane in 3D using any available backend

plot_matplotlib(*theme='Paired'*, *scale_grid=False*)
Use matplotlib to plot a phenotype phase plane in 3D.

theme: color theme to use (requires brewer2mpl)

returns: maptlotlib 3d subplot object

plot_mayavi()
Use mayavi to plot a phenotype phase plane in 3D. The resulting figure will be quick to interact with in real time, but might be difficult to save as a vector figure. returns: mlab figure object

segment(*threshold=0.01*)
attempt to segment the data and identify the various phases

cobra.flux_analysis.reaction module

cobra.flux_analysis.reaction.**assess**(*model*, *reaction*, *flux_coefficient_cutoff=0.001*)
Assesses the capacity of the model to produce the precursors for the reaction and absorb the production of the reaction while the reaction is operating at, or above, the specified cutoff.

model: A *Model* object

reaction: A *Reaction* object

flux_coefficient_cutoff: Float. The minimum flux that reaction must carry to be considered active.

returns: True if the model can produce the precursors and absorb the products for the reaction operating at, or above, flux_coefficient_cutoff. Otherwise, a dictionary of {'precursor': Status, 'product': Status}. Where Status is the results from assess_precursors and assess_products, respectively.

cobra.flux_analysis.reaction.**assess_precursors**(*model*, *reaction*, *flux_coefficient_cutoff=0.001*)
Assesses the ability of the model to provide sufficient precursors for a reaction operating at, or beyond, the specified cutoff.

model: A *Model* object

reaction: A *Reaction* object

flux_coefficient_cutoff: Float. The minimum flux that reaction must carry to be considered active.

returns: True if the precursors can be simultaneously produced at the specified cutoff. False, if the model has the capacity to produce each individual precursor at the specified threshold but not all precursors at the required level simultaneously. Otherwise a dictionary of the required and the produced fluxes for each reactant that is not produced in sufficient quantities.

cobra.flux_analysis.reaction.**assess_products**(*model*, *reaction*, *flux_coefficient_cutoff=0.001*)
Assesses whether the model has the capacity to absorb the products of a reaction at a given flux rate. Useful for identifying which components might be blocking a reaction from achieving a specific flux rate.

model: A *Model* object

reaction: A *Reaction* object

flux_coefficient_cutoff: Float. The minimum flux that reaction must carry to be considered active.

returns: True if the model has the capacity to absorb all the reaction products being simultaneously given the specified cutoff. False, if the model has the capacity to absorb each individual product but not all products at the required level simultaneously. Otherwise a dictionary of the required and the capacity fluxes for each product that is not absorbed in sufficient quantities.

cobra.flux_analysis.single_deletion module

cobra.flux_analysis.single_deletion.**single_deletion**(*cobra_model*, *element_list=None*, *method='fba'*, *element_type='gene'*, *solver=None*)

Wrapper for single_gene_deletion and the single_reaction_deletion functions

cobra_model: a cobra.Model object

element_list: Is None or a list of elements (genes or reactions) to delete.

method: 'fba' or 'moma'

the_problem: Is None, 'return', or an LP model object for the solver.

element_type: 'gene' or 'reaction'

solver: 'glpk', 'gurobi', or 'cplex'.

error_reporting: None or True to disable or enable printing errors encountered when trying to find the optimal solution.

discard_problems: Boolean. If True do not save problems. This will help with memory issues related to gurobi. .. warning:: This is deprecated.

Returns a list of dictionaries: growth_rate_dict, solution_status_dict, problem_dict where the key corresponds to each element in element_list.

cobra.flux_analysis.single_deletion.**single_gene_deletion**(*cobra_model*, *element_list=None*, *method='fba'*, *the_problem='reuse'*, *solver=None*, *error_reporting=None*)

Performs optimization simulations to realize the objective defined from cobra_model.reactions[:].objective_coefficients after deleting each gene in gene_list from the model.

cobra_model: a cobra.Model object

element_list: Is None or a list of genes to delete. If None then disable each reaction associated with each gene in cobra_model.genes.

method: 'fba' or 'moma'

the_problem: Is None or 'reuse'.

solver: 'glpk', 'gurobi', or 'cplex'.

Returns a list of dictionaries: growth_rate_dict, solution_status_dict, problem_dict where the key corresponds to each reaction in reaction_list.

TODO: Add in a section that allows copying and collection of problem for debugging purposes.

cobra.flux_analysis.single_deletion.**single_gene_deletion_fba**(*cobra_model*, *gene_list=None*, *solver=None*)

cobra.flux_analysis.single_deletion.**single_reaction_deletion**(*cobra_model*, *element_list=None*, *method='fba'*, *the_problem='return'*, *solver=None*, *error_reporting=None*, *discard_problems=True*)

Performs optimization simulations to realize the objective defined from cobra_model.reactions[:].objective_coefficients after deleting each reaction from the model.

cobra_model: a cobra.Model object

element_list: Is None or a list of cobra.Reactions in cobra_model to disable. If None then disable each reaction in cobra_model.reactions and optimize for the objective function defined from cobra_model.reactions[:].objective_coefficients.

method: 'fba' is the only option at the moment.

the_problem: Is None, 'reuse', or an LP model object for the solver.

solver: 'glpk', 'gurobi', or 'cplex'.

discard_problems: Boolean. If True do not save problems. This will help with memory issues related to gurobi.

