

---

# **cobra Documentation**

***Release 0.11.1***

**The cobrapy core team**

**Feb 06, 2018**



---

## Contents

---

<b>1</b>	<b>Getting Started</b>	<b>3</b>
1.1	Loading a model and inspecting it . . . . .	3
1.2	Reactions . . . . .	4
1.3	Metabolites . . . . .	5
1.4	Genes . . . . .	6
1.5	Making changes reversibly using models as contexts . . . . .	7
<b>2</b>	<b>Building a Model</b>	<b>9</b>
<b>3</b>	<b>Reading and Writing Models</b>	<b>13</b>
3.1	SBML . . . . .	13
3.2	JSON . . . . .	14
3.3	YAML . . . . .	14
3.4	MATLAB . . . . .	14
3.5	Pickle . . . . .	15
<b>4</b>	<b>Simulating with FBA</b>	<b>17</b>
4.1	Running FBA . . . . .	17
4.2	Changing the Objectives . . . . .	19
4.3	Running FVA . . . . .	19
4.4	Running pFBA . . . . .	21
<b>5</b>	<b>Simulating Deletions</b>	<b>23</b>
5.1	Knocking out single genes and reactions . . . . .	23
5.2	Single Deletions . . . . .	24
5.3	Double Deletions . . . . .	25
<b>6</b>	<b>Production envelopes</b>	<b>27</b>
<b>7</b>	<b>Flux sampling</b>	<b>29</b>
7.1	Basic usage . . . . .	29
7.2	Advanced usage . . . . .	30
7.3	Adding constraints . . . . .	31
<b>8</b>	<b>Loopless FBA</b>	<b>33</b>
8.1	Loopless solution . . . . .	33
8.2	Loopless model . . . . .	34

8.3	Method . . . . .	36
<b>9</b>	<b>Gapfilling</b>	<b>39</b>
<b>10</b>	<b>Solvers</b>	<b>43</b>
10.1	Internal solver interfaces . . . . .	43
<b>11</b>	<b>Tailored constraints, variables and objectives</b>	<b>45</b>
11.1	Constraints . . . . .	45
11.2	Objectives . . . . .	46
11.3	Variables . . . . .	47
<b>12</b>	<b>Using the COBRA toolbox with cobrapy</b>	<b>49</b>
<b>13</b>	<b>FAQ</b>	<b>51</b>
13.1	How do I install cobrapy? . . . . .	51
13.2	How do I cite cobrapy? . . . . .	51
13.3	How do I rename reactions or metabolites? . . . . .	51
13.4	How do I delete a gene? . . . . .	52
13.5	How do I change the reversibility of a Reaction? . . . . .	52
13.6	How do I generate an LP file from a COBRA model? . . . . .	52
<b>14</b>	<b>cobra</b>	<b>55</b>
14.1	cobra package . . . . .	55
<b>15</b>	<b>Indices and tables</b>	<b>111</b>
	<b>Python Module Index</b>	<b>113</b>

For installation instructions, please see [INSTALL.rst](#).

Many of the examples below are viewable as IPython notebooks, which can be viewed at [nbviewer](#).



### 1.1 Loading a model and inspecting it

To begin with, `cobrapy` comes with bundled models for *Salmonella* and *E. coli*, as well as a “textbook” model of *E. coli* core metabolism. To load a test model, type

```
In [1]: from __future__ import print_function

import cobra
import cobra.test

# "ecoli" and "salmonella" are also valid arguments
model = cobra.test.create_test_model("textbook")
```

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

```
In [2]: print(len(model.reactions))
print(len(model.metabolites))
print(len(model.genes))

95
72
137
```

When using `Jupyter notebook` this type of information is rendered as a table.

```
In [3]: model

Out[3]: <Model e_coli_core at 0x1116ea9e8>
```

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

```
In [4]: model.reactions[29]

Out[4]: <Reaction EX_glu__L_e at 0x11b8643c8>
```

Additionally, items can be retrieved by their `id` using the `DictList.get_by_id()` function. For example, to get the cytosolic atp metabolite object (the `id` is “atp\_c”), we can do the following:

```
In [5]: model.metabolites.get_by_id("atp_c")
Out[5]: <Metabolite atp_c at 0x11b7f82b0>
```

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:

```
In [6]: model.reactions.EX_glc__D_e.bounds
Out[6]: (-10.0, 1000.0)
```

## 1.2 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`.

```
In [7]: pgi = model.reactions.get_by_id("PGI")
        pgi
```

```
Out[7]: <Reaction PGI at 0x11b886a90>
```

We can view the full name and reaction catalyzed as strings

```
In [8]: 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.

```
In [9]: 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.

```
In [10]: pgi.check_mass_balance()
Out[10]: {}
```

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

```
In [11]: pgi.add_metabolites({model.metabolites.get_by_id("h_c"): -1})
        pgi.reaction

Out[11]: 'g6p_c + h_c <=> f6p_c'
```

The reaction is no longer mass balanced

```
In [11]: pgi.check_mass_balance()
Out[11]: {'H': -1.0, 'charge': -1.0}
```

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



```
In [12]: pgi.subtract_metabolites({model.metabolites.get_by_id("h_c"): -1})
         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.

```
In [13]: pgi.reaction = "g6p_c --> f6p_c + h_c + green_eggs + ham"

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

In [14]: pgi.reaction

Out[14]: 'g6p_c --> f6p_c + green_eggs + h_c + ham'

In [15]: pgi.reaction = "g6p_c <=> f6p_c"
         pgi.reaction

Out[15]: 'g6p_c <=> f6p_c'
```

## 1.3 Metabolites

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

```
In [16]: atp = model.metabolites.get_by_id("atp_c")
         atp

Out[16]: <Metabolite atp_c at 0x11b7f82b0>
```

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

```
In [17]: print(atp.name)
         print(atp.compartment)
```

```
ATP
c
```

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

```
In [18]: atp.charge

Out[18]: -4
```

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

```
In [19]: 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.

```
In [20]: len(atp.reactions)

Out[20]: 13
```

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

```
In [21]: model.metabolites.get_by_id("g6p_c").reactions
```

```
Out [21]: frozenset({<Reaction G6PDH2r at 0x11b870c88>,
                    <Reaction GLCpts at 0x11b870f98>,
                    <Reaction PGI at 0x11b886a90>,
                    <Reaction Biomass_Ecoli_core at 0x11b85a5f8>})
```

## 1.4 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.

```
In [22]: gpr = pgi.gene_reaction_rule
        gpr
```

```
Out [22]: 'b4025'
```

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

```
In [23]: pgi.genes
Out [23]: frozenset({<Gene b4025 at 0x11b844cc0>})
In [24]: pgi_gene = model.genes.get_by_id("b4025")
        pgi_gene
Out [24]: <Gene b4025 at 0x11b844cc0>
```

Each gene keeps track of the reactions it catalyzes

```
In [25]: pgi_gene.reactions
Out [25]: frozenset({<Reaction PGI at 0x11b886a90>})
```

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

```
In [26]: pgi.gene_reaction_rule = "(spam or eggs)"
        pgi.genes
Out [26]: frozenset({<Gene spam at 0x11b850908>, <Gene eggs at 0x11b850eb8>})
In [27]: pgi_gene.reactions
Out [27]: frozenset()
```

Newly created genes are also added to the model

```
In [28]: model.genes.get_by_id("spam")
Out [28]: <Gene spam at 0x11b850908>
```

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.

```
In [29]: cobra.manipulation.delete_model_genes(
        model, ["spam"], cumulative_deletions=True)
        print("after 1 KO: %4d < flux_PGI < %4d" % (pgi.lower_bound, pgi.upper_bound))

        cobra.manipulation.delete_model_genes(
        model, ["eggs"], cumulative_deletions=True)
        print("after 2 KO:  %4d < flux_PGI < %4d" % (pgi.lower_bound, pgi.upper_bound))

after 1 KO: -1000 < flux_PGI < 1000
after 2 KO:      0 < flux_PGI <      0
```

The `undelete_model_genes` can be used to reset a gene deletion

```
In [30]: cobra.manipulation.undelete_model_genes(model)
         print(pgi.lower_bound, "< pgi <", pgi.upper_bound)

-1000 < pgi < 1000
```

## 1.5 Making changes reversibly using models as contexts

Quite often, one wants to make small changes to a model and evaluate the impacts of these. For example, we may want to knock-out all reactions sequentially, and see what the impact of this is on the objective function. One way of doing this would be to create a new copy of the model before each knock-out with `model.copy()`. However, even with small models, this is a very slow approach as models are quite complex objects. Better then would be to do the knock-out, optimizing and then manually resetting the reaction bounds before proceeding with the next reaction. Since this is such a common scenario however, `cobrapy` allows us to use the model as a context, to have changes reverted automatically.

```
In [31]: model = cobra.test.create_test_model('textbook')
         for reaction in model.reactions[:5]:
             with model as model:
                 reaction.knock_out()
                 model.optimize()
                 print('%s blocked (bounds: %s), new growth rate %f' %
                       (reaction.id, str(reaction.bounds), model.objective.value))
```

```
ACALD blocked (bounds: (0, 0)), new growth rate 0.873922
ACALDt blocked (bounds: (0, 0)), new growth rate 0.873922
ACKr blocked (bounds: (0, 0)), new growth rate 0.873922
ACONTa blocked (bounds: (0, 0)), new growth rate -0.000000
ACONTb blocked (bounds: (0, 0)), new growth rate -0.000000
```

If we look at those knocked reactions, see that their bounds have all been reverted.

```
In [32]: [reaction.bounds for reaction in model.reactions[:5]]

Out[32]: [(-1000.0, 1000.0),
          (-1000.0, 1000.0),
          (-1000.0, 1000.0),
          (-1000.0, 1000.0),
          (-1000.0, 1000.0)]
```

Nested contexts are also supported

```
In [33]: print('original objective: ', model.objective.expression)
         with model:
             model.objective = 'ATPM'
             print('print objective in first context:', model.objective.expression)
             with model:
                 model.objective = 'ACALD'
                 print('print objective in second context:', model.objective.expression)
                 print('objective after exiting second context:',
                       model.objective.expression)
             print('back to original objective:', model.objective.expression)

original objective: -1.0*Biomass_Ecoli_core_reverse_2cdba + 1.0*Biomass_Ecoli_core
print objective in first context: -1.0*ATPM_reverse_5b752 + 1.0*ATPM
print objective in second context: 1.0*ACALD - 1.0*ACALD_reverse_fda2b
objective after exiting second context: -1.0*ATPM_reverse_5b752 + 1.0*ATPM
back to original objective: -1.0*Biomass_Ecoli_core_reverse_2cdba + 1.0*Biomass_Ecoli_core
```

Most methods that modify the model are supported like this including adding and removing reactions and metabolites and setting the objective. Supported methods and functions mention this in the corresponding documentation.

While it does not have any actual effect, for syntactic convenience it is also possible to refer to the model by a different name than outside the context. Such as

```
In [34]: with model as inner:
         inner.reactions.PFK.knock_out
```

## CHAPTER 2

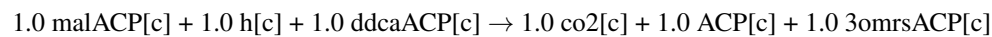
---

### Building a Model

---

This simple example 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:



First, create the model and reaction.

```
In [1]: from __future__ import print_function
In [2]: from cobra import Model, Reaction, Metabolite
        # Best practise: SBML compliant IDs
        model = Model('example_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
```

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

```
In [3]: 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.

```
In [4]: 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
```

```
Out[4]: 'ddcaACP_c + h_c + malACP_c --> 3omrsACP_c + ACP_c + co2_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.

```
In [5]: reaction.gene_reaction_rule = '( STM2378 or STM1197 )'
        reaction.genes

Out[5]: frozenset({<Gene STM1197 at 0x7f2d85786898>, <Gene STM2378 at 0x7f2dc45437f0>})
```

At this point in time, the model is still empty

```
In [6]: print('%i reactions initially' % len(model.reactions))
        print('%i metabolites initially' % len(model.metabolites))
        print('%i genes initially' % len(model.genes))

0 reactions initially
0 metabolites initially
0 genes initially
```

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

```
In [7]: model.add_reactions([reaction])

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

1 reaction
6 metabolites
2 genes
```

We can iterate through the model objects to observe the contents

```
In [8]: # Iterate through the the objects in the model
        print("Reactions")
        print("-----")
        for x in model.reactions:
            print("%s : %s" % (x.id, x.reaction))
```

```

print("")
print("Metabolites")
print("-----")
for x in model.metabolites:
    print('%9s : %s' % (x.id, x.formula))

print("")
print("Genes")
print("-----")
for x in model.genes:
    associated_ids = (i.id for i in x.reactions)
    print("%s is associated with reactions: %s" %
          (x.id, "{" + ", ".join(associated_ids) + "}"))

Reactions
-----
3OAS140 : ddcaACP_c + h_c + malACP_c --> 3omrsACP_c + ACP_c + co2_c

Metabolites
-----
    co2_c : CO2
    malACP_c : C14H22N2O10PRS
    h_c : H
    3omrsACP_c : C25H45N2O9PRS
    ddcaACP_c : C23H43N2O8PRS
    ACP_c : C11H21N2O7PRS

Genes
-----
STM1197 is associated with reactions: {3OAS140}
STM2378 is associated with reactions: {3OAS140}

```

Last we need to set the objective of the model. Here, we just want this to be the maximization of the flux in the single reaction we added and we do this by assigning the reaction's identifier to the `objective` property of the model.

```
In [9]: model.objective = '3OAS140'
```

The created objective is a symbolic algebraic expression and we can examine it by printing it

```
In [10]: print(model.objective.expression)
         print(model.objective.direction)

-1.0*3OAS140_reverse_65ddc + 1.0*3OAS140
max

```

which here shows that the solver will maximize the flux in the forward direction.





---

## Reading and Writing Models

---

Cobrapy supports reading and writing models in SBML (with and without FBC), JSON, YAML, MAT, and pickle formats. Generally, SBML with FBC version 2 is the preferred format for general use. The JSON format may be more useful for cobrapy-specific functionality.

The package also ships with test models in various formats for testing purposes.

```
In [1]: import cobra.test
import os
from os.path import join

data_dir = cobra.test.data_dir

print("mini test files: ")
print(", ".join(i for i in os.listdir(data_dir) if i.startswith("mini")))

textbook_model = cobra.test.create_test_model("textbook")
ecoli_model = cobra.test.create_test_model("ecoli")
salmonella_model = cobra.test.create_test_model("salmonella")

mini test files:
mini.json, mini.mat, mini.pickle, mini.yml, mini_cobra.xml, mini_fbc1.xml, mini_fbc2.xml, mini_fbc2.yml
```

### 3.1 SBML

The [Systems Biology Markup Language](#) is an XML-based standard format for distributing models which has support for COBRA models through the [FBC extension](#) version 2.

Cobrapy has native support for reading and writing SBML with FBCv2. Please note that all id's in the model must conform to the SBML SID requirements in order to generate a valid SBML file.

```
In [2]: cobra.io.read_sbml_model(join(data_dir, "mini_fbc2.xml"))

Out[2]: <Model mini_textbook at 0x1074fd080>

In [3]: cobra.io.write_sbml_model(textbook_model, "test_fbc2.xml")
```

There are other dialects of SBML prior to FBC 2 which have previously been used to encode COBRA models. The primary one is the “COBRA” dialect which used the “notes” fields in SBML files.

Cobrapy can use `libsbml`, which must be installed separately (see installation instructions) to read and write these files. When reading in a model, it will automatically detect whether FBC was used or not. When writing a model, the `use_fbc_package` flag can be used to write files in this legacy “cobra” format.

Consider having the `lxml` package installed as it can speed up parsing considerably.

```
In [4]: cobra.io.read_sbml_model(join(data_dir, "mini_cobra.xml"))
Out[4]: <Model mini_textbook at 0x112fa6b38>
In [5]: cobra.io.write_sbml_model(
        textbook_model, "test_cobra.xml", use_fbc_package=False)
```

## 3.2 JSON

Cobrapy models have a **JSON** (JavaScript Object Notation) representation. This format was created for interoperability with `escher`.

```
In [6]: cobra.io.load_json_model(join(data_dir, "mini.json"))
Out[6]: <Model mini_textbook at 0x113061080>
In [7]: cobra.io.save_json_model(textbook_model, "test.json")
```

## 3.3 YAML

Cobrapy models have a **YAML** (YAML Ain’t Markup Language) representation. This format was created for more human readable model representations and automatic diffs between models.

```
In [8]: cobra.io.load_yaml_model(join(data_dir, "mini.yaml"))
Out[8]: <Model mini_textbook at 0x113013390>
In [9]: cobra.io.save_yaml_model(textbook_model, "test.yaml")
```

## 3.4 MATLAB

Often, models may be imported and exported solely 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`.

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:

```
In [10]: cobra.io.load_matlab_model(
        join(data_dir, "mini.mat"), variable_name="mini_textbook")
Out[10]: <Model mini_textbook at 0x113000b70>
```

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

```
In [11]: cobra.io.load_matlab_model(join(data_dir, "mini.mat"))
Out[11]: <Model mini_textbook at 0x113758438>
```

Saving models to mat files is also relatively straightforward

```
In [12]: cobra.io.save_matlab_model(textbook_model, "test.mat")
```

## 3.5 Pickle

Cobra models can be serialized using the python serialization format, [pickle](#).

Please note that use of the pickle format is generally not recommended for most use cases. JSON, SBML, and MAT are generally the preferred formats.



---

## Simulating with FBA

---

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.

```
In [1]: import cobra.test
        model = cobra.test.create_test_model("textbook")
```

### 4.1 Running FBA

```
In [2]: solution = model.optimize()
        print(solution)
```

```
<Solution 0.874 at 0x112eb3d30>
```

The `Model.optimize()` function will return a `Solution` object. A solution object has several attributes:

- `objective_value`: the objective value
- `status`: the status from the linear programming solver
- `fluxes`: a pandas series with flux indexed by reaction identifier. The flux for a reaction variable is the difference of the primal values for the forward and reverse reaction variables.
- `shadow_prices`: a pandas series with shadow price indexed by the metabolite identifier.

For example, after the last call to `model.optimize()`, if the optimization succeeds it's status will be `optimal`. In case the model is infeasible an error is raised.

```
In [3]: solution.objective_value
```

```
Out[3]: 0.8739215069684307
```

The solvers that can be used with `cobrapy` are so fast that for many small to mid-size models computing the solution can be even faster than it takes to collect the values from the solver and convert to them python objects. With `model.optimize`, we gather values for all reactions and metabolites and that can take a significant amount of time if done repeatedly. If we are only interested in the flux value of a single reaction or the objective, it is faster to instead use `model.slim_optimize` which only does the optimization and returns the objective value leaving it up to you to fetch other values that you may need.

```
In [4]: %%time
        model.optimize().objective_value

CPU times: user 3.84 ms, sys: 672 µs, total: 4.51 ms
Wall time: 6.16 ms

Out[4]: 0.8739215069684307

In [5]: %%time
        model.slim_optimize()

CPU times: user 229 µs, sys: 19 µs, total: 248 µs
Wall time: 257 µs

Out[5]: 0.8739215069684307
```

### 4.1.1 Analyzing FBA solutions

Models solved using FBA can be further analyzed by using summary methods, which output printed text to give a quick representation of model behavior. Calling the summary method on the entire model displays information on the input and output behavior of the model, along with the optimized objective.

```
In [6]: model.summary()
```

IN FLUXES		OUT FLUXES		OBJECTIVES
o2_e	21.8	h2o_e	29.2	Biomass_Ecol... 0.874
glc__D_e	10	co2_e	22.8	
nh4_e	4.77	h_e	17.5	
pi_e	3.21			

In addition, the input-output behavior of individual metabolites can also be inspected using summary methods. For instance, the following commands can be used to examine the overall redox balance of the model

```
In [7]: model.metabolites.nadh_c.summary()
```

PRODUCING REACTIONS -- Nicotinamide adenine dinucleotide - reduced (nadh\_c)

%	FLUX	RXN ID	REACTION
42%	16	GAPD	g3p_c + nad_c + pi_c <=> 13dpg_c + h_c + nadh_c
24%	9.28	PDH	coa_c + nad_c + pyr_c --> accoa_c + co2_c + nadh_c
13%	5.06	AKGDH	akg_c + coa_c + nad_c --> co2_c + nadh_c + succ...
13%	5.06	MDH	mal__L_c + nad_c <=> h_c + nadh_c + oaa_c
8%	3.1	Biomass...	1.496 3pg_c + 3.7478 accoa_c + 59.81 atp_c + 0...

CONSUMING REACTIONS -- Nicotinamide adenine dinucleotide - reduced (nadh\_c)

%	FLUX	RXN ID	REACTION
100%	38.5	NADH16	4.0 h_c + nadh_c + q8_c --> 3.0 h_e + nad_c + q...

Or to get a sense of the main energy production and consumption reactions

```
In [8]: model.metabolites.atp_c.summary()
```

PRODUCING REACTIONS -- ATP (atp\_c)

%	FLUX	RXN ID	REACTION
67%	45.5	ATPS4r	adp_c + 4.0 h_e + pi_c <=> atp_c + h2o_c + 3.0 h_c
23%	16	PGK	3pg_c + atp_c <=> 13dpg_c + adp_c
7%	5.06	SUCOAS	atp_c + coa_c + succ_c <=> adp_c + pi_c + succoa_c

```
3%      1.76    PYK      adp_c + h_c + pep_c --> atp_c + pyr_c
```

```
CONSUMING REACTIONS -- ATP (atp_c)
```

```
-----
%      FLUX  RXN ID      REACTION
-----
76%    52.3   Biomass...  1.496 3pg_c + 3.7478 accoa_c + 59.81 atp_c + 0...
12%    8.39   ATPM        atp_c + h2o_c --> adp_c + h_c + pi_c
11%    7.48   PFK         atp_c + f6p_c --> adp_c + fdp_c + h_c
0%     0.223  GLNS        atp_c + glu__L_c + nh4_c --> adp_c + gln__L_c +...
```

## 4.2 Changing the Objectives

The objective function is determined from the `objective_coefficient` attribute of the objective reaction(s). Generally, a “biomass” function which describes the composition of metabolites which make up a cell is used.

```
In [9]: biomass_rxn = model.reactions.get_by_id("Biomass_Ecoli_core")
```

Currently in the model, there is only one reaction in the objective (the biomass reaction), with a linear coefficient of 1.

```
In [10]: from cobra.util.solver import linear_reaction_coefficients
         linear_reaction_coefficients(model)
```

```
Out[10]: {<Reaction Biomass_Ecoli_core at 0x112eab4a8>: 1.0}
```

The objective function can be changed by assigning `Model.objective`, which can be a reaction object (or just its name), or a dict of {Reaction: `objective_coefficient`}.

```
In [11]: # change the objective to ATPM
         model.objective = "ATPM"

         # The upper bound should be 1000, so that we get
         # the actual optimal value
         model.reactions.get_by_id("ATPM").upper_bound = 1000.
         linear_reaction_coefficients(model)
```

```
Out[11]: {<Reaction ATPM at 0x112eab470>: 1.0}
```

```
In [12]: model.optimize().objective_value
```

```
Out[12]: 174.99999999999996
```

We can also have more complicated objectives including quadratic terms.

