cobamp Documentation

Release 0.0.1

Vítor Vieira

Dec 18, 2019

Contents:

CHAPTER 1

Installation

1.1 Basic requirements

- Python 3.x
- CPLEX along with its Python wrapper installed in your current distribution (please note that the Python version must be compatible with CPLEX)

1.2 Optional requirements

For easier model loading and analysis (using constraint-based methods), the following libraries can be used:

- cobrapy
- framed

Additionally, the escher library can be used to display elementary flux modes on metabolic maps and the networkx library can also be used to plot trees generated from EFM or MCS enumeration.

1.3 Via pip

The easiest method is to use pip to install the package from PyPI:

```
pip install cobamp
```

1.4 From source

- Download the latest source files from github
- Unpack the source files into a directory of your choosing

- Open the operating system's command-line interface
- Change into the source file directory
- Run the following command

```
python setup.py install
```

It is highly recommended that this package along with its requirements are installed in a separate Python environment. Tools such as virtualenv or conda can be used to create Python environments.

CHAPTER 2

Quick start

This notebook provides an example on how to use *cobamp* to easily calculate elementary flux modes (EFMs) and minimal cut sets (MCSs) coupled with the model reading and flux analysis capabilities of the *COBRApy* package.

2.1 Model reading and setup

Download the model in SBML format from the BiGG models repository. In this example we use the central carbon metabolic model of *Escherichia coli*, downloaded and stored in a temporary folder/path stored as *model_path*.

[1]:
```
import urllib

model_url = "http://bigg.ucsd.edu/static/models/e_coli_core.xml"
model_path, model_content = urllib.request.urlretrieve(model_url)
```

Read the SBML temporary file generated from the HTTP request content and generate a COBRA Model object from it. Two functions are defined to assist in further analyses: * *restore_bounds()* - restores all reaction bounds to their original values * *apply_kos(**kos*)* - knockout reactions contained in *kos* (any iterable containing reaction IDs)

[2]:
```
import cobra

model = cobra.io.sbml.read_sbml_model(model_path)
model_bounds = {r.id:(r.lower_bound, r.upper_bound) for r in model.reactions}

# >>> model_bounds
# ... {'ACALD': (-1000.0, 1000.0),
# ... 'ACALDt': (-1000.0, 1000.0),
# ... 'ACKr': (-1000.0, 1000.0),
# ... 'ACONTa': (-1000.0, 1000.0),
# ... 'ACONTb': (-1000.0, 1000.0), ...

def restore_bounds():
    for r in model.reactions:
        r.lower_bound, r.upper_bound = model_bounds[r.id]
```

(continues on next page)

(continued from previous page)

```
def apply_kos(kos):
    for ko in kos:
        model.reactions.get_by_id(ko).knock_out()

model
```

[2]:
```
<Model e_coli_core at 0x7f6bbab80588>
```

Extracting environmental conditions from this particular model, assuming the maximum capacity constant N ($N = 1000$). This assumes the media is defined when a lower or upper bound is different than 0, N or $-N$.

This will be needed some steps further for the MCSEnumerator approach.

[3]:
```
env_conditions = {r.id:(r.lower_bound, r.upper_bound)
 for r in model.reactions
 if (r.lower_bound not in [-1000, 0]) or (r.upper_bound not in [0, 1000])}

env_conditions
```

[3]:
```
{'ATPM': (8.39, 1000.0), 'EX_glc__D_e': (-10.0, 1000.0)}
```

2.2 Enumerating minimal cut sets

Import the MCSEnumerator wrapper class (*KShortestMCSEnumeratorWrapper*) that works with COBRA models. *KShortestMCSEnumeratorWrapper* provides a simple interface to enumerate minimal cut sets (MCSs) using the K-shortest algorithm (commonly referred to as the MCSEnumerator approach).

MCSs are sets of reactions that disable a certain phenotype in a model. This phenotype is defined as a set of flux or yield constraints. In this example, we will enumerate some MCSs.

[4]:
```
from cobamp.wrappers import KShortestMCSEnumeratorWrapper
```

Define the region of the flux space that the MCSs are supposed to block. In this scenario, we are attempting to improve succinate production using glucose as a carbon source. The following reaction IDs are required: * Succinate exchange : 'EX_succ_e' * Glucose exchange: 'EX_glc__D_e' * Maintenance ATP: 'ATPM'

As such, we wish to block low production phenotypes by adding a lower bound on the yield between succinate and glucose. Since the latter is only consumed, we assume its flux value is always negative. Thus, we want yields to be as low as possible, hence the lower bound definition.

This yield constraint can be loosely represented as the following:

$Y\frac{succinate}{glucose} > -0.001$

MCSs will block this phenotype and thus, we will obtain negative yields.

[5]:
```
yield_space = {
   #('product_id'),('substrate_id'):(lower_bound, upper_bound, deviation - set to␣
↪zero for most cases)
    ('EX_succ_e','EX_glc__D_e'):(-0.001, None, 0),
}

flux_space = {
   # flux id     : (lower_bound, upper_bound)
    'EX_glc__D_e': (-10, None),
```

(continues on next page)

(continued from previous page)

```
    'ATPM' : (8.39, None)
}
```

Having defined the flux space, we can now start the algorithm. The wrapper for MCSEnumerator requires the following parameters: * *model*: A model instance from *COBRA*, *framed* (or any other framework if a *ModelReader* subclass is implemented for it and added to the list of readers) * *target_flux_space_dict*: A dictionary (str -> tuple(float, float)) mapping reaction identifiers to flux bounds representing the target flux space for minimal cut sets. * *target_yield_space_dict*: A dictionary ((str, str) -> tuple(float, float, float)) mapping product/substrate pairs to a lower and upper bound for the yield constraint between both fluxes and a deviation value for the constraint. * *algorithm_type*: Can be one of two values: * *KShortestMCSEnumeratorWrapper.ALGORITHM_TYPE_POPULATE*: Enumerate one size at a time * *KShortestMCSEnumeratorWrapper.ALGORITHM_TYPE_ITERATIVE*: Enumerate one solution at a time * *stop_criteria*: An integer defining the maximum number of iterations. If the *POPULATE* routine is used, the algorithm will yield all solutions that have size *stop_criteria* and below. On the other hand, if *ITERATIVE* is used, the algorithm will yield the *stop_criteria*-shortest solutions. * *solver*: A string that specifies the solver to be used

It is worth noting that CPLEX is currently the best alternative for elementary mode enumeration due to some limitation on both the solvers and the *optlang* framework.

With CPLEX, *cobamp* uses indicator constraints, while other solvers (GLPK and GUROBI) rely on Big-M formulations. The latter requires a parameter (M), which is to be set at a very large value. However, higher values of M result in numerical instability which negatively affects the solutions yielded by the optimizer.

Indicator constraints can be used with GUROBI, although *optlang* does not currently support their addition. If these features are implemented in *optlang*, *cobamp* will also support them.

[6]:
```
ksefm = KShortestMCSEnumeratorWrapper(
    model=model,
    target_flux_space_dict=flux_space,
    target_yield_space_dict=yield_space,
    algorithm_type=KShortestMCSEnumeratorWrapper.ALGORITHM_TYPE_POPULATE,
    stop_criteria=5,
    solver='CPLEX') # or 'GUROBI', 'GLPK'
```

The MILP problems aren't actually run once the KShortestMCSEnumeratorWrapper instance is created. To obtain the solutions, the *get_enumerator()* method must be called. This method returns a generator that can be called in several ways.

[7]:
```
enumerator = ksefm.get_enumerator()
```

Depending on time constraints, one can interactively call the *next* function on the enumerator so it iterates just once. This is useful when the problem is large or when exploring new constraints.

[8]:
```
solutions = [sol for sol in enumerator]
```

Solutions are always lists of dictionaries mapping fluxes with values for all active fluxes. For MCSs, one simply has to obtain the dictionary keys for all dictionaries in the list.

[9]:
```
solutions[0]
```

[9]:
```
[{'ENO': 1548.6591179977859},
 {'NADH16': 357.40166865316615},
 {'GAPD': 1548.5399284864625},
 {'PGK': -1548.5399284864627},
 {'PGM': -1548.6591179977856},
 {'GLCpts': 2.324195470798569},
 {'EX_glc__D_e': -2.0858164481525625},
```

(continues on next page)

(continued from previous page)

```
 {'EX_succ_e': 2084.743742550656},
 {'EX_h2o_e': 1131.6895113231285},
 {'H2Ot': -1131.6895113231285},
 {'SUCCt3': 2084.743742550656},
 {'EX_h_e': 1042.3718712753266},
 {'ATPM': 1.0}]
```

[10]:
```
mcs = [set(d.keys()) for d in solutions[0]]
mcs
```

[10]:
```
[{'ENO'},
 {'NADH16'},
 {'GAPD'},
 {'PGK'},
 {'PGM'},
 {'GLCpts'},
 {'EX_glc__D_e'},
 {'EX_succ_e'},
 {'EX_h2o_e'},
 {'H2Ot'},
 {'SUCCt3'},
 {'EX_h_e'},
 {'ATPM'}]
```

Unlike the previous approach, we can try and extensively enumerate all MCSs. We must redefine the enumerator as it already iterated once. We can import the *chain* function and use it to get a list of solutions until the stop criteria is reached.

[11]:
```
enumerator = ksefm.get_enumerator()
```

[12]:
```
from itertools import chain
solutions = list(chain(*ksefm.get_enumerator()))
```

[13]:
```
mcs = [set(e.keys()) for e in solutions]
mcs[1:20]
```

[13]:
```
[{'NADH16'},
 {'GAPD'},
 {'PGK'},
 {'PGM'},
 {'GLCpts'},
 {'EX_glc__D_e'},
 {'EX_succ_e'},
 {'EX_h2o_e'},
 {'H2Ot'},
 {'SUCCt3'},
 {'EX_h_e'},
 {'ATPM'},
 {'NADTRHD', 'PFK'},
 {'EX_co2_e', 'TPI'},
 {'G6PDH2r', 'PGI'},
 {'CO2t', 'TPI'},
 {'PGI', 'PGL'},
 {'CYTBD', 'FBA'},
 {'PGI', 'RPE'}]
```

2.3 Enumerating elementary flux modes

We can import the *KShortestEFMEnumeratorWrapper* to enumerate elementary flux modes (EFMs) using the K-shortest EFM algorithm.

EFMs can be considered as the smallest functional units in a metabolic model that can achieve a certain function and are useful in finding novel metabolic functions.