Returns a list of dictionaries: growth_rate_dict, solution_status_dict, problem_dict where the key corresponds to each reaction in reaction_list.

cobra.flux_analysis.single_deletion.**single_reaction_deletion_fba**(*cobra_model*, *reaction_list=None*, *solver=None*)

cobra.flux_analysis.variability module

cobra.flux_analysis.variability.**find_blocked_reactions**(*cobra_model*, *reaction_list=None*, *solver=None*, *zero_cutoff=1e-09*, *open_exchanges=False*, ***kwargs*)

Finds reactions that cannot carry a flux with the current exchange reaction settings for cobra_model, using flux variability analysis.

cobra.flux_analysis.variability.**flux_variability_analysis**(*cobra_model*, *reaction_list=None*, *fraction_of_optimum=1.0*, *solver=None*, *objective_sense='maximize'*, ***solver_args*)

Runs flux variability analysis to find max/min flux values

cobra_model : *Model*:

reaction_list [list of *Reaction*: or their id's] The id's for which FVA should be run. If this is None, the bounds will be comptued for all reactions in the model.

fraction_of_optimum [fraction of optimum which must be maintained.] The original objective reaction is constrained to be greater than maximal_value * fraction_of_optimum

solver [string of solver name] If None is given, the default solver will be used.

Module contents

11.1.3 cobra.io package

Submodules

cobra.io.json module

cobra.io.json.**from_json**(*jsons*)
Load cobra model from a json string

cobra.io.json.**load_json_model**(*file_name*)
Load a cobra model stored as a json file

file_name : str or file-like object

cobra.io.json.**save_json_model**(*model*, *file_name*)
Save the cobra model as a json file.

model : *Model* object

file_name : str or file-like object

cobra.io.json.**to_json**(*model*)
Save the cobra model as a json string

cobra.io.mat module

cobra.io.mat.**create_mat_dict**(*model*)
create a dict mapping model attributes to arrays

cobra.io.mat.**from_mat_struct**(*mat_struct*, *model_id=None*)
create a model from the COBRA toolbox struct

The struct will be a dict read in by scipy.io.loadmat

cobra.io.mat.**load_matlab_model**(*infile_path*, *variable_name=None*)
Load a cobra model stored as a .mat file

infile_path : str

variable_name [str, optional] The variable name of the model in the .mat file. If this is not specified, then the first MATLAB variable which looks like a COBRA model will be used

cobra.io.mat.**save_matlab_model**(*model*, *file_name*)
Save the cobra model as a .mat file.

This .mat file can be used directly in the MATLAB version of COBRA.

model : *Model* object

file_name : str or file-like object

cobra.io.sbml module

cobra.io.sbml.**add_sbml_species**(*sbml_model*, *cobra_metabolite*, *note_start_tag*, *note_end_tag*, *boundary_metabolite=False*)

A helper function for adding cobra metabolites to an sbml model.

sbml_model: sbml_model object

cobra_metabolite: a cobra.Metabolite object

note_start_tag: the start tag for parsing cobra notes. this will eventually be supplanted when COBRA is worked into sbml.

note_end_tag: the end tag for parsing cobra notes. this will eventually be supplanted when COBRA is worked into sbml.

cobra.io.sbml.**create_cobra_model_from_sbml_file**(*sbml_filename*, *old_sbml=False*, *legacy_metabolite=False*, *print_time=False*, *use_hyphens=False*)

convert an SBML XML file into a cobra.Model object. Supports SBML Level 2 Versions 1 and 4. The function will detect if the SBML fbc package is used in the file and run the converter if the fbc package is used.

sbml_filename: String.

old_sbml: Boolean. Set to True if the XML file has metabolite formula appended to metabolite names. This was a poorly designed artifact that persists in some models.

legacy_metabolite: Boolean. If True then assume that the metabolite id has the compartment id appended after an underscore (e.g. _c for cytosol). This has not been implemented but will be soon.

print_time: deprecated

use_hyphens: Boolean. If True, double underscores (__) in an SBML ID will be converted to hyphens

cobra.io.sbml.**fix_legacy_id**(*id*, *use_hyphens=False*, *fix_compartments=False*)

cobra.io.sbml.**parse_legacy_id**(*the_id*, *the_compartment=None*, *the_type='metabolite'*, *use_hyphens=False*)

Deals with a bunch of problems due to bigg.ucsd.edu not following SBML standards

the_id: String.

the_compartment: String.

the_type: String. Currently only 'metabolite' is supported

use_hyphens: Boolean. If True, double underscores (__) in an SBML ID will be converted to hyphens

cobra.io.sbml.**parse_legacy_sbml_notes**(*note_string*, *note_delimiter=':'*)

Deal with legacy SBML format issues arising from the COBRA Toolbox for MATLAB and BiGG.ucsd.edu developers.