## 4.3 Running FVA

FBA will not always give unique solution, because multiple flux states can achieve the same optimum. FVA (or flux variability analysis) finds the ranges of each metabolic flux at the optimum.

```
In [13]: from cobra.flux_analysis import flux_variability_analysis
```

```
In [14]: flux_variability_analysis(model, model.reactions[:10])
```

```
Out[14]: maximum      minimum
         ACALD  -2.208811e-30 -5.247085e-14
         ACALDt  0.000000e+00 -5.247085e-14
         ACKr    0.000000e+00 -8.024953e-14
         ACONTa  2.000000e+01  2.000000e+01
         ACONTh  2.000000e+01  2.000000e+01
```

```

ACT2r    0.000000e+00 -8.024953e-14
ADK1     3.410605e-13  0.000000e+00
AKGDH    2.000000e+01  2.000000e+01
AKGt2r   0.000000e+00 -2.902643e-14
ALCD2x   0.000000e+00 -4.547474e-14

```

Setting parameter `fraction_of_optimum=0.90` would give the flux ranges for reactions at 90% optimality.

```
In [15]: cobra.flux_analysis.flux_variability_analysis(
        model, model.reactions[:10], fraction_of_optimum=0.9)
```

```

Out[15]: maximum    minimum
ACALD    0.000000e+00 -2.692308
ACALDt   0.000000e+00 -2.692308
ACKr     6.635712e-30 -4.117647
ACONTa   2.000000e+01  8.461538
ACONTb   2.000000e+01  8.461538
ACT2r    0.000000e+00 -4.117647
ADK1     1.750000e+01  0.000000
AKGDH    2.000000e+01  2.500000
AKGt2r   2.651196e-16 -1.489362
ALCD2x   0.000000e+00 -2.333333

```

The standard FVA may contain loops, i.e. high absolute flux values that only can be high if they are allowed to participate in loops (a mathematical artifact that cannot happen in vivo). Use the `loopless` argument to avoid such loops. Below, we can see that FRD7 and SUCDi reactions can participate in loops but that this is avoided when using the `loopless` FVA.

```
In [16]: loop_reactions = [model.reactions.FRD7, model.reactions.SUCDi]
        flux_variability_analysis(model, reaction_list=loop_reactions, loopless=False)
```

```

Out[16]: maximum    minimum
FRD7      980.0      0.0
SUCDi     1000.0     20.0

```

```
In [17]: flux_variability_analysis(model, reaction_list=loop_reactions, loopless=True)
```

```

Out[17]: maximum    minimum
FRD7        0.0      0.0
SUCDi       20.0     20.0

```

### 4.3.1 Running FVA in summary methods

Flux variability analysis can also be embedded in calls to summary methods. For instance, the expected variability in substrate consumption and product formation can be quickly found by

```
In [18]: model.optimize()
        model.summary(fva=0.95)
```

IN FLUXES			OUT FLUXES			OBJECTIVES	
id	Flux	Range	id	Flux	Range	ATPM	175
o2_e	60	[55.9, 60]	co2_e	60	[54.2, 60]		
glc__D_e	10	[9.5, 10]	h2o_e	60	[54.2, 60]		
nh4_e	0	[0, 0.673]	for_e	0	[0, 5.83]		
pi_e	0	[0, 0.171]	h_e	0	[0, 5.83]		
			ac_e	0	[0, 2.06]		
			acald_e	0	[0, 1.35]		
			pyr_e	0	[0, 1.35]		
			etoh_e	0	[0, 1.17]		



```

lac__D_e      0 [0, 1.13]
succ_e        0 [0, 0.875]
akg_e         0 [0, 0.745]
glu__L_e      0 [0, 0.673]

```

Similarly, variability in metabolite mass balances can also be checked with flux variability analysis.

```
In [19]: model.metabolites.pyr_c.summary(fva=0.95)
```

```
PRODUCING REACTIONS -- Pyruvate (pyr_c)
```

%	FLUX	RANGE	RXN ID	REACTION
50%	10	[1.25, 18.8]	PYK	adp_c + h_c + pep_c --> atp_c + pyr_c
50%	10	[9.5, 10]	GLCpts	glc__D_e + pep_c --> g6p_c + pyr_c
0%	0	[0, 8.75]	ME1	mal__L_c + nad_c --> co2_c + nadh_c + ...
0%	0	[0, 8.75]	ME2	mal__L_c + nadp_c --> co2_c + nadph_c...

```
CONSUMING REACTIONS -- Pyruvate (pyr_c)
```

%	FLUX	RANGE	RXN ID	REACTION
100%	20	[13, 28.8]	PDH	coa_c + nad_c + pyr_c --> accoa_c + c...
0%	0	[0, 8.75]	PPS	atp_c + h2o_c + pyr_c --> amp_c + 2.0...
0%	0	[0, 5.83]	PFL	coa_c + pyr_c --> accoa_c + for_c
0%	0	[0, 1.35]	PYRt2	h_e + pyr_e <=> h_c + pyr_c
0%	0	[0, 1.13]	LDH_D	lac__D_c + nad_c <=> h_c + nadh_c + p...
0%	0	[0, 0.132]	Biomass...	1.496 3pg_c + 3.7478 accoa_c + 59.81 ...

In these summary methods, the values are reported as a the center point +/- the range of the FVA solution, calculated from the maximum and minimum values.

## 4.4 Running pFBA

Parsimonious FBA (often written pFBA) finds a flux distribution which gives the optimal growth rate, but minimizes the total sum of flux. This involves solving two sequential linear programs, but is handled transparently by cobrapy. For more details on pFBA, please see [Lewis et al. \(2010\)](#).

```
In [20]: model.objective = 'Biomass_Ecoli_core'
         fba_solution = model.optimize()
         pfba_solution = cobra.flux_analysis.pfba(model)
```

These functions should give approximately the same objective value.

```
In [21]: abs(fba_solution.fluxes["Biomass_Ecoli_core"] - pfba_solution.fluxes[
         "Biomass_Ecoli_core"])
```

```
Out[21]: 7.7715611723760958e-16
```



---

## Simulating Deletions

---

```
In [1]: import pandas
        from time import time

        import cobra.test
        from cobra.flux_analysis import (
            single_gene_deletion, single_reaction_deletion, double_gene_deletion,
            double_reaction_deletion)

        cobra_model = cobra.test.create_test_model("textbook")
        ecoli_model = cobra.test.create_test_model("ecoli")
```

### 5.1 Knocking out single genes and reactions

A commonly asked question when analyzing metabolic models is what will happen if a certain reaction was not allowed to have any flux at all. This can be tested using `cobrapy` by

```
In [2]: print('complete model: ', cobra_model.optimize())
        with cobra_model:
            cobra_model.reactions.PFK.knock_out()
            print('pfk knocked out: ', cobra_model.optimize())

complete model:  <Solution 0.874 at 0x1118cc898>
pfk knocked out:  <Solution 0.704 at 0x1118cc5c0>
```

For evaluating genetic manipulation strategies, it is more interesting to examine what happens if given genes are knocked out as doing so can affect no reactions in case of redundancy, or more reactions if a gene is participating in more than one reaction.

```
In [3]: print('complete model: ', cobra_model.optimize())
        with cobra_model:
            cobra_model.genes.b1723.knock_out()
            print('pfkA knocked out: ', cobra_model.optimize())
            cobra_model.genes.b3916.knock_out()
            print('pfkB knocked out: ', cobra_model.optimize())
```

```
complete model: <Solution 0.874 at 0x1108b81d0>
pfkA knocked out: <Solution 0.874 at 0x1108b80b8>
pfkB knocked out: <Solution 0.704 at 0x1108b8128>
```

## 5.2 Single Deletions

Perform all single gene deletions on a model

```
In [4]: deletion_results = single_gene_deletion(cobra_model)
```

These can also be done for only a subset of genes

```
In [5]: single_gene_deletion(cobra_model, cobra_model.genes[:20])
```

```
Out[5]: flux    status
        b0116  0.782351  optimal
        b0118  0.873922  optimal
        b0351  0.873922  optimal
        b0356  0.873922  optimal
        b0474  0.873922  optimal
        b0726  0.858307  optimal
        b0727  0.858307  optimal
        b1241  0.873922  optimal
        b1276  0.873922  optimal
        b1478  0.873922  optimal
        b1849  0.873922  optimal
        b2296  0.873922  optimal
        b2587  0.873922  optimal
        b3115  0.873922  optimal
        b3732  0.374230  optimal
        b3733  0.374230  optimal
        b3734  0.374230  optimal
        b3735  0.374230  optimal
        b3736  0.374230  optimal
        s0001  0.211141  optimal
```

This can also be done for reactions

```
In [6]: single_reaction_deletion(cobra_model, cobra_model.reactions[:20])
```

```
Out[6]: flux    status
        ACALD           8.739215e-01  optimal
        ACALDt          8.739215e-01  optimal
        ACKr            8.739215e-01  optimal
        ACONTa          -5.039994e-13  optimal
        ACONtb          -1.477823e-12  optimal
        Act2r           8.739215e-01  optimal
        ADK1            8.739215e-01  optimal
        AKGDH           8.583074e-01  optimal
        AKGt2r          8.739215e-01  optimal
        ALCD2x          8.739215e-01  optimal
        ATPM            9.166475e-01  optimal
        ATPS4r          3.742299e-01  optimal
        Biomass_Ecoli_core 0.000000e+00  optimal
        CO2t            4.616696e-01  optimal
        CS              1.129472e-12  optimal
        CYTBD           2.116629e-01  optimal
        D_LACt2         8.739215e-01  optimal
        ENO             1.161773e-14  optimal
```

ETOht2r	8.739215e-01	optimal
EX_ac_e	8.739215e-01	optimal

## 5.3 Double Deletions

Double deletions run in a similar way. Passing in `return_frame=True` will cause them to format the results as a `pandas.DataFrame`.

```
In [7]: double_gene_deletion(
        cobra_model, cobra_model.genes[-5:], return_frame=True).round(4)
```

```
Out[7]: b2464  b0008  b2935  b2465  b3919
        b2464  0.8739  0.8648  0.8739  0.8739  0.704
        b0008  0.8648  0.8739  0.8739  0.8739  0.704
        b2935  0.8739  0.8739  0.8739  0.0000  0.704
        b2465  0.8739  0.8739  0.0000  0.8739  0.704
        b3919  0.7040  0.7040  0.7040  0.7040  0.704
```

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 specified as well. Setting use of a single core will disable use of the multiprocessing library, which often aids debugging.

```
In [8]: start = time() # start timer()
        double_gene_deletion(
            ecoli_model, ecoli_model.genes[:300], number_of_processes=2)
        t1 = time() - start
        print("Double gene deletions for 200 genes completed in "
              "%.2f sec with 2 cores" % t1)

        start = time() # start timer()
        double_gene_deletion(
            ecoli_model, ecoli_model.genes[:300], number_of_processes=1)
        t2 = time() - start
        print("Double gene deletions for 200 genes completed in "
              "%.2f sec with 1 core" % t2)

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

```
Double gene deletions for 200 genes completed in 33.26 sec with 2 cores
Double gene deletions for 200 genes completed in 45.38 sec with 1 core
Speedup of 1.36x
```

Double deletions can also be run for reactions.

```
In [9]: double_reaction_deletion(
        cobra_model, cobra_model.reactions[2:7], return_frame=True).round(4)
```

```
Out[9]: ACKr  ACONTa  ACONtb  Act2r  ADK1
        ACKr    0.8739    0.0    0.0    0.8739  0.8739
        ACONTa  0.0000    0.0    0.0    0.0000  0.0000
        ACONtb  0.0000    0.0    0.0    0.0000 -0.0000
        Act2r   0.8739    0.0    0.0    0.8739  0.8739
        ADK1    0.8739    0.0   -0.0    0.8739  0.8739
```



---

## Production envelopes

---

Production envelopes (aka phenotype phase planes) will show distinct phases of optimal growth with different use of two different substrates. For more information, see [Edwards et al.](#)

Cobrapy supports calculating these production envelopes and they can easily be plotted using your favorite plotting package. Here, we will make one for the “textbook” *E. coli* core model and demonstrate plotting using [matplotlib](#).

```
In [1]: import cobra.test
        from cobra.flux_analysis import production_envelope

        model = cobra.test.create_test_model("textbook")
```

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

```
In [2]: prod_env = production_envelope(model, ["EX_glc__D_e", "EX_o2_e"])
```

```
In [3]: prod_env.head()
```

```
Out[3]: carbon_source  carbon_yield_maximum  carbon_yield_minimum  flux_maximum  \
0    EX_glc__D_e          1.442300e-13              0.0          0.000000
1    EX_glc__D_e          1.310050e+00              0.0          0.072244
2    EX_glc__D_e          2.620100e+00              0.0          0.144488
3    EX_glc__D_e          3.930150e+00              0.0          0.216732
4    EX_glc__D_e          5.240200e+00              0.0          0.288975

      flux_minimum  mass_yield_maximum  mass_yield_minimum  EX_glc__D_e  \
0              0.0              NaN              NaN          -10.0
1              0.0              NaN              NaN          -10.0
2              0.0              NaN              NaN          -10.0
3              0.0              NaN              NaN          -10.0
4              0.0              NaN              NaN          -10.0

      EX_o2_e
0 -60.000000
1 -56.842105
2 -53.684211
3 -50.526316
4 -47.368421
```

If we specify the carbon source, we can also get the carbon and mass yield. For example, temporarily setting the objective to produce acetate instead we could get production envelope as follows and pandas to quickly plot the results.

```
In [4]: prod_env = production_envelope(
        model, ["EX_o2_e"], objective="EX_ac_e", carbon_sources="EX_glc__D_e")
```

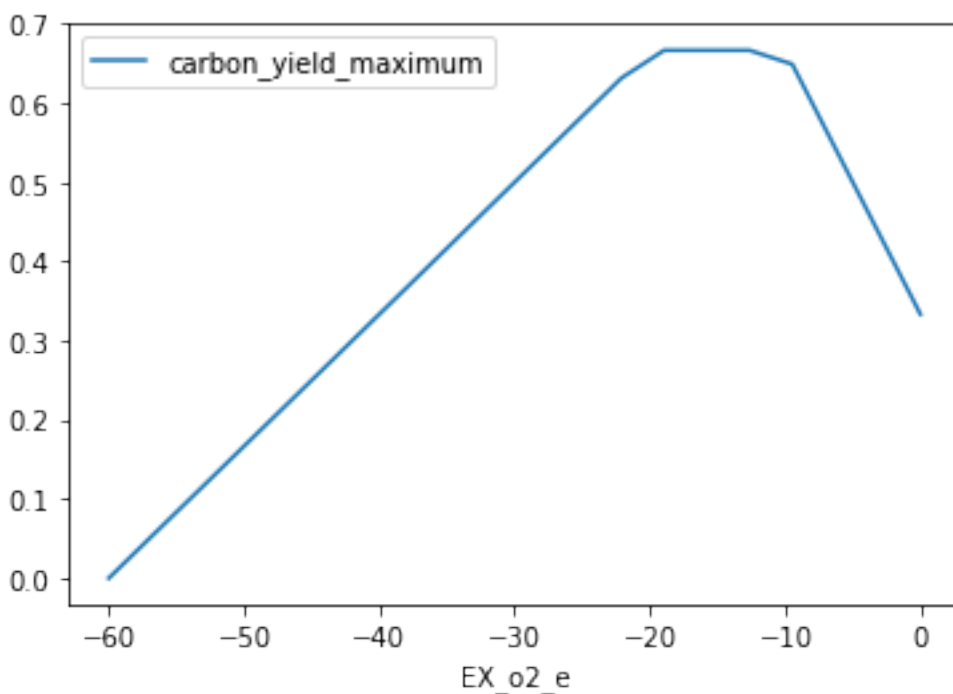
```
In [5]: prod_env.head()
```

```
Out[5]: carbon_source  carbon_yield_maximum  carbon_yield_minimum  flux_maximum \
0    EX_glc__D_e          2.385536e-15          0.0          0.000000
1    EX_glc__D_e          5.263158e-02          0.0          1.578947
2    EX_glc__D_e          1.052632e-01          0.0          3.157895
3    EX_glc__D_e          1.578947e-01          0.0          4.736842
4    EX_glc__D_e          2.105263e-01          0.0          6.315789

      flux_minimum  mass_yield_maximum  mass_yield_minimum  EX_o2_e
0              0.0      2.345496e-15              0.0 -60.000000
1              0.0      5.174819e-02              0.0 -56.842105
2              0.0      1.034964e-01              0.0 -53.684211
3              0.0      1.552446e-01              0.0 -50.526316
4              0.0      2.069927e-01              0.0 -47.368421
```

```
In [6]: %matplotlib inline
```

```
In [7]: prod_env.plot(
        kind='line', x='EX_o2_e', y='carbon_yield_maximum');
```



Previous versions of cobrapy included more tailored plots for phase planes which have now been dropped in order to improve maintainability and enhance the focus of cobrapy. Plotting for cobra models is intended for another package.



## 7.1 Basic usage

The easiest way to get started with flux sampling is using the `sample` function in the `flux_analysis` submodule. `sample` takes at least two arguments: a cobra model and the number of samples you want to generate.

```
In [1]: from cobra.test import create_test_model
        from cobra.flux_analysis import sample

        model = create_test_model("textbook")
        s = sample(model, 100)
        s.head()
```

```
Out[1]: ACALD      ACALDt      ACKr      ACONTa      ACONTb      Act2r      ADK1  \
0 -3.706944 -0.163964 -0.295823  8.975852  8.975852 -0.295823  4.847986
1 -1.340710 -0.175665 -0.429169 11.047827 11.047827 -0.429169  2.901598
2 -1.964087 -0.160334 -0.618029  9.811474  9.811474 -0.618029 17.513791
3 -0.838442 -0.123865 -0.376067 11.869552 11.869552 -0.376067  7.769872
4 -0.232088 -0.034346 -1.067684  7.972039  7.972039 -1.067684  5.114975

          AKGDH      AKGt2r      ALCD2x      ...      RPI      SUCct2_2      SUCct3  \
0  6.406533 -0.081797 -3.542980      ...      -1.649393  20.917568  20.977290
1  7.992916 -0.230564 -1.165045      ...      -0.066975  24.735567  24.850041
2  8.635576 -0.284992 -1.803753      ...      -4.075515  23.425719  23.470968
3  9.765178 -0.325219 -0.714577      ...      -0.838094  23.446704  23.913036
4  5.438125 -0.787864 -0.197742      ...      -3.109205  8.902309  9.888083

          SUCDi      SUCOAS      TALA      THD2      TKT1      TKT2      TPI
0  744.206008 -6.406533  1.639515  1.670533  1.639515  1.635542  6.256787
1  710.481004 -7.992916  0.056442  9.680476  0.056442  0.052207  7.184752
2  696.114154 -8.635576  4.063291  52.316496  4.063291  4.058376  5.122237
3  595.787313 -9.765178  0.822987  36.019720  0.822987  0.816912  8.364314
4  584.552692 -5.438125  3.088152  12.621811  3.088152  3.079686  6.185089

[5 rows x 95 columns]
```

By default sample uses the `optgp` method based on the [method presented here](#) as it is suited for larger models and can run in parallel. By default the sampler uses a single process. This can be changed by using the `processes` argument.

```
In [2]: print("One process:")
        %time s = sample(model, 1000)
        print("Two processes:")
        %time s = sample(model, 1000, processes=2)
```

```
One process:
CPU times: user 5.31 s, sys: 433 ms, total: 5.74 s
Wall time: 5.27 s
Two processes:
CPU times: user 217 ms, sys: 488 ms, total: 705 ms
Wall time: 2.8 s
```

Alternatively you can also use Artificial Centering Hit-and-Run for sampling by setting the method to `achr`. `achr` does not support parallel execution but has good convergence and is almost Markovian.

```
In [3]: s = sample(model, 100, method="achr")
```

In general setting up the sampler is expensive since initial search directions are generated by solving many linear programming problems. Thus, we recommend to generate as many samples as possible in one go. However, this might require finer control over the sampling procedure as described in the following section.

## 7.2 Advanced usage

### 7.2.1 Sampler objects

The sampling process can be controlled on a lower level by using the sampler classes directly.

```
In [4]: from cobra.flux_analysis.sampling import OptGPSampler, ACHRSampler
```

Both sampler classes have standardized interfaces and take some additional argument. For instance the `thinning` factor. “Thinning” means only recording samples every `n` iterations. A higher thinning factor means less correlated samples but also larger computation times. By default the samplers use a thinning factor of 100 which creates roughly uncorrelated samples. If you want less samples but better mixing feel free to increase this parameter. If you want to study convergence for your own model you might want to set it to 1 to obtain all iterates.

```
In [5]: achr = ACHRSampler(model, thinning=10)
```

`OptGPSampler` has an additional `processes` argument specifying how many processes are used to create parallel sampling chains. This should be in the order of your CPU cores for maximum efficiency. As noted before class initialization can take up to a few minutes due to generation of initial search directions. Sampling on the other hand is quick.

```
In [6]: optgp = OptGPSampler(model, processes=4)
```

### 7.2.2 Sampling and validation

Both samplers have a `sample` function that generates samples from the initialized object and act like the `sample` function described above, only that this time it will only accept a single argument, the number of samples. For `OptGPSampler` the number of samples should be a multiple of the number of processes, otherwise it will be increased to the nearest multiple automatically.

```
In [7]: s1 = achr.sample(100)

        s2 = optgp.sample(100)
```

You can call `sample` repeatedly and both samplers are optimized to generate large amount of samples without falling into “numerical traps”. All sampler objects have a `validate` function in order to check if a set of points are feasible and give detailed information about feasibility violations in a form of a short code denoting feasibility. Here the short code is a combination of any of the following letters:

- “v” - valid point
- “l” - lower bound violation
- “u” - upper bound violation
- “e” - equality violation (meaning the point is not a steady state)

For instance for a random flux distribution (should not be feasible):

```
In [8]: import numpy as np
```

```
bad = np.random.uniform(-1000, 1000, size=len(model.reactions))
achr.validate(np.atleast_2d(bad))
```

```
Out[8]: array(['le'],
             dtype='<U3')
```

And for our generated samples:

```
In [9]: achr.validate(s1)
```

```
Out[9]: array(['v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v',
              'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v',
              'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v',
              'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v',
              'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v',
              'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v',
              'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v',
              'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v', 'v'],
             dtype='<U3')
```

## 7.2.3 Batch sampling

Sampler objects are made for generating billions of samples, however using the `sample` function might quickly fill up your RAM when working with genome-scale models. Here, the `batch` method of the sampler objects might come in handy. `batch` takes two arguments, the number of samples in each batch and the number of batches. This will make sense with a small example.

Let’s assume we want to quantify what proportion of our samples will grow. For that we might want to generate 10 batches of 50 samples each and measure what percentage of the individual 100 samples show a growth rate larger than 0.1. Finally, we want to calculate the mean and standard deviation of those individual percentages.

```
In [10]: counts = [np.mean(s.Biomass_Ecoli_core > 0.1) for s in optgp.batch(100, 10)]
          print("Usually {:.2f}% +- {:.2f}% grow...".format(
              np.mean(counts) * 100.0, np.std(counts) * 100.0))
```

Usually 8.70% +- 2.72% grow...

## 7.3 Adding constraints

Flux sampling will respect additional constraints defined in the model. For instance we can add a constraint enforcing growth in a similar manner as the section before.

```
In [11]: co = model.problem.Constraint(model.reactions.Biomass_Ecoli_core.flux_expression, lb=0.1)
         model.add_cons_vars([co])
```

*Note that this is only for demonstration purposes. usually you could set the lower bound of the reaction directly instead of creating a new constraint.*

```
In [12]: s = sample(model, 10)
         print(s.Biomass_Ecoli_core)

0    0.175547
1    0.111499
2    0.123073
3    0.151874
4    0.122541
5    0.121878
6    0.147333
7    0.106499
8    0.174448
9    0.143273
Name: Biomass_Ecoli_core, dtype: float64
```

As we can see our new constraint was respected.

The goal of this procedure is identification of a thermodynamically consistent flux state without loops, as implied by the name. You can find a more detailed description in the *method* section at the end of the notebook.