[14]:
```
from cobamp.wrappers import KShortestEFMEnumeratorWrapper
```

For this example, we will enumerate EFMs in the model we have previously loaded. Similarly to the example above, we intend to produce succinate with this organism. For that purpose, we will try to enumerate the smallest EFMs that allow the conversion of glucose into succinate and biomass (to ensure growth).

To specify these metabolites, they have to be present in the model. First, we will create a metabolite to represent cell growth. We then add a reaction to represent the fictional secretion of this metabolite. Both components are added to a copy of the model to preserve the original version.

[15]:
```
biomass_c = cobra.Metabolite(compartment='c', id='biomass_c', name='Biomass')
biomass_production = cobra.Reaction(id='EX_biomass_e', name='Biomass production',␣
↪lower_bound=0, upper_bound=1000)

model_biomass = model.copy()
model_biomass.add_metabolites([biomass_c])
```

The biomass metabolite is added to the biomass reaction already present in the model as a product. On the new reaction, the same metabolite is added as a reactant, effectively creating a drain on the model.

[16]:
```
model_biomass.reactions.BIOMASS_Ecoli_core_w_GAM.add_metabolites({biomass_c: 1})
biomass_production.add_metabolites({biomass_c: -1})
```

The K-shortest enumeration algorithm is then instantiated using mostly the same arguments as in the example above, with the addition of: * *consumed* : A list of metabolite IDs that are guaranteed to be consumed by the cell * *non_consumed*: A list of external metabolite IDs that cannot be consumed * *produced* : A list of external metabolite IDs that must be produced by the cell

These parameters are used to narrow the space of possible EFMs to avoid unwanted metabolic functions and speed up the search.

[17]:
```
ksefm = KShortestEFMEnumeratorWrapper(
    model=model_biomass,
    non_consumed=[],
    consumed=['glc__D_e'],
    produced=['biomass_c','succ_e'],
    algorithm_type=KShortestEFMEnumeratorWrapper.ALGORITHM_TYPE_POPULATE,
    stop_criteria=100,
    solver='CPLEX'
)
```

Having set a limit of 100 for the size of the EFMs, we will now attempt to enumerate them. In this example we will only obtain with the smallest. Once the algorithm reaches first iteration that yields at least one EFM, we will no longer call the *enumerator* instance.

[18]:
```
enumerator = ksefm.get_enumerator()

efm_list = []
```

(continues on next page)

(continued from previous page)

```
while len(efm_list) == 0:
    efm_list += next(enumerator)
```

The 17 smallest EFMs include 42 reactions

[19]:
```
len(efm_list)
```

[19]:
```
17
```

2.4 Analysis of elementary flux modes

2.4.1 Inspecting flux coefficients

The EFMs can be displayed as a set of reactions with coefficients. Please note that the drains that correspond with the external metabolites defined as being consumed or produced will not show up as active reactions, even though they are part of the EFM. This occurs since the mass balance equation for these metabolites is changed from $\sum_{j=1}^{n} S_{ij}v = 0$ to $\sum_{j=1}^{n} S_{ij}v > 1$ or $\sum_{j=1}^{n} S_{ij}v < -1$, assuming v as the flux vector.

The values for these fluxes can be calculated by replacing v with the flux values from the EFM and obtaining the result from the left hand side of the inequation

[20]:
```
efm_list[0]
```

[20]:
```
{'ACONTa': 509.7485477179139,
 'ACONTb': 509.7485477179139,
 'AKGDH': 491.84149377600266,
 'BIOMASS_Ecoli_core_w_GAM': 16.597510373446326,
 'CO2t': -1034.139419087282,
 'CS': 509.7485477179139,
 'CYTBD': 2521.7435684650845,
 'ENO': 1166.9892116184212,
 'EX_co2_e': 1034.139419087282,
 'EX_h2o_e': 1646.8838174276166,
 'EX_h_e': 1316.6290456433399,
 'EX_nh4_e': -90.50290456432813,
 'EX_o2_e': -1260.8717842325423,
 'EX_pi_e': -61.05726141079617,
 'FBA': 601.4605809129475,
 'GAPD': 1191.8190871370969,
 'GLCpts': 617.9701244814147,
 'GLNS': 90.50290456432813,
 'GLUSy': 86.2589211618379,
 'H2Ot': -1646.8838174276166,
 'ICDHyr': 509.7485477179139,
 'NADH16': 2521.7435684650845,
 'NADTRHD': 5576.853941909496,
 'NH4t': 90.50290456432813,
 'O2t': 1260.8717842325423,
 'PDH': 571.952697095516,
 'PFK': 601.4605809129475,
 'PGI': 614.5676348548582,
 'PGK': -1191.8190871370969,
 'PGM': -1166.9892116184212,
 'PIt2r': 61.05726141079617,
```

(continues on next page)

(continued from previous page)

```
 'PPC': 539.4033195021505,
 'PYK': 1.0,
 'RPE': -11.93029045643322,
 'RPI': -11.93029045643322,
 'SUCCt3': 491.84149377600266,
 'SUCOAS': -491.84149377600266,
 'TALA': -2.969294605809549,
 'THD2': 5369.595020747642,
 'TKT1': -2.969294605809549,
 'TKT2': -8.96099585062367,
 'TPI': 601.4605809129475}
```

[21]:
```
metabolite = model_biomass.metabolites.biomass_c
cur_efm = efm_list[0]
{r:cur_efm[r.id] for r in metabolite.reactions if r.id in cur_efm}
```

[21]:
```
{<Reaction BIOMASS_Ecoli_core_w_GAM at 0x7f6b7bff8780>: 16.597510373446326}
```

2.4.2 Visualizing EFMs in metabolic maps

Alternatively, external packages can be used to provide a more interactive result. The *escher* package (https://escher.github.io/), once installed in your Python distribution, can be used to display metabolic maps. The snippet below shows how the EFMs from *cobamp* can be used directly as inputs for *escher* maps.

[22]:
```
import escher

escher_builder = escher.Builder(
    map_name='e_coli_core.Core metabolism', # map that matches our model
    reaction_scale=[{'type': 'min', 'color': '#0000ff', 'size': 10}, # make active␣
↪fluxes more visible
                {'type': 'mean', 'color': '#551a8b', 'size': 20},
                {'type': 'max', 'color': '#ff0000', 'size': 40}],

    hide_secondary_metabolites = True,
    reaction_data = efm_list[0], # provides flux values from the first EFM

)
escher_builder.display_in_notebook(js_source='web')
```

[22]:
```
<IPython.core.display.HTML object>
```

2.4.3 Representing multiple EFMs as trees

Additionally, we can also display EFMs as ordered trees where each path from the root to a leaf is a single EFM. This allows us to observe patterns among large sets of EFMs to identify common mechanisms between them

The *analysis.tree* module contains some functions to achieve this. Please note that this functionality is still highly experimental

[23]:
```
from cobamp.analysis.graph import *
```

The function below provides an example workflow to organize EFMs (or MCSs) as trees

Firstly, the root node is created and the tree is populated with the EFMs. This procedure is performed by iteratively adding one node representing the most common reaction within a the set of EFMs. Each of these nodes is then recursively populated using the subset of EFMs containing reactions from the nodes above it.

The tree is then compressed by merging linear sequences of nodes into a single node whose value is a list of reactions. This list is then converted to a concatenated string for easier reading.

If needed, very large nodes can be ignored, with their value set to an arbitrary value. Additionally the tree can be pruned below a supplied depth value, replacing all nodes below it by the relative frequency of each reaction in the set of EFMs represented under these new leaves.

[24]:
```
def generate_efm_results_tree(efm_sets, ignore_greater_than=10, pruning_level=6,␣
↪merge_dupes=False):
    root = Tree('ROOT')
    # fill the tree with EFMs
    fill_tree(root, efm_sets)
    # compress sequences of nodes with only 1 inward and 1 outward edge
    compress_linear_paths(root)
    if ignore_greater_than:
        ignore_compressed_nodes_by_size(root, ignore_greater_than) # if a node has␣
↪too many fluxes, ignore it

    # apply a function to all nodes in this example, the node value
    # is converted to a string if it is currently a list of strings
    # this is the case of compressed nodes
    apply_fx_to_all_node_values(root, lambda x: ' + '.join(sorted(x)) if isinstance(x,
↪ list) else x if x is not None else "None")

    # if the tree is too deep, a depth value can be specified
    # so the tree is cut past that point
    # the leaf that replaces it will specify the relative
    # frequency of the reactions in all EFMs represented by it
    if pruning_level:
        probabilistic_tree_prune(root, target_level=pruning_level, cut_leaves=False,␣
↪name_separator='+')

    # compress the tree again
    compress_linear_paths(root)

    # merge duplicate nodes. this effectively replaces all
    # objects of the same value with a single object
    if merge_dupes:
        merge_duplicate_nodes(root)
    return root
```

The tree is then generated using this function, returning the populated *tree* object

[25]:
```
tree = generate_efm_results_tree(
    efm_sets=efm_list, # list of elementary modes (list[dict[str,float]])
    ignore_greater_than=None, # ignore nodes with more than *ignore_greater_than*␣
↪elements
    pruning_level=None, # prune tree from the nodes past depth=*pruning_level*
    merge_dupes=False # do not merge duplicate nodes
)
```

A simple print function is supplied to quickly view the tree.

[26]:
```
print(pretty_print_tree(tree))
```

```
|-- ACONTa + ACONTb + AKGDH + BIOMASS_Ecoli_core_w_GAM + CO2t + CS + CYTBD + ENO + EX_
↪co2_e + EX_h2o_e + EX_h_e + EX_nh4_e + EX_o2_e + EX_pi_e + FBA + GAPD + GLCpts +␣
↪GLNS + H2Ot + ICDHyr + NADH16 + NH4t + O2t + PDH + PFK + PGI + PGK + PGM + PIt2r +␣
↪PPC + PYK + ROOT + RPE + RPI + SUCCt3 + SUCOAS + TALA + TKT1 + TKT2 + TPI(None)
|       |-- ATPS4r(13)
|       |       |-- GLUSy(7)
|       |       |       |-- GLUDy(1)
|       |       |       |-- SUCCt2_2(1)
|       |       |       |-- FBP(1)
|       |       |       |-- ATPM(1)
|       |       |       |-- GLUN(1)
|       |       |       |-- THD2(1)
|       |       |       |-- PPCK(1)
|       |       |-- GLUDy(6)
|       |       |       |-- SUCCt2_2(1)
|       |       |       |-- FBP(1)
|       |       |       |-- ATPM(1)
|       |       |       |-- GLUN(1)
|       |       |       |-- THD2(1)
|       |       |       |-- PPCK(1)
|       |-- NADTRHD(4)
|       |       |-- GLUSy(2)
|       |       |       |-- SUCCt2_2(1)
|       |       |       |-- THD2(1)
|       |       |-- GLUDy(2)
|       |       |       |-- SUCCt2_2(1)
|       |       |       |-- THD2(1)
```

Another way to draw this would be to used the NetworkX (https://networkx.github.io/) with GraphViz (https://www.graphviz.org/) installed.