cobra.io.sbml.**read_legacy_sbml**(*filename*, *use_hyphens=False*)

read in an sbml file and fix the sbml id's

cobra.io.sbml.**write_cobra_model_to_sbml_file**(*cobra_model*, *sbml_filename*, *sbml_level=2*, *sbml_version=1*, *print_time=False*, *use_fbc_package=True*)

Write a cobra.Model object to an SBML XML file.

cobra_model: *Model* object

sbml_filename: The file to write the SBML XML to.

sbml_level: 2 is the only level supported at the moment.

sbml_version: 1 is the only version supported at the moment.

use_fbc_package: Boolean. Convert the model to the FBC package format to improve portability. http://sbml.org/Documents/Specifications/SBML_Level_3/Packages/Flux_Balance_Constraints_(flux)

TODO: Update the NOTES to match the SBML standard and provide support for Level 2 Version 4

Module contents

11.1.4 cobra.manipulation package

Submodules

cobra.manipulation.delete module

cobra.manipulation.delete.**delete_model_genes**(*cobra_model*, *gene_list*, *cumulative_deletions=True*, *disable_orphans=False*)

delete_model_genes will set the upper and lower bounds for reactions catalysed by the genes in gene_list if deleting the genes means that the reaction cannot proceed according to cobra_model.reactions[:].gene_reaction_rule

cumulative_deletions: False or True. If True then any previous deletions will be maintained in the model.

cobra.manipulation.delete.**find_gene_knockout_reactions**(*cobra_model*, *gene_list*, *compiled_gene_reaction_rules={}*)

identify reactions which will be disabled when the genes are knocked out

cobra_model: *Model*

gene_list: iterable of *Gene*

compiled_gene_reaction_rules: dict of {reaction_id: compiled_string} If provided, this gives pre-compiled gene_reaction_rule strings. The compiled rule strings can be evaluated much faster. If a rule is not provided, the regular expression evaluation will be used. Because not all gene_reaction_rule strings can be evaluated, this dict must exclude any rules which can not be used with eval.

cobra.manipulation.delete.**get_compiled_gene_reaction_rules**(*cobra_model*)

Generates a dict of compiled gene_reaction_rules

Any gene_reaction_rule expressions which cannot be compiled or do not evaluate after compiling will be excluded. The result can be used in the find_gene_knockout_reactions function to speed up evaluation of these rules.

cobra.manipulation.delete.**prune_unused_metabolites**(*cobra_model*)

Removes metabolites that aren't involved in any reactions in the model

cobra_model: A Model object.

cobra.manipulation.delete.**prune_unused_reactions**(*cobra_model*)

Removes reactions from cobra_model.

cobra_model: A Model object.

reactions_to_prune: None, a string matching a reaction.id, a cobra.Reaction, or as list of the ids / Reactions to remove from cobra_model. If None then the function will delete reactions that have no active metabolites in the model.

cobra.manipulation.delete.**undelete_model_genes**(*cobra_model*)

Undoes the effects of a call to delete_model_genes in place.

cobra_model: A cobra.Model which will be modified in place

cobra.manipulation.modify module

cobra.manipulation.modify.**convert_to_irreversible**(*cobra_model*)
Split reversible reactions into two irreversible reactions

These two reactions will proceed in opposite directions. This guarentees that all reactions in the model will only allow positive flux values, which is useful for some modeling problems.

cobra_model: A Model object which will be modified in place.

cobra.manipulation.modify.**initialize_growth_medium**(*cobra_model*, *the_medium='MgM'*, *external_boundary_compartment='e'*, *external_boundary_reactions=None*, *reaction_lower_bound=0.0*, *reaction_upper_bound=1000.0*, *irreversible=False*, *reactions_to_disable=None*)
Sets all of the input fluxes to the model to zero and then will initialize the input fluxes to the values specified in the_medium if it is a dict or will see if the model has a composition dict and use that to do the initialization.

cobra_model: A cobra.Model object.

the_medium: A string, or a dictionary. If a string then the initialize_growth_medium function expects that the_model has an attribute dictionary called media_compositions, which is a dictionary of dictionaries for various medium compositions. Where a medium composition is a dictionary of external boundary reaction ids for the medium components and the external boundary fluxes for each medium component.

external_boundary_compartment: None or a string. If not None then it specifies the compartment in which to disable all of the external systems boundaries.

external_boundary_reactions: None or a list of external_boundaries that are to have their bounds reset. This acts in conjunction with external_boundary_compartment.

reaction_lower_bound: Float. The default value to use for the lower bound for the boundary reactions.

reaction_upper_bound: Float. The default value to use for the upper bound for the boundary.

irreversible: Boolean. If the model is irreversible then the medium composition is taken as the upper bound

reactions_to_disable: List of reactions for which the upper and lower bounds are disabled. This is superceded by the contents of media_composition

cobra.manipulation.modify.**revert_to_reversible**(*cobra_model*, *update_solution=True*)
This function will convert a reversible model made by convert_to_irreversible into a reversible model.

cobra_model: A cobra.Model which will be modified in place.