```
In [1]: %matplotlib inline
import plot_helper

import cobra.test
from cobra import Reaction, Metabolite, Model
from cobra.flux_analysis.loopless import add_loopless, loopless_solution
from cobra.flux_analysis import pfba
```

## 8.1 Loopless solution

Classical loopless approaches as described below are computationally expensive to solve due to the added mixed-integer constraints. A much faster, and pragmatic approach is instead to post-process flux distributions to simply set fluxes to zero wherever they can be zero without changing the fluxes of any exchange reactions in the model. *CycleFreeFlux* is an algorithm that can be used to achieve this and in cobrapy it is implemented in the `cobra.flux_analysis.loopless_solution` function. `loopless_solution` will identify the closest flux distribution (using only loopless elementary flux modes) to the original one. Note that this will not remove loops which you explicitly requested, for instance by forcing a loop reaction to carry non-zero flux.

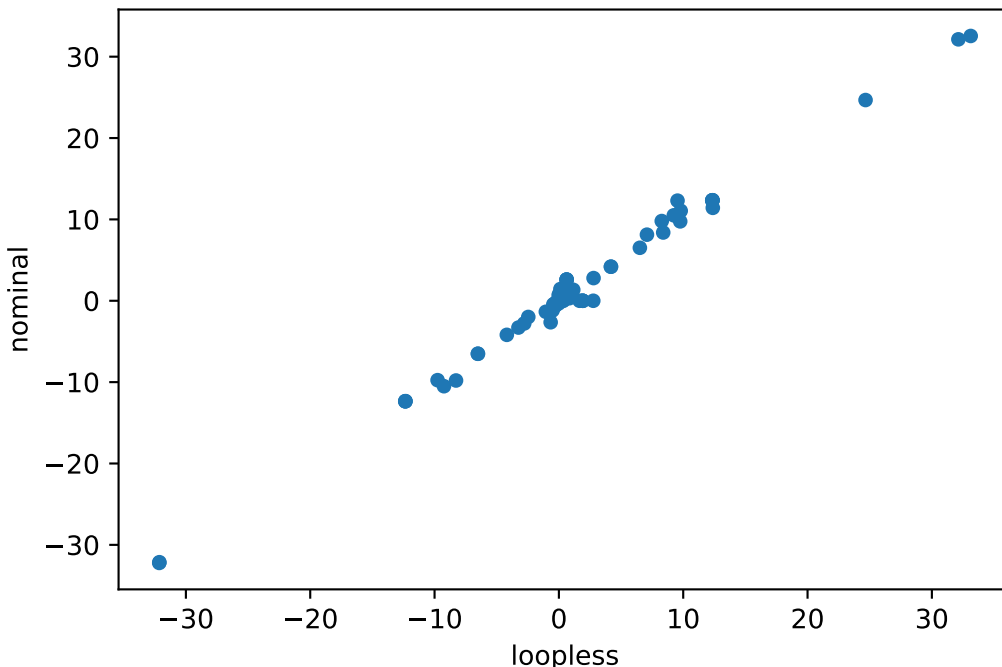
Using a larger model than the simple example above, this can be demonstrated as follows

```
In [2]: salmonella = cobra.test.create_test_model('salmonella')
        nominal = salmonella.optimize()
        loopless = loopless_solution(salmonella)

In [3]: import pandas
        df = pandas.DataFrame(dict(loopless=loopless.fluxes, nominal=nominal.fluxes))

In [4]: df.plot.scatter(x='loopless', y='nominal')

Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x10f7cb3c8>
```



This functionality can also be used in FVA by using the `loopless=True` argument to avoid getting high flux ranges for reactions that essentially only can reach high fluxes if they are allowed to participate in loops (see the simulation notebook) leading to much narrower flux ranges.

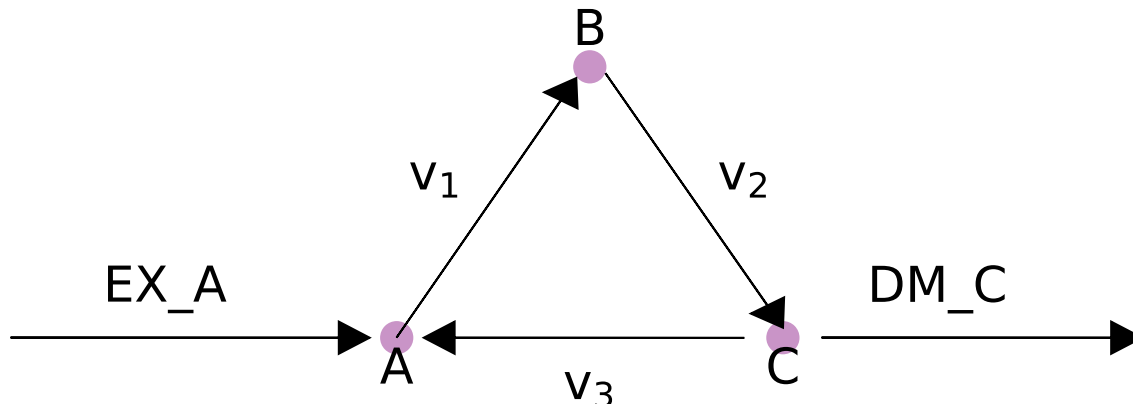
## 8.2 Loopless model

Cobrapy also includes the “classical” loopless formulation by [Schellenberger et. al.](#) implemented in `cobra.flux_analysis.add_loopless` modify the model with additional mixed-integer constraints that make thermodynamically infeasible loops impossible. This is much slower than the strategy provided above and should only be used if one of the two following cases applies:

1. You want to combine a non-linear (e.g. quadratic) objective with the loopless condition
2. You want to force the model to be infeasible in the presence of loops independent of the set reaction bounds.

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

```
In [5]: plot_helper.plot_loop()
```



```

In [6]: model = Model()
        model.add_metabolites([Metabolite(i) for i in "ABC"])
        model.add_reactions([Reaction(i) for i in ["EX_A", "DM_C", "v1", "v2", "v3"]])

        model.reactions.EX_A.add_metabolites({"A": 1})
        model.reactions.DM_C.add_metabolites({"C": -1})

        model.reactions.v1.add_metabolites({"A": -1, "B": 1})
        model.reactions.v2.add_metabolites({"B": -1, "C": 1})
        model.reactions.v3.add_metabolites({"C": -1, "A": 1})

        model.objective = 'DM_C'

```

While this model contains a loop, a flux state exists which has no flux through reaction  $v_3$ , and is identified by loopless FBA.

```

In [7]: with model:
        add_loopless(model)
        solution = model.optimize()
        print("loopless solution: status = " + solution.status)
        print("loopless solution flux: v3 = %.1f" % solution.fluxes["v3"])

loopless solution: status = optimal
loopless solution flux: v3 = 0.0

```

If there is no forced flux through a loopless reaction, parsimonious FBA will also have no flux through the loop.

```

In [8]: solution = pfba(model)
        print("parsimonious solution: status = " + solution.status)
        print("loopless solution flux: v3 = %.1f" % solution.fluxes["v3"])

parsimonious solution: status = optimal
loopless solution flux: v3 = 0.0

```

However, if flux is forced through  $v_3$ , then there is no longer a feasible loopless solution, but the parsimonious solution will still exist.

```

In [9]: model.reactions.v3.lower_bound = 1
        with model:
            add_loopless(model)
            try:
                solution = model.optimize()

```

```
except:
    print('model is infeasible')
model is infeasible
cobra/util/solver.py:398 UserWarning: solver status is 'infeasible'
In [10]: solution = pfba(model)
         print("parsimonious solution: status = " + solution.status)
         print("loopless solution flux: v3 = %.1f" % solution.fluxes["v3"])
parsimonious solution: status = optimal
loopless solution flux: v3 = 1.0
```

## 8.3 Method

`loopless_solution` is based on a given reference flux distribution. It will look for a new flux distribution with the following requirements:

1. The objective value is the same as in the reference fluxes.
2. All exchange fluxes have the same value as in the reference distribution.
3. All non-exchange fluxes have the same sign (flow in the same direction) as the reference fluxes.
4. The sum of absolute non-exchange fluxes is minimized.

As proven in the [original publication](#) this will identify the “least-loopy” solution closest to the reference fluxes.

If you are using `add_loopless` this will use the method [described here](#). In summary, it will add  $G \approx \Delta G$  proxy variables and make loops thermodynamically infeasible. This is achieved by the following formulation.

to

$$\begin{aligned} & \text{maximize } v_{obj} \\ & s.t. Sv = 0 \\ & lb_j \leq v_j \leq ub_j \\ & -M \cdot (1 - a_i) \leq v_i \leq M \cdot a_i \\ & -1000a_i + (1 - a_i) \leq G_i \leq -a_i + 1000(1 - a_i) \\ & N_{int}G = 0 \\ & a_i \in \{0, 1\} \end{aligned} \quad (8.1)$$

$$\begin{aligned} & Sv = 0 \\ & -M \cdot (1 - a_i) \leq v_i \leq M \cdot a_i \\ & N_{int}G = 0 \end{aligned}$$



Here the index  $j$  runs over all reactions and the index  $i$  only over internal ones.  $a_i$  are indicator variables which equal one if the reaction flux flows in the forward direction and 0 otherwise. They are used to force the  $G$  proxies to always carry the opposite sign of the flux (as it is the case for the “real”  $\Delta G$  values).  $N_{int}$  is the nullspace matrix for internal reactions and is used to find thermodynamically “correct” values for  $G$ .



## Gapfilling

Model gap filling is the task of figuring out which reactions have to be added to a model to make it feasible. Several such algorithms have been reported e.g. [Kumar et al. 2009](#) and [Reed et al. 2006](#). Cobrapy has a gap filling implementation that is very similar to that of Reed et al. where we use a mixed-integer linear program to figure out the smallest number of reactions that need to be added for a user-defined collection of reactions, i.e. a universal model. Briefly, the problem that we try to solve is

Minimize:

$$\sum_i c_i * z_i$$

subject to

$$Sv = 0$$

$$v^* \geq t$$

$$l_i \leq v_i \leq u_i$$

$$v_i = 0 \text{ if } z_i = 0$$

Where  $l$ ,  $u$  are lower and upper bounds for reaction  $i$  and  $z$  is an indicator variable that is zero if the reaction is not used and otherwise 1,  $c$  is a user-defined cost associated with using the  $i$ th reaction,  $v^*$  is the flux of the objective and  $t$  a lower bound for that objective. To demonstrate, let's take a model and remove some essential reactions from it.

```
In [1]: import cobra.test
        from cobra.flux_analysis import gapfill
        model = cobra.test.create_test_model("salmonella")
```

In this model D-Fructose-6-phosphate is an essential metabolite. We will remove all the reactions using it, and at them to a separate model.

```
In [2]: universal = cobra.Model("universal_reactions")
        for i in [i.id for i in model.metabolites.f6p_c.reactions]:
            reaction = model.reactions.get_by_id(i)
            universal.add_reaction(reaction.copy())
            model.remove_reactions([reaction])
```

Now, because of these gaps, the model won't grow.

```
In [3]: model.optimize().objective_value
```

```
Out[3]: 0.0
```

We will use can use the model's original objective, growth, to figure out which of the removed reactions are required for the model be feasible again. This is very similar to making the 'no-growth but growth (NGG)' predictions of Kumar et al. 2009.

```
In [4]: solution = gapfill(model, universal, demand_reactions=False)
        for reaction in solution[0]:
            print(reaction.id)
```

```
GF6PTA
F6PP
TKT2
FBP
MAN6PI
```

We can obtain multiple possible reaction sets by having the algorithm go through multiple iterations.

```
In [5]: result = gapfill(model, universal, demand_reactions=False, iterations=4)
        for i, entries in enumerate(result):
            print("---- Run %d ----" % (i + 1))
            for e in entries:
                print(e.id)
```

```
---- Run 1 ----
GF6PTA
F6PP
TKT2
FBP
MAN6PI
---- Run 2 ----
GF6PTA
TALA
PGI
F6PA
MAN6PI
---- Run 3 ----
GF6PTA
F6PP
TKT2
FBP
MAN6PI
---- Run 4 ----
GF6PTA
TALA
PGI
F6PA
MAN6PI
```

We can also instead of using the original objective, specify a given metabolite that we want the model to be able to produce.

```
In [6]: with model:
        model.objective = model.add_boundary(model.metabolites.f6p_c, type='demand')
        solution = gapfill(model, universal)
        for reaction in solution[0]:
            print(reaction.id)
```

```
FBP
```

Finally, note that using mixed-integer linear programming is computationally quite expensive and for larger models you may want to consider alternative [gap filling methods](#) and [reconstruction methods](#).



A constraints-based reconstruction and analysis model for biological systems is actually just an application of a class of discrete optimization problems typically solved with [linear](#), [mixed integer](#) or [quadratic programming](#) techniques. Cobrapy does not implement any algorithms to find solutions to such problems but rather creates an biologically motivated abstraction to these techniques to make it easier to think of how metabolic systems work without paying much attention to how that formulates to an optimization problem.

The actual solving is instead done by tools such as the free software [glpk](#) or commercial tools [gurobi](#) and [cplex](#) which are all made available as a common programmers interface via the [optlang](#) package.

When you have defined your model, you can switch solver backend by simply assigning to the `model.solver` property.

```
In [1]: import cobra.test
        model = cobra.test.create_test_model('textbook')

In [2]: model.solver = 'glpk'
        # or if you have cplex installed
        model.solver = 'cplex'
```

For information on how to configure and tune the solver, please see the [documentation for optlang project](#) and note that `model.solver` is simply an object `optlang` of class `Model`.

```
In [3]: type(model.solver)

Out[3]: optlang.cplex_interface.Model
```

## 10.1 Internal solver interfaces

Cobrapy also contains its own solver interfaces but these are now deprecated and will be removed completely in the near future. For documentation of how to use these, please refer to [older documentation](#).





---

## Tailored constraints, variables and objectives

---

Thanks to the use of symbolic expressions via the `optlang` mathematical modeling package, it is relatively straightforward to add new variables, constraints and advanced objectives that can not easily be formulated as a combination of different reaction and their corresponding upper and lower bounds. Here we demonstrate this `optlang` functionality which is exposed via the `model.solver.interface`.

### 11.1 Constraints

Suppose we want to ensure that two reactions have the same flux in our model. We can add this criteria as constraint to our model using the `optlang` solver interface by simply defining the relevant expression as follows.

```
In [1]: import cobra.test
        model = cobra.test.create_test_model('textbook')

In [2]: same_flux = model.problem.Constraint(
        model.reactions.FBA.flux_expression - model.reactions.NH4t.flux_expression,
        lb=0,
        ub=0)
        model.add_cons_vars(same_flux)
```

The flux for our reaction of interest is obtained by the `model.reactions.FBA.flux_expression` which is simply the sum of the forward and reverse flux, i.e.,

```
In [3]: model.reactions.FBA.flux_expression
```

```
Out[3]: 1.0*FBA - 1.0*FBA_reverse_84806
```

Now I can maximize growth rate whilst the fluxes of reactions 'FBA' and 'NH4t' are constrained to be (near) identical.

```
In [4]: solution = model.optimize()
        print(solution.fluxes['FBA'], solution.fluxes['NH4t'],
              solution.objective_value)
```

```
4.66274904774 4.66274904774 0.855110960926157
```

## 11.2 Objectives

Simple objective such as the maximization of the flux through one or more reactions can conveniently be done by simply assigning to the `model.objective` property as we have seen in previous chapters, e.g.,

```
In [5]: model = cobra.test.create_test_model('textbook')
        with model:
            model.objective = {model.reactions.Biomass_Ecoli_core: 1}
            model.optimize()
            print(model.reactions.Biomass_Ecoli_core.flux)

0.8739215069684307
```

The objectives mathematical expression is seen by

```
In [6]: model.objective.expression

Out[6]: -1.0*Biomass_Ecoli_core_reverse_2cdba + 1.0*Biomass_Ecoli_core
```

But suppose we need a more complicated objective, such as minimizing the Euclidean distance of the solution to the origin minus another variable, while subject to additional linear constraints. This is an objective function with both linear and quadratic components.

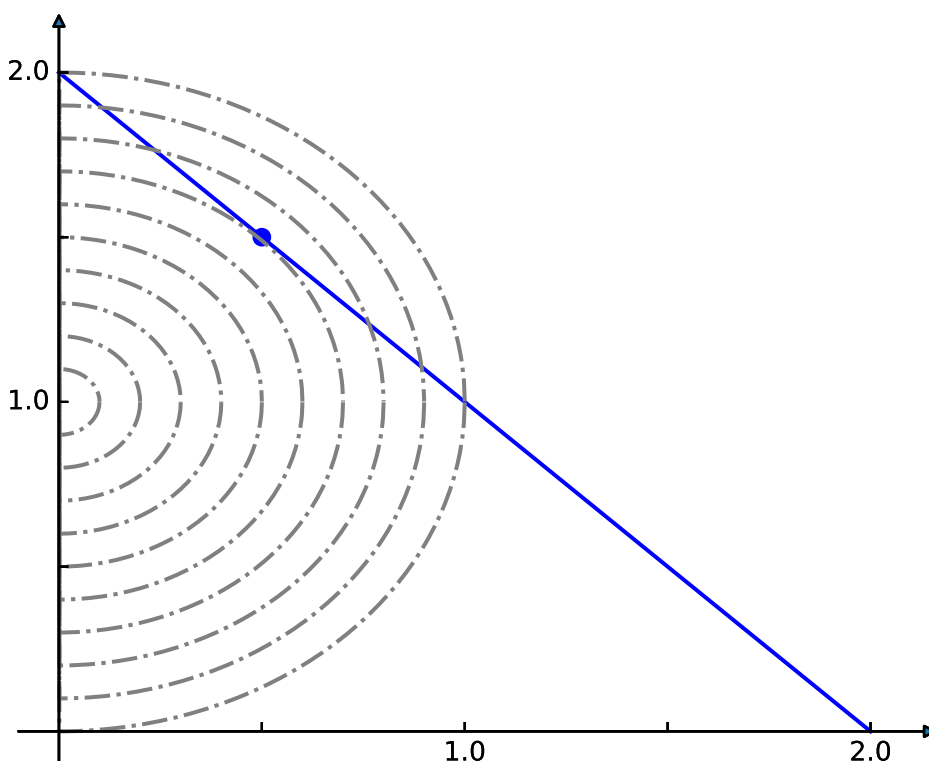
Consider the example problem:

$$\begin{aligned} \min \quad & \frac{1}{2} (x^2 + y^2) - y \\ \text{subject to} \quad & \\ & x + y = 2 \\ & x \geq 0 \\ & y \geq 0 \end{aligned}$$

This (admittedly very artificial) problem can be visualized graphically where the optimum is indicated by the blue dot on the line of feasible solutions.

```
In [7]: %matplotlib inline
        import plot_helper

        plot_helper.plot_qp2()
```



We return to the textbook model and set the solver to one that can handle quadratic objectives such as cplex. We then add the linear constraint that the sum of our x and y reactions, that we set to FBA and NH4t, must equal 2.

```
In [8]: model.solver = 'cplex'
        sum_two = model.problem.Constraint(
            model.reactions.FBA.flux_expression + model.reactions.NH4t.flux_expression,
            lb=2,
            ub=2)
        model.add_cons_vars(sum_two)
```

Next we add the quadratic objective

```
In [9]: quadratic_objective = model.problem.Objective(
        0.5 * model.reactions.NH4t.flux_expression**2 + 0.5 *
        model.reactions.FBA.flux_expression**2 -
        model.reactions.FBA.flux_expression,
        direction='min')
        model.objective = quadratic_objective
        solution = model.optimize(objective_sense=None)
```

```
In [10]: print(solution.fluxes['NH4t'], solution.fluxes['FBA'])
```

```
0.5 1.5
```

## 11.3 Variables

We can also create additional variables to facilitate studying the effects of new constraints and variables. Suppose we want to study the difference in flux between nitrogen and carbon uptake whilst we block other reactions. For this it will may help to add another variable representing this difference.

```
In [11]: model = cobra.test.create_test_model('textbook')
         difference = model.problem.Variable('difference')
```

We use constraints to define what values this variable shall take

```
In [12]: constraint = model.problem.Constraint(
         model.reactions.EX_glc__D_e.flux_expression -
         model.reactions.EX_nh4_e.flux_expression - difference,
         lb=0,
         ub=0)
         model.add_cons_vars([difference, constraint])
```

Now we can access that difference directly during our knock-out exploration by looking at its primal value.

```
In [13]: for reaction in model.reactions[:5]:
         with model:
             reaction.knock_out()
             model.optimize()
             print(model.solver.variables.difference.primal)
```

```
-5.234680806802543
-5.2346808068025386
-5.234680806802525
-1.8644444444444337
-1.8644444444444466
```

---

## Using the COBRA toolbox with cobrapy

---

This example demonstrates using COBRA toolbox commands in MATLAB from python through `pymatbridge`.

```
In [1]: %load_ext pymatbridge
```

```
Starting MATLAB on ZMQ socket ipc:///tmp/pymatbridge-57ff5429-02d9-4e1a-8ed0-44e391fb0df7
Send 'exit' command to kill the server
...MATLAB started and connected!
```

```
In [2]: import cobra.test
        m = cobra.test.create_test_model("textbook")
```

The `model_to_pymatbridge` function will send the model to the workspace with the given variable name.

```
In [3]: from cobra.io.mat import model_to_pymatbridge
        model_to_pymatbridge(m, variable_name="model")
```

Now in the MATLAB workspace, the variable name 'model' holds a COBRA toolbox struct encoding the model.

```
In [4]: %%matlab
        model
```

```
model =
```

```

        rev: [95x1 double]
    metNames: {72x1 cell}
         b: [72x1 double]
    metCharge: [72x1 double]
         c: [95x1 double]
     csense: [72x1 char]
      genes: {137x1 cell}
metFormulas: {72x1 cell}
       rxns: {95x1 cell}
     grRules: {95x1 cell}
    rxnNames: {95x1 cell}
description: [11x1 char]
         S: [72x95 double]
         ub: [95x1 double]
```

```
lb: [95x1 double]
mets: {72x1 cell}
subSystems: {95x1 cell}
```

First, we have to initialize the COBRA toolbox in MATLAB.

```
In [5]: %%matlab --silent
warning('off'); % this works around a pymatbridge bug
addpath(genpath('~\cobratoolbox/'));
initCobraToolbox();
```

Commands from the COBRA toolbox can now be run on the model

```
In [6]: %%matlab
optimizeCbModel(model)
```

ans =

```

x: [95x1 double]
f: 0.8739
y: [71x1 double]
w: [95x1 double]
stat: 1
origStat: 5
solver: 'glpk'
time: 3.2911
```

FBA in the COBRA toolbox should give the same result as cobrapy (but maybe just a little bit slower :))

```
In [7]: %time
m.optimize().f
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 5.48 µs
```

```
Out[7]: 0.8739215069684909
```

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

## 13.1 How do I install cobrapy?

Please see the [INSTALL.rst](#) file.

## 13.2 How do I cite cobrapy?

Please cite the 2013 publication: [10.1186/1752-0509-7-74](#)

## 13.3 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:

```
In [1]: 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))
```

The `Model.repair` function will rebuild the necessary indexes

```
In [2]: model.repair()
        model.metabolites.get_by_id(model.metabolites[0].id)
```

```
Out[2]: <Metabolite test_dcaACP_c at 0x110f09630>
```

## 13.4 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.manipulation.delete_model_genes` function. The effects of this function are reversed by `cobra.manipulation.undelete_model_genes`.

```
In [3]: 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.

## 13.5 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.

```
In [4]: model = cobra.test.create_test_model()
        model.reactions.get_by_id("PGI").reversibility
```

```
Out[4]: True
```

Trying to set it directly will result in an error or warning:

```
In [5]: try:
        model.reactions.get_by_id("PGI").reversibility = False
        except Exception as e:
            print(repr(e))
```

```
cobra/core/reaction.py:501 UserWarning: Setting reaction reversibility is ignored
```

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

```
In [6]: model.reactions.get_by_id("PGI").lower_bound = 10
        model.reactions.get_by_id("PGI").reversibility
```

```
Out[6]: False
```

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

### 13.6.1 For optlang based solvers

With optlang solvers, the LP formulation of a model is obtained by its string representation. All solvers behave the same way.



```
In [7]: with open('test.lp', 'w') as out:
        out.write(str(model.solver))
```

### 13.6.2 For cobrapy's internal solvers

With the internal solvers, we first create the problem and use functions bundled with the solver.

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

```
In [8]: model = cobra.test.create_test_model()
        # glpk through cglpk
        glpk = cobra.solvers.cglpk.create_problem(model)
        glpk.write("test.lp")
        glpk.write("test.mps") # will not rewrite objective
        # cplex
        cplex = cobra.solvers.cplex_solver.create_problem(model)
        cplex.write("test.lp")
        cplex.write("test.mps") # rewrites objective
```

### 13.6.3 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.



## 14.1 cobra package

### 14.1.1 Subpackages

cobra.core package

Submodules

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.

**add**(x)

Opposite of *remove*. Mirrors set.add

**append**(object)

append object to end

**extend**(iterable)

extend list by appending elements from the iterable

**get\_by\_any**(iterable)

Get a list of members using several different ways of indexing

**Parameters** *iterable* (list (if not, turned into single element list)) – list where each element is either int (referring to an index in in this DictList), string (a id of a member in this DictList) or member of this DictList for pass-through

**Returns** a list of members

**Return type** list

**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=None*)

Query the list

#### Parameters

- **search\_function** (*a string, regular expression or function*) – Used to find the matching elements in the list. - a regular expression (possibly compiled), in which case the given attribute of the object should match the regular expression. - a function which takes one argument and returns True for desired values
- **attribute** (*string or None*) – the name attribute of the object to passed as argument to the *search\_function*. If this is None, the object itself is used.