[27]:
```
%matplotlib inline

tree = generate_efm_results_tree(
    efm_sets=efm_list, # list of elementary modes (list[dict[str,float]])
    ignore_greater_than=10, # ignore nodes with more than *ignore_greater_than*␣
↪elements
    pruning_level=10, # prune tree from the nodes past depth=*pruning_level*
    merge_dupes=True # do not merge duplicate nodes
)

old_value = tree.value # save the old value of the root node
print('Old root string:', tree.value)
tree.value = "Glycolysis + TCA" # change the root node value to a shorter string

import networkx as nx
import matplotlib.pyplot as plt

G = nx.DiGraph() # define a new networkx Graph object
populate_nx_graph(tree, G, unique_nodes=True) # populate it with data from the Tree␣
↪object

pos = nx.nx_pydot.graphviz_layout(G) # generate the graphviz layout

# create a figure and plot the nodes, edges and labels
plt.figure(figsize=(20,10))
nx.draw_networkx_nodes(G, pos, node_size=10)
```

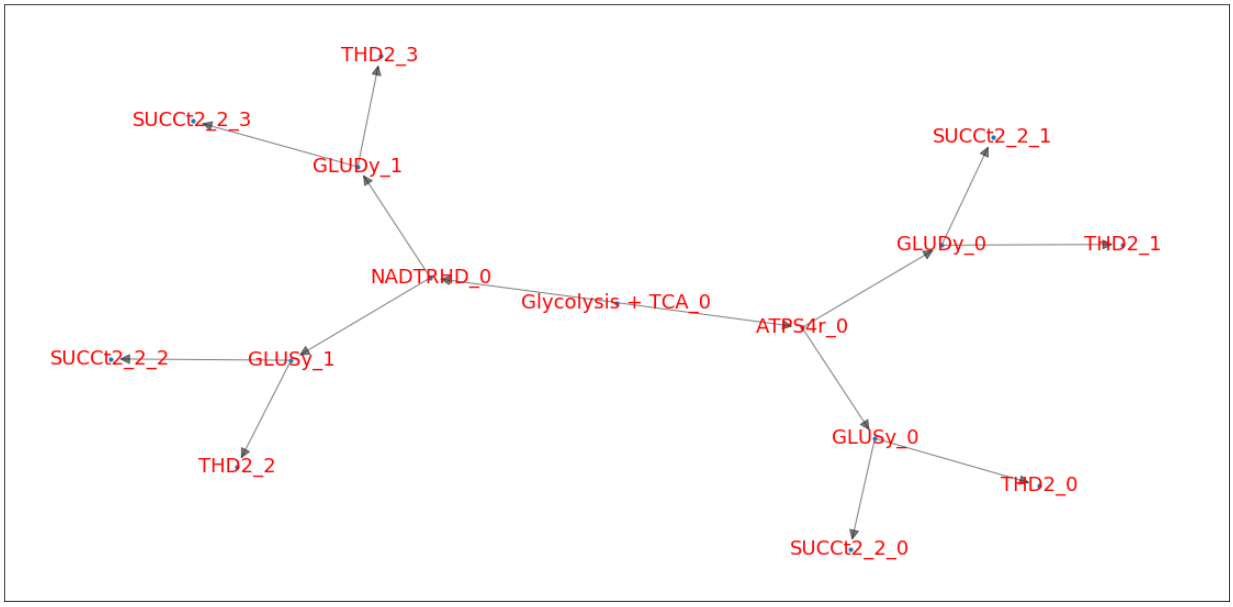
(continues on next page)

(continued from previous page)

```
nx.draw_networkx_edges(G, pos, alpha=0.5, arrowsize=20)
nx.draw_networkx_labels(G, pos, font_size=18, font_color='red')

plt.show()
```

```
Old root string: ACONTa + ACONTb + AKGDH + BIOMASS_Ecoli_core_w_GAM + CO2t + CS +␣
↪CYTBD + ENO + EX_co2_e + EX_h2o_e + EX_h_e + EX_nh4_e + EX_o2_e + EX_pi_e + FBA +␣
↪GAPD + GLCpts + GLNS + H2Ot + ICDHyr + NADH16 + NH4t + O2t + PDH + PFK + PGI + PGK␣
↪+ PGM + PIt2r + PPC + PYK + ROOT + RPE + RPI + SUCCt3 + SUCOAS + TALA + TKT1 + TKT2␣
↪+ TPI
```



2.4.4 Frequency analysis of multiple sets of EFMs

Heatmaps can be used as graphical representations of matrices. If multiple sets of EFMs are enumerated using different conditions, they can be compared in terms of the frequency at which their active reactions are displayed. *cobamp* provides a simple analysis function to generate a *pandas* dataframe containing reaction frequencies for one or more sets of EFMs.

A dictionary mapping a group identifier (as string) and a set of reaction indices from the EFMs is required as input.

Firstly, the heatmap and frequency dataframe generating functions must be imported as shown below

[28]:
```
from cobamp.analysis.plotting import display_heatmap
from cobamp.analysis.frequency import get_frequency_dataframe
```

As an example, one can try to enumerate the 50 smallest EFMs for consuming glucose and producing a wide array of interesting compounds as well as ensuring cell growth through the production of biomass. In this test case, we will try to get at least 50 EFMs to produce: * Succinate (succ_e) * D-Lactate (lac__D_e) * Glutamine (gln__L_e) * Glutamate (glu__L_e) * Ethanol (etoh_e)

Some functions should be defined to achieve this specific purpose. First, we can generalize the EFM enumerators that were previously used to accept any product. The function *get_efm_enumerator_for_product(**model*, **product**)* returns an EFM enumerator for the **model** while forcing **product** to be secreted.

[29]:
```
def get_efm_enumerator_for_product(model, product):
    ksefm = KShortestEFMEnumeratorWrapper(
        model=model,
        non_consumed=[],
        consumed=['glc__D_e'],
        produced=['biomass_c', product],
        algorithm_type=KShortestEFMEnumeratorWrapper.ALGORITHM_TYPE_POPULATE,
        stop_criteria=100,
            solver='CPLEX'
    )

    return ksefm.get_enumerator()
```

Another function can also be created to obtain at least 50 EFMs and convert them to sets of reactions. This function, *get_production_efms(**model*, **product**)**, calls the function defined above to obtain the enumerator and enumerates EFM up to size 100 but stops if 50 EFMs have been added.

[30]:
```
def get_production_efms(model, product):
    enumerator = get_efm_enumerator_for_product(model, product)
    efm_list = []
    stopped = False

    while len(efm_list) < 50 and not stopped:
        try:
            efm_list += next(enumerator)
        except StopIteration as si:
            stopped = True
            print('Stopping criteria reached!')

    return [set(efm.keys()) for efm in efm_list]
```

Finally, we can define the desired products and enumerate the EFMs for each. A dictionary mapping products and EFMs is the preferred format for storing and analysing these EFMs.

[31]:
```
products = ['succ_e','lac__D_e','gln__L_e','glu__L_e','etoh_e']

production_efms = {product:get_production_efms(model_biomass, product) for product in␣
↪products}
```

```
Stopping criteria reached!
```

The enumerated EFMs for each product can be summarized in terms on reaction frequencies for each product. The function *get_frequency_dataframe(**efm_groups*)** from the *frequency* submodule accepts the dictionaries created above and generates a dataframe. Further preprocessing using other methods can enhance this analysis.

[32]:
```
efm_df = get_frequency_dataframe(production_efms) # obtain the dataframe
efm_df = efm_df.loc[:,~ efm_df.isna().all()].fillna(0).astype(int) # remove entirely␣
↪null columns (with no EFMs)

# convert absolute to relative frequency in percentage values
efm_df = efm_df * 100 / [len(production_efms[col]) for col in efm_df.columns]
efm_df
```

[32]:

	etoh_e	glu__L_e	lac__D_e	succ_e
ACALD	100.000000	3.448276	1.481481	0.000000
ACALDt	0.523560	3.448276	1.481481	0.000000
ACONTa	100.000000	100.000000	100.000000	100.000000
ACONTb	100.000000	100.000000	100.000000	100.000000

(continues on next page)

(continued from previous page)

ADK1	6.806283	29.655172	9.629630	6.060606
AKGDH	0.000000	0.689655	0.000000	36.363636
AKGt2r	13.612565	25.517241	28.148148	0.000000
ALCD2x	100.000000	0.000000	0.000000	0.000000
ATPM	7.329843	6.206897	4.444444	7.272727
ATPS4r	49.214660	64.827586	56.296296	63.030303
BIOMASS_Ecoli_core_w_GAM	100.000000	100.000000	100.000000	100.000000
CO2t	73.821990	100.000000	100.000000	80.000000
CS	100.000000	100.000000	100.000000	100.000000
CYTBD	57.591623	100.000000	92.592593	91.515152
D_LACt2	1.047120	11.724138	100.000000	6.060606
ENO	100.000000	100.000000	100.000000	100.000000
ETOHt2r	100.000000	0.000000	0.000000	0.000000
EX_acald_e	0.523560	3.448276	1.481481	0.000000
EX_akg_e	13.612565	25.517241	28.148148	0.000000
EX_co2_e	73.821990	100.000000	100.000000	80.000000
EX_glu__L_e	2.617801	0.000000	12.592593	0.606061
EX_h2o_e	100.000000	100.000000	100.000000	100.000000
EX_h_e	100.000000	100.000000	100.000000	100.000000
EX_lac__D_e	1.047120	11.724138	0.000000	6.060606
EX_nh4_e	100.000000	100.000000	100.000000	100.000000
EX_o2_e	57.591623	100.000000	92.592593	91.515152
EX_pi_e	100.000000	100.000000	100.000000	100.000000
EX_pyr_e	26.701571	20.689655	9.629630	23.636364
EX_succ_e	42.408377	0.689655	7.407407	0.000000
FBA	93.193717	85.517241	90.370370	97.575758
...	...	...	...	...
MALS	0.000000	0.000000	0.000000	35.151515
MDH	62.303665	40.689655	45.185185	69.090909
ME1	21.465969	15.172414	19.259259	27.272727
ME2	37.172775	26.896552	34.814815	52.121212
NADH16	100.000000	100.000000	100.000000	100.000000
NADTRHD	30.366492	35.172414	28.148148	24.242424
NH4t	100.000000	100.000000	100.000000	100.000000
O2t	57.591623	100.000000	92.592593	91.515152
PDH	100.000000	100.000000	100.000000	100.000000
PFK	93.193717	86.896552	90.370370	97.575758
PGI	97.905759	95.172414	97.037037	100.000000
PGK	100.000000	100.000000	100.000000	100.000000
PGL	34.031414	46.896552	49.629630	2.424242
PGM	100.000000	100.000000	100.000000	100.000000
PIt2r	100.000000	100.000000	100.000000	100.000000
PPC	100.000000	100.000000	100.000000	100.000000
PPCK	7.329843	6.206897	4.444444	7.272727
PPS	6.806283	29.655172	9.629630	6.060606
PYK	39.790576	22.068966	43.703704	40.606061
PYRt2	26.701571	20.689655	9.629630	23.636364
RPE	97.905759	95.862069	98.518519	100.000000
RPI	100.000000	100.000000	100.000000	100.000000
SUCCt2_2	18.848168	11.724138	2.962963	29.696970
SUCCt3	50.785340	11.724138	10.370370	100.000000
SUCOAS	0.000000	0.689655	0.000000	36.363636
TALA	82.198953	86.206897	71.851852	100.000000
THD2	51.832461	44.137931	42.222222	37.575758
TKT1	82.198953	86.206897	71.851852	100.000000
TKT2	97.905759	95.862069	98.518519	100.000000
TPI	93.193717	85.517241	90.370370	97.575758