Module contents

11.1.5 cobra.mlab package

Submodules

cobra.mlab.mlab module

cobra.mlab.mlab.**cobra_model_object_to_cobra_matlab_struct**(*cobra_model*)
This function converts all of the object values to the corresponding model fields to update the mlab model.

cobra.mlab.mlab.**init_matlab_toolbox**(*matlab_cobra_path=None*, *discover_functions=True*)
initialize the matlab cobra toolbox, and load its functions into mlab's namespace (very useful for ipython tab completion)

matlab_cobra_path: the path to the directory containing the MATLAB cobra installation. Using the default None will attempt to find the toolbox in the MATLAB path

discover_functions: Whether mlabwrap should autodiscover all cobra toolbox functions in matlab. This is convenient for tab completion, but may take some time.

cobra.mlab.mlab.**matlab_cell_to_python_list**(*the_cell*)

cobra.mlab.mlab.**matlab_cobra_struct_to_python_cobra_object**(*matlab_struct*)
Converts a COBRA toolbox 2.0 struct into a cobra.Model object using the mlabwrap matlab proxy

cobra.mlab.mlab.**matlab_logical_to_python_logical**(*the_logical*)

cobra.mlab.mlab.**matlab_sparse_to_numpy_array**(*matlab_sparse_matrix*)

cobra.mlab.mlab.**matlab_sparse_to_scipy_sparse**(*matlab_sparse_matrix*)

cobra.mlab.mlab.**numpy_array_to_mlab_sparse**(*numpy_array*)
A more efficient method is needed for when the matrix is so big that making a dense version is a waste of computer effort.

cobra.mlab.mlab.**python_list_to_matlab_cell**(*the_list*, *transpose=False*)

cobra.mlab.mlab.**scipy_sparse_to_mlab_sparse**(*scipy_sparse_matrix*)
A more efficient method is needed for when the matrix is so big that making a dense version is a waste of computer effort.

Module contents

Provides python functions which are useful for interacting with MATLAB by using the mlabwrap library.

To use MATLAB functions directly, see cobra.matlab

11.1.6 cobra.solvers package

Submodules

cobra.solvers.cglpk module

cobra.solvers.cplex_solver module

cobra.solvers.glpk_solver module

cobra.solvers.glpk_solver.**change_coefficient**(*lp*, *met_index*, *rxn_index*, *value*)

cobra.solvers.glpk_solver.**change_variable_bounds**(*lp*, *index*, *lower_bound*, *upper_bound*)

cobra.solvers.glpk_solver.**change_variable_objective**(*lp*, *index*, *objective*)

cobra.solvers.glpk_solver.**create_problem**(*cobra_model*, ***kwargs*)
Solver-specific method for constructing a solver problem from a cobra.Model. This can be tuned for performance using kwargs

cobra.solvers.glpk_solver.**format_solution**(*lp*, *cobra_model*, ***kwargs*)

cobra.solvers.glpk_solver.**get_objective_value**(*lp*)

cobra.solvers.glpk_solver.**get_status**(*lp*)

cobra.solvers.glpk_solver.**set_parameter**(*lp*, *parameter_name*, *parameter_value*)
with pyglpk the parameters are set during the solve phase, with the exception of objective sense.

cobra.solvers.glpk_solver.**solve**(*cobra_model*, ***kwargs*)
Smart interface to optimization solver functions that will convert the cobra_model to a solver object, set the parameters, and try multiple methods to get an optimal solution before returning the solver object and a cobra.Solution (which is attached to cobra_model.solution)

cobra_model: a cobra.Model

returns a dict: {'the_problem': solver specific object, 'the_solution': cobra.Solution for the optimization problem'}

cobra.solvers.glpk_solver.**solve_problem**(*lp*, ***kwargs*)
A performance tunable method for updating a model problem file

lp: a pyGLPK 0.3 problem

For pyGLPK it is necessary to provide the following parameters, if they are not provided then the default settings will be used: tolerance_feasibility, tolerance_integer, lp_method, and objective_sense

cobra.solvers.glpk_solver.**update_problem**(*lp*, *cobra_model*, ***kwargs*)
A performance tunable method for updating a model problem file

lp: A gurobi problem object

cobra_model: the cobra.Model corresponding to 'lp'

cobra.solvers.gurobi_solver module

cobra.solvers.gurobi_solver.**change_coefficient**(*lp*, *met_index*, *rxn_index*, *value*)

cobra.solvers.gurobi_solver.**change_variable_bounds**(*lp*, *index*, *lower_bound*, *upper_bound*)

cobra.solvers.gurobi_solver.**change_variable_objective**(*lp*, *index*, *objective*)

cobra.solvers.gurobi_solver.**create_problem**(*cobra_model*, *quadratic_component=None*, ***kwargs*)

Solver-specific method for constructing a solver problem from a cobra.Model. This can be tuned for performance using kwargs

cobra.solvers.gurobi_solver.**format_solution**(*lp*, *cobra_model*, ***kwargs*)

cobra.solvers.gurobi_solver.**get_objective_value**(*lp*)

cobra.solvers.gurobi_solver.**get_status**(*lp*)

cobra.solvers.gurobi_solver.**set_parameter**(*lp*, *parameter_name*, *parameter_value*)

cobra.solvers.gurobi_solver.**set_quadratic_objective**(*lp*, *quadratic_objective*)

cobra.solvers.gurobi_solver.**solve**(*cobra_model*, ***kwargs*)

cobra.solvers.gurobi_solver.**solve_problem**(*lp*, ***kwargs*)