**Returns** a new list of objects which match the query

**Return type** *DictList*

## Examples

```
>>> import cobra.test
>>> model = cobra.test.create_test_model('textbook')
>>> model.reactions.query(lambda x: x.boundary)
>>> import re
>>> regex = re.compile('^g', flags=re.IGNORECASE)
>>> model.metabolites.query(regex, attribute='name')
```

**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

**Parameters** **formula** (*string*) – A legal formula string contains only letters and numbers.

**parse\_composition** ()

Breaks the chemical formula down by element.

**weight**

Calculate the mol mass of the compound

**Returns** the mol mass

**Return type** *float*

## cobra.core.gene module

**class** cobra.core.gene.**GPRCleaner**

Bases: *ast.NodeTransformer*

Parses compiled ast of a gene\_reaction\_rule and identifies genes

Parts of the tree are rewritten to allow periods in gene ID's and bitwise boolean operations

**visit\_BinOp** (*node*)

**visit\_Name** (*node*)

**class** cobra.core.gene.**Gene** (*id=None, name="", functional=True*)

Bases: *cobra.core.species.Species*

A Gene in a cobra model

**Parameters**

- **id** (*string*) – The identifier to associate the gene with
- **name** (*string*) – A longer human readable name for the gene
- **functional** (*bool*) – Indicates whether the gene is functional. If it is not functional then it cannot be used in an enzyme complex nor can its products be used.

**functional**

A flag indicating if the gene is functional.

Changing the flag is reverted upon exit if executed within the model as context.

**knock\_out** ()

Knockout gene by marking it as non-functional and setting all associated reactions bounds to zero.

The change is reverted upon exit if executed within the model as context.

**remove\_from\_model** (*model=None, make\_dependent\_reactions\_nonfunctional=True*)

Removes the association

**Parameters**

- **model** (*cobra model*) – The model to remove the gene from
- **make\_dependent\_reactions\_nonfunctional** (*bool*) – If True then replace the gene with ‘False’ in the gene association, else replace the gene with ‘True’

Deprecated since version 0.4: Use `cobra.manipulation.delete_model_genes` to simulate knockouts and `cobra.manipulation.remove_genes` to remove genes from the model.

`cobra.core.gene.ast2str` (*expr, level=0, names=None*)  
convert compiled ast to gene\_reaction\_rule str

#### Parameters

- **expr** (*str*) – string for a gene reaction rule, e.g “a and b”
- **level** (*int*) – internal use only
- **names** (*dict*) – Dict where each element id a gene identifier and the value is the gene name. Use this to get a rule str which uses names instead. This should be done for display purposes only. All gene\_reaction\_rule strings which are computed with should use the id.

**Returns** The gene reaction rule

**Return type** `string`

`cobra.core.gene.eval_gpr` (*expr, knockouts*)  
evaluate compiled ast of gene\_reaction\_rule with knockouts

#### Parameters

- **expr** (*Expression*) – The ast of the gene reaction rule
- **knockouts** (*DictList, set*) – Set of genes that are knocked out

**Returns** True if the gene reaction rule is true with the given knockouts otherwise false

**Return type** `bool`

`cobra.core.gene.parse_gpr` (*str\_expr*)  
parse gpr into AST

**Parameters** **str\_expr** (*string*) – string with the gene reaction rule to parse

**Returns** elements ast\_tree and gene\_ids as a set

**Return type** `tuple`

## cobra.core.metabolite module

**class** `cobra.core.metabolite.Metabolite` (*id=None, formula=None, name="", charge=None, compartment=None*)

Bases: `cobra.core.species.Species`

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

#### Parameters

- **id** (*str*) – the identifier to associate with the metabolite
- **formula** (*str*) – Chemical formula (e.g. H<sub>2</sub>O)
- **name** (*str*) – A human readable name.
- **charge** (*float*) – The charge number of the metabolite
- **compartment** (*str or None*) – Compartment of the metabolite.

**constraint**

Get the constraints associated with this metabolite from the solve

**Returns** the optlang constraint for this metabolite

**Return type** optlang.<interface>.Constraint

**elements**

Dictionary of elements as keys and their count in the metabolite as integer. When set, the *formula* property is update accordingly

**formula\_weight**

Calculate the formula weight

**remove\_from\_model** (*destructive=False*)

Removes the association from self.model

The change is reverted upon exit when using the model as a context.

**Parameters** **destructive** (*bool*) – If False then the metabolite is removed from all associated reactions. If True then all associated reactions are removed from the Model.

**shadow\_price**

The shadow price in the most recent solution.

Shadow price is the dual value of the corresponding constraint in the model.

**Warning:**

- Accessing shadow prices through a *Solution* object is the safer, preferred, and only guaranteed to be correct way. You can see how to do so easily in the examples.
- Shadow price is retrieved from the currently defined *self.\_model.solver*. The solver status is checked but there are no guarantees that the current solver state is the one you are looking for.
- If you modify the underlying model after an optimization, you will retrieve the old optimization values.

**Raises**

- `RuntimeError` – If the underlying model was never optimized beforehand or the metabolite is not part of a model.
- `OptimizationError` – If the solver status is anything other than ‘optimal’.

**Examples**

```
>>> import cobra
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.metabolites.glc__D_e.shadow_price
-0.09166474637510488
>>> solution.shadow_prices.glc__D_e
-0.091664746375104883
```

**summary** (*solution=None, threshold=0.01, fva=False, floatfmt='.3g'*)

Print a summary of the reactions which produce and consume this metabolite.

This method requires the model for which this metabolite is a part to be solved.

### Parameters

- **solution** (*cobra.core.Solution*) – A previously solved model solution to use for generating the summary. If none provided (default), the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method.
- **threshold** (*float*) – a value below which to ignore reaction fluxes
- **fva** (*float (0->1), or None*) – Whether or not to include flux variability analysis in the output. If given, fva should be a float between 0 and 1, representing the fraction of the optimum objective to be searched.
- **floatfmt** (*string*) – format method for floats, passed to tabulate. Default is `‘.3g’`.

### 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` (*id\_or\_model=None, name=None*)

Bases: `cobra.core.object.Object`

Class representation for a cobra model

### Parameters

- **id\_or\_model** (*Model, string*) – Either an existing Model object in which case a new model object is instantiated with the same properties as the original model, or a the identifier to associate with the model as a string.
- **name** (*string*) – Human readable name for the model

### reactions

*DictList* – A DictList where the key is the reaction identifier and the value a Reaction

### metabolites

*DictList* – A DictList where the key is the metabolite identifier and the value a Metabolite

### genes

*DictList* – A DictList where the key is the gene identifier and the value a Gene

### solution

*Solution* – The last obtained solution from optimizing the model.

**add\_boundary** (*metabolite, type='exchange', reaction\_id=None, lb=None, ub=1000.0*)

Add a boundary reaction for a given metabolite.

There are three different types of pre-defined boundary reactions: exchange, demand, and sink reactions. An exchange reaction is a reversible, imbalanced reaction that adds to or removes an extracellular metabolite from the extracellular compartment. A demand reaction is an irreversible reaction that consumes an intracellular metabolite. A sink is similar to an exchange but specifically for intracellular metabolites.

If you set the reaction *type* to something else, you must specify the desired identifier of the created reaction along with its upper and lower bound. The name will be given by the metabolite name and the given *type*.

### Parameters



- **metabolite** (*cobra.Metabolite*) – Any given metabolite. The compartment is not checked but you are encouraged to stick to the definition of exchanges and sinks.
- **type** (*str*, {"exchange", "demand", "sink"}) – Using one of the pre-defined reaction types is easiest. If you want to create your own kind of boundary reaction choose any other string, e.g., 'my-boundary'.
- **reaction\_id** (*str*, *optional*) – The ID of the resulting reaction. Only used for custom reactions.
- **lb** (*float*, *optional*) – The lower bound of the resulting reaction. Only used for custom reactions.
- **ub** (*float*, *optional*) – The upper bound of the resulting reaction. For the pre-defined reactions this default value determines all bounds.

**Returns** The created boundary reaction.

**Return type** cobra.Reaction

## Examples

```
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> demand = model.add_boundary(model.metabolites.atp_c, type="demand")
>>> demand.id
'DM_atp_c'
>>> demand.name
'ATP demand'
>>> demand.bounds
(0, 1000.0)
>>> demand.build_reaction_string()
'atp_c --> '
```

**add\_cons\_vars** (*what*, *\*\*kwargs*)

Add constraints and variables to the model's mathematical problem.

Useful for variables and constraints that can not be expressed with reactions and simple lower and upper bounds.

Additions are reversed upon exit if the model itself is used as context.

### Parameters

- **what** (*list or tuple of optlang variables or constraints.*) – The variables or constraints to add to the model. Must be of class *optlang.interface.Variable* or *optlang.interface.Constraint*.
- **\*\*kwargs** (*keyword arguments*) – Passed to *solver.add()*

**add\_metabolites** (*metabolite\_list*)

Will add a list of metabolites to the model object and add new constraints accordingly.

The change is reverted upon exit when using the model as a context.

**Parameters** **metabolite\_list** (A list of *cobra.core.Metabolite* objects) –

**add\_reaction** (*reaction*)

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

### Parameters

- **reaction** (*cobra.Reaction*) – The reaction to add
- (0.6) Use `~cobra.Model.add_reactions instead` (*Deprecated*) –

**add\_reactions** (*reaction\_list*)

Add reactions to the model.

Reactions with identifiers identical to a reaction already in the model are ignored.

The change is reverted upon exit when using the model as a context.

**Parameters** **reaction\_list** (*list*) – A list of *cobra.Reaction* objects

**compartments**

**constraints**

The constraints in the cobra model.

In a cobra model, most constraints are metabolites and their stoichiometries. However, for specific use cases, it may also be useful to have other types of constraints. This property defines all constraints currently associated with the model's problem.

**Returns** A container with all associated constraints.

**Return type** `optlang.container.Container`

**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

**description**

**exchanges**

Exchange reactions in model.

Reactions that either don't have products or substrates.

**get\_metabolite\_compartments** ()

Return all metabolites' compartments.

**medium**

**merge** (*right*, *prefix\_existing=None*, *inplace=True*, *objective='left'*)

Merge two models to create a model with the reactions from both models.

Custom constraints and variables from right models are also copied to left model, however note that, constraints and variables are assumed to be the same if they have the same name.

**right** [*cobra.Model*] The model to add reactions from

**prefix\_existing** [*string*] Prefix the reaction identifier in the right that already exist in the left model with this string.

**inplace** [*bool*] Add reactions from right directly to left model object. Otherwise, create a new model leaving the left model untouched. When done within the model as context, changes to the models are reverted upon exit.

**objective** [*string*] One of 'left', 'right' or 'sum' for setting the objective of the resulting model to that of the corresponding model or the sum of both.

**objective**

Get or set the solver objective

Before introduction of the optlang based problems, this function returned the objective reactions as a list. With optlang, the objective is not limited a simple linear summation of individual reaction fluxes,

making that return value ambiguous. Henceforth, use `cobra.util.solver.linear_reaction_coefficients` to get a dictionary of reactions with their linear coefficients (empty if there are none)

The set value can be dictionary (reactions as keys, linear coefficients as values), string (reaction identifier), int (reaction index), Reaction or problem.Objective or sympy expression directly interpreted as objectives.

When using a *HistoryManager* context, this attribute can be set temporarily, reversed when the exiting the context.

### **objective\_direction**

Get or set the objective direction.

When using a *HistoryManager* context, this attribute can be set temporarily, reversed when exiting the context.

**optimize** (*objective\_sense=None, raise\_error=False*)

Optimize the model using flux balance analysis.

#### **Parameters**

- **objective\_sense** (*{None, 'maximize' 'minimize'}, optional*) – Whether fluxes should be maximized or minimized. In case of None, the previous direction is used.
- **raise\_error** (*bool*) –  
If true, raise an **OptimizationError** if solver status is not optimal.

### **Notes**

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

### **problem**

The interface to the model's underlying mathematical problem.

Solutions to cobra models are obtained by formulating a mathematical problem and solving it. Cobrapy uses the optlang package to accomplish that and with this property you can get access to the problem interface directly.

**Returns** The problem interface that defines methods for interacting with the problem and associated solver directly.

**Return type** optlang.interface

**remove\_cons\_vars** (*what*)

Remove variables and constraints from the model's mathematical problem.

Remove variables and constraints that were added directly to the model's underlying mathematical problem. Removals are reversed upon exit if the model itself is used as context.

**Parameters** **what** (*list or tuple of optlang variables or constraints.*) – The variables or constraints to add to the model. Must be of class *optlang.interface.Variable* or *optlang.interface.Constraint*.

**remove\_metabolites** (*metabolite\_list, destructive=False*)

Remove a list of metabolites from the the object.

The change is reverted upon exit when using the model as a context.

#### **Parameters**

- **metabolite\_list** (*list*) – A list with *cobra.Metabolite* objects as elements.

- **destructive** (*bool*) – If False then the metabolite is removed from all associated reactions. If True then all associated reactions are removed from the Model.

**remove\_reactions** (*reactions*, *remove\_orphans=False*)

Remove reactions from the model.

The change is reverted upon exit when using the model as a context.

#### Parameters

- **reactions** (*list*) – A list with reactions (*cobra.Reaction*), or their id's, to remove
- **remove\_orphans** (*bool*) – 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

#### Parameters

- **rebuild\_index** (*bool*) – rebuild the indices kept in reactions, metabolites and genes
- **rebuild\_relationships** (*bool*) – reset all associations between genes, metabolites, model and then re-add them.

**slim\_optimize** (*error\_value=nan*, *message=None*)

Optimize model without creating a solution object.

Creating a full solution object implies fetching shadow prices and flux values for all reactions and metabolites from the solver object. This necessarily takes some time and in cases where only one or two values are of interest, it is recommended to instead use this function which does not create a solution object returning only the value of the objective. Note however that the *optimize()* function uses efficient means to fetch values so if you need fluxes/shadow prices for more than say 4 reactions/metabolites, then the total speed increase of *slim\_optimize* versus *optimize* is expected to be small or even negative depending on how you fetch the values after optimization.

#### Parameters

- **error\_value** (*float*, *None*) – The value to return if optimization failed due to e.g. infeasibility. If None, raise *OptimizationError* if the optimization fails.
- **message** (*string*) – Error message to use if the model optimization did not succeed.

**Returns** The objective value.

**Return type** *float*

**solver**

Get or set the attached solver instance.

The associated the solver object, which manages the interaction with the associated solver, e.g. glpk.

This property is useful for accessing the optimization problem directly and to define additional non-metabolic constraints.

## Examples

```
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> new = model.problem.Constraint(model.objective.expression,
>>> lb=0.99)
>>> model.solver.add(new)
```

**summary** (*solution=None, threshold=1e-08, fva=None, floatfmt='.3g'*)

Print a summary of the input and output fluxes of the model. This method requires the model to have been previously solved.

#### Parameters

- **solution** (*cobra.core.Solution*) – A previously solved model solution to use for generating the summary. If none provided (default), the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method.
- **threshold** (*float*) – tolerance for determining if a flux is zero (not printed)
- **fva** (*int or None*) – Whether or not to calculate and report flux variability in the output summary
- **floatfmt** (*string*) – format method for floats, passed to tabulate. Default is `‘.3g’`.

#### variables

The mathematical variables in the cobra model.

In a cobra model, most variables are reactions. However, for specific use cases, it may also be useful to have other types of variables. This property defines all variables currently associated with the model’s problem.

**Returns** A container with all associated variables.

**Return type** `optlang.container.Container`

### cobra.core.object module

**class** `cobra.core.object.Object` (*id=None, name=*)

Bases: `object`

Defines common behavior of object in cobra.core

**id**

### cobra.core.reaction module

**class** `cobra.core.reaction.Reaction` (*id=None, name=*, *subsystem=*, *lower\_bound=0.0, upper\_bound=1000.0, objective\_coefficient=0.0*)

Bases: `cobra.core.object.Object`

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

#### Parameters

- **id** (*string*) – The identifier to associate with this reaction
- **name** (*string*) – A human readable name for the reaction
- **subsystem** (*string*) – Subsystem where the reaction is meant to occur
- **lower\_bound** (*float*) – The lower flux bound
- **upper\_bound** (*float*) – The upper flux bound

**add\_metabolites** (*metabolites\_to\_add*, *combine=True*, *reversibly=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.

The change is reverted upon exit when using the model as a context.

#### Parameters

- **metabolites\_to\_add** (*dict*) – Dictionary with metabolite objects or metabolite identifiers as keys and coefficients as values. If keys are strings (name of a metabolite) the reaction must already be part of a model and a metabolite with the given name must exist in the model.
- **combine** (*bool*) – Describes behavior a metabolite already exists in the reaction. True causes the coefficients to be added. False causes the coefficient to be replaced.
- **reversibly** (*bool*) – Whether to add the change to the context to make the change reversibly or not (primarily intended for internal use).

#### boundary

Whether or not this reaction is an exchange reaction.

Returns *True* if the reaction has either no products or reactants.

#### bounds

Get or set the bounds directly from a tuple

Convenience method for setting upper and lower bounds in one line using a tuple of lower and upper bound. Invalid bounds will raise an *AssertionError*.

When using a *HistoryManager* context, this attribute can be set temporarily, reversed when the exiting the context.

**build\_reaction\_from\_string** (*reaction\_str*, *verbose=True*, *fwd\_arrow=None*,  
*rev\_arrow=None*, *reversible\_arrow=None*, *term\_split='+'*)

Builds reaction from reaction equation *reaction\_str* using parser

Takes a string and using the specifications supplied in the optional arguments infers a set of metabolites, metabolite compartments and stoichiometries for the reaction. It also infers the reversibility of the reaction from the reaction arrow.

Changes to the associated model are reverted upon exit when using the model as a context.

#### Parameters

- **reaction\_str** (*string*) – a string containing a reaction formula (equation)
- **verbose** (*bool*) – setting verbosity of function
- **fwd\_arrow** (*re.compile*) – for forward irreversible reaction arrows
- **rev\_arrow** (*re.compile*) – for backward irreversible reaction arrows
- **reversible\_arrow** (*re.compile*) – for reversible reaction arrows
- **term\_split** (*string*) – dividing individual metabolite entries

**build\_reaction\_string** (*use\_metabolite\_names=False*)

Generate a human readable reaction string

**check\_mass\_balance** ()

Compute mass and charge balance for the reaction

returns a dict of {element: amount} for unbalanced elements. “charge” is treated as an element in this dict  
This should be empty for balanced reactions.

**compartments**

lists compartments the metabolites are in

**copy()**

Copy a reaction

The referenced metabolites and genes are also copied.

**delete(remove\_orphans=False)**

Removes the reaction from a model.

This removes all associations between a reaction the associated model, metabolites and genes.

The change is reverted upon exit when using the model as a context.

Deprecated, use *reaction.remove\_from\_model* instead.

**Parameters** **remove\_orphans** (*bool*) – Remove orphaned genes and metabolites from the model as well

**flux**

The flux value in the most recent solution.

Flux is the primal value of the corresponding variable in the model.

**Warning:**

- Accessing reaction fluxes through a *Solution* object is the safer, preferred, and only guaranteed to be correct way. You can see how to do so easily in the examples.
- Reaction flux is retrieved from the currently defined *self.\_model.solver*. The solver status is checked but there are no guarantees that the current solver state is the one you are looking for.
- If you modify the underlying model after an optimization, you will retrieve the old optimization values.

**Raises**

- *RuntimeError* – If the underlying model was never optimized beforehand or the reaction is not part of a model.
- *OptimizationError* – If the solver status is anything other than ‘optimal’.
- *AssertionError* – If the flux value is not within the bounds.

**Examples**

```
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.reactions.PFK.flux
7.477381962160283
>>> solution.fluxes.PFK
7.4773819621602833
```

**flux\_expression**

Forward flux expression

**Returns** The expression representing the the forward flux (if associated with model), otherwise None. Representing the net flux if model.reversible\_encoding == 'unsplit' or None if reaction is not associated with a model

**Return type** sympy expression

#### **forward\_variable**

An optlang variable representing the forward flux

**Returns** An optlang variable for the forward flux or None if reaction is not associated with a model.

**Return type** optlang.interface.Variable

#### **functional**

All required enzymes for reaction are functional.

**Returns** True if the gene-protein-reaction (GPR) rule is fulfilled for this reaction, or if reaction is not associated to a model, otherwise False.

**Return type** bool

#### **gene\_name\_reaction\_rule**

Display gene\_reaction\_rule with names instead.

Do NOT use this string for computation. It is intended to give a representation of the rule using more familiar gene names instead of the often cryptic ids.

#### **gene\_reaction\_rule**

#### **genes**

#### **get\_coefficient** (*metabolite\_id*)

Return the stoichiometric coefficient of a metabolite.

**Parameters** *metabolite\_id* (*str* or *cobra.Metabolite*) –

#### **get\_coefficients** (*metabolite\_ids*)

Return the stoichiometric coefficients for a list of metabolites.

**Parameters** *metabolite\_ids* (*iterable*) – Containing *str* or “cobra.Metabolite”s.

#### **get\_compartments** ()

lists compartments the metabolites are in

#### **knock\_out** ()

Knockout reaction by setting its bounds to zero.

#### **lower\_bound**

Get or set the lower bound

Setting the lower bound (float) will also adjust the associated optlang variables associated with the reaction. Infeasible combinations, such as a lower bound higher than the current upper bound will update the other bound.

When using a *HistoryManager* context, this attribute can be set temporarily, reversed when the exiting the context.

#### **metabolites**

#### **model**

returns the model the reaction is a part of

#### **objective\_coefficient**

Get the coefficient for this reaction in a linear objective (float)



Assuming that the objective of the associated model is summation of fluxes from a set of reactions, the coefficient for each reaction can be obtained individually using this property. A more general way is to use the *model.objective* property directly.

#### **products**

Return a list of products for the reaction

#### **reactants**

Return a list of reactants for the reaction.

#### **reaction**

Human readable reaction string

#### **reduced\_cost**

The reduced cost in the most recent solution.

Reduced cost is the dual value of the corresponding variable in the model.

#### **Warning:**

- Accessing reduced costs through a *Solution* object is the safer, preferred, and only guaranteed to be correct way. You can see how to do so easily in the examples.
- Reduced cost is retrieved from the currently defined *self.\_model.solver*. The solver status is checked but there are no guarantees that the current solver state is the one you are looking for.
- If you modify the underlying model after an optimization, you will retrieve the old optimization values.

#### **Raises**

- `RuntimeError` – If the underlying model was never optimized beforehand or the reaction is not part of a model.
- `OptimizationError` – If the solver status is anything other than ‘optimal’.

### **Examples**

```
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.reactions.PFK.reduced_cost
-8.673617379884035e-18
>>> solution.reduced_costs.PFK
-8.673617379884035e-18
```

#### **remove\_from\_model** (*remove\_orphans=False*)

Removes the reaction from a model.

This removes all associations between a reaction the associated model, metabolites and genes.

The change is reverted upon exit when using the model as a context.

**Parameters** `remove_orphans` (*bool*) – Remove orphaned genes and metabolites from the model as well

#### **reverse\_id**

Generate the id of reverse\_variable from the reaction's id.

**reverse\_variable**

An optlang variable representing the reverse flux

**Returns** An optlang variable for the reverse flux or None if reaction is not associated with a model.

**Return type** optlang.interface.Variable

**reversibility**

Whether the reaction can proceed in both directions (reversible)

This is computed from the current upper and lower bounds.

**subtract\_metabolites** (*metabolites*, *combine=True*, *reversibly=True*)

Subtract metabolites from a reaction.

That means add the metabolites with  $-1 \times \text{coefficient}$ . If the final coefficient for a metabolite is 0 then the metabolite is removed from the reaction.

**Notes**

- A final coefficient  $< 0$  implies a reactant.
- The change is reverted upon exit when using the model as a context.

**Parameters**

- **metabolites** (*dict*) – Dictionary where the keys are of class Metabolite and the values are the coefficients. These metabolites will be added to the reaction.
- **combine** (*bool*) – Describes behavior a metabolite already exists in the reaction. True causes the coefficients to be added. False causes the coefficient to be replaced.
- **reversibly** (*bool*) – Whether to add the change to the context to make the change reversibly or not (primarily intended for internal use).