(continues on next page)

(continued from previous page)

```
[76 rows x 4 columns]
```

For the purpose of this tutorial, we are only interested in the differences between EFMs for different products and thus, we will only consider rows (reactions) whose frequency is variable amongst products.

In this example, we identify the 20 reactions with highest variance and subset the dataframe to be represented as a heatmap.

[33]:
```
variance = efm_df.T.apply(lambda x: x.var())
high_variance = variance.sort_values(ascending= False).iloc[:20]
display_heatmap(efm_df.loc[high_variance.index,:])
```


etoh_e
glu__L_e
lac__D_e
succ_e
ALCD2x 100.0% 0.0% 0.0% 0.0%
ETOHt2r 100.0% 0.0% 0.0% 0.0%
ACALD 100.0% 3.4% 1.5% 0.0%
GLUt2r 2.6% 100.0% 12.6% 0.6%
LDH_D 1.0% 11.7% 100.0% 6.1%
D_LACt2 1.0% 11.7% 100.0% 6.1%
SUCCt3 50.8% 11.7% 10.4% 100.0%
PGL 34.0% 46.9% 49.6% 2.4%
GND 34.0% 46.9% 49.6% 2.4%
G6PDH2r 34.0% 46.9% 49.6% 2.4%
EX_succ_e 42.4% 0.7% 7.4% 0.0%
FRD7 42.4% 0.0% 7.4% 28.5%
FUM 42.4% 0.0% 7.4% 28.5%
O2t 57.6% 100.0% 92.6% 91.5%
CYTBD 57.6% 100.0% 92.6% 91.5%
EX_o2_e 57.6% 100.0% 92.6% 91.5%
AKGDH 0.0% 0.7% 0.0% 36.4%
SUCOAS 0.0% 0.7% 0.0% 36.4%
ICL 0.0% 0.0% 0.0% 35.2%
MALS 0.0% 0.0% 0.0% 35.2%
100
80
60
40
20
0
reaction frequency [%]

2.5 Constraint-based analysis of mutant phenotypes

With MCSs stored as dictionaries, we can easily use COBRA to analyse these knockouts. In this snippet, a dictionary is created to store various simulation values (Flux Balance/Variability Analysis) obtained after applying the reaction knockouts suggested by the MCSs. These are: * *product_minimum* and *product_maximum* - Flux range for succinate production * *max_growth* - Maximum growth rate * *product_at_max_growth* - Predicted production rate at maximum growth * *glucose_consumption* - Glucose consumption rate at maximum growth

[34]:
```
def analyse(kos):
    apply_kos(kos)
    try:
        # Flux Balance Analysis
        sol = model.optimize()

        biomass = sol.fluxes.BIOMASS_Ecoli_core_w_GAM
        product = sol.fluxes.EX_succ_e
        substrate = sol.fluxes.EX_glc__D_e

        # Flux Variability Analysis
        fva = cobra.flux_analysis.variability.flux_variability_analysis(
            model, reaction_list=['EX_succ_e']).loc['EX_succ_e',:]

        return {
        'product_maximum': fva['maximum'],
        'product_minimum': fva['minimum'],
        'max_growth': biomass,
        'product_at_max_growth':product,
        'glucose_consumption': substrate
        }

    except:
        pass
    finally:
        # restore reaction bounds to their original state
        # so the next iteration applies knockouts over them
        restore_bounds()
    return
```

pandas can be used to store these values as a dataframe for easier manipulation and analysis

[35]:
```
import pandas as pd

df = pd.DataFrame({frozenset(ko): analyse(ko) for ko in mcs}).T
```

```
cobra/util/solver.py:416 UserWarning: solver status is 'infeasible'
```

Solutions with null or very low growth, as well as solutions with very low production rates are filtered out. This way, only the best candidates are displayed

[36]:
```
null_growth = df.max_growth.isnull() # growth is None - due to infeasibility
zero_growth = df.max_growth <= 1e-10 # growth is too low (near 0)
no_product = df.product_at_max_growth <= 1e-5 # no production at maximum biomass

df.index.names = ['knockouts']

df[~ (zero_growth | null_growth | no_product)].sort_values(by='max_growth',␣
↪ascending=False).iloc[1:20,:]
```

[36]:

knockouts	glucose_consumption	max_growth	product_at_max_growth	product_maximum	product_minimum
(PPS, LDH_D, ACALD, CYTBD, PYRt2)	-10	0.110886	9.09864	9.09864	9.09864
(PPS, D_LACt2, PYRt2, CYTBD, ACALD)	-10	0.110886	9.09864	9.09864	9.09864
(PYRt2, PPS, EX_lac__D_e, CYTBD, ACALD)	-10	0.110886	9.09864	9.09864	9.09864
(PYRt2, PPS, EX_lac__D_e, O2t, ACALD)	-10	0.110886	9.09864	9.09864	9.09864
(PYRt2, ADK1, EX_lac__D_e, EX_o2_e, ACALD)	-10	0.110886	9.09864	9.09864	9.09864
(PYRt2, ADK1, EX_lac__D_e, O2t, ACALD)	-10	0.110886	9.09864	9.09864	9.09864
(O2t, LDH_D, ADK1, ACALD, EX_pyr_e)	-10	0.110886	9.09864	9.09864	9.09864
(LDH_D, ADK1, ACALD, EX_o2_e, EX_pyr_e)	-10	0.110886	9.09864	9.09864	9.09864
(PYRt2, LDH_D, ADK1, ACALD, EX_o2_e)	-10	0.110886	9.09864	9.09864	9.09864
(O2t, LDH_D, ADK1, ACALD, PYRt2)	-10	0.110886	9.09864	9.09864	9.09864
(EX_pyr_e, ADK1, EX_lac__D_e, EX_o2_e, ACALD)	-10	0.110886	9.09864	9.09864	9.09864
(O2t, ADK1, ACALD, D_LACt2, EX_pyr_e)	-10	0.110886	9.09864	9.09864	9.09864
(ADK1, ACALD, D_LACt2, EX_o2_e, EX_pyr_e)	-10	0.110886	9.09864	9.09864	9.09864
(EX_pyr_e, O2t, ADK1, EX_lac__D_e, ACALD)	-10	0.110886	9.09864	9.09864	9.09864
(PYRt2, ADK1, ACALD, D_LACt2, EX_o2_e)	-10	0.110886	9.09864	9.09864	9.09864
(LDH_D, ADK1, ACALD, CYTBD, EX_pyr_e)	-10	0.110886	9.09864	9.09864	9.09864
(PPS, D_LACt2, ACALD, CYTBD, EX_pyr_e)	-10	0.110886	9.09864	9.09864	9.09864
(PPS, LDH_D, ACALD, CYTBD, EX_pyr_e)	-10	0.110886	9.09864	9.09864	9.09864
(EX_pyr_e, ADK1, EX_lac__D_e, CYTBD, ACALD)	-10	0.110886	9.09864	9.09864	9.09864

(continues on next page)

(continued from previous page)

```
(O2t, ADK1, ACALD, D_LACt2, EX_pyr_e)                9.09864        9.09864
(ADK1, ACALD, D_LACt2, EX_o2_e, EX_pyr_e)            9.09864        9.09864
(EX_pyr_e, O2t, ADK1, EX_lac__D_e, ACALD)            9.09864        9.09864
(PYRt2, ADK1, ACALD, D_LACt2, EX_o2_e)               9.09864        9.09864
(LDH_D, ADK1, ACALD, CYTBD, EX_pyr_e)                9.09864        9.09864
(PPS, D_LACt2, ACALD, CYTBD, EX_pyr_e)               9.09864        9.09864
(PPS, LDH_D, ACALD, CYTBD, EX_pyr_e)                 9.09864        9.09864
(EX_pyr_e, ADK1, EX_lac__D_e, CYTBD, ACALD)          9.09864        9.09864
```

[ ]:

CHAPTER 3

src

3.1 cobamp package

3.1.1 Subpackages

cobamp.algorithms package

Submodules

cobamp.algorithms.kshortest module

This module includes classes that implement the K-shortest EFM enumeration algorithms. Please refer to the original authors' paper describing the method [1]. Additional improvements concerning enumeration of EFMs by size have been adapted from the methods described by Von Kamp et al. [2]

References:

> [1] De Figueiredo, Luis F., et al. "Computing the shortest elementary flux modes in genome-scale metabolic networks" Bioinformatics 25.23 (2009): 3158-3165. [2] von Kamp, Axel, and Steffen Klamt. "Enumeration of smallest intervention strategies in genome-scale metabolic networks" PLoS computational biology 10.1 (2014): e1003378.

class cobamp.algorithms.kshortest.**AbstractConstraint**
Bases: object

Object representing a constraint to be used within the intervention problem structures provided in this package.

from_tuple()
Generates a constraint from a tuple. Refer to subclasses for each specific format.

materialize(*n*)
Generates a matrix T 1-by-n or 2-by-n and a list b of length 1 or 2 to be used for target flux vector definition within the intervention problem framework

Parameters:

n: Number of columns to include in the target matrix

Returns: Tuple with Iterable[ndarray] and list of float/int

class cobamp.algorithms.kshortest.**DefaultFluxbound**(*lb*, *ub*, *r_index*)