A performance tunable method for updating a model problem file

cobra.solvers.gurobi_solver.**update_problem**(*lp*, *cobra_model*, ***kwargs*)

A performance tunable method for updating a model problem file

lp: A gurobi problem object

cobra_model: the cobra.Model corresponding to 'lp'

cobra.solvers.parameters module

Module contents

exception cobra.solvers.**SolverNotFound**

Bases: exceptions.Exception

cobra.solvers.**add_solver**(*solver_name*, *use_name=None*)

add a solver module to the solvers

cobra.solvers.**get_solver_name**(*mip=False*, *qp=False*)

returns a solver name

raises SolverNotFound if a suitable solver is not found

cobra.solvers.**optimize**(*cobra_model*, *solver=None*, ***kwargs*)

Wrapper to optimization solvers

solver [str] Name of the LP solver from solver_dict to use. If None is given, the default one will be used

11.1.7 cobra.topology package

Submodules

cobra.topology.reporter_metabolites module

cobra.topology.reporter_metabolites.**identify_reporter_metabolites**(*cobra_model*, *reaction_scores_dict*, *number_of_randomizations=1000*, *number_of_layers=1*, *scoring_metric='default'*, *score_type='p'*, *entire_network=False*, *background_correction=True*, *ignore_external_boundary_reactions=*

Calculate the aggregate Z-score for the metabolites in the model. Ignore reactions that are solely spontaneous or orphan. Allow the scores to have multiple columns / experiments. This will change the way the output is represented.

cobra_model: A cobra.Model object

TODO: CHANGE TO USING DICTIONARIES for the_reactions: the_scores

reaction_scores_dict: A dictionary where the keys are reactions in cobra_model.reactions and the values are the scores. Currently, only supports a single numeric value as the value; however, this will be updated to allow for lists

number_of_randomizations: Integer. Number of random shuffles of the scores to assess which are significant.

number_of_layers: 1 is the only option supported

scoring_metric: default means divide by k**0.5

score_type: 'p' Is the only option at the moment and indicates p-value.

entire_network: Boolean. Currently, only compares scores calculated from the_reactions

background_correction: Boolean. If True apply background correction to the aggreagate Z-score

ignore_external_boundary_reactions: Not yet implemented. Boolean. If True do not count exchange reactions when calculating the score.

cobra.topology.reporter_metabolites.**ppmap_identify_reporter_metabolites**(*keywords*)

A function that receives a dict with all of the parameters for identify_reporter_metabolites Serves to make it possible to call the reporter metabolites function from ppmap. It only will be useful for parallel experiments not for breaking up a single experiment.

Module contents

11.2 Module contents

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

C

cobra, 65
cobra.core, 49
cobra.core.ArrayBasedModel, 41
cobra.core.DictList, 42
cobra.core.Formula, 43
cobra.core.Gene, 43
cobra.core.Metabolite, 44
cobra.core.Model, 44
cobra.core.Object, 45
cobra.core.Reaction, 45
cobra.core.Solution, 48
cobra.core.Species, 48
cobra.flux_analysis, 58
cobra.flux_analysis.deletion_worker, 49
cobra.flux_analysis.double_deletion, 50
cobra.flux_analysis.essentiality, 52
cobra.flux_analysis.loopless, 52
cobra.flux_analysis.moma, 53
cobra.flux_analysis.objective, 53
cobra.flux_analysis.parsimonious, 53
cobra.flux_analysis.phenotype_phase_plane, 54
cobra.flux_analysis.reaction, 55
cobra.flux_analysis.single_deletion, 56
cobra.flux_analysis.variability, 57
cobra.io, 60
cobra.io.json, 58
cobra.io.mat, 58
cobra.io.sbml, 59
cobra.manipulation, 62
cobra.manipulation.delete, 60
cobra.manipulation.modify, 61
cobra.mlab, 62
cobra.mlab.mlab, 62
cobra.solvers, 64
cobra.solvers.glpk_solver, 63
cobra.solvers.gurobi_solver, 63
cobra.solvers.parameters, 64
cobra.topology, 65
cobra.topology.reporter_metabolites, 65

A

add_gene() (cobra.core.Reaction.Reaction method), 46
add_gene_reaction_rule() (cobra.core.Reaction.Reaction method), 46
add_metabolites() (cobra.core.ArrayBasedModel.ArrayBasedModel method), 41
add_metabolites() (cobra.core.Model.Model method), 44
add_metabolites() (cobra.core.Reaction.Reaction method), 46
add_reaction() (cobra.core.Model.Model method), 44
add_reactions() (cobra.core.ArrayBasedModel.ArrayBasedModel method), 41
add_reactions() (cobra.core.Model.Model method), 44
add_sbml_species() (in module cobra.io.sbml), 59
add_solver() (in module cobra.solvers), 64
append() (cobra.core.DictList.DictList method), 42
ArrayBasedModel (class in cobra.core.ArrayBasedModel), 41
assess() (in module cobra.flux_analysis.reaction), 55
assess_medium_component_essentiality() (in module cobra.flux_analysis.essentiality), 52
assess_objective() (in module cobra.flux_analysis.objective), 53
assess_precursors() (in module cobra.flux_analysis.reaction), 55
assess_products() (in module cobra.flux_analysis.reaction), 55