**upper\_bound**

Get or set the upper bound

Setting the upper bound (float) will also adjust the associated optlang variables associated with the reaction. Infeasible combinations, such as a upper bound lower than the current lower bound will update the other bound.

When using a *HistoryManager* context, this attribute can be set temporarily, reversed when the exiting the context.

**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.

**y**

The reduced cost of the reaction in the most recent solution.

Reduced costs are computed from the dual values of the variables in the solution.

`cobra.core.reaction.separate_forward_and_reverse_bounds` (*lower\_bound*, *upper\_bound*)

Split a given (*lower\_bound*, *upper\_bound*) interval into a negative component and a positive component. Negative components are negated (returns positive ranges) and flipped for usage with forward and reverse reactions bounds

**Parameters**

- **lower\_bound** (*float*) – The lower flux bound
- **upper\_bound** (*float*) – The upper flux bound

`cobra.core.reaction.update_forward_and_reverse_bounds` (*reaction*, *direction*='both')

For the given reaction, update the bounds in the forward and reverse variable bounds.

**Parameters**

- **reaction** (*cobra.Reaction*) – The reaction to operate on
- **direction** (*string*) – Either 'both', 'upper' or 'lower' for updating the corresponding flux bounds.

**cobra.core.solution module**

Provide unified interfaces to optimization solutions.

**class** `cobra.core.solution.Solution` (*objective\_value*, *status*, *fluxes*, *reduced\_costs*=None, *shadow\_prices*=None, *\*\*kwargs*)

Bases: `object`

A unified interface to a *cobra.Model* optimization solution.

**Notes**

Solution is meant to be constructed by *get\_solution* please look at that function to fully understand the *Solution* class.

**objective\_value**

*float* – The (optimal) value for the objective function.

**status**

*str* – The solver status related to the solution.

**fluxes**

*pandas.Series* – Contains the reaction fluxes (primal values of variables).

**reduced\_costs**

*pandas.Series* – Contains reaction reduced costs (dual values of variables).

**shadow\_prices**

*pandas.Series* – Contains metabolite shadow prices (dual values of constraints).

**Deprecated Attributes**

-----

**f**

*float* – Use *objective\_value* instead.

**x**

*list* – Use *fluxes.values* instead.

**x\_dict**

*pandas.Series* – Use *fluxes* instead.

**y**

*list* – Use *reduced\_costs.values* instead.

**y\_dict**

*pandas.Series* – Use *reduced\_costs* instead.

**f**

Deprecated property for getting the objective value.

**get\_primal\_by\_id** (*reaction\_id*)

Return the flux of a reaction.

**Parameters** **reaction** (*str*) – A model reaction ID.

**to\_frame** ()

Return the fluxes and reduced costs as a data frame

**x**

Deprecated property for getting flux values.

**x\_dict**

Deprecated property for getting fluxes.

**y**

Deprecated property for getting reduced cost values.

**y\_dict**

Deprecated property for getting reduced costs.

**class** cobra.core.solution.**LegacySolution** (*f, x=None, x\_dict=None, y=None, y\_dict=None, solver=None, the\_time=0, status='NA', \*\*kwargs*)

Bases: *object*

Legacy support for an interface to a *cobra.Model* optimization solution.

**f**

*float* – The objective value

**solver**

*str* – A string indicating which solver package was used.

**x**

*iterable* – List or Array of the fluxes (primal values).

**x\_dict**

*dict* – A dictionary of reaction IDs that maps to the respective primal values.

**y**

*iterable* – List or Array of the dual values.

**y\_dict**

*dict* – A dictionary of reaction IDs that maps to the respective dual values.

<b>Warning:</b> The LegacySolution class and its interface is deprecated.
---

**dress\_results** (*model*)

Method could be intended as a decorator.

<b>Warning:</b> deprecated
----------------------------

```
cobra.core.solution.get_solution(model, reactions=None, metabolites=None,
                                raise_error=False)
```

Generate a solution representation of the current solver state.

#### Parameters

- **model** (*cobra.Model*) – The model whose reactions to retrieve values for.
- **reactions** (*list, optional*) – An iterable of *cobra.Reaction* objects. Uses *model.reactions* by default.
- **metabolites** (*list, optional*) – An iterable of *cobra.Metabolite* objects. Uses *model.metabolites* by default.
- **raise\_error** (*bool*) – If true, raise an *OptimizationError* if solver status is not optimal.

#### Returns

**Return type** *cobra.Solution*

---

**Note:** This is only intended for the *optlang* solver interfaces and not the legacy solvers.

---

## cobra.core.species module

```
class cobra.core.species.Species(id=None, name=None)
```

Bases: *cobra.core.object.Object*

Species is a class for holding information regarding a chemical Species

#### Parameters

- **id** (*string*) – An identifier for the chemical species
- **name** (*string*) – A human readable name.

#### 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.\_\_deepcopy\_\_* if possible

**model**

**reactions**

## Module contents

### cobra.flux\_analysis package

#### Submodules

#### cobra.flux\_analysis.deletion module

```
cobra.flux_analysis.deletion.double_gene_deletion(model, gene_list1=None,
                                                    gene_list2=None, method='fba',
                                                    processes=None)
```

Knock out each gene pair from the combination of two given lists.

We say ‘pair’ here but the order order does not matter.

##### Parameters

- **model** (*cobra.Model*) – The metabolic model to perform deletions in.
- **gene\_list1** (*iterable, optional*) – First iterable of “cobra.Gene”s to be deleted. If not passed, all the genes from the model are used.
- **gene\_list2** (*iterable, optional*) – Second iterable of “cobra.Gene”s to be deleted. If not passed, all the genes from the model are used.
- **method** ({*"fba", "moma", "linear moma"*}, *optional*) – Method used to predict the growth rate.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.

##### Returns

A representation of all combinations of gene deletions. The columns are ‘growth’ and ‘status’, where

**index** [frozenset([str])] The gene identifiers that were knocked out.

**growth** [float] The growth rate of the adjusted model.

**status** [str] The solution’s status.

**Return type** pandas.DataFrame

```
cobra.flux_analysis.deletion.double_reaction_deletion(model, reaction_list1=None,
                                                       reaction_list2=None,
                                                       method='fba',
                                                       processes=None)
```

Knock out each reaction pair from the combinations of two given lists.

We say ‘pair’ here but the order order does not matter.

##### Parameters

- **model** (*cobra.Model*) – The metabolic model to perform deletions in.
- **reaction\_list1** (*iterable, optional*) – First iterable of “cobra.Reaction”s to be deleted. If not passed, all the reactions from the model are used.
- **reaction\_list2** (*iterable, optional*) – Second iterable of “cobra.Reaction”s to be deleted. If not passed, all the reactions from the model are used.

- **method** (`{"fba", "moma", "linear moma"}`, *optional*) – Method used to predict the growth rate.
- **processes** (*int*, *optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.

### Returns

A representation of all combinations of reaction deletions. The columns are ‘growth’ and ‘status’, where

**index** [frozenset([str])] The reaction identifiers that were knocked out.

**growth** [float] The growth rate of the adjusted model.

**status** [str] The solution’s status.

**Return type** pandas.DataFrame

```
cobra.flux_analysis.deletion.single_gene_deletion(model, gene_list=None,
                                                  method='fba', processes=None)
```

Knock out each gene from a given list.

### Parameters

- **model** (*cobra.Model*) – The metabolic model to perform deletions in.
- **gene\_list** (*iterable*) – “cobra.Gene”s to be deleted. If not passed, all the genes from the model are used.
- **method** (`{"fba", "moma", "linear moma"}`, *optional*) – Method used to predict the growth rate.
- **processes** (*int*, *optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.

### Returns

A representation of all single gene deletions. The columns are ‘growth’ and ‘status’, where

**index** [frozenset([str])] The gene identifier that was knocked out.

**growth** [float] The growth rate of the adjusted model.

**status** [str] The solution’s status.

**Return type** pandas.DataFrame

```
cobra.flux_analysis.deletion.single_reaction_deletion(model, reaction_list=None,
                                                       method='fba', processes=None)
```

Knock out each reaction from a given list.

### Parameters

- **model** (*cobra.Model*) – The metabolic model to perform deletions in.
- **reaction\_list** (*iterable*) – “cobra.Reaction”s to be deleted. If not passed, all the reactions from the model are used.
- **method** (`{"fba", "moma", "linear moma"}`, *optional*) – Method used to predict the growth rate.

- **processes** (*int*, *optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.

### Returns

A representation of all single reaction deletions. The columns are ‘growth’ and ‘status’, where

**index** [frozenset([str])] The reaction identifier that was knocked out.

**growth** [float] The growth rate of the adjusted model.

**status** [str] The solution’s status.

**Return type** pandas.DataFrame

## cobra.flux\_analysis.gapfilling module

```
class cobra.flux_analysis.gapfilling.GapFiller(model, universal=None,
                                             lower_bound=0.05, penal-
                                             ties=None, exchange_reactions=False,
                                             demand_reactions=True,
                                             integer_threshold=1e-06)
```

Bases: `object`

Class for performing gap filling.

This class implements gap filling based on a mixed-integer approach, very similar to that described in<sup>1</sup> and the ‘no-growth but growth’ part of [2]<sub>1</sub> but with minor adjustments. In short, we add indicator variables for using the reactions in the universal model,  $z_i$  and then solve problem

minimize  $\sum_i c_i * z_i$  s.t.  $Sv = 0$

$v_o \geq t$   $lb_i \leq v_i \leq ub_i$   $v_i = 0$  if  $z_i = 0$

where  $lb$ ,  $ub$  are the upper, lower flux bounds for reaction  $i$ ,  $c_i$  is a cost parameter and the objective  $v_o$  is greater than the lower bound  $t$ . The default costs are 1 for reactions from the universal model, 100 for exchange (uptake) reactions added and 1 for added demand reactions.

Note that this is a mixed-integer linear program and as such will be expensive to solve for large models. Consider using alternatives [3]<sub>1</sub> such as CORDA instead [4,5]<sub>1</sub>.

### Parameters

- **model** (*cobra.Model*) – The model to perform gap filling on.
- **universal** (*cobra.Model*) – A universal model with reactions that can be used to complete the model.
- **lower\_bound** (*float*) – The minimally accepted flux for the objective in the filled model.

<sup>1</sup> Reed, Jennifer L., Trina R. Patel, Keri H. Chen, Andrew R. Joyce, Margaret K. Applebee, Christopher D. Herring, Olivia T. Bui, Eric M. Knight, Stephen S. Fong, and Bernhard O. Palsson. “Systems Approach to Refining Genome Annotation.” *Proceedings of the National Academy of Sciences* 103, no. 46 (2006): 17480–17484.

[2] Kumar, Vinay Satish, and Costas D. Maranas. “GrowMatch: An Automated Method for Reconciling In Silico/In Vivo Growth Predictions.” Edited by Christos A. Ouzounis. *PLoS Computational Biology* 5, no. 3 (March 13, 2009): e1000308. doi:10.1371/journal.pcbi.1000308.

[3] <http://opencobra.github.io/cobrapy/tags/gapfilling/>

[4] Schultz, André, and Amina A. Qutub. “Reconstruction of Tissue-Specific Metabolic Networks Using CORDA.” Edited by Costas D. Maranas. *PLOS Computational Biology* 12, no. 3 (March 4, 2016): e1004808. doi:10.1371/journal.pcbi.1004808.

[5] Diener, Christian <https://github.com/cdiener/corda>



- **penalties** (*dict*, *None*) – A dictionary with keys being ‘universal’ (all reactions included in the universal model), ‘exchange’ and ‘demand’ (all additionally added exchange and demand reactions) for the three reaction types. Can also have reaction identifiers for reaction specific costs. Defaults are 1, 100 and 1 respectively.
- **integer\_threshold** (*float*) – The threshold at which a value is considered non-zero (aka integrality threshold). If gapfilled models fail to validate, you may want to lower this value.
- **exchange\_reactions** (*bool*) – Consider adding exchange (uptake) reactions for all metabolites in the model.
- **demand\_reactions** (*bool*) – Consider adding demand reactions for all metabolites.

## References

### **add\_switches\_and\_objective** ()

Update gapfilling model with switches and the indicator objective.

### **extend\_model** (*exchange\_reactions=False*, *demand\_reactions=True*)

Extend gapfilling model.

Add reactions from universal model and optionally exchange and demand reactions for all metabolites in the model to perform gapfilling on.

#### Parameters

- **exchange\_reactions** (*bool*) – Consider adding exchange (uptake) reactions for all metabolites in the model.
- **demand\_reactions** (*bool*) – Consider adding demand reactions for all metabolites.

### **fill** (*iterations=1*)

Perform the gapfilling by iteratively solving the model, updating the costs and recording the used reactions.

**Parameters** **iterations** (*int*) – The number of rounds of gapfilling to perform. For every iteration, the penalty for every used reaction increases linearly. This way, the algorithm is encouraged to search for alternative solutions which may include previously used reactions. I.e., with enough iterations pathways including 10 steps will eventually be reported even if the shortest pathway is a single reaction.

**Returns** A list of lists where each element is a list reactions that were used to gapfill the model.

**Return type** iterable

**Raises** `RuntimeError` – If the model fails to be validated (i.e. the original model with the proposed reactions added, still cannot get the required flux through the objective).

### **update\_costs** ()

Update the coefficients for the indicator variables in the objective.

Done incrementally so that second time the function is called, active indicators in the current solutions gets higher cost than the unused indicators.

### **validate** (*reactions*)

`cobra.flux_analysis.gapfilling.gapfill` (*model*, *universal=None*, *lower\_bound=0.05*, *penalties=None*, *demand\_reactions=True*, *exchange\_reactions=False*, *iterations=1*)

Perform gapfilling on a model.

See documentation for the class GapFiller.

### Parameters

- **model** (*cobra.Model*) – The model to perform gap filling on.
- **universal** (*cobra.Model*, *None*) – A universal model with reactions that can be used to complete the model. Only gapfill considering demand and exchange reactions if left missing.
- **lower\_bound** (*float*) – The minimally accepted flux for the objective in the filled model.
- **penalties** (*dict*, *None*) – A dictionary with keys being ‘universal’ (all reactions included in the universal model), ‘exchange’ and ‘demand’ (all additionally added exchange and demand reactions) for the three reaction types. Can also have reaction identifiers for reaction specific costs. Defaults are 1, 100 and 1 respectively.
- **iterations** (*int*) – The number of rounds of gapfilling to perform. For every iteration, the penalty for every used reaction increases linearly. This way, the algorithm is encouraged to search for alternative solutions which may include previously used reactions. I.e., with enough iterations pathways including 10 steps will eventually be reported even if the shortest pathway is a single reaction.
- **exchange\_reactions** (*bool*) – Consider adding exchange (uptake) reactions for all metabolites in the model.
- **demand\_reactions** (*bool*) – Consider adding demand reactions for all metabolites.

**Returns** list of lists with on set of reactions that completes the model per requested iteration.

**Return type** iterable

### Examples

```
>>> import cobra.test as ct
>>> from cobra import Model
>>> from cobra.flux_analysis import gapfill
>>> model = ct.create_test_model("salmonella")
>>> universal = Model('universal')
>>> universal.add_reactions(model.reactions.GF6PTA.copy())
>>> model.remove_reactions([model.reactions.GF6PTA])
>>> gapfill(model, universal)
```

### cobra.flux\_analysis.loopless module

Provides functions to remove thermodynamically infeasible loops.

`cobra.flux_analysis.loopless.add_loopless(model, zero_cutoff=1e-12)`

Modify a model so all feasible flux distributions are loopless.

In most cases you probably want to use the much faster *loopless\_solution*. May be used in cases where you want to add complex constraints and objectives (for instance quadratic objectives) to the model afterwards or use an approximation of Gibbs free energy directions in you model. Adds variables and constraints to a model which will disallow flux distributions with loops. The used formulation is described in [1]. This function *will* modify your model.

### Parameters

- **model** (*cobra.Model*) – The model to which to add the constraints.

- **zero\_cutoff** (*positive float, optional*) – Cutoff used for null space. Coefficients with an absolute value smaller than *zero\_cutoff* are considered to be zero.

**Returns****Return type** Nothing**References**

`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.loopless.loopless_fva_iter` (*model*, *reaction*, *solution=False*, *zero\_cutoff=1e-06*)

Plugin to get a loopless FVA solution from single FVA iteration.

Assumes the following about *model* and *reaction*: 1. the model objective is set to be *reaction* 2. the model has been optimized and contains the minimum/maximum flux for

*reaction*

3. the model contains an auxiliary variable called “fva\_old\_objective” denoting the previous objective

**Parameters**

- **model** (*cobra.Model*) – The model to be used.
- **reaction** (*cobra.Reaction*) – The reaction currently minimized/maximized.
- **solution** (*boolean, optional*) – Whether to return the entire solution or only the minimum/maximum for *reaction*.
- **zero\_cutoff** (*positive float, optional*) – Cutoff used for loop removal. Fluxes with an absolute value smaller than *zero\_cutoff* are considered to be zero.

**Returns** Returns the minimized/maximized flux through *reaction* if *all\_fluxes* == False (default). Otherwise returns a loopless flux solution containing the minimum/maximum flux for *reaction*.

**Return type** single float or *dict*

`cobra.flux_analysis.loopless.loopless_solution` (*model*, *fluxes=None*)

Convert an existing solution to a loopless one.

Removes as many loops as possible (see Notes). Uses the method from CycleFreeFlux [1] and is much faster than *add\_loopless* and should therefore be the preferred option to get loopless flux distributions.

**Parameters**

- **model** (*cobra.Model*) – The model to which to add the constraints.
- **fluxes** (*dict*) – A dictionary {rxn\_id: flux} that assigns a flux to each reaction. If not None will use the provided flux values to obtain a close loopless solution.

**Returns** A solution object containing the fluxes with the least amount of loops possible or None if the optimization failed (usually happening if the flux distribution in *fluxes* is infeasible).

**Return type** *cobra.Solution*

## Notes

The returned flux solution has the following properties:

- it contains the minimal number of loops possible and no loops at all if all flux bounds include zero
- it has an objective value close to the original one and the same objective value if the objective expression can not form a cycle (which is usually true since it consumes metabolites)
- it has the same exact exchange fluxes as the previous solution
- all fluxes have the same sign (flow in the same direction) as the previous solution

## References

### cobra.flux\_analysis.moma module

Contains functions to run minimization of metabolic adjustment (MOMA).

`cobra.flux_analysis.moma.add_moma(model, solution=None, linear=False)`

Add constraints and objective representing for MOMA.

This adds variables and constraints for the minimization of metabolic adjustment (MOMA) to the model.

#### Parameters

- **model** (*cobra.Model*) – The model to add MOMA constraints and objective to.
- **solution** (*cobra.Solution*) – A previous solution to use as a reference.
- **linear** (*bool*) – Whether to use the linear MOMA formulation or not.

#### Returns

**Return type** Nothing.

## Notes

In the original MOMA specification one looks for the flux distribution of the deletion ( $v^d$ ) closest to the fluxes without the deletion ( $v$ ). In math this means:

$$\text{minimize } \sum_i (v^d_i - v_i)^2 \text{ s.t. } Sv^d = 0$$

$$lb_i \leq v^d_i \leq ub_i$$

Here, we use a variable transformation  $v^t := v^d_i - v_i$ . Substituting and using the fact that  $Sv = 0$  gives:

$$\text{minimize } \sum_i (v^t_i)^2 \text{ s.t. } Sv^d = 0$$

$$v^t = v^d_i - v_i \quad lb_i \leq v^d_i \leq ub_i$$

So basically we just re-center the flux space at the old solution and then find the flux distribution closest to the new zero (center). This is the same strategy as used in cameo.

In the case of linear MOMA, we instead minimize  $\sum_i \text{abs}(v^t_i)$ . The linear MOMA is typically significantly faster. Also quadratic MOMA tends to give flux distributions in which all fluxes deviate from the reference fluxes a little bit whereas linear MOMA tends to give flux distributions where the majority of fluxes are the same reference which few fluxes deviating a lot (typical effect of L2 norm vs L1 norm).

The former objective function is saved in the optlang solver interface as “moma\_old\_objective” and this can be used to immediately extract the value of the former objective after MOMA optimization.

## cobra.flux\_analysis.parsimonious module

```
cobra.flux_analysis.parsimonious.add_pfba(model, objective=None, fraction_of_optimum=1.0)
```

Add pFBA objective

Add objective to minimize the summed flux of all reactions to the current objective.

See also:

`pfba()`

### Parameters

- **model** (*cobra.Model*) – The model to add the objective to
- **objective** – An objective to set in combination with the pFBA objective.
- **fraction\_of\_optimum** (*float*) – Fraction of optimum which must be maintained. The original objective reaction is constrained to be greater than `maximal_value * fraction_of_optimum`.

```
cobra.flux_analysis.parsimonious.optimize_minimal_flux(*args, **kwargs)
```

```
cobra.flux_analysis.parsimonious.pfba(model, fraction_of_optimum=1.0, objective=None, reactions=None)
```

Perform basic pFBA (parsimonious Enzyme Usage Flux Balance Analysis) to minimize total flux.

pFBA [1] adds the minimization of all fluxes the the objective of the model. This approach is motivated by the idea that high fluxes have a higher enzyme turn-over and that since producing enzymes is costly, the cell will try to minimize overall flux while still maximizing the original objective function, e.g. the growth rate.

### Parameters

- **model** (*cobra.Model*) – The model
- **fraction\_of\_optimum** (*float, optional*) – Fraction of optimum which must be maintained. The original objective reaction is constrained to be greater than `maximal_value * fraction_of_optimum`.
- **objective** (*dict or model.problem.Objective*) – A desired objective to use during optimization in addition to the pFBA objective. Dictionaries (reaction as key, coefficient as value) can be used for linear objectives.
- **reactions** (*iterable*) – List of reactions or reaction identifiers. Implies `return_frame` to be true. Only return fluxes for the given reactions. Faster than fetching all fluxes if only a few are needed.

**Returns** The solution object to the optimized model with pFBA constraints added.

**Return type** `cobra.Solution`

## References

## cobra.flux\_analysis.phenotype\_phase\_plane module

```
cobra.flux_analysis.phenotype_phase_plane.add_envelope(model, reactions, grid, c_input, c_output, threshold)
```

`cobra.flux_analysis.phenotype_phase_plane.find_carbon_sources(model)`

Find all active carbon source reactions.

**Parameters** `model` (`Model`) – A genome-scale metabolic model.

**Returns** The medium reactions with carbon input flux.

**Return type** `list`

`cobra.flux_analysis.phenotype_phase_plane.production_envelope(model, reactions, objective=None, carbon_sources=None, points=20, threshold=1e-07)`

Calculate the objective value conditioned on all combinations of fluxes for a set of chosen reactions

The production envelope can be used to analyze a model's ability to produce a given compound conditional on the fluxes for another set of reactions, such as the uptake rates. The model is alternately optimized with respect to minimizing and maximizing the objective and the obtained fluxes are recorded. Ranges to compute production is set to the effective bounds, i.e., the minimum / maximum fluxes that can be obtained given current reaction bounds.

#### Parameters

- **model** (`cobra.Model`) – The model to compute the production envelope for.
- **reactions** (`list` or `string`) – A list of reactions, reaction identifiers or a single reaction.
- **objective** (`string`, `dict`, `model.solver.interface.Objective`, `optional`) – The objective (reaction) to use for the production envelope. Use the model's current objective if left missing.
- **carbon\_sources** (`list` or `string`, `optional`) – One or more reactions or reaction identifiers that are the source of carbon for computing carbon (mol carbon in output over mol carbon in input) and mass yield (gram product over gram output). Only objectives with a carbon containing input and output metabolite is supported. Will identify active carbon sources in the medium if none are specified.
- **points** (`int`, `optional`) – The number of points to calculate production for.
- **threshold** (`float`, `optional`) – A cut-off under which flux values will be considered to be zero.