Bases: *cobamp.algorithms.kshortest.AbstractConstraint*

Class representing bounds for a single flux with a lower and an upper bound.

from_tuple()

materialize(*n*)

Generates a matrix T 1-by-n or 2-by-n and a list b of length 1 or 2 to be used for target flux vector definition within the intervention problem framework

Parameters:

n: Number of columns to include in the target matrix

Returns: Tuple with Iterable[ndarray] and list of float/int

class cobamp.algorithms.kshortest.**DefaultYieldbound**(*lb*, *ub*, *numerator_index*, *denominator_index*, *deviation=0*)

Bases: *cobamp.algorithms.kshortest.AbstractConstraint*

Class representing a constraint on a yield between two fluxes. Formally, this constraint can be represented as n - yd < b, assuming n and d as the numerator and denominator fluxes (yield(N,D) = N/D), y as the maximum yield and b as a deviation value.

from_tuple()

Returns a DefaultYieldbound instance from a tuple containing numerator and denominator indices, yield lower and upper bounds, a flag indicating whether it's an upper bound and a deviation (optional) ——-

materialize(*n*)

Generates a matrix T 1-by-n or 2-by-n and a list b of length 1 or 2 to be used for target flux vector definition within the intervention problem framework

Parameters:

n: Number of columns to include in the target matrix

Returns: Tuple with Iterable[ndarray] and list of float/int

class cobamp.algorithms.kshortest.**InterventionProblem**(*S*)

Bases: object

Class containing functions useful when defining an problem using the intervention problem framework. References:

[1] Hädicke, O., & Klamt, S. (2011). Computing complex metabolic intervention strategies using constrained minimal cut sets. Metabolic engineering, 13(2), 204-213.

generate_target_matrix(*constraints*)

constraints: An iterable containing valid constraints of

Returns a tuple (T,b) with two elements: T is numpy 2D array with as many rows specifying individual bounds (lower and upper bounds count as two) for each reaction.

b is a numpy 1D array with the right hand side of the T.v > b inequality. This represents the value of the bound.

class cobamp.algorithms.kshortest.**KShortestEFMAlgorithm**(*configuration*, *verbose=True*)

Bases: object

A higher level class to use the K-Shortest algorithms. This encompasses the standard routine for enumeration of EFMs. Requires a configuration defining an algorithms type. See <KShortestProperties>

enumerate(*linear_system*, *excluded_sets=None*, *forced_sets=None*)

Enumerates the elementary modes for a linear system

Parameters

linear_system: A KShortestCompatibleLinearSystem instance

excluded_sets: Iterable[Tuple[Solution/Tuple]] with solutions to exclude from the enumeration

forced_sets: Iterable[Tuple[Solution/Tuple]] with solutions to force

Returns a list with solutions encoding elementary flux modes.

get_enumerator(*linear_system*, *excluded_sets*, *forced_sets*)

Parameters

linear_system: A KShortestCompatibleLinearSystem instance

excluded_sets: Iterable[Tuple[Solution/Tuple]] with solutions to exclude from the enumeration

forced_sets: Iterable[Tuple[Solution/Tuple]] with solutions to force

Returns an iterator that yields one or multiple EFMs at each iteration, depending on the properties.

class cobamp.algorithms.kshortest.**KShortestEnumerator**(*linear_system*, *m_value=None*, *force_non_cancellation=True*, *is_efp_problem=False*, *n_threads=0*, *workmem=None*, *force_big_m=False*, *max_populate_sols=2100000000*, *max_time=0*)

Bases: object

Class implementing the k-shortest elementary flux mode algorithms. This is a lower level class implemented using the optlang solver framework.

ENUMERATION_METHOD_ITERATE = 'iterate'

ENUMERATION_METHOD_POPULATE = 'populate'

exclude_solutions(*sols*)

Excludes the supplied solutions from the search by adding them as integer cuts. Use at your own discretion as this will yield different EFMs than would be intended. This can also be used to exclude certain reactions from the search by supplying solutions with one reaction.

Parameters

sols: An Iterable containing list/tuples with active reaction combinations to exclude or Solution instances.

force_solutions(*sols*)

Forces a set of reactions encoded as solutions to appear in the subsequent elementary modes to be calculated.

Parameters

sols: An Iterable containing list/tuples with active reaction combinations to exclude or Solution instances.

get_model()
Returns the solver instance.

get_single_solution(*cut=True*, *allow_suboptimal=False*)
Returns a single solution. Use the solution_iterator method instead.

populate_current_size()
Returns the solutions for the current size. Use the population_iterator method instead.

population_iterator(*max_size*)
Generates a solution iterator that yields a list of solutions. Each next call returns all EFMs for a single size starting from 1 up to max_size.

Parameters

max_size: The maximum solution size.

Returns a list of KShortestSolution instances.

reset_enumerator_state()
Resets all integer cuts and size constraints.

set_indicator_activity(*forced_off=None*, *forced_on=None*)

set_objective_expression(*mask*)

set_size_constraint(*start_at*, *equal=False*)
Defines the size constraint for the K-shortest algorithms.

Parameters

start_at: Size from which the solutions will be obtained.

equal: Boolean indicating whether the solutions will match the size or can be higher than it.

solution_iterator(*maximum_amount=2147483647*)
Generates a solution iterator. Each next call will yield a single solution. This method should be used to allow flexibility when enumerating EFMs for large problems. Since it uses the optimize routine, this may be slower in the longer term.

class cobamp.algorithms.kshortest.**KShortestProperties**
Bases: *cobamp.utilities.property_management.PropertyDictionary*

Class defining a configuration for the K-shortest algorithms. The following fields are mandatory: K_SHORTEST_MPROPERTY_METHOD:

- K_SHORTEST_METHOD_ITERATE : Iterative enumeration (one EFM at a time)
- K_SHORTEST_METHOD_POPULATE : Enumeration by size (EFMs of a certain size at a time)

cobamp.algorithms.kshortest.**decompose_list**(*a*)

cobamp.algorithms.kshortest.**value_map_apply**(*single_fx*, *pair_fx*, *value_map*, ***kwargs*)
Applies functions to the elements of an ordered dictionary, using one of two functions that process, respectively, a single item or a tuple of items.

Functions must receive the dictionary key, the dictionary itself and optional arguments

Parameters

- **single_fx** – A function that receives a single object as argument
- **pair_fx** – A function that receives a tuple as argument
- **value_map** – An ordered dictionary mapping keys with values
- **kwargs** – Optional function arguments

Returns An iterable containing the results of the applied functions

Module contents

Module containing elementary flux mode enumeration methods

cobamp.analysis package

Submodules

cobamp.analysis.frequency module

cobamp.analysis.frequency.**get_frequency_dataframe**(*pathway_dict*, *k_min=1*, *k_max=1*)

cobamp.analysis.graph module

cobamp.analysis.graph.**apply_fx_to_all_node_values**(*tree*, *fx*)
Applies a function to all nodes below the tree, modifying their value to its result. Parameters ———- tree: A Tree instance fx: A function to apply ——-

cobamp.analysis.graph.**compress_linear_paths**(*tree*)
Collapses sequences of nodes contained in a Tree with only one children as a single node containing all values of those nodes. Parameters ———- tree: A Tree instance. ——-

cobamp.analysis.graph.**find_all_tree_nodes**(*tree*)
tree: A Tree instance.

cobamp.analysis.graph.**ignore_compressed_nodes_by_size**(*tree*, *size*)
Modifies the values of a tree's children that have been previously compressed with the <compress_linear_paths> function if they contain more than a certain number of elements. The node's value is changed to "REMAINING". Parameters ———- tree: A Tree instance size: An integer with the size threshold ——-

cobamp.analysis.graph.**merge_duplicate_nodes**(*tree*)
Merges all nodes with similar values, replacing every instance reference of all nodes with the same object if its value is identical

cobamp.analysis.graph.**populate_nx_graph**(*tree*, *G*, *previous=None*, *name_separator='\n'*, *unique_nodes=True*, *node_dict=None*)

cobamp.analysis.graph.**pretty_print_tree**(*tree*, *write_path=None*)
tree: A Tree instance write_path: Path to store a text file. Use None if the string is not to be stored in a file.

cobamp.analysis.graph.**probabilistic_tree_compression**(*tree*, *data=None*, *total_count=None*, *name_separator=' and '*)
Compresses a node and subsequent children by removing them and modifying the value to a dictionary with the relative frequency of each element in the subsequent nodes. Requires values on the extra_info field.

tree: A Tree instance data: Local count if not available in extra_info total_count: Total amount of sets if not available in extra_info name_separator: Separator to use when representing multiple elements ——-

cobamp.analysis.graph.**probabilistic_tree_prune**(*tree*, *target_level*, *current_level=0*, *cut_leaves=False*, *name_separator=' and '*)
Cuts a tree's nodes under a certain height (*target_level*) and converts ensuing nodes into a single one whose value represents the relative frequency of an element in the nodes below. Requires values on the extra_info

field. Parameters ———- tree: A Tree instance target_level: An int representing the level at which the tree will be cut current_level: The current level of the tree (int). Default is 0 for root nodes. cut_leaves: A boolean indicating whether the node at the target level is excluded or displays probabilities. name_separator: Separator to use when representing multiple elements ——-

cobamp.analysis.plotting module

cobamp.analysis.plotting.**annotate_heatmap**(*im, data=None, valfmt='{x:.2f}', textcolors=['black', 'white'], threshold=None, **textkw*)

A function to annotate a heatmap.

Arguments: im : The AxesImage to be labeled.

Optional arguments: data : Data used to annotate. If None, the image's data is used. valfmt : The format of the annotations inside the heatmap.

This should either use the string format method, e.g. "$ {x:.2f}", or be a `matplotlib.ticker.Formatter`.

textcolors [A list or array of two color specifications. The first is] used for values below a threshold, the second for those above.

threshold [Value in data units according to which the colors from] textcolors are applied. If None (the default) uses the middle of the colormap as separation.

Further arguments are passed on to the created text labels.

cobamp.analysis.plotting.**display_heatmap**(*df*)

cobamp.analysis.plotting.**heatmap**(*data*, *row_labels*, *col_labels*, *ax=None*, *cbar_kw={}*, *cbarlabel=''*, ***kwargs*)

Create a heatmap from a numpy array and two lists of labels.

Arguments: data : A 2D numpy array of shape (N,M) row_labels : A list or array of length N with the labels

for the rows

col_labels [A list or array of length M with the labels] for the columns

Optional arguments:

ax [A matplotlib.axes.Axes instance to which the heatmap] is plotted. If not provided, use current axes or create a new one.

cbar_kw [A dictionary with arguments to] `matplotlib.Figure.colorbar()`.

cbarlabel : The label for the colorbar

All other arguments are directly passed on to the imshow call.