B

b (cobra.core.ArrayBasedModel.ArrayBasedModel attribute), 41
boundary (cobra.core.Reaction.Reaction attribute), 46
build_reaction_from_string() (cobra.core.Reaction.Reaction method), 46
build_reaction_string() (cobra.core.Reaction.Reaction method), 46

C

calculate_phenotype_phase_plane() (in module cobra.flux_analysis.phenotype_phase_plane), 54
change_coefficient() (in module cobra.solvers.glpk_solver), 63
change_coefficient() (in module cobra.solvers.gurobi_solver), 63
change_objective() (cobra.core.Model.Model method), 44
change_variable_bounds() (in module cobra.solvers.glpk_solver), 63
change_variable_bounds() (in module cobra.solvers.gurobi_solver), 63
change_variable_objective() (in module cobra.solvers.glpk_solver), 63
change_variable_objective() (in module cobra.solvers.gurobi_solver), 63
check_mass_balance() (cobra.core.Reaction.Reaction method), 46
clear_metabolites() (cobra.core.Reaction.Reaction method), 46
cobra (module), 65
cobra.core (module), 49
cobra.core.ArrayBasedModel (module), 41
cobra.core.DictList (module), 42
cobra.core.Formula (module), 43
cobra.core.Gene (module), 43
cobra.core.Metabolite (module), 44
cobra.core.Model (module), 44
cobra.core.Object (module), 45
cobra.core.Reaction (module), 45
cobra.core.Solution (module), 48
cobra.core.Species (module), 48
cobra.flux_analysis (module), 58
cobra.flux_analysis.deletion_worker (module), 49
cobra.flux_analysis.double_deletion (module), 50
cobra.flux_analysis.essentiality (module), 52
cobra.flux_analysis.loopless (module), 52
cobra.flux_analysis.moma (module), 53
cobra.flux_analysis.objective (module), 53
cobra.flux_analysis.parsimonious (module), 53
cobra.flux_analysis.phenotype_phase_plane (module), 54
cobra.flux_analysis.reaction (module), 55

cobra.flux_analysis.single_deletion (module), 56
cobra.flux_analysis.variability (module), 57
cobra.io (module), 60
cobra.io.json (module), 58
cobra.io.mat (module), 58
cobra.io.sbml (module), 59
cobra.manipulation (module), 62
cobra.manipulation.delete (module), 60
cobra.manipulation.modify (module), 61
cobra.mlab (module), 62
cobra.mlab.mlab (module), 62
cobra.solvers (module), 64
cobra.solvers.glpk_solver (module), 63
cobra.solvers.gurobi_solver (module), 63
cobra.solvers.parameters (module), 64
cobra.topology (module), 65
cobra.topology.reporter_metabolites (module), 65
cobra_model_object_to_cobra_matlab_struct() (in module cobra.mlab.mlab), 62
CobraDeletionMockPool (class in cobra.flux_analysis.deletion_worker), 49
CobraDeletionPool (class in cobra.flux_analysis.deletion_worker), 49
compute_fba_deletion() (in module cobra.flux_analysis.deletion_worker), 49
compute_fba_deletion_worker() (in module cobra.flux_analysis.deletion_worker), 49
constraint_sense (cobra.core.ArrayBasedModel.ArrayBasedModel attribute), 42
construct_difference_model() (in module cobra.flux_analysis.moma), 53
construct_loopless_model() (in module cobra.flux_analysis.loopless), 52
convert_to_irreversible() (in module cobra.manipulation.modify), 61
copy() (cobra.core.ArrayBasedModel.ArrayBasedModel method), 42
copy() (cobra.core.Model.Model method), 44
copy() (cobra.core.Reaction.Reaction method), 46
copy() (cobra.core.Species.Species method), 48
create_cobra_model_from_sbml_file() (in module cobra.io.sbml), 59
create_mat_dict() (in module cobra.io.mat), 58
create_problem() (in module cobra.solvers.glpk_solver), 63
create_problem() (in module cobra.solvers.gurobi_solver), 64

D

delete() (cobra.core.Reaction.Reaction method), 46
delete_model_genes() (in module cobra.manipulation.delete), 60
deletion_analysis() (in module cobra.flux_analysis.essentiality), 52
DictList (class in cobra.core.DictList), 42
double_deletion() (in module cobra.flux_analysis.double_deletion), 50
double_gene_deletion_fba() (in module cobra.flux_analysis.double_deletion), 50
double_gene_deletion_moma() (in module cobra.flux_analysis.double_deletion), 50
double_reaction_deletion_fba() (in module cobra.flux_analysis.double_deletion), 51
dress_results() (cobra.core.Solution.Solution method), 48