#### Returns

A data frame with one row per evaluated point and

- `reaction_id` : one column per input reaction indicating the flux at each given point,
- `carbon_source`: identifiers of carbon exchange reactions

A column for the maximum and minimum each for the following types:

- `flux`: the objective flux
- `carbon_yield`: if carbon source is defined and the product is a single metabolite (mol carbon product per mol carbon feeding source)
- `mass_yield`: if carbon source is defined and the product is a single metabolite (gram product per 1 g of feeding source)

**Return type** `pandas.DataFrame`

## Examples

```
>>> import cobra.test
>>> from cobra.flux_analysis import production_envelope
>>> model = cobra.test.create_test_model("textbook")
>>> production_envelope(model, ["EX_glc__D_e", "EX_o2_e"])
```

`cobra.flux_analysis.phenotype_phase_plane.reaction_elements` (*reaction*)

Split metabolites into the atoms times their stoichiometric coefficients.

**Parameters** *reaction* (*Reaction*) – The metabolic reaction whose components are desired.

**Returns** Each of the reaction's metabolites' desired carbon elements (if any) times that metabolite's stoichiometric coefficient.

**Return type** *list*

`cobra.flux_analysis.phenotype_phase_plane.reaction_weight` (*reaction*)

Return the metabolite weight times its stoichiometric coefficient.

`cobra.flux_analysis.phenotype_phase_plane.total_components_flux` (*flux*, *com-*  
*ponents*,  
*consump-*  
*tion=True*)

Compute the total components consumption or production flux.

**Parameters**

- **flux** (*float*) – The reaction flux for the components.
- **components** (*list*) – List of stoichiometrically weighted components.
- **consumption** (*bool*, *optional*) – Whether to sum up consumption or production fluxes.

`cobra.flux_analysis.phenotype_phase_plane.total_yield` (*input\_fluxes*, *input\_elements*,  
*output\_flux*, *out-*  
*put\_elements*)

Compute total output per input unit.

Units are typically mol carbon atoms or gram of source and product.

**Parameters**

- **input\_fluxes** (*list*) – A list of input reaction fluxes in the same order as the *input\_components*.
- **input\_elements** (*list*) – A list of reaction components which are in turn list of numbers.
- **output\_flux** (*float*) – The output flux value.
- **output\_elements** (*list*) – A list of stoichiometrically weighted output reaction components.

**Returns** The ratio between output (mol carbon atoms or grams of product) and input (mol carbon atoms or grams of source compounds).

**Return type** *float*

## cobra.flux\_analysis.reaction module

functions for analyzing / creating objective functions

```
cobra.flux_analysis.reaction.assess(model, reaction, flux_coefficient_cutoff=0.001,  
                                   solver=None)
```

Assesses production capacity.

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.

### Parameters

- **model** (*cobra.Model*) – The cobra model to assess production capacity for
- **reaction** (*reaction identifier or cobra.Reaction*) – The reaction to assess
- **flux\_coefficient\_cutoff** (*float*) – The minimum flux that reaction must carry to be considered active.
- **solver** (*basestring*) – Solver name. If None, the default solver will be used.

**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.

**Return type** *bool* or *dict*

```
cobra.flux_analysis.reaction.assess_component(model, reaction, side,  
                                              flux_coefficient_cutoff=0.001,  
                                              solver=None)
```

Assesses the ability of the model to provide sufficient precursors, or absorb products, for a reaction operating at, or beyond, the specified cutoff.

### Parameters

- **model** (*cobra.Model*) – The cobra model to assess production capacity for
- **reaction** (*reaction identifier or cobra.Reaction*) – The reaction to assess
- **side** (*basestring*) – Side of the reaction, 'products' or 'reactants'
- **flux\_coefficient\_cutoff** (*float*) – The minimum flux that reaction must carry to be considered active.
- **solver** (*basestring*) – Solver name. If None, the default solver will be used.

**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.

**Return type** *bool* or *dict*

```
cobra.flux_analysis.reaction.assess_precursors(model, reaction,  
                                              flux_coefficient_cutoff=0.001,  
                                              solver=None)
```

Assesses the ability of the model to provide sufficient precursors for a reaction operating at, or beyond, the specified cutoff.

Deprecated: use assess\_component instead



**Parameters**

- **model** (*cobra.Model*) – The cobra model to assess production capacity for
- **reaction** (*reaction identifier or cobra.Reaction*) – The reaction to assess
- **flux\_coefficient\_cutoff** (*float*) – The minimum flux that reaction must carry to be considered active.
- **solver** (*basestring*) – Solver name. If None, the default solver will be used.

**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.

**Return type** *bool* or *dict*

```
cobra.flux_analysis.reaction.assess_products(model, reaction,
                                             flux_coefficient_cutoff=0.001,
                                             solver=None)
```

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.

Deprecated: use `assess_component` instead

**Parameters**

- **model** (*cobra.Model*) – The cobra model to assess production capacity for
- **reaction** (*reaction identifier or cobra.Reaction*) – The reaction to assess
- **flux\_coefficient\_cutoff** (*float*) – The minimum flux that reaction must carry to be considered active.
- **solver** (*basestring*) – Solver name. If None, the default solver will be used.

**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.

**Return type** *bool* or *dict*

**cobra.flux\_analysis.sampling module**

Module implementing flux sampling for cobra models.

New samplers should derive from the abstract *HRSampler* class where possible to provide a uniform interface.

```
class cobra.flux_analysis.sampling.ACHRSampler(model, thinning=100, seed=None)
    Bases: cobra.flux_analysis.sampling.HRSampler
```

Artificial Centering Hit-and-Run sampler.

A sampler with low memory footprint and good convergence.

**Parameters**

- **model** (*a cobra model*) – The cobra model from which to generate samples.

- **thinning** (*int*, *optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.
- **seed** (*positive integer*, *optional*) – Sets the random number seed. Initialized to the current time stamp if None.

**model**

*cobra.Model* – The cobra model from which the samples get generated.

**thinning**

*int* – The currently used thinning factor.

**n\_samples**

*int* – The total number of samples that have been generated by this sampler instance.

**problem**

*collections.namedtuple* – A python object whose attributes define the entire sampling problem in matrix form. See docstring of *Problem*.

**warmup**

*a numpy matrix* – A matrix of with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

**seed**

*positive integer, optional* – Sets the random number seed. Initialized to the current time stamp if None.

**fwd\_idx**

*np.array* – Has one entry for each reaction in the model containing the index of the respective forward variable.

**rev\_idx**

*np.array* – Has one entry for each reaction in the model containing the index of the respective reverse variable.

**prev**

*numpy array* – The current/last flux sample generated.

**center**

*numpy array* – The center of the sampling space as estimated by the mean of all previously generated samples.

## Notes

ACHR generates samples by choosing new directions from the sampling space's center and the warmup points. The implementation used here is the same as in the Matlab Cobra Toolbox [2] and uses only the initial warmup points to generate new directions and not any other previous iterates. This usually gives better mixing since the startup points are chosen to span the space in a wide manner. This also makes the generated sampling chain quasi-markovian since the center converges rapidly.

Memory usage is roughly in the order of  $(2 * \text{number reactions})^2$  due to the required nullspace matrices and warmup points. So large models easily take up a few GB of RAM.

## References

**sample** (*n*, *fluxes=True*)

Generate a set of samples.

This is the basic sampling function for all hit-and-run samplers.

**Parameters**

- **n** (*int*) – The number of samples that are generated at once.
- **fluxes** (*boolean*) – Whether to return fluxes or the internal solver variables. If set to False will return a variable for each forward and backward flux as well as all additional variables you might have defined in the model.

**Returns** Returns a matrix with *n* rows, each containing a flux sample.

**Return type** `numpy.matrix`

**Notes**

Performance of this function linearly depends on the number of reactions in your model and the thinning factor.

**class** `cobra.flux_analysis.sampling.HRSampler` (*model, thinning, seed=None*)

Bases: `object`

The abstract base class for hit-and-run samplers.

**Parameters**

- **model** (*cobra.Model*) – The cobra model from which to generate samples.
- **thinning** (*int*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.

**model**

*cobra.Model* – The cobra model from which the samples get generated.

**thinning**

*int* – The currently used thinning factor.

**n\_samples**

*int* – The total number of samples that have been generated by this sampler instance.

**problem**

*collections.namedtuple* – A python object whose attributes define the entire sampling problem in matrix form. See docstring of *Problem*.

**warmup**

*a numpy matrix* – A matrix of with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

**seed**

*positive integer, optional* – Sets the random number seed. Initialized to the current time stamp if None.

**fwd\_idx**

*np.array* – Has one entry for each reaction in the model containing the index of the respective forward variable.

**rev\_idx**

*np.array* – Has one entry for each reaction in the model containing the index of the respective reverse variable.

**batch** (*batch\_size, batch\_num, fluxes=True*)

Create a batch generator.

This is useful to generate n batches of m samples each.

**Parameters**

- **batch\_size** (*int*) – The number of samples contained in each batch (m).
- **batch\_num** (*int*) – The number of batches in the generator (n).
- **fluxes** (*boolean*) – Whether to return fluxes or the internal solver variables. If set to False will return a variable for each forward and backward flux as well as all additional variables you might have defined in the model.

**Yields** *pandas.DataFrame* – A DataFrame with dimensions (batch\_size x n\_r) containing a valid flux sample for a total of n\_r reactions (or variables if fluxes=False) in each row.

**generate\_fva\_warmup** ()

Generate the warmup points for the sampler.

Generates warmup points by setting each flux as the sole objective and minimizing/maximizing it. Also caches the projection of the warmup points into the nullspace for non-homogeneous problems (only if necessary).

**sample** (*n*, *fluxes=True*)

Abstract sampling function.

Should be overwritten by child classes.

**validate** (*samples*)

Validate a set of samples for equality and inequality feasibility.

Can be used to check whether the generated samples and warmup points are feasible.

**Parameters** **samples** (*numpy.matrix*) – Must be of dimension (n\_samples x n\_reactions).  
Contains the samples to be validated. Samples must be from fluxes.

#### Returns

A one-dimensional numpy array of length containing a code of 1 to 3 letters denoting the validation result:

- 'v' means feasible in bounds and equality constraints
- 'l' means a lower bound violation
- 'u' means a lower bound validation
- 'e' means and equality constraint violation

**Return type** *numpy.array*

`cobra.flux_analysis.sampling.LOGGER = <logging.Logger object>`

The logger for the package.

**class** `cobra.flux_analysis.sampling.OptGPSampler` (*model*, *processes*, *thinning=100*,  
*seed=None*)

Bases: `cobra.flux_analysis.sampling.HRSampler`

A parallel optimized sampler.

A parallel sampler with fast convergence and parallel execution. See [1] for details.

#### Parameters

- **model** (*cobra.Model*) – The cobra model from which to generate samples.
- **processes** (*int*) – The number of processes used during sampling.
- **thinning** (*int*, *optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.

- **seed**(*positive integer, optional*) – Sets the random number seed. Initialized to the current time stamp if None.

**model**

*cobra.Model* – The cobra model from which the samples get generated.

**thinning**

*int* – The currently used thinning factor.

**n\_samples**

*int* – The total number of samples that have been generated by this sampler instance.

**problem**

*collections.namedtuple* – A python object whose attributes define the entire sampling problem in matrix form. See docstring of *Problem*.

**warmup**

*a numpy matrix* – A matrix of with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

**seed**

*positive integer, optional* – Sets the random number seed. Initialized to the current time stamp if None.

**fwd\_idx**

*np.array* – Has one entry for each reaction in the model containing the index of the respective forward variable.

**rev\_idx**

*np.array* – Has one entry for each reaction in the model containing the index of the respective reverse variable.

**prev**

*numpy.array* – The current/last flux sample generated.

**center**

*numpy.array* – The center of the sampling space as estimated by the mean of all previously generated samples.

## Notes

The sampler is very similar to artificial centering where each process samples its own chain. Initial points are chosen randomly from the warmup points followed by a linear transformation that pulls the points towards the a little bit towards the center of the sampling space.

If the number of processes used is larger than one the requested number of samples is adjusted to the smallest multiple of the number of processes larger than the requested sample number. For instance, if you have 3 processes and request 8 samples you will receive 9.

Memory usage is roughly in the order of  $(2 * \text{number reactions})^2$  due to the required nullspace matrices and warmup points. So large models easily take up a few GB of RAM. However, most of the large matrices are kept in shared memory. So the RAM usage is independent of the number of processes.

## References

**sample**(*n, fluxes=True*)

Generate a set of samples.

This is the basic sampling function for all hit-and-run samplers.

**n** [int] The minimum number of samples that are generated at once (see Notes).

**fluxes** [boolean] Whether to return fluxes or the internal solver variables. If set to False will return a variable for each forward and backward flux as well as all additional variables you might have defined in the model.

**Returns** Returns a matrix with  $n$  rows, each containing a flux sample.

**Return type** `numpy.matrix`

## Notes

Performance of this function linearly depends on the number of reactions in your model and the thinning factor.

If the number of processes is larger than one, computation is split across as the CPUs of your machine. This may shorten computation time. However, there is also overhead in setting up parallel computation so we recommend to calculate large numbers of samples at once ( $n > 1000$ ).

```
class cobra.flux_analysis.sampling.Problem(equalities, b, inequalities, bounds, variable_fixed, variable_bounds, nullspace, homogeneous)
```

Bases: tuple

Defines the matrix representation of a sampling problem.

### **equalities**

*numpy.array* – All equality constraints in the model.

### **b**

*numpy.array* – The right side of the equality constraints.

### **inequalities**

*numpy.array* – All inequality constraints in the model.

### **bounds**

*numpy.array* – The lower and upper bounds for the inequality constraints.

### **variable\_bounds**

*numpy.array* – The lower and upper bounds for the variables.

### **homogeneous**

*boolean* – Indicates whether the sampling problem is homogenous, e.g. whether there exist no non-zero fixed variables or constraints.

### **nullspace**

*numpy.matrix* – A matrix containing the nullspace of the equality constraints. Each column is one basis vector.

### **b**

Alias for field number 1

### **bounds**

Alias for field number 3

### **equalities**

Alias for field number 0

### **homogeneous**

Alias for field number 7

**inequalities**

Alias for field number 2

**nullspace**

Alias for field number 6

**variable\_bounds**

Alias for field number 5

**variable\_fixed**

Alias for field number 4

`cobra.flux_analysis.sampling.bounds_tol`

The tolerance used for checking bounds feasibility.

`cobra.flux_analysis.sampling.feasibility_tol`

The tolerance used for checking equalities feasibility.

`cobra.flux_analysis.sampling.mp_init(obj)`

Initialize the multiprocessing pool.

`cobra.flux_analysis.sampling.nproj = 1000000`

Reproject the solution into the feasibility space every nproj iterations.

`cobra.flux_analysis.sampling.nproj_center = 10000`

Reproject the center into the nullspace every nproj\_center iterations. Only used for inhomogeneous problems.

`cobra.flux_analysis.sampling.sample(model, n, method='optgp', thinning=100, processes=1, seed=None)`

Sample valid flux distributions from a cobra model.

The function samples valid flux distributions from a cobra model. Currently we support two methods:

1. **'optgp' (default) which uses the OptGPSampler that supports parallel** [\[1\]](#). Requires large numbers of samples to be performant ( $n < 1000$ ). For smaller samples 'achr' might be better suited.

or

2. 'achr' which uses artificial centering hit-and-run. This is a single process method with good convergence [\[2\]](#).

**Parameters**

- **model** (`cobra.Model`) – The model from which to sample flux distributions.
- **n** (`int`) – The number of samples to obtain. When using 'optgp' this must be a multiple of *processes*, otherwise a larger number of samples will be returned.
- **method** (`str`, *optional*) – The sampling algorithm to use.
- **thinning** (`int`, *optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps. Defaults to 100 which in benchmarks gives approximately uncorrelated samples. If set to one will return all iterates.
- **processes** (`int`, *optional*) – Only used for 'optgp'. The number of processes used to generate samples.
- **seed** (*positive integer*, *optional*) – The random number seed to be used. Initialized to current time stamp if None.

**Returns** The generated flux samples. Each row corresponds to a sample of the fluxes and the columns are the reactions.

**Return type** `pandas.DataFrame`

## Notes

The samplers have a correction method to ensure equality feasibility for long-running chains, however this will only work for homogeneous models, meaning models with no non-zero fixed variables or constraints (right-hand side of the equalities are zero).

## References

`cobra.flux_analysis.sampling.shared_np_array` (*shape*, *data=None*, *integer=False*)

Create a new numpy array that resides in shared memory.

### Parameters

- **shape** (*tuple of ints*) – The shape of the new array.
- **data** (*numpy.array*) – Data to copy to the new array. Has to have the same shape.
- **integer** (*boolean*) – Whether to use an integer array. Defaults to False which means float array.

## cobra.flux\_analysis.summary module

`cobra.flux_analysis.summary.metabolite_summary` (*met*, *solution=None*, *threshold=0.01*,  
*fva=False*, *floatfmt='.3g'*)

Print a summary of the reactions which produce and consume this metabolite

**solution** [`cobra.core.Solution`] A previously solved model solution to use for generating the summary. If none provided (default), the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method.

**threshold** [float] a value below which to ignore reaction fluxes

**fva** [float (0->1), or None] Whether or not to include flux variability analysis in the output. If given, fva should be a float between 0 and 1, representing the fraction of the optimum objective to be searched.

**floatfmt** [string] format method for floats, passed to tabulate. Default is `‘.3g’`.

`cobra.flux_analysis.summary.model_summary` (*model*, *solution=None*, *threshold=1e-08*,  
*fva=None*, *floatfmt='.3g'*)

Print a summary of the input and output fluxes of the model.

**solution** [`cobra.core.Solution`] A previously solved model solution to use for generating the summary. If none provided (default), the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method.

**threshold** [float] tolerance for determining if a flux is zero (not printed)

**fva** [int or None] Whether or not to calculate and report flux variability in the output summary

**floatfmt** [string] format method for floats, passed to tabulate. Default is `‘.3g’`.



**cobra.flux\_analysis.variability module**

`cobra.flux_analysis.variability.find_blocked_reactions` (*model*, *reaction\_list=None*,  
*zero\_cutoff=1e-09*,  
*open\_exchanges=False*)

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

**Parameters**

- **model** (*cobra.Model*) – The model to analyze
- **reaction\_list** (*list*) – List of reactions to consider, use all if left missing
- **zero\_cutoff** (*float*) – Flux value which is considered to effectively be zero.
- **open\_exchanges** (*bool*) – If true, set bounds on exchange reactions to very high values to avoid that being the bottle-neck.

**Returns** List with the blocked reactions

**Return type** `list`

`cobra.flux_analysis.variability.find_essential_genes` (*model*, *threshold=None*, *processes=None*)

Return a set of essential genes.

A gene is considered essential if restricting the flux of all reactions that depends on it to zero causes the objective (e.g. the growth rate) to also be zero.

**Parameters**

- **model** (*cobra.Model*) – The model to find the essential genes for.
- **threshold** (*float*, *optional*) – Minimal objective flux to be considered viable. By default this is 0.01 times the growth rate.
- **processes** (*int*, *optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.

**Returns** Set of essential genes

**Return type** `set`

`cobra.flux_analysis.variability.find_essential_reactions` (*model*, *threshold=None*,  
*processes=None*)

Return a set of essential reactions.

A reaction is considered essential if restricting its flux to zero causes the objective (e.g. the growth rate) to also be zero.

**Parameters**

- **model** (*cobra.Model*) – The model to find the essential reactions for.
- **threshold** (*float*, *optional*) – Minimal objective flux to be considered viable. By default this is 0.01 times the growth rate.
- **processes** (*int*, *optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.

**Returns** Set of essential reactions

**Return type** `set`

```
cobra.flux_analysis.viability.flux_variability_analysis(model, reaction_list=None, loopless=False, fraction_of_optimum=1.0, pfba_factor=None)
```

Runs flux variability analysis to find the min/max flux values for each reaction in *reaction\_list*.

#### Parameters

- **model** (a *cobra model*) – The model for which to run the analysis. It will *not* be modified.
- **reaction\_list** (list of *cobra.Reaction* or *str*, optional) – The reactions for which to obtain min/max fluxes. If *None* will use all reactions in the model.
- **loopless** (*boolean*, optional) – Whether to return only loopless solutions. Ignored for legacy solvers, also see *Notes*.
- **fraction\_of\_optimum** (*float*, optional) – Must be  $\leq 1.0$ . Requires that the objective value is at least  $\text{fraction} * \text{max\_objective\_value}$ . A value of 0.85 for instance means that the objective has to be at least at 85% percent of its maximum.
- **pfba\_factor** (*float*, optional) – Add additional constraint to the model that the total sum of absolute fluxes must not be larger than this value times the smallest possible sum of absolute fluxes, i.e., by setting the value to 1.1 then the total sum of absolute fluxes must not be more than 10% larger than the pfba solution. Since the pfba solution is the one that optimally minimizes the total flux sum, the pfba\_factor should, if set, be larger than one. Setting this value may lead to more realistic predictions of the effective flux bounds.

#### Returns

DataFrame with reaction identifier as the index columns

- maximum: indicating the highest possible flux
- minimum: indicating the lowest possible flux

**Return type** pandas.DataFrame

#### Notes

This implements the fast version as described in<sup>1</sup>. Please note that the flux distribution containing all minimal/maximal fluxes does not have to be a feasible solution for the model. Fluxes are minimized/maximized individually and a single minimal flux might require all others to be suboptimal.

Using the loopless option will lead to a significant increase in computation time (about a factor of 100 for large models). However, the algorithm used here (see<sup>2</sup>) is still more than 1000x faster than the “naive” version using *add\_loopless(model)*. Also note that if you have included constraints that force a loop (for instance by setting all fluxes in a loop to be non-zero) this loop will be included in the solution.

---

<sup>1</sup> Computationally efficient flux variability analysis. Gudmundsson S, Thiele I. BMC Bioinformatics. 2010 Sep 29;11:489. doi: 10.1186/1471-2105-11-489, PMID: 20920235

<sup>2</sup> CycleFreeFlux: efficient removal of thermodynamically infeasible loops from flux distributions. Desouki AA, Jarre F, Gelius-Dietrich G, Lercher MJ. Bioinformatics. 2015 Jul 1;31(13):2159-65. doi: 10.1093/bioinformatics/btv096.

## References

### Module contents

#### cobra.io package

#### Submodules

#### cobra.io.dict module

`cobra.io.dict.gene_from_dict(gene)`

`cobra.io.dict.gene_to_dict(gene)`

`cobra.io.dict.metabolite_from_dict(metabolite)`

`cobra.io.dict.metabolite_to_dict(metabolite)`

`cobra.io.dict.model_from_dict(obj)`

Build a model from a dict.

Models stored in json are first formulated as a dict that can be read to cobra model using this function.

**Parameters** `obj` (`dict`) – A dictionary with elements, ‘genes’, ‘compartments’, ‘id’, ‘metabolites’, ‘notes’ and ‘reactions’; where ‘metabolites’, ‘genes’ and ‘metabolites’ are in turn lists with dictionaries holding all attributes to form the corresponding object.

**Returns** The generated model.

**Return type** `cora.core.Model`

**See also:**

`cobra.io.model_to_dict()`

`cobra.io.dict.model_to_dict(model, sort=False)`

Convert model to a dict.

**Parameters**

- **model** (`cobra.Model`) – The model to reformulate as a dict.
- **sort** (`bool`, *optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

**Returns** A dictionary with elements, ‘genes’, ‘compartments’, ‘id’, ‘metabolites’, ‘notes’ and ‘reactions’; where ‘metabolites’, ‘genes’ and ‘metabolites’ are in turn lists with dictionaries holding all attributes to form the corresponding object.

**Return type** `OrderedDict`

**See also:**

`cobra.io.model_from_dict()`

`cobra.io.dict.reaction_from_dict(reaction, model)`

`cobra.io.dict.reaction_to_dict(reaction)`

## cobra.io.json module

`cobra.io.json.from_json(document)`

Load a cobra model from a JSON document.

**Parameters** `document` (*str*) – The JSON document representation of a cobra model.

**Returns** The cobra model as represented in the JSON document.

**Return type** `cobra.Model`

See also:

`load_json_model()` Load directly from a file.

`cobra.io.json.load_json_model(filename)`

Load a cobra model from a file in JSON format.

**Parameters** `filename` (*str* or *file-like*) – File path or descriptor that contains the JSON document describing the cobra model.

**Returns** The cobra model as represented in the JSON document.

**Return type** `cobra.Model`

See also:

`from_json()` Load from a string.

`cobra.io.json.save_json_model(model, filename, sort=False, pretty=False, **kwargs)`

Write the cobra model to a file in JSON format.

`kwargs` are passed on to `json.dump`.