Module contents

cobamp.core package

Submodules

cobamp.core.linear_systems module

class cobamp.core.linear_systems.**BendersMasterSystem**(*F*, *c*, *g*, *lb*, *ub*, *solver*)
Bases: *cobamp.core.linear_systems.GenericLinearSystem*

add_combinatorial_benders_cut(*x_sol*)

Args: x_sol:

build_problem()
Builds a CPLEX model with the constraints specified in the constructor arguments. This method must be implemented by any <LinearSystem>. Refer to the constructor ——-

remove_cuts()

class cobamp.core.linear_systems.**BendersSlaveSystem**(*A*, *M*, *D*, *b*, *e*, *lb_y*, *ub_y*, *solver=None*)
Bases: *cobamp.core.linear_systems.GenericLinearSystem*

parametrize(*x_sol*)

Args: x_sol:

class cobamp.core.linear_systems.**DualLinearSystem**(*S*, *lb*, *ub*, *T*, *b*, *solver=None*)
Bases: *cobamp.core.linear_systems.KShortestCompatibleLinearSystem*, *cobamp.core.linear_systems.GenericLinearSystem*

Class representing a dual system based on a steady-state metabolic network whose elementary flux modes are minimal cut sets for use with the KShortest algorithms. Based on previous work by Ballerstein et al. and Von Kamp et al.

[1] von Kamp, A., & Klamt, S. (2014). Enumeration of smallest intervention strategies in genome-scale metabolic networks. PLoS computational biology, 10(1), e1003378. [2] Ballerstein, K., von Kamp, A., Klamt, S., & Haus, U. U. (2011). Minimal cut sets in a metabolic network are elementary modes in a dual network. Bioinformatics, 28(3), 381-387.

generate_dual_problem(*S*, *irrev*, *T*, *b*)

Args: S: irrev: T: b:

class cobamp.core.linear_systems.**GenericLinearSystem**(*S*, *var_types*, *lb*, *ub*, *b_lb*, *b_ub*, *var_names*, *solver=None*)
Bases: *cobamp.core.linear_systems.LinearSystem*

Class representing a generic system of linear (in)equations Used as arguments for various algorithms implemented in the package.

build_problem()
Builds a CPLEX model with the constraints specified in the constructor arguments. This method must be implemented by any <LinearSystem>. Refer to the constructor ——-

class cobamp.core.linear_systems.**IrreversibleLinearPatternSystem**(*S*, *lb*, *ub*, *subset*, ***kwargs*)
Bases: *cobamp.core.linear_systems.IrreversibleLinearSystem*

build_problem()
Builds a CPLEX model with the constraints specified in the constructor arguments. This method must be implemented by any <LinearSystem>. Refer to the constructor ——-

class cobamp.core.linear_systems.**IrreversibleLinearSystem**(*S*, *lb*, *ub*, *non_consumed=()*, *consumed=()*, *produced=()*, *solver=None*, *force_bounds={}*)

Bases: *cobamp.core.linear_systems.KShortestCompatibleLinearSystem*, *cobamp.core.linear_systems.GenericLinearSystem*

Class representing a steady-state biological system of metabolites and reactions without dynamic parameters. All irreversible reactions are split into their forward and backward components so every lower bound is 0. Used as arguments for various algorithms implemented in the package.

class cobamp.core.linear_systems.**KShortestCompatibleLinearSystem**

Bases: *cobamp.core.linear_systems.LinearSystem*

Abstract class representing a linear system that can be passed as an argument for the KShortestAlgorithm class. Subclasses must instantiate the following variables:

__dvar_mapping: A dictionary mapping reaction indexes with variable names

__dvars: A list of variable names (str) or Tuple[str] if two linear system variables represent a single flux. Should be kept as type *list* to maintain order.

__c: str representing the variable to be used as constant for indicator constraints

add_c_variable()

c = None

dvar_mapping = None

dvars = None

get_c_variable()

get_dvar_mapping()

get_dvars()

class cobamp.core.linear_systems.**LinearSystem**

Bases: object

An abstract class defining the template for subclasses implementing linear systems that can be used with optimizers such as CPLEX and passed onto other algorithms supplied with the package.

Must instantiate the following variables:

S: System of linear equations represented as a n-by-m ndarray, preferrably with dtype as float or int

__model: Linear model as an instance of the solver.

add_columns_to_model(*S_new*, *var_names*, *lb*, *ub*, *var_types*)

Args: S_new: var_names: lb: ub: var_types:

add_rows_to_model(*S_new*, *b_lb*, *b_ub*, *only_nonzero=False*, *indicator_rows=None*, *vars=None*, *names=None*)

Args: S_new: b_lb: b_ub: only_nonzero: indicator_rows: vars: names:

add_variables_to_model(*var_names*, *lb*, *ub*, *var_types*)

Args: var_names: lb: ub: var_types:

build_problem()

Builds a CPLEX model with the constraints specified in the constructor arguments. This method must be implemented by any <LinearSystem>. Refer to the constructor ——-

dummy_variable()

empty_constraint(*b_lb*, *b_ub*, ***kwargs*)

Args: b_lb: b_ub: **kwargs:

get_configuration()

get_constraint_bounds(*constraints=None*)

Args: constraints:

get_constraint_matrices(*constraints=None*, *vars=None*)

Args: constraints: vars:

get_model()
Returns the model instance. Must call <self.build_problem()> to return a CPLEX model.

get_stoich_matrix_shape()
Returns a tuple with the shape (rows, columns) of the supplied stoichiometric matrix.

get_stuff(*what*, *index*)

Args: what: index:

get_system_matrix(*constraints=None*, *vars=None*)

Args: constraints: vars:

model = None

populate_constraints_from_matrix(*S*, *constraints*, *vars*, *only_nonzero=False*)

Args: S: Two-dimensional np.ndarray instance constraints (side of all): vars: list of variable instances only_nonzero:

populate_model_from_matrix(*S*, *var_types*, *lb*, *ub*, *b_lb*, *b_ub*, *var_names*, *only_nonzero=False*, *indicator_rows=None*)

Args: S: Two-dimensional np.ndarray instance var_types: list or tuple with length equal to the amount of columns of S

containing variable types (e.g.

lb: list or tuple with length equal to the amount of columns of S containing the lower bounds for all variables

ub: list or tuple with length equal to the amount of columns of S containing the upper bounds for all variables

b_lb: list or tuple with length equal to the amount of rows of S containing the lower bound for the right hand

b_ub: list or tuple with length equal to the amount of rows of S containing the upper bound for the right hand

var_names: string identifiers for the variables only_nonzero: indicator_rows:

remove_from_model(*index*, *what*)

Args: index: what:

select_solver(*solver=None*)
Args: solver:

set_constraint_bounds(*constraints*, *lb*, *ub*)

Args: constraints: lb: ub:

set_default_configuration()

set_number_of_threads(*n_threads=0*)
Defines the amount of threads available for the solver to use. :param n_threads: Number of threads available to the solver. Set to 0 if default parameters are needed :return:

Args: n_threads:

set_objective(*coefficients*, *minimize*, *vars=None*)

Args: coefficients: minimize: vars:

set_variable_bounds(*vars*, *lb*, *ub*)

Args: vars: lb: ub:

set_variable_types(*vars*, *types*)

Args: vars: types:

set_working_memory_limit(*workmem*)
Defines the amount of memory available for the solver to use. Use this at your own peril! :param n_threads: Memory in MegaBytes available to the solver :return:

Args: workmem:

was_built()

write_to_lp(*filename*)

Args: filename:

class cobamp.core.linear_systems.**SteadyStateLinearSystem**(*S*, *lb*, *ub*, *var_names*, *solver=None*)
Bases: *cobamp.core.linear_systems.GenericLinearSystem*

Class representing a steady-state biological system of metabolites and reactions without dynamic parameters Used as arguments for various algorithms implemented in the package.

cobamp.core.linear_systems.**bak_irrev**(*lb*, *ub*)

Args: lb: ub:

cobamp.core.linear_systems.**fix_backwards_irreversible_reactions**(*S*, *lb*, *ub*)

Args: S: lb: ub:

cobamp.core.linear_systems.**fwd_irrev**(*lb*, *ub*)

Args: lb: ub:

cobamp.core.linear_systems.**get_default_solver**()

cobamp.core.linear_systems.**get_solver_interfaces**()

cobamp.core.linear_systems.**make_irreversible_model**(*S*, *lb*, *ub*)

Args: S: lb: ub:

cobamp.core.linear_systems.**prepare_irreversible_system**(*self*, *S*, *lb*, *ub*, *non_consumed*, *consumed*, *produced*, *solver*, *force_bounds*)

Args: self: S: lb: ub: non_consumed: consumed: produced: solver: force_bounds:

cobamp.core.models module

class cobamp.core.models.**CORSOModel**(*cbmodel*, *corso_element_names=('R_PSEUDO_CORSO', 'M_PSEUDO_CORSO')*, *solver=None*)

Bases: *cobamp.core.models.ConstraintBasedModel*

optimize_corso(*cost*, *of_dict*, *minimize=False*, *constraint=1*, *constraintby='val'*, *eps=1e-06*, *flux1=None*)

set_corso_objective()

set_costs(*cost*)

solve_original_model(*of_dict*, *minimize=False*)

class cobamp.core.models.**ConstraintBasedModel**(*S*, *thermodynamic_constraints*, *reaction_names=None*, *metabolite_names=None*, *optimizer=True*, *solver=None*)

Bases: object

add_metabolite(*arg*, *name=None*)

add_reaction(*arg*, *bounds*, *name=None*)

decode_index(*index*, *labels*)

flux_limits(*reaction*)

get_bounds_as_list()

get_reaction_bounds(*index*)

get_stoichiometric_matrix(*rows=None*, *columns=None*)

initialize_optimizer()

is_reversible_reaction(*index*)

make_irreversible()

optimize(*coef_dict=None*, *minimize=False*)

remove_metabolite(*index*)

remove_reaction(*index*)

revert_to_original_bounds()

set_objective(*coef_dict*, *minimize=False*)

set_reaction_bounds(*index*, ***kwargs*)

set_stoichiometric_matrix(*values*, *rows=None*, *columns=None*)

class cobamp.core.models.**GIMMEModel**(*cbmodel*, *solver=None*)

Bases: *cobamp.core.models.ConstraintBasedModel*

optimize_gimme(*exp_vector*, *objectives*, *obj_frac*, *flux_thres*)

cobamp.core.models.**make_irreversible_model_raven**(*S*, *lb*, *ub*, *inverse_reverse_reactions=False*)

cobamp.core.optimization module

class cobamp.core.optimization.**BatchOptimizer**(*linear_system: cobamp.core.linear_systems.LinearSystem*, *threads=4*)