E

endswith() (cobra.core.Object.Object method), 45
extend() (cobra.core.DictList.DictList method), 42

F

find_blocked_reactions() (in module cobra.flux_analysis.variability), 57
find_gene_knockout_reactions() (in module cobra.manipulation.delete), 60
fix_legacy_id() (in module cobra.io.sbml), 59
flux_variability_analysis() (in module cobra.flux_analysis.variability), 57
format_solution() (in module cobra.solvers.glpk_solver), 63
format_solution() (in module cobra.solvers.gurobi_solver), 64
Formula (class in cobra.core.Formula), 43
from_json() (in module cobra.io.json), 58
from_mat_struct() (in module cobra.io.mat), 58
Frozendict (class in cobra.core.Reaction), 45

G

Gene (class in cobra.core.Gene), 43
gene_reaction_rule (cobra.core.Reaction.Reaction attribute), 46
genes (cobra.core.Reaction.Reaction attribute), 46
get_by_id() (cobra.core.DictList.DictList method), 42
get_coefficient() (cobra.core.Reaction.Reaction method), 46
get_coefficients() (cobra.core.Reaction.Reaction method), 46
get_compartments() (cobra.core.Reaction.Reaction method), 47
get_compiled_gene_reaction_rules() (in module cobra.manipulation.delete), 60
get_gene() (cobra.core.Reaction.Reaction method), 47
get_model() (cobra.core.Reaction.Reaction method), 47
get_model() (cobra.core.Species.Species method), 48
get_objective_value() (in module cobra.solvers.glpk_solver), 63
get_objective_value() (in module cobra.solvers.gurobi_solver), 64

get_products() (cobra.core.Reaction.Reaction method), 47
get_reactants() (cobra.core.Reaction.Reaction method), 47
get_reaction() (cobra.core.Species.Species method), 48
get_solver_name() (in module cobra.solvers), 64
get_status() (in module cobra.solvers.glpk_solver), 63
get_status() (in module cobra.solvers.gurobi_solver), 64
guided_copy() (cobra.core.Model.Model method), 44
guided_copy() (cobra.core.Object.Object method), 45
guided_copy() (cobra.core.Reaction.Reaction method), 47
guided_copy() (cobra.core.Species.Species method), 48

H

has_id() (cobra.core.DictList.DictList method), 42

I

identify_reporter_metabolites() (in module cobra.topology.reporter_metabolites), 65
index() (cobra.core.DictList.DictList method), 42
init_matlab_toolbox() (in module cobra.mlab.mlab), 62
initialize_growth_medium() (in module cobra.manipulation.modify), 61
insert() (cobra.core.DictList.DictList method), 42

K

knock_out() (cobra.core.Reaction.Reaction method), 47

L

list_attr() (cobra.core.DictList.DictList method), 42
load_json_model() (in module cobra.io.json), 58
load_matlab_model() (in module cobra.io.mat), 58
lower_bounds (cobra.core.ArrayBasedModel.ArrayBasedModel attribute), 42

M

matlab_cell_to_python_list() (in module cobra.mlab.mlab), 62
matlab_cobra_struct_to_python_cobra_object() (in module cobra.mlab.mlab), 62
matlab_logical_to_python_logical() (in module cobra.mlab.mlab), 62
matlab_sparse_to_numpy_array() (in module cobra.mlab.mlab), 62
matlab_sparse_to_scipy_sparse() (in module cobra.mlab.mlab), 62
Metabolite (class in cobra.core.Metabolite), 44
metabolites (cobra.core.Reaction.Reaction attribute), 47
Model (class in cobra.core.Model), 44
model (cobra.core.Reaction.Reaction attribute), 47
model (cobra.core.Species.Species attribute), 49
moma() (in module cobra.flux_analysis.moma), 53

N

numpy_array_to_mlab_sparse() (in module cobra.mlab.mlab), 62

O

Object (class in cobra.core.Object), 45
objective_coefficients (cobra.core.ArrayBasedModel.ArrayBasedModel attribute), 42
optimize() (cobra.core.Model.Model method), 44
optimize() (in module cobra.solvers), 64
optimize_minimal_flux() (in module cobra.flux_analysis.parsimonious), 53

P

parse_composition() (cobra.core.Formula.Formula method), 43
parse_composition() (cobra.core.Species.Species method), 49
parse_gene_association() (cobra.core.Reaction.Reaction method), 47
parse_legacy_id() (in module cobra.io.sbml), 59
parse_legacy_sbml_notes() (in module cobra.io.sbml), 59
phenotypePhasePlaneData (class in cobra.flux_analysis.phenotype_phase_plane), 54
pids (cobra.flux_analysis.deletion_worker.CobraDeletionPool attribute), 49
plot() (cobra.flux_analysis.phenotype_phase_plane.phenotypePhasePlaneDa method), 54
plot_matplotlib() (cobra.flux_analysis.phenotype_phase_plane.phenotypePh method), 55
plot_mayavi() (cobra.flux_analysis.phenotype_phase_plane.phenotypePhas method), 55
pop() (cobra.core.DictList.DictList method), 42
pop() (cobra.core.Reaction.Frozendict method), 45
pop() (cobra.core.Reaction.Reaction method), 47
popitem() (cobra.core.Reaction.Frozendict method), 46
ppmap_identify_reporter_metabolites() (in module cobra.topology.reporter_metabolites), 65
print_values() (cobra.core.Reaction.Reaction method), 47
products (cobra.core.Reaction.Reaction attribute), 47
prune_unused_metabolites() (in module cobra.manipulation.delete), 60
prune_unused_reactions() (in module cobra.manipulation.delete), 60
python_list_to_matlab_cell() (in module cobra.mlab.mlab), 62