### Parameters

- **model** (*cobra.Model*) – The cobra model to represent.
- **filename** (*str* or *file-like*) – File path or descriptor that the JSON representation should be written to.
- **sort** (*bool*, *optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.
- **pretty** (*bool*, *optional*) – Whether to format the JSON more compactly (default) or in a more verbose but easier to read fashion. Can be partially overwritten by the `kwargs`.

See also:

`to_json()` Return a string representation.

`json.dump()` Base function.

`cobra.io.json.to_json(model, sort=False, **kwargs)`

Return the model as a JSON document.

`kwargs` are passed on to `json.dumps`.

### Parameters

- **model** (*cobra.Model*) – The cobra model to represent.
- **sort** (*bool*, *optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

**Returns** String representation of the cobra model as a JSON document.

**Return type** `str`

See also:

`save_json_model()` Write directly to a file.

`json.dumps()` Base function.

## cobra.io.mat module

`cobra.io.mat.create_mat_dict(model)`

create a dict mapping model attributes to arrays

`cobra.io.mat.create_mat_metabolite_id(model)`

`cobra.io.mat.from_mat_struct(mat_struct, model_id=None, inf=<Mock object>)`

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, inf=<Mock object>)`

Load a cobra model stored as a .mat file

### Parameters

- **infile\_path** (`str`) – path to the file to to read
- **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
- **inf** (*value*) – The value to use for infinite bounds. Some solvers do not handle infinite values so for using those, set this to a high numeric value.

**Returns** The resulting cobra model

**Return type** `cobra.core.Model.Model`

`cobra.io.mat.model_to_pymatbridge(model, variable_name='model', matlab=None)`

send the model to a MATLAB workspace through pymatbridge

This model can then be manipulated through the COBRA toolbox

### Parameters

- **variable\_name** (`str`) – The variable name to which the model will be assigned in the MATLAB workspace
- **matlab** (*None* or `pymatbridge.Matlab instance`) – The MATLAB workspace to which the variable will be sent. If this is *None*, then this will be sent to the same environment used in IPython magics.

`cobra.io.mat.save_matlab_model(model, file_name, varname=None)`

Save the cobra model as a .mat file.

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

### Parameters

- **model** (`cobra.core.Model.Model object`) – The model to save
- **file\_name** (`str` or *file-like object*) – The file to save to

- **varname** (*string*) – The name of the variable within the workspace

## 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.

### Parameters

- **sbml\_model** (*sbml\_model object*) –
- **cobra\_metabolite** (*a cobra.Metabolite object*) –
- **note\_start\_tag** (*string*) – the start tag for parsing cobra notes. this will eventually be supplanted when COBRA is worked into sbml.
- **note\_end\_tag** (*string*) – the end tag for parsing cobra notes. this will eventually be supplanted when COBRA is worked into sbml.
- **boundary\_metabolite** (*bool*) – if metabolite boundary condition should be set or not

### Returns string

**Return type** the created metabolite identifier

`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.

### Parameters

- **sbml\_filename** (*string*) –
- **old\_sbml** (*bool*) – 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** (*bool*) –  
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** (*bool*) – deprecated
- **use\_hyphens** (*bool*) – If True, double underscores (`__`) in an SBML ID will be converted to hyphens

### Returns Model

**Return type** The parsed cobra model

`cobra.io.sbml.fix_legacy_id` (*id*, *use\_hyphens=False*, *fix\_compartments=False*)

`cobra.io.sbml.get_libsbml_document` (*cobra\_model*, *sbml\_level=2*, *sbml\_version=1*,  
*print\_time=False*, *use\_fbc\_package=True*)

Return a libsbml document object for writing to a file. This function is used by `write_cobra_model_to_sbml_file()`.

`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

#### Parameters

- **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

#### Returns string

**Return type** the identifier

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

Deal with various legacy SBML format issues.

`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.

#### Parameters

- **cobra\_model** (*cobra.core.Model.Model*) – The model object to write
- **sbml\_filename** (*string*) – The file to write the SBML XML to.
- **sbml\_level** (*int*) – 2 is the only supported level.
- **sbml\_version** (*int*) – 1 is the only supported version.
- **print\_time** (*bool*) – deprecated
- **use\_fbc\_package** (*bool*) – Convert the model to the FBC package format to improve portability. [http://sbml.org/Documents/Specifications/SBML\\_Level\\_3/Packages/Flux\\_Balance\\_Constraints\\_flux](http://sbml.org/Documents/Specifications/SBML_Level_3/Packages/Flux_Balance_Constraints_flux)

#### Notes

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

### cobra.io.sbml3 module

**exception** `cobra.io.sbml3.CobraSBMLError`

Bases: Exception

`cobra.io.sbml3.annotate_cobra_from_sbml(cobra_element, sbml_element)`

`cobra.io.sbml3.annotate_sbml_from_cobra(sbml_element, cobra_element)`

`cobra.io.sbml3.clip(string, prefix)`

clips a prefix from the beginning of a string if it exists

```
>>> clip("R_pgi", "R_")
"pgi"
```

`cobra.io.sbml3.construct_gpr_xml` (*parent*, *expression*)  
create gpr xml under parent node

`cobra.io.sbml3.get_attrib` (*tag*, *attribute*, *type*=<function <lambda>>, *require*=False)

`cobra.io.sbml3.indent_xml` (*elem*, *level*=0)  
indent xml for pretty printing

`cobra.io.sbml3.model_to_xml` (*cobra\_model*, *units*=True)

`cobra.io.sbml3.ns` (*query*)  
replace prefixes with namespace

`cobra.io.sbml3.parse_stream` (*filename*)  
parses filename or compressed stream to xml

`cobra.io.sbml3.parse_xml_into_model` (*xml*, *number*=<class 'float'>)

`cobra.io.sbml3.read_sbml_model` (*filename*, *number*=<class 'float'>, *\*\*kwargs*)

`cobra.io.sbml3.set_attrib` (*xml*, *attribute\_name*, *value*)

`cobra.io.sbml3.strnum` (*number*)  
Utility function to convert a number to a string

`cobra.io.sbml3.validate_sbml_model` (*filename*, *check\_model*=True)  
Returns the model along with a list of errors.

#### Parameters

- **filename** (*str*) – The filename of the SBML model to be validated.
- **check\_model** (*bool*, *optional*) – Whether to also check some basic model properties such as reaction boundaries and compartment formulas.

#### Returns

- **model** (Model object) – The cobra model if the file could be read successfully or None otherwise.
- **errors** (*dict*) – Warnings and errors grouped by their respective types.

**Raises** *CobraSBMLError* – If the file is not a valid SBML Level 3 file with FBC.

`cobra.io.sbml3.write_sbml_model` (*cobra\_model*, *filename*, *use\_fbc\_package*=True, *\*\*kwargs*)

## cobra.io.yaml module

`cobra.io.yaml.from_yaml` (*document*)  
Load a cobra model from a YAML document.

**Parameters** *document* (*str*) – The YAML document representation of a cobra model.

**Returns** The cobra model as represented in the YAML document.

**Return type** cobra.Model

See also:

[`load\_yaml\_model\(\)`](#) Load directly from a file.



`cobra.io.yaml.load_yaml_model(filename)`

Load a cobra model from a file in YAML format.

**Parameters** `filename` (*str* or *file-like*) – File path or descriptor that contains the YAML document describing the cobra model.

**Returns** The cobra model as represented in the YAML document.

**Return type** `cobra.Model`

**See also:**

`from_yaml()` Load from a string.

`cobra.io.yaml.save_yaml_model(model, filename, sort=False, **kwargs)`

Write the cobra model to a file in YAML format.

`kwargs` are passed on to `yaml.dump`.

**Parameters**

- **model** (*cobra.Model*) – The cobra model to represent.
- **filename** (*str* or *file-like*) – File path or descriptor that the YAML representation should be written to.
- **sort** (*bool*, *optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

**See also:**

`to_yaml()` Return a string representation.

`ruamel.yaml.dump()` Base function.

`cobra.io.yaml.to_yaml(model, sort=False, **kwargs)`

Return the model as a YAML document.

`kwargs` are passed on to `yaml.dump`.

**Parameters**

- **model** (*cobra.Model*) – The cobra model to represent.
- **sort** (*bool*, *optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

**Returns** String representation of the cobra model as a YAML document.

**Return type** *str*

**See also:**

`save_yaml_model()` Write directly to a file.

`ruamel.yaml.dump()` Base function.

## Module contents

### cobra.manipulation package

#### Submodules

#### cobra.manipulation.annotate module

`cobra.manipulation.annotate.add_SBO(model)`

adds SBO terms for demands and exchanges

This works for models which follow the standard convention for constructing and naming these reactions.

The reaction should only contain the single metabolite being exchanged, and the id should be EX\_metid or DM\_metid

#### 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=None)`

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)`

Remove metabolites that are not involved in any reactions

**Parameters** `cobra_model` (`cobra.Model`) – the model to remove unused metabolites from

**Returns** list of metabolites that were removed

**Return type** list

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

Remove reactions that have no assigned metabolites

**Parameters** `cobra_model` (`cobra.Model`) – the model to remove unused reactions from

**Returns** list of reactions that were removed

**Return type** list

`cobra.manipulation.delete.remove_genes(cobra_model, gene_list, remove_reactions=True)`  
 remove genes entirely from the model

This will also simplify all gene\_reaction\_rules with this gene inactivated.

`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 guarantees 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.escape_ID(cobra_model)`  
 makes all ids SBML compliant

`cobra.manipulation.modify.rename_genes(cobra_model, rename_dict)`  
 renames genes in a model from the rename\_dict

`cobra.manipulation.modify.revert_to_reversible(cobra_model, update_solution=True)`  
 This function will convert an irreversible model made by convert\_to\_irreversible into a reversible model.

**cobra\_model** [cobra.Model] A model which will be modified in place.

**update\_solution: bool** This option is ignored since *model.solution* was removed.

## cobra.manipulation.validate module

`cobra.manipulation.validate.check_mass_balance(model)`

`cobra.manipulation.validate.check_metabolite_compartment_formula(model)`

`cobra.manipulation.validate.check_reaction_bounds(model)`

## Module contents

### cobra.test package

#### Submodules

##### cobra.test.conftest module

##### cobra.test.test\_flux\_analysis module

##### cobra.test.test\_io module

##### cobra.test.test\_io\_order module

##### cobra.test.test\_manipulation module

##### cobra.test.test\_model module

##### cobra.test.test\_solver\_model module

##### cobra.test.test\_solver\_utils module

##### cobra.test.test\_util module

## Module contents

`cobra.test.create_test_model(model_name='salmonella')`

Returns a cobra model for testing

**model\_name:** str One of 'ecoli', 'textbook', or 'salmonella', or the path to a pickled cobra.Model

`cobra.test.test_all(args=None)`

alias for running all unit-tests on installed cobra

### cobra.util package

#### Submodules

##### cobra.util.array module

`cobra.util.array.constraint_matrices(model, array_type='dense', include_vars=False, zero_tol=1e-06)`

Create a matrix representation of the problem.

This is used for alternative solution approaches that do not use optlang. The function will construct the equality matrix, inequality matrix and bounds for the complete problem.

## Notes

To accomodate non-zero equalities the problem will add the variable “const\_one” which is a variable that equals one.

### Parameters

- **model** (*cobra.Model*) – The model from which to obtain the LP problem.
- **array\_type** (*string*) – The type of array to construct. if ‘dense’, return a standard numpy.array, ‘dok’, or ‘lil’ will construct a sparse array using scipy of the corresponding type and ‘DataFrame’ will give a pandas *DataFrame* with metabolite indices and reaction columns.
- **zero\_tol** (*float*) – The zero tolerance used to judge whether two bounds are the same.

### Returns

A named tuple consisting of 6 matrices and 2 vectors: - “equalities” is a matrix  $S$  such that  $S \cdot \text{vars} = b$ . It includes a row

for each constraint and one column for each variable.

- “b” the right side of the equality equation such that  $S \cdot \text{vars} = b$ .
- “inequalities” is a matrix  $M$  such that  $\text{lb} \leq M \cdot \text{vars} \leq \text{ub}$ . It contains a row for each inequality and as many columns as variables.
- “bounds” is a compound matrix [lb ub] containing the lower and upper bounds for the inequality constraints in  $M$ .
- “variable\_fixed” is a boolean vector indicating whether the variable at that index is fixed (lower bound == upper\_bound) and is thus bounded by an equality constraint.
- “variable\_bounds” is a compound matrix [lb ub] containing the lower and upper bounds for all variables.

### Return type `collections.namedtuple`

`cobra.util.array.create_stoichiometric_matrix(model, array_type='dense', dtype=None)`  
Return a stoichiometric array representation of the given model.

The columns represent the reactions and rows represent metabolites.  $S[i,j]$  therefore contains the quantity of metabolite  $i$  produced (negative for consumed) by reaction  $j$ .

### Parameters

- **model** (*cobra.Model*) – The cobra model to construct the matrix for.
- **array\_type** (*string*) – The type of array to construct. if ‘dense’, return a standard numpy.array, ‘dok’, or ‘lil’ will construct a sparse array using scipy of the corresponding type and ‘DataFrame’ will give a pandas *DataFrame* with metabolite indices and reaction columns
- **dtype** (*data-type*) – The desired data-type for the array. If not given, defaults to float.

**Returns** The stoichiometric matrix for the given model.

**Return type** matrix of class *dtype*

`cobra.util.array.nullspace(A, atol=1e-13, rtol=0)`

Compute an approximate basis for the nullspace of  $A$ . The algorithm used by this function is based on the singular value decomposition of  $A$ .

### Parameters

- **A** (*numpy.ndarray*) – A should be at most 2-D. A 1-D array with length k will be treated as a 2-D with shape (1, k)
- **atol** (*float*) – The absolute tolerance for a zero singular value. Singular values smaller than *atol* are considered to be zero.
- **rtol** (*float*) – The relative tolerance. Singular values less than  $\text{rtol} * \text{smax}$  are considered to be zero, where *smax* is the largest singular value.
- **both atol and rtol are positive, the combined tolerance is the** (*If*) –

:param maximum of the two; that is::: :param *tol* =  $\max(\text{atol}, \text{rtol} * \text{smax})$ : :param Singular values smaller than *tol* are considered to be zero.:

**Returns** If *A* is an array with shape (m, k), then *ns* will be an array with shape (k, n), where n is the estimated dimension of the nullspace of *A*. The columns of *ns* are a basis for the nullspace; each element in  $\text{numpy.dot}(A, ns)$  will be approximately zero.

**Return type** *numpy.ndarray*

### Notes

Taken from the numpy cookbook.

## cobra.util.context module

**class** `cobra.util.context.HistoryManager`

Bases: `object`

Record a list of actions to be taken at a later time. Used to implement context managers that allow temporary changes to a `Model`.

**reset** ()

Trigger executions for all items in the stack in reverse order

`cobra.util.context.get_context (obj)`

Search for a context manager

`cobra.util.context.resettable (f)`

A decorator to simplify the context management of simple object attributes. Gets the value of the attribute prior to setting it, and stores a function to set the value to the old value in the `HistoryManager`.

## cobra.util.solver module

Additional helper functions for the optlang solvers.

All functions integrate well with the context manager, meaning that all operations defined here are automatically reverted when used in a *with model:* block.

The functions defined here together with the existing model functions should allow you to implement custom flux analysis methods with ease.

`cobra.util.solver.add_absolute_expression (model, expression, name='abs_var', ub=None, difference=0, add=True)`

Add the absolute value of an expression to the model.

Also defines a variable for the absolute value that can be used in other objectives or constraints.

#### Parameters

- **model** (*a cobra model*) – The model to which to add the absolute expression.
- **expression** (*A sympy expression*) – Must be a valid expression within the Model's solver object. The absolute value is applied automatically on the expression.
- **name** (*string*) – The name of the newly created variable.
- **ub** (*positive float*) – The upper bound for the variable.
- **difference** (*positive float*) – The difference between the expression and the variable.
- **add** (*bool*) – Whether to add the variable to the model at once.

**Returns** A named tuple with variable and two constraints (upper\_constraint, lower\_constraint) describing the new variable and the constraints that assign the absolute value of the expression to it.

**Return type** namedtuple

```
cobra.util.solver.add_cons_vars_to_problem(model, what, **kwargs)
```

Add variables and constraints to a Model's solver object.

Useful for variables and constraints that can not be expressed with reactions and lower/upper bounds. Will integrate with the Model's context manager in order to revert changes upon leaving the context.

#### Parameters

- **model** (*a cobra model*) – The model to which to add the variables and constraints.
- **what** (*list or tuple of optlang variables or constraints.*) – The variables or constraints to add to the model. Must be of class *model.problem.Variable* or *model.problem.Constraint*.
- **\*\*kwargs** (*keyword arguments*) – passed to solver.add()

```
cobra.util.solver.assert_optimal(model, message='optimization failed')
```

Assert model solver status is optimal.

Do nothing if model solver status is optimal, otherwise throw appropriate exception depending on the status.

#### Parameters

- **model** (*cobra.Model*) – The model to check the solver status for.
- **message** (*str (optional)*) – Message to for the exception if solver status was not optimal.

```
cobra.util.solver.check_solver_status(status, raise_error=False)
```

Perform standard checks on a solver's status.

```
cobra.util.solver.choose_solver(model, solver=None, qp=False)
```

Choose a solver given a solver name and model.

This will choose a solver compatible with the model and required capabilities. Also respects model.solver where it can.

#### Parameters

- **model** (*a cobra model*) – The model for which to choose the solver.
- **solver** (*str, optional*) – The name of the solver to be used.

- **qp** (*boolean, optional*) – Whether the solver needs Quadratic Programming capabilities.

**Returns** *solver* – Returns a valid solver for the problem.

**Return type** an optlang solver interface

**Raises** *SolverNotFound* – If no suitable solver could be found.

```
cobra.util.solver.fix_objective_as_constraint(model, fraction=1, bound=None,
                                             name='fixed_objective_{}')

```

Fix current objective as an additional constraint.

When adding constraints to a model, such as done in pFBA which minimizes total flux, these constraints can become too powerful, resulting in solutions that satisfy optimality but sacrifice too much for the original objective function. To avoid that, we can fix the current objective value as a constraint to ignore solutions that give a lower (or higher depending on the optimization direction) objective value than the original model.

When done with the model as a context, the modification to the objective will be reverted when exiting that context.

#### Parameters

- **model** (*cobra.Model*) – The model to operate on
- **fraction** (*float*) – The fraction of the optimum the objective is allowed to reach.
- **bound** (*float, None*) – The bound to use instead of fraction of maximum optimal value. If not None, fraction is ignored.
- **name** (*str*) – Name of the objective. May contain one `{}` placeholder which is filled with the name of the old objective.

```
cobra.util.solver.get_solver_name(mip=False, qp=False)

```

Select a solver for a given optimization problem.

#### Parameters

- **mip** (*bool*) – Does the solver require mixed integer linear programming capabilities?
- **qp** (*bool*) – Does the solver require quadratic programming capabilities?

**Returns** The name of feasible solver.

**Return type** *string*

**Raises** *SolverNotFound* – If no suitable solver could be found.

```
cobra.util.solver.interface_to_str(interface)

```

Give a string representation for an optlang interface.

**Parameters** *interface* (*string, ModuleType*) – Full name of the interface in optlang or cobra representation. For instance ‘optlang.glpk\_interface’ or ‘optlang-glpk’.

**Returns** The name of the interface as a string

**Return type** *string*

```
cobra.util.solver.linear_reaction_coefficients(model, reactions=None)

```

Coefficient for the reactions in a linear objective.

#### Parameters

- **model** (*cobra model*) – the model object that defined the objective
- **reactions** (*list*) – an optional list for the reactions to get the coefficients for. All reactions if left missing.



**Returns** A dictionary where the key is the reaction object and the value is the corresponding coefficient. Empty dictionary if there are no linear terms in the objective.

**Return type** *dict*

`cobra.util.solver.remove_cons_vars_from_problem(model, what)`

Remove variables and constraints from a Model's solver object.

Useful to temporarily remove variables and constraints from a Models's solver object.

#### Parameters

- **model** (*a cobra model*) – The model from which to remove the variables and constraints.
- **what** (*list or tuple of optlang variables or constraints.*) – The variables or constraints to remove from the model. Must be of class *model.problem.Variable* or *model.problem.Constraint*.

`cobra.util.solver.set_objective(model, value, additive=False)`

Set the model objective.

#### Parameters

- **model** (*cobra model*) – The model to set the objective for
- **value** (*model.problem.Objective,*) – e.g. `optlang.glpk_interface.Objective`, `sympy.Basic` or `dict`  
  
If the model objective is linear, the value can be a new `Objective` object or a dictionary with linear coefficients where each key is a reaction and the element the new coefficient (float).  
  
If the objective is not linear and *additive* is true, only values of class `Objective`.
- **additive** (*boolmodel.reactions.Biomass\_Ecoli\_core.bounds = (0.1, 0.1)*) – If true, add the terms to the current objective, otherwise start with an empty objective.

## cobra.util.util module

**class** `cobra.util.util.AutoVivification`

Bases: `dict`

Implementation of perl's autovivification feature. Checkout <http://stackoverflow.com/a/652284/280182>

`cobra.util.util.format_long_string(string, max_length=50)`

## cobra.util.version\_info module

`cobra.util.version_info.show_versions()`

Print the formatted information to standard out.

## Module contents

### 14.1.2 Submodules

**cobra.config module**

**cobra.exceptions module**

**exception** `cobra.exceptions.FeasibleButNotOptimal` (*message*)  
Bases: `cobra.exceptions.OptimizationError`

**exception** `cobra.exceptions.Infeasible` (*message*)  
Bases: `cobra.exceptions.OptimizationError`

**exception** `cobra.exceptions.OptimizationError` (*message*)  
Bases: `Exception`

**exception** `cobra.exceptions.SolverNotFound`  
Bases: `Exception`

A simple Exception when a solver can not be found.