Bases: object

batch_optimize(*bounds*, *objective_coefs*, *objective_senses*)

class cobamp.core.optimization.**BendersDecompositionOptimizer**(*master_system*, *slave_system*, *hard_fail=False*, *build=True*)

Bases: object

optimize(*max_iterations=10000*, *slave_pool=20*)

class cobamp.core.optimization.**BendersSlaveOptimizer**(*slave_system*, *hard_fail*, *build*)

Bases: *cobamp.core.optimization.LinearSystemOptimizer*

optimize()

Internal function to instantiate the solver and return a solution to the optimization problem

Parameters

objective: A List[Tuple[coef,name]], where coef is an objective coefficient and name is the name of the variable to be optimized.

minimize: A boolean that, when True, defines the problem as a minimization

class cobamp.core.optimization.**CORSOSolution**(*sol_max*, *sol_min*, *f*, *index_map*, *var_names*, *eps=1e-08*)

Bases: *cobamp.core.optimization.Solution*

class cobamp.core.optimization.**GIMMESolution**(*sol*, *exp_vector*, *var_names*, *mapping=None*)

Bases: *cobamp.core.optimization.Solution*

get_reaction_activity(*flux_threshold*)

class cobamp.core.optimization.**KShortestSolution**(*value_map*, *status*, *indicator_map*, *dvar_mapping*, *dvars*, ***kwargs*)

Bases: *cobamp.core.optimization.Solution*

A Solution subclass that also contains attributes suitable for elementary flux modes such as non-cancellation sums of split reactions and reaction activity.

SIGNED_INDICATOR_SUM = 'signed_indicator_map'

SIGNED_VALUE_MAP = 'signed_value_map'

get_active_indicator_varids()

Returns the indices of active indicator variables (maps with variables on the original stoichiometric matrix)

class cobamp.core.optimization.**LinearSystemOptimizer**(*linear_system*, *hard_fail=False*, *build=True*)

Bases: object

Class with methods to solve a <LinearSystem> as a linear optimization problem.

optimize()

Internal function to instantiate the solver and return a solution to the optimization problem

Parameters

objective: A List[Tuple[coef,name]], where coef is an objective coefficient and name is the name of the variable to be optimized.

minimize: A boolean that, when True, defines the problem as a minimization

populate(*limit=None*)

class cobamp.core.optimization.**Solution**(*value_map*, *status*, ***kwargs*)
Bases: object

Class representing a solution to a given linear optimization problem. Includes an internal dictionary for additional information to be included.

attribute_names()
Returns all keys present in the attribute dictionary.

attribute_value(*attribute_name*)
Parameters

attribute_name: A dictionary key (preferrably str)

objective_value()
Returns the objective value for this solution

set_attribute(*key*, *value*)
Sets the value of a given <key> as <value>.

Parameters

key - A string

value - Any object to be associated with the supplied key

status()

to_series()

var_values()

x()
Returns a ndarray with the solution values in order (from the variables)

cobamp.core.optimization.**optimization_pool**(*lsystem*, *bound_change_list*, *objective_coef_list*, *objective_sense_list*, *threads=4*)

cobamp.core.optimization.**random_string_generator**(*N*)
Parameters

N : an integer

cobamp.core.transformer module

class cobamp.core.transformer.**ModelTransformer**
Bases: object

transform(*args*, *properties*)

transform_array(*S*, *lb*, *ub*, *properties*)

class cobamp.core.transformer.**ReactionIndexMapping**(*otn*, *nto*)
Bases: object

from_new(*idx*)

from_original(*idx*)

multiply(*new_ids*)

Module contents

cobamp.nullspace package

Submodules

cobamp.nullspace.nullspace module

cobamp.nullspace.nullspace.**compute_nullspace**(*A*, *eps=1e-09*, *left=True*)
Computes the nullspace of the matrix A.

Parameters

A: A 2D-ndarray

eps: Tolerance value for 0

left: A boolean value indicating whether the result is the left nullspace (right if False)

cobamp.nullspace.nullspace.**nullspace_blocked_reactions**(*K*, *tolerance*)
Parameters

K: A nullspace matrix as a 2D ndarray

tolerance: Tolerance value for 0

Returns indices of the rows of K where all values are 0

cobamp.nullspace.subset_reduction module

Inspired by Metatool's code

class cobamp.nullspace.subset_reduction.**SubsetReducer**
Bases: *cobamp.core.transformer.ModelTransformer*

ABSOLUTE_BOUNDS = 'SUBSET_REDUCER-ABSOLUTE_BOUNDS'

TO_BLOCK = 'SUBSET_REDUCER-TO_BLOCK'

TO_KEEP_SINGLE = 'SUBSET_REDUCER-TO_KEEP_SINGLE'

get_transform_maps(*sub*)

reduce(*S*, *lb*, *ub*, *keep=()*, *block=()*, *absolute_bounds=False*)

transform_array(*S*, *lb*, *ub*, *properties*)

class cobamp.nullspace.subset_reduction.**SubsetReducerProperties**(*keep=None*, *block=None*, *absolute_bounds=False*, *reaction_id_sep='_+_'*)
Bases: *cobamp.utilities.property_management.PropertyDictionary*

cobamp.nullspace.subset_reduction.**reduce**(*S*, *sub*, *irrev_reduced=None*)

Reduces a stoichiometric matrix according to the subset information present in the sub matrix and irrev_reduced.

Parameters

S: Stoichiometric matrix

sub: Subset matrix as computed by <subset_correlation_matrix>

irrev_reduced: Irreversibility vector regarding the subsets.

Returns reduced, reduced_indexes, irrev_reduced

cobamp.nullspace.subset_reduction.**subset_candidates**(*kernel*, *tol=None*)

Computes a matrix of subset candidates from the nullspace of the S matrix

Parameters

kernel: Nullspace of the S matrix

tol: Tolerance to 0.

cobamp.nullspace.subset_reduction.**subset_correlation_matrix**(*S*, *kernel*, *irrev*, *cr*, *keepSingle=None*)

S: Stoichiometric matrix as ndarray

kernel: The nullspace of S

irrev: List of booleans representing irreversible reactions (when True)

cr: The subset candidate matrix, computed using <subset_candidates>

keepSingle: List of reaction indices that will not be compressed.

Returns sub, irrev_sub, irrev_violating_subsets

sub : subset matrix, n-subsets by n-reactions -> numpy.array

irrev_sub : subset reversibilities -> numpy.array of type bool

irrev_violating_subsets : same as sub, but list if empty. Contains subsets discarded due to irreversibility faults

cobamp.nullspace.subset_reduction.**subset_reduction**(*S*, *irrev*, *to_remove=[]*, *to_keep_single=[]*)

Reduces a stoichiometric matrix using nullspace analysis by identifying linearly dependent (enzyme) subsets. These reactions are then compressed.

S: Stoichiometric matrix as an ndarray.

irrev: A boolean array with size equal to the number of columns in the S matrix.

to_remove: A list of indices specifying columns of the S matrix to remove before the compression (usually blocked reactions)

to_keep_single: A list of indices specifying columns of the S matrix not to compress.

Returns rd, sub, irrev_reduced, rdind, irrv_subsets, kept_reactions, kernel, correlation_matrix

rd : compressed stoichiometric matrix -> numpy.array

sub : subset matrix, n-subsets by n-reactions -> numpy.array

irrev_reduced : subset reversibilities -> numpy.array of type bool

rdind : metabolite indices -> numpy.array of type int

irrv_subsets : same as sub, but list if empty

kept_reactions : indexes for reactions used in the network compression

kernel : numpy.array with the right nullspace of S

correlation_matrix : numpy.array with reaction correlation matrix

Module contents

cobamp.utilities package

Submodules

cobamp.utilities.file_utils module

cobamp.utilities.file_utils.**open_file**(*path*, *mode*)

cobamp.utilities.file_utils.**pickle_object**(*obj*, *path*)
Stores an object as a file. Parameters ———- obj: The object instance path: Full path as a str where the file will be stored.

cobamp.utilities.file_utils.**read_pickle**(*path*)
Reads a file containing a pickled object and returns it Parameters ———- path: Full path as a str where the file is stored.

cobamp.utilities.postfix_expressions module

class cobamp.utilities.postfix_expressions.**Queue**
Bases: list

push(*x*)

top()

class cobamp.utilities.postfix_expressions.**Stack**
Bases: list

push(*x*)

top()

cobamp.utilities.postfix_expressions.**boolean_precedence**(*token*)

cobamp.utilities.postfix_expressions.**eval_boolean_operator**(*operator*, *o1*, *o2*)

cobamp.utilities.postfix_expressions.**eval_math_operator**(*operator*, *o1*, *o2*)

cobamp.utilities.postfix_expressions.**evaluate_postfix_expression**(*op*, *eval_fx*, *type_conv=<class 'int'>*)

cobamp.utilities.postfix_expressions.**is_boolean_operator**(*token*)

cobamp.utilities.postfix_expressions.**is_boolean_value**(*token*)

cobamp.utilities.postfix_expressions.**is_number_token**(*token*)

cobamp.utilities.postfix_expressions.**is_operator_token**(*token*)

cobamp.utilities.postfix_expressions.**is_string_token**(*token*)

cobamp.utilities.postfix_expressions.**left_operator_association**(*op*)

cobamp.utilities.postfix_expressions.**op_prec**(*op*)

cobamp.utilities.postfix_expressions.**parse_infix_expression**(*op*, *is_operand_fx*, *is_operator_fx*, *precedence_fx*)

cobamp.utilities.postfix_expressions.**tokenize_boolean_expression**(*inf_exp_str*)

cobamp.utilities.postfix_expressions.**tokenize_infix_expression**(*inf_exp_str*)

cobamp.utilities.property_management module

class cobamp.utilities.property_management.**PropertyDictionary**(*mandatory_properties={}*, *optional_properties={}*)

Bases: object

Implements a dict with additional control on which objects can be added to which keys and whether these are optional or mandatory.

add_if_not_none(*key*, *value*)

add_new_properties(*mandatory_properties*, *optional_properties*)

Adds new properties to the dictionary and/or updates existing ones, if present. Parameters ———- mandatory_properties: A dict[str, function] optional_properties: A dict[str, function] ——-

get_mandatory_properties()

get_optional_properties()

has_required_properties()

cobamp.utilities.set_utils module

cobamp.utilities.set_utils.**has_no_overlap**(*set1*, *set2*)

set1: A set of frozenset instance. set2: A set of frozenset instance.

cobamp.utilities.set_utils.**is_identical**(*set1*, *set2*)

set1: A set of frozenset instance. set2: A set of frozenset instance.

cobamp.utilities.set_utils.**is_subset**(*which*, *of_what*)

which: A set or frozenset instance to be checked as a possible subset of *of_what* of_what: A set or frozenset instance

cobamp.utilities.test_utils module

cobamp.utilities.test_utils.**timeit**(*method*)

Timer decorator for methods. Courtesy of Fahim Sakri from PythonHive Args:

method:

Returns:

Module contents

cobamp.wrappers package

Submodules

cobamp.wrappers.external_wrappers module

class cobamp.wrappers.external_wrappers.**AbstractObjectReader**(*model*)
Bases: object

An abstract class for reading metabolic model objects from external frameworks, and extracting the data needed for pathway analysis methods. Also deals with name conversions.

convert_constraint_ids(*tup*, *yield_constraint*)

convert_gprs_to_list(*rx*, *apply_fx*)

decode_k_shortest_solution(*solarg*)

g2rx(*expression*, *and_fx=<built-in function min>*, *or_fx=<built-in function max>*, *as_vector=False*, *apply_fx=None*)

get_gene_protein_reaction_rule(*id*)

get_irreversibilities(*as_index*)
Returns a vector representing irreversible reactions, either as a vector of booleans (each value is a flux, ordered in the same way as reaction identifiers) or as a vector of reaction indexes.