Q

query() (cobra.core.DictList.DictList method), 42

R

reactants (cobra.core.Reaction.Reaction attribute), 47

Reaction (class in cobra.core.Reaction), 46
reaction (cobra.core.Reaction.Reaction attribute), 47
reactions (cobra.core.Species.Species attribute), 49
read_legacy_sbml() (in module cobra.io.sbml), 59
receive_all() (cobra.flux_analysis.deletion_worker.CobraDeletionMockPool method), 49
receive_all() (cobra.flux_analysis.deletion_worker.CobraDeletionPool method), 49
receive_one() (cobra.flux_analysis.deletion_worker.CobraDeletionMockPool method), 49
receive_one() (cobra.flux_analysis.deletion_worker.CobraDeletionPool method), 49
remove() (cobra.core.DictList.DictList method), 43
remove_from_model() (cobra.core.Gene.Gene method), 43
remove_from_model() (cobra.core.Metabolite.Metabolite method), 44
remove_from_model() (cobra.core.Reaction.Reaction method), 47
remove_gene() (cobra.core.Reaction.Reaction method), 47
remove_reactions() (cobra.core.ArrayBasedModel.ArrayBasedModel method), 42
remove_reactions() (cobra.core.Model.Model method), 45
repair() (cobra.core.Model.Model method), 45
reverse() (cobra.core.DictList.DictList method), 43
reversibility (cobra.core.Reaction.Reaction attribute), 47
revert_to_reversible() (in module cobra.manipulation.modify), 61

S

S (cobra.core.ArrayBasedModel.ArrayBasedModel attribute), 41
save_json_model() (in module cobra.io.json), 58
save_matlab_model() (in module cobra.io.mat), 58
scipy_sparse_to_mlab_sparse() (in module cobra.mlab.mlab), 62
segment() (cobra.flux_analysis.phenotype_phase_plane.phenotypePhasePlaneData method), 55
set_parameter() (in module cobra.solvers.glpk_solver), 63
set_parameter() (in module cobra.solvers.gurobi_solver), 64
set_quadratic_objective() (in module cobra.solvers.gurobi_solver), 64
single_deletion() (in module cobra.flux_analysis.single_deletion), 56
single_gene_deletion() (in module cobra.flux_analysis.single_deletion), 56
single_gene_deletion_fba() (in module cobra.flux_analysis.single_deletion), 56
single_reaction_deletion() (in module cobra.flux_analysis.single_deletion), 56
single_reaction_deletion_fba() (in module cobra.flux_analysis.single_deletion), 57
Solution (class in cobra.core.Solution), 48
solve() (in module cobra.solvers.glpk_solver), 63
solve() (in module cobra.solvers.gurobi_solver), 64
solve_problem() (in module cobra.solvers.glpk_solver), 63
solve_problem() (in module cobra.solvers.gurobi_solver), 64
SolverNotFound, 64
sort() (cobra.core.DictList.DictList method), 43
Species (class in cobra.core.Species), 48
start() (cobra.flux_analysis.deletion_worker.CobraDeletionMockPool method), 49
start() (cobra.flux_analysis.deletion_worker.CobraDeletionPool method), 49
startswith() (cobra.core.Object.Object method), 45
submit() (cobra.flux_analysis.deletion_worker.CobraDeletionMockPool method), 49
submit() (cobra.flux_analysis.deletion_worker.CobraDeletionPool method), 49
subtract_metabolites() (cobra.core.Reaction.Reaction method), 47

T

terminate() (cobra.flux_analysis.deletion_worker.CobraDeletionMockPool method), 49
terminate() (cobra.flux_analysis.deletion_worker.CobraDeletionPool method), 49
to_array_based_model() (cobra.core.Model.Model method), 45
to_json() (in module cobra.io.json), 58

U

undelete_model_genes() (in module cobra.manipulation.delete), 60
union() (cobra.core.DictList.DictList method), 43
update() (cobra.core.ArrayBasedModel.ArrayBasedModel method), 42
update() (cobra.core.Model.Model method), 45
update_objective() (in module cobra.flux_analysis.objective), 53
update_problem() (in module cobra.solvers.glpk_solver), 63
update_problem() (in module cobra.solvers.gurobi_solver), 64
upper_bounds (cobra.core.ArrayBasedModel.ArrayBasedModel attribute), 42

W

weight (cobra.core.Formula.Formula attribute), 43
write_cobra_model_to_sbml_file() (in module cobra.io.sbml), 59

X

x (cobra.core.Reaction.Reaction attribute), 48

Y

y (cobra.core.Metabolite.Metabolite attribute), 44