**exception** `cobra.exceptions.Unbounded` (*message*)  
Bases: `cobra.exceptions.OptimizationError`

**exception** `cobra.exceptions.UndefinedSolution` (*message*)  
Bases: `cobra.exceptions.OptimizationError`

### 14.1.3 Module contents

## CHAPTER 15

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### C

- `cobra`, 110
- `cobra.config`, 110
- `cobra.core`, 74
  - `cobra.core.dictlist`, 55
  - `cobra.core.formula`, 57
  - `cobra.core.gene`, 57
  - `cobra.core.metabolite`, 58
  - `cobra.core.model`, 60
  - `cobra.core.object`, 65
  - `cobra.core.reaction`, 65
  - `cobra.core.solution`, 71
  - `cobra.core.species`, 73
- `cobra.exceptions`, 110
- `cobra.flux_analysis`, 95
  - `cobra.flux_analysis.deletion`, 74
  - `cobra.flux_analysis.gapfilling`, 76
  - `cobra.flux_analysis.loopless`, 78
  - `cobra.flux_analysis.moma`, 80
  - `cobra.flux_analysis.parsimonious`, 81
  - `cobra.flux_analysis.phenotype_phase_plane`, 81
  - `cobra.flux_analysis.reaction`, 84
  - `cobra.flux_analysis.sampling`, 85
  - `cobra.flux_analysis.summary`, 92
  - `cobra.flux_analysis.variability`, 93
- `cobra.io`, 102
  - `cobra.io.dict`, 95
  - `cobra.io.json`, 96
  - `cobra.io.mat`, 97
  - `cobra.io.sbml`, 98
  - `cobra.io.sbml3`, 99
  - `cobra.io.yaml`, 100
- `cobra.manipulation`, 104
  - `cobra.manipulation.annotate`, 102
  - `cobra.manipulation.delete`, 102
  - `cobra.manipulation.modify`, 103
  - `cobra.manipulation.validate`, 103
- `cobra.test`, 104
- `cobra.util`, 110
  - `cobra.util.array`, 104
  - `cobra.util.context`, 106
  - `cobra.util.solver`, 106
  - `cobra.util.util`, 109
  - `cobra.util.version_info`, 109



## A

ACHRSampler (class in cobra.flux\_analysis.sampling), 85

add() (cobra.core.dictlist.DictList method), 55

add\_absolute\_expression() (in module cobra.util.solver), 106

add\_boundary() (cobra.core.model.Model method), 60

add\_cons\_vars() (cobra.core.model.Model method), 61

add\_cons\_vars\_to\_problem() (in module cobra.util.solver), 107

add\_envelope() (in module cobra.flux\_analysis.phenotype\_phase\_plane), 81

add\_loopless() (in module cobra.flux\_analysis.loopless), 78

add\_metabolites() (cobra.core.model.Model method), 61

add\_metabolites() (cobra.core.reaction.Reaction method), 65

add\_moma() (in module cobra.flux\_analysis.moma), 80

add\_pfba() (in module cobra.flux\_analysis.parsimonious), 81

add\_reaction() (cobra.core.model.Model method), 61

add\_reactions() (cobra.core.model.Model method), 62

add\_sbml\_species() (in module cobra.io.sbml), 98

add\_SBO() (in module cobra.manipulation.annotate), 102

add\_switches\_and\_objective() (cobra.flux\_analysis.gapfilling.GapFiller method), 77

annotate\_cobra\_from\_sbml() (in module cobra.io.sbml3), 99

annotate\_sbml\_from\_cobra() (in module cobra.io.sbml3), 99

append() (cobra.core.dictlist.DictList method), 55

assert\_optimal() (in module cobra.util.solver), 107

assess() (in module cobra.flux\_analysis.reaction), 84

assess\_component() (in module cobra.flux\_analysis.reaction), 84

assess\_precursors() (in module cobra.flux\_analysis.reaction), 84

assess\_products() (in module cobra.flux\_analysis.reaction), 85

ast2str() (in module cobra.core.gene), 58

AutoVivification (class in cobra.util.util), 109

## B

b (cobra.flux\_analysis.sampling.Problem attribute), 90

batch() (cobra.flux\_analysis.sampling.HRSampler method), 87

boundary (cobra.core.reaction.Reaction attribute), 66

bounds (cobra.core.reaction.Reaction attribute), 66

bounds (cobra.flux\_analysis.sampling.Problem attribute), 90

bounds\_tol (in module cobra.flux\_analysis.sampling), 91

build\_reaction\_from\_string() (cobra.core.reaction.Reaction method), 66

build\_reaction\_string() (cobra.core.reaction.Reaction method), 66

## C

center (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86

center (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89

check\_mass\_balance() (cobra.core.reaction.Reaction method), 66

check\_mass\_balance() (in module cobra.manipulation.validate), 103

check\_metabolite\_compartment\_formula() (in module cobra.manipulation.validate), 103

check\_reaction\_bounds() (in module cobra.manipulation.validate), 103

check\_solver\_status() (in module cobra.util.solver), 107

choose\_solver() (in module cobra.util.solver), 107

clip() (in module cobra.io.sbml3), 99

cobra (module), 110

cobra.config (module), 110

cobra.core (module), 74

cobra.core.dictlist (module), 55

[cobra.core.formula \(module\)](#), 57  
[cobra.core.gene \(module\)](#), 57  
[cobra.core.metabolite \(module\)](#), 58  
[cobra.core.model \(module\)](#), 60  
[cobra.core.object \(module\)](#), 65  
[cobra.core.reaction \(module\)](#), 65  
[cobra.core.solution \(module\)](#), 71  
[cobra.core.species \(module\)](#), 73  
[cobra.exceptions \(module\)](#), 110  
[cobra.flux\\_analysis \(module\)](#), 95  
[cobra.flux\\_analysis.deletion \(module\)](#), 74  
[cobra.flux\\_analysis.gapfilling \(module\)](#), 76  
[cobra.flux\\_analysis.loopless \(module\)](#), 78  
[cobra.flux\\_analysis.moma \(module\)](#), 80  
[cobra.flux\\_analysis.parsimonious \(module\)](#), 81  
[cobra.flux\\_analysis.phenotype\\_phase\\_plane \(module\)](#), 81  
[cobra.flux\\_analysis.reaction \(module\)](#), 84  
[cobra.flux\\_analysis.sampling \(module\)](#), 85  
[cobra.flux\\_analysis.summary \(module\)](#), 92  
[cobra.flux\\_analysis.variability \(module\)](#), 93  
[cobra.io \(module\)](#), 102  
[cobra.io.dict \(module\)](#), 95  
[cobra.io.json \(module\)](#), 96  
[cobra.io.mat \(module\)](#), 97  
[cobra.io.sbml \(module\)](#), 98  
[cobra.io.sbml3 \(module\)](#), 99  
[cobra.io.yaml \(module\)](#), 100  
[cobra.manipulation \(module\)](#), 104  
[cobra.manipulation.annotate \(module\)](#), 102  
[cobra.manipulation.delete \(module\)](#), 102  
[cobra.manipulation.modify \(module\)](#), 103  
[cobra.manipulation.validate \(module\)](#), 103  
[cobra.test \(module\)](#), 104  
[cobra.util \(module\)](#), 110  
[cobra.util.array \(module\)](#), 104  
[cobra.util.context \(module\)](#), 106  
[cobra.util.solver \(module\)](#), 106  
[cobra.util.util \(module\)](#), 109  
[cobra.util.version\\_info \(module\)](#), 109  
[CobraSBMLError](#), 99  
[compartments \(cobra.core.model.Model attribute\)](#), 62  
[compartments \(cobra.core.reaction.Reaction attribute\)](#), 66  
[constraint \(cobra.core.metabolite.Metabolite attribute\)](#), 58  
[constraint\\_matrices\(\) \(in module cobra.util.array\)](#), 104  
[constraints \(cobra.core.model.Model attribute\)](#), 62  
[construct\\_gpr\\_xml\(\) \(in module cobra.io.sbml3\)](#), 100  
[construct\\_loopless\\_model\(\) \(in module cobra.flux\\_analysis.loopless\)](#), 79  
[convert\\_to\\_irreversible\(\) \(in module cobra.manipulation.modify\)](#), 103  
[copy\(\) \(cobra.core.model.Model method\)](#), 62  
[copy\(\) \(cobra.core.reaction.Reaction method\)](#), 67  
[copy\(\) \(cobra.core.species.Species method\)](#), 73

[create\\_cobra\\_model\\_from\\_sbml\\_file\(\) \(in module cobra.io.sbml\)](#), 98  
[create\\_mat\\_dict\(\) \(in module cobra.io.mat\)](#), 97  
[create\\_mat\\_metabolite\\_id\(\) \(in module cobra.io.mat\)](#), 97  
[create\\_stoichiometric\\_matrix\(\) \(in module cobra.util.array\)](#), 105  
[create\\_test\\_model\(\) \(in module cobra.test\)](#), 104

## D

[delete\(\) \(cobra.core.reaction.Reaction method\)](#), 67  
[delete\\_model\\_genes\(\) \(in module cobra.manipulation.delete\)](#), 102  
[description \(cobra.core.model.Model attribute\)](#), 62  
[DictList \(class in cobra.core.dictlist\)](#), 55  
[double\\_gene\\_deletion\(\) \(in module cobra.flux\\_analysis.deletion\)](#), 74  
[double\\_reaction\\_deletion\(\) \(in module cobra.flux\\_analysis.deletion\)](#), 74  
[dress\\_results\(\) \(cobra.core.solution.LegacySolution method\)](#), 72

## E

[elements \(cobra.core.metabolite.Metabolite attribute\)](#), 59  
[equalities \(cobra.flux\\_analysis.sampling.Problem attribute\)](#), 90  
[escape\\_ID\(\) \(in module cobra.manipulation.modify\)](#), 103  
[eval\\_gpr\(\) \(in module cobra.core.gene\)](#), 58  
[exchanges \(cobra.core.model.Model attribute\)](#), 62  
[extend\(\) \(cobra.core.dictlist.DictList method\)](#), 55  
[extend\\_model\(\) \(cobra.flux\\_analysis.gapfilling.GapFiller method\)](#), 77

## F

[f \(cobra.core.solution.LegacySolution attribute\)](#), 72  
[f \(cobra.core.solution.Solution attribute\)](#), 71, 72  
[feasibility\\_tol \(in module cobra.flux\\_analysis.sampling\)](#), 91  
[FeasibleButNotOptimal](#), 110  
[fill\(\) \(cobra.flux\\_analysis.gapfilling.GapFiller method\)](#), 77  
[find\\_blocked\\_reactions\(\) \(in module cobra.flux\\_analysis.variability\)](#), 93  
[find\\_carbon\\_sources\(\) \(in module cobra.flux\\_analysis.phenotype\\_phase\\_plane\)](#), 81  
[find\\_essential\\_genes\(\) \(in module cobra.flux\\_analysis.variability\)](#), 93  
[find\\_essential\\_reactions\(\) \(in module cobra.flux\\_analysis.variability\)](#), 93  
[find\\_gene\\_knockout\\_reactions\(\) \(in module cobra.manipulation.delete\)](#), 102  
[fix\\_legacy\\_id\(\) \(in module cobra.io.sbml\)](#), 98  
[fix\\_objective\\_as\\_constraint\(\) \(in module cobra.util.solver\)](#), 108



flux (cobra.core.reaction.Reaction attribute), 67  
 flux\_expression (cobra.core.reaction.Reaction attribute), 67  
 flux\_variability\_analysis() (in module cobra.flux\_analysis.variability), 94  
 fluxes (cobra.core.solution.Solution attribute), 71  
 format\_long\_string() (in module cobra.util.util), 109  
 Formula (class in cobra.core.formula), 57  
 formula\_weight (cobra.core.metabolite.Metabolite attribute), 59  
 forward\_variable (cobra.core.reaction.Reaction attribute), 68  
 from\_json() (in module cobra.io.json), 96  
 from\_mat\_struct() (in module cobra.io.mat), 97  
 from\_yaml() (in module cobra.io.yaml), 100  
 functional (cobra.core.gene.Gene attribute), 57  
 functional (cobra.core.reaction.Reaction attribute), 68  
 fwd\_idx (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
 fwd\_idx (cobra.flux\_analysis.sampling.HRSampler attribute), 87  
 fwd\_idx (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89

## G

gapfill() (in module cobra.flux\_analysis.gapfilling), 77  
 GapFiller (class in cobra.flux\_analysis.gapfilling), 76  
 Gene (class in cobra.core.gene), 57  
 gene\_from\_dict() (in module cobra.io.dict), 95  
 gene\_name\_reaction\_rule (cobra.core.reaction.Reaction attribute), 68  
 gene\_reaction\_rule (cobra.core.reaction.Reaction attribute), 68  
 gene\_to\_dict() (in module cobra.io.dict), 95  
 generate\_fva\_warmup() (cobra.flux\_analysis.sampling.HRSampler method), 88  
 genes (cobra.core.model.Model attribute), 60  
 genes (cobra.core.reaction.Reaction attribute), 68  
 get\_attr() (in module cobra.io.sbml3), 100  
 get\_by\_any() (cobra.core.dictlist.DictList method), 55  
 get\_by\_id() (cobra.core.dictlist.DictList method), 56  
 get\_coefficient() (cobra.core.reaction.Reaction method), 68  
 get\_coefficients() (cobra.core.reaction.Reaction method), 68  
 get\_compartments() (cobra.core.reaction.Reaction method), 68  
 get\_compiled\_gene\_reaction\_rules() (in module cobra.manipulation.delete), 102  
 get\_context() (in module cobra.util.context), 106  
 get\_libsbml\_document() (in module cobra.io.sbml), 98  
 get\_metabolite\_compartments() (cobra.core.model.Model method), 62

get\_primal\_by\_id() (cobra.core.solution.Solution method), 72  
 get\_solution() (in module cobra.core.solution), 72  
 get\_solver\_name() (in module cobra.util.solver), 108  
 GPRCleaner (class in cobra.core.gene), 57

## H

has\_id() (cobra.core.dictlist.DictList method), 56  
 HistoryManager (class in cobra.util.context), 106  
 homogeneous (cobra.flux\_analysis.sampling.Problem attribute), 90  
 HRSampler (class in cobra.flux\_analysis.sampling), 87

## I

id (cobra.core.object.Object attribute), 65  
 indent\_xml() (in module cobra.io.sbml3), 100  
 index() (cobra.core.dictlist.DictList method), 56  
 inequalities (cobra.flux\_analysis.sampling.Problem attribute), 90  
 Infeasible, 110  
 insert() (cobra.core.dictlist.DictList method), 56  
 interface\_to\_str() (in module cobra.util.solver), 108

## K

knock\_out() (cobra.core.gene.Gene method), 57  
 knock\_out() (cobra.core.reaction.Reaction method), 68

## L

LegacySolution (class in cobra.core.solution), 72  
 linear\_reaction\_coefficients() (in module cobra.util.solver), 108  
 list\_attr() (cobra.core.dictlist.DictList method), 56  
 load\_json\_model() (in module cobra.io.json), 96  
 load\_matlab\_model() (in module cobra.io.mat), 97  
 load\_yaml\_model() (in module cobra.io.yaml), 100  
 LOGGER (in module cobra.flux\_analysis.sampling), 88  
 loopless\_fva\_iter() (in module cobra.flux\_analysis.loopless), 79  
 loopless\_solution() (in module cobra.flux\_analysis.loopless), 79  
 lower\_bound (cobra.core.reaction.Reaction attribute), 68

## M

medium (cobra.core.model.Model attribute), 62  
 merge() (cobra.core.model.Model method), 62  
 Metabolite (class in cobra.core.metabolite), 58  
 metabolite\_from\_dict() (in module cobra.io.dict), 95  
 metabolite\_summary() (in module cobra.flux\_analysis.summary), 92  
 metabolite\_to\_dict() (in module cobra.io.dict), 95  
 metabolites (cobra.core.model.Model attribute), 60  
 metabolites (cobra.core.reaction.Reaction attribute), 68  
 Model (class in cobra.core.model), 60

model (cobra.core.reaction.Reaction attribute), 68  
 model (cobra.core.species.Species attribute), 73  
 model (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
 model (cobra.flux\_analysis.sampling.HRSampler attribute), 87  
 model (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89  
 model\_from\_dict() (in module cobra.io.dict), 95  
 model\_summary() (in module cobra.flux\_analysis.summary), 92  
 model\_to\_dict() (in module cobra.io.dict), 95  
 model\_to\_pymatbridge() (in module cobra.io.mat), 97  
 model\_to\_xml() (in module cobra.io.sbml3), 100  
 mp\_init() (in module cobra.flux\_analysis.sampling), 91

## N

n\_samples (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
 n\_samples (cobra.flux\_analysis.sampling.HRSampler attribute), 87  
 n\_samples (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89  
 nproj (in module cobra.flux\_analysis.sampling), 91  
 nproj\_center (in module cobra.flux\_analysis.sampling), 91  
 ns() (in module cobra.io.sbml3), 100  
 nullspace (cobra.flux\_analysis.sampling.Problem attribute), 90, 91  
 nullspace() (in module cobra.util.array), 105

## O

Object (class in cobra.core.object), 65  
 objective (cobra.core.model.Model attribute), 62  
 objective\_coefficient (cobra.core.reaction.Reaction attribute), 68  
 objective\_direction (cobra.core.model.Model attribute), 63  
 objective\_value (cobra.core.solution.Solution attribute), 71  
 OptGPSampler (class in cobra.flux\_analysis.sampling), 88  
 OptimizationError, 110  
 optimize() (cobra.core.model.Model method), 63  
 optimize\_minimal\_flux() (in module cobra.flux\_analysis.parsimonious), 81

## P

parse\_composition() (cobra.core.formula.Formula method), 57  
 parse\_gpr() (in module cobra.core.gene), 58  
 parse\_legacy\_id() (in module cobra.io.sbml), 98  
 parse\_legacy\_sbml\_notes() (in module cobra.io.sbml), 99  
 parse\_stream() (in module cobra.io.sbml3), 100

parse\_xml\_into\_model() (in module cobra.io.sbml3), 100  
 pfba() (in module cobra.flux\_analysis.parsimonious), 81  
 pop() (cobra.core.dictlist.DictList method), 56  
 prev (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
 prev (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89  
 Problem (class in cobra.flux\_analysis.sampling), 90  
 problem (cobra.core.model.Model attribute), 63  
 problem (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
 problem (cobra.flux\_analysis.sampling.HRSampler attribute), 87  
 problem (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89  
 production\_envelope() (in module cobra.flux\_analysis.phenotype\_phase\_plane), 82  
 products (cobra.core.reaction.Reaction attribute), 69  
 prune\_unused\_metabolites() (in module cobra.manipulation.delete), 102  
 prune\_unused\_reactions() (in module cobra.manipulation.delete), 102

## Q

query() (cobra.core.dictlist.DictList method), 56

## R

reactants (cobra.core.reaction.Reaction attribute), 69  
 Reaction (class in cobra.core.reaction), 65  
 reaction (cobra.core.reaction.Reaction attribute), 69  
 reaction\_elements() (in module cobra.flux\_analysis.phenotype\_phase\_plane), 83  
 reaction\_from\_dict() (in module cobra.io.dict), 95  
 reaction\_to\_dict() (in module cobra.io.dict), 95  
 reaction\_weight() (in module cobra.flux\_analysis.phenotype\_phase\_plane), 83  
 reactions (cobra.core.model.Model attribute), 60  
 reactions (cobra.core.species.Species attribute), 73  
 read\_legacy\_sbml() (in module cobra.io.sbml), 99  
 read\_sbml\_model() (in module cobra.io.sbml3), 100  
 reduced\_cost (cobra.core.reaction.Reaction attribute), 69  
 reduced\_costs (cobra.core.solution.Solution attribute), 71  
 remove() (cobra.core.dictlist.DictList method), 56  
 remove\_cons\_vars() (cobra.core.model.Model method), 63  
 remove\_cons\_vars\_from\_problem() (in module cobra.util.solver), 109  
 remove\_from\_model() (cobra.core.gene.Gene method), 57  
 remove\_from\_model() (cobra.core.metabolite.Metabolite method), 59

- [remove\\_from\\_model\(\)](#) (cobra.core.reaction.Reaction method), 69  
[remove\\_genes\(\)](#) (in module cobra.manipulation.delete), 103  
[remove\\_metabolites\(\)](#) (cobra.core.model.Model method), 63  
[remove\\_reactions\(\)](#) (cobra.core.model.Model method), 64  
[rename\\_genes\(\)](#) (in module cobra.manipulation.modify), 103  
[repair\(\)](#) (cobra.core.model.Model method), 64  
[reset\(\)](#) (cobra.util.context.HistoryManager method), 106  
[resettable\(\)](#) (in module cobra.util.context), 106  
[rev\\_idx](#) (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
[rev\\_idx](#) (cobra.flux\_analysis.sampling.HRSampler attribute), 87  
[rev\\_idx](#) (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89  
[reverse\(\)](#) (cobra.core.dictlist.DictList method), 56  
[reverse\\_id](#) (cobra.core.reaction.Reaction attribute), 69  
[reverse\\_variable](#) (cobra.core.reaction.Reaction attribute), 69  
[reversibility](#) (cobra.core.reaction.Reaction attribute), 70  
[revert\\_to\\_reversible\(\)](#) (in module cobra.manipulation.modify), 103
- ## S
- [sample\(\)](#) (cobra.flux\_analysis.sampling.ACHRSampler method), 86  
[sample\(\)](#) (cobra.flux\_analysis.sampling.HRSampler method), 88  
[sample\(\)](#) (cobra.flux\_analysis.sampling.OptGPSampler method), 89  
[sample\(\)](#) (in module cobra.flux\_analysis.sampling), 91  
[save\\_json\\_model\(\)](#) (in module cobra.io.json), 96  
[save\\_matlab\\_model\(\)](#) (in module cobra.io.mat), 97  
[save\\_yaml\\_model\(\)](#) (in module cobra.io.yaml), 101  
[seed](#) (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
[seed](#) (cobra.flux\_analysis.sampling.HRSampler attribute), 87  
[seed](#) (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89  
[separate\\_forward\\_and\\_reverse\\_bounds\(\)](#) (in module cobra.core.reaction), 70  
[set\\_attr\(\)](#) (in module cobra.io.sbml3), 100  
[set\\_objective\(\)](#) (in module cobra.util.solver), 109  
[shadow\\_price](#) (cobra.core.metabolite.Metabolite attribute), 59  
[shadow\\_prices](#) (cobra.core.solution.Solution attribute), 71  
[shared\\_np\\_array\(\)](#) (in module cobra.flux\_analysis.sampling), 92
- [show\\_versions\(\)](#) (in module cobra.util.version\_info), 109  
[single\\_gene\\_deletion\(\)](#) (in module cobra.flux\_analysis.deletion), 75  
[single\\_reaction\\_deletion\(\)](#) (in module cobra.flux\_analysis.deletion), 75  
[slim\\_optimize\(\)](#) (cobra.core.model.Model method), 64  
[Solution](#) (class in cobra.core.solution), 71  
[solution](#) (cobra.core.model.Model attribute), 60  
[solver](#) (cobra.core.model.Model attribute), 64  
[solver](#) (cobra.core.solution.LegacySolution attribute), 72  
[SolverNotFound](#), 110  
[sort\(\)](#) (cobra.core.dictlist.DictList method), 56  
[Species](#) (class in cobra.core.species), 73  
[status](#) (cobra.core.solution.Solution attribute), 71  
[strnum\(\)](#) (in module cobra.io.sbml3), 100  
[subtract\\_metabolites\(\)](#) (cobra.core.reaction.Reaction method), 70  
[summary\(\)](#) (cobra.core.metabolite.Metabolite method), 59  
[summary\(\)](#) (cobra.core.model.Model method), 64
- ## T
- [test\\_all\(\)](#) (in module cobra.test), 104  
[thinning](#) (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
[thinning](#) (cobra.flux\_analysis.sampling.HRSampler attribute), 87  
[thinning](#) (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89  
[to\\_frame\(\)](#) (cobra.core.solution.Solution method), 72  
[to\\_json\(\)](#) (in module cobra.io.json), 96  
[to\\_yaml\(\)](#) (in module cobra.io.yaml), 101  
[total\\_components\\_flux\(\)](#) (in module cobra.flux\_analysis.phenotype\_phase\_plane), 83  
[total\\_yield\(\)](#) (in module cobra.flux\_analysis.phenotype\_phase\_plane), 83
- ## U
- [Unbounded](#), 110  
[UndefinedSolution](#), 110  
[undele\\_model\\_genes\(\)](#) (in module cobra.manipulation.delete), 103  
[union\(\)](#) (cobra.core.dictlist.DictList method), 56  
[update\\_costs\(\)](#) (cobra.flux\_analysis.gapfilling.GapFiller method), 77  
[update\\_forward\\_and\\_reverse\\_bounds\(\)](#) (in module cobra.core.reaction), 71  
[upper\\_bound](#) (cobra.core.reaction.Reaction attribute), 70
- ## V
- [validate\(\)](#) (cobra.flux\_analysis.gapfilling.GapFiller method), 77

validate() (cobra.flux\_analysis.sampling.HRSampler method), 88  
 validate\_sbml\_model() (in module cobra.io.sbml3), 100  
 variable\_bounds (cobra.flux\_analysis.sampling.Problem attribute), 90, 91  
 variable\_fixed (cobra.flux\_analysis.sampling.Problem attribute), 91  
 variables (cobra.core.model.Model attribute), 65  
 visit\_BinOp() (cobra.core.gene.GPRCleaner method), 57  
 visit\_Name() (cobra.core.gene.GPRCleaner method), 57

## W

warmup (cobra.flux\_analysis.sampling.ACHRSampler attribute), 86  
 warmup (cobra.flux\_analysis.sampling.HRSampler attribute), 87  
 warmup (cobra.flux\_analysis.sampling.OptGPSampler attribute), 89  
 weight (cobra.core.formula.Formula attribute), 57  
 write\_cobra\_model\_to\_sbml\_file() (in module cobra.io.sbml), 99  
 write\_sbml\_model() (in module cobra.io.sbml3), 100

## X

x (cobra.core.reaction.Reaction attribute), 70  
 x (cobra.core.solution.LegacySolution attribute), 72  
 x (cobra.core.solution.Solution attribute), 71, 72  
 x\_dict (cobra.core.solution.LegacySolution attribute), 72  
 x\_dict (cobra.core.solution.Solution attribute), 71, 72

## Y

y (cobra.core.metabolite.Metabolite attribute), 60  
 y (cobra.core.reaction.Reaction attribute), 70  
 y (cobra.core.solution.LegacySolution attribute), 72  
 y (cobra.core.solution.Solution attribute), 71, 72  
 y\_dict (cobra.core.solution.LegacySolution attribute), 72  
 y\_dict (cobra.core.solution.Solution attribute), 71, 72