Parameters

as_dict: A boolean value that controls whether the result is a vector of indexes

get_model_bounds(*as_dict*, *separate_list*)
Returns the lower and upper bounds for all fluxes in the model. This either comes in the form of an ordered list with tuples of size 2 (lb,ub) or a dictionary with the same tuples mapped by strings with reaction identifiers.

Parameters

as_dict: A boolean value that controls whether the result is a dictionary mapping str to tuple of size 2 separate: A boolean value that controls whether the result is two numpy.array(), one for lb and the other

to ub

get_model_genes()
Returns the identifiers for the genes contained in the model

get_model_gprs(*apply_fx=None*)
Returns the model gene-protein-reaction rules associated with each reaction

get_reaction_and_metabolite_ids()
Returns two ordered iterables containing the metabolite and reaction ids respectively.

get_rx_instances()
Returns the reaction instances contained in the model. Varies depending on the framework.

get_stoichiometric_matrix()
Returns a 2D numpy array with the stoichiometric matrix whose metabolite and reaction indexes match the names defined in the class variables r_ids and m_ids

get_working_gene_names()

metabolite_id_to_index(*id*)
Returns the numerical index of a metabolite when given a string representing its identifier.

Parameters

id: A metabolite identifier as a string

reaction_id_to_index(*id*)
Returns the numerical index of a reaction when given a string representing its identifier.

Parameters

id: A reaction identifier as a string

to_cobamp_cbm(*solver=None*)

class cobamp.wrappers.external_wrappers.**COBRAModelObjectReader**(*model*)
Bases: *cobamp.wrappers.external_wrappers.AbstractObjectReader*

convert_gprs_to_list(*rx*, *apply_fx*)

get_irreversibilities(*as_index*)
Returns a vector representing irreversible reactions, either as a vector of booleans (each value is a flux, ordered in the same way as reaction identifiers) or as a vector of reaction indexes.

Parameters

as_dict: A boolean value that controls whether the result is a vector of indexes

get_model_bounds(*as_dict=False*, *separate_list=False*)
Returns the lower and upper bounds for all fluxes in the model. This either comes in the form of an ordered list with tuples of size 2 (lb,ub) or a dictionary with the same tuples mapped by strings with reaction identifiers.

Parameters

as_dict: A boolean value that controls whether the result is a dictionary mapping str to tuple of size 2 separate: A boolean value that controls whether the result is two numpy.array(), one for lb and the other

to ub

get_model_genes()
Returns the identifiers for the genes contained in the model

get_model_gprs(*apply_fx=None*, *token_to_gene_ratio=20*)
Returns the model gene-protein-reaction rules associated with each reaction

get_reaction_and_metabolite_ids()
Returns two ordered iterables containing the metabolite and reaction ids respectively.

get_rx_instances()
Returns the reaction instances contained in the model. Varies depending on the framework.

get_stoichiometric_matrix()
Returns a 2D numpy array with the stoichiometric matrix whose metabolite and reaction indexes match the names defined in the class variables r_ids and m_ids

class cobamp.wrappers.external_wrappers.**CobampModelObjectReader**(*model*)
Bases: *cobamp.wrappers.external_wrappers.AbstractObjectReader*

get_irreversibilities(*as_index*)
Returns a vector representing irreversible reactions, either as a vector of booleans (each value is a flux, ordered in the same way as reaction identifiers) or as a vector of reaction indexes.

Parameters

as_dict: A boolean value that controls whether the result is a vector of indexes

get_model_bounds(*as_dict*, *separate_list=False*)
Returns the lower and upper bounds for all fluxes in the model. This either comes in the form of an ordered list with tuples of size 2 (lb,ub) or a dictionary with the same tuples mapped by strings with reaction identifiers.

Parameters

as_dict: A boolean value that controls whether the result is a dictionary mapping str to tuple of size 2 separate: A boolean value that controls whether the result is two numpy.array(), one for lb and the other

to ub

get_reaction_and_metabolite_ids()
Returns two ordered iterables containing the metabolite and reaction ids respectively.

get_rx_instances()
Returns the reaction instances contained in the model. Varies depending on the framework.

get_stoichiometric_matrix()
Returns a 2D numpy array with the stoichiometric matrix whose metabolite and reaction indexes match the names defined in the class variables r_ids and m_ids

class cobamp.wrappers.external_wrappers.**FramedModelObjectReader**(*model*)
Bases: *cobamp.wrappers.external_wrappers.AbstractObjectReader*

convert_gprs_to_list(*rx*)

get_irreversibilities(*as_index*)
Returns a vector representing irreversible reactions, either as a vector of booleans (each value is a flux, ordered in the same way as reaction identifiers) or as a vector of reaction indexes.

Parameters

as_dict: A boolean value that controls whether the result is a vector of indexes

get_model_bounds(*as_dict=False*, *separate_list=False*)
Returns the lower and upper bounds for all fluxes in the model. This either comes in the form of an ordered list with tuples of size 2 (lb,ub) or a dictionary with the same tuples mapped by strings with reaction identifiers.

Parameters

as_dict: A boolean value that controls whether the result is a dictionary mapping str to tuple of size 2 separate: A boolean value that controls whether the result is two numpy.array(), one for lb and the other

to ub

get_reaction_and_metabolite_ids()
Returns two ordered iterables containing the metabolite and reaction ids respectively.

get_rx_instances()
Returns the reaction instances contained in the model. Varies depending on the framework.

get_stoichiometric_matrix()
Returns a 2D numpy array with the stoichiometric matrix whose metabolite and reaction indexes match the names defined in the class variables r_ids and m_ids

class cobamp.wrappers.external_wrappers.**MatFormatReader**(*model*)
Bases: *cobamp.wrappers.external_wrappers.AbstractObjectReader*

convert_gprs_to_list(*rx*, *apply_fx*)

get_irreversibilities(*as_index*)
Returns a vector representing irreversible reactions, either as a vector of booleans (each value is a flux, ordered in the same way as reaction identifiers) or as a vector of reaction indexes.

Parameters

as_dict: A boolean value that controls whether the result is a vector of indexes

get_model_bounds(*as_dict=False*, *separate_list=False*)
Returns the lower and upper bounds for all fluxes in the model. This either comes in the form of an ordered list with tuples of size 2 (lb,ub) or a dictionary with the same tuples mapped by strings with reaction identifiers.

Parameters

as_dict: A boolean value that controls whether the result is a dictionary mapping str to tuple of size 2 separate: A boolean value that controls whether the result is two numpy.array(), one for lb and the other

to ub

get_model_genes()
Returns the identifiers for the genes contained in the model

get_model_gprs(*apply_fx=None*)
Returns the model gene-protein-reaction rules associated with each reaction

get_reaction_and_metabolite_ids()
Returns two ordered iterables containing the metabolite and reaction ids respectively.

get_rx_instances()
Returns the reaction instances contained in the model. Varies depending on the framework.

get_stoichiometric_matrix()
Returns a 2D numpy array with the stoichiometric matrix whose metabolite and reaction indexes match the names defined in the class variables r_ids and m_ids

cobamp.wrappers.external_wrappers.**normalize_boolean_expression**(*rule*)

cobamp.wrappers.method_wrappers module

class cobamp.wrappers.method_wrappers.**KShortestEFMEnumeratorWrapper**(*model*, *non_consumed*, *consumed*, *produced*, *subset=None*, ***kwargs*)
Bases: *cobamp.wrappers.method_wrappers.KShortestEnumeratorWrapper*

Extension of the abstract class KShortestEnumeratorWrapper that takes a metabolic model as input and yields elementary flux modes.

get_linear_system()

class cobamp.wrappers.method_wrappers.**KShortestEFPEnumeratorWrapper**(*model*, *subset*, *non_consumed=[]*, *consumed=[]*, *produced=[]*, ***kwargs*)

Bases: *cobamp.wrappers.method_wrappers.KShortestEnumeratorWrapper*

Extension of the abstract class KShortestEnumeratorWrapper that takes a metabolic model as input and yields elementary flux patterns.

get_linear_system()

class cobamp.wrappers.method_wrappers.**KShortestEnumeratorWrapper**(*model*, *algorithm_type='kse_populate'*, *stop_criteria=1*, *forced_solutions=None*, *excluded_solutions=None*, *solver='CPLEX'*, *force_bounds={}*, *n_threads=0*, *workmem=None*, *big_m=False*, *max_populate_sols_override=None*, *time_limit=None*, *big_m_value=None*)

Bases: object

ALGORITHM_TYPE_ITERATIVE = 'kse_iterative'

ALGORITHM_TYPE_POPULATE = 'kse_populate'

get_enumerator()

Returns an iterator that yields a single EFM or a list of multiple EFMs of the same size. Call next(iterator) to obtain the next set of EFMs.

get_linear_system()

class cobamp.wrappers.method_wrappers.**KShortestMCSEnumeratorWrapper**(*model*, *target_flux_space_dict*, *target_yield_space_dict*, ***kwargs*)

Bases: *cobamp.wrappers.method_wrappers.KShortestEnumeratorWrapper*

Extension of the abstract class KShortestEnumeratorWrapper that takes a metabolic model as input and yields minimal cut sets.

get_linear_system()

Module contents

3.1.2 Module contents

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